Proof Script Pragmatics in IMPS*

William M. Farmer, Joshua D. Guttman, Mark E. Nadel, F. Javier Thayer

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420, USA

{farmer,guttman,men,jt}@mitre.org

March 28, 1994

Abstract. This paper introduces the IMPS proof script mechanism and some practical methods for exploiting it.

1 Introduction

IMPS, an Interactive Mathematical Proof System [4, 2], is intended to serve three ultimate purposes:

- To provide mathematics education with a mathematics laboratory for students to develop axiomatic theories, proofs, and rigorous methods of symbolic computation.
- To provide mathematical research with mechanized support covering a range of concrete and abstract mathematics, eventually with the help of a large theory library of formal mathematics.
- To allow applied formal methods to use flexible approaches to formalizing problem domains and proof techniques, in showing software or hardware correctness.

Thus, the goal of IMPS is to provide mechanical support for traditional methods and activities of mathematics, and for traditional styles of mathematical proof. Other automated theorem provers may be intended for quite different sorts of problems, and they can therefore be designed on quite different principles. For instance, some are meant to act as the back ends for AI systems that need to prove simple theorems about simplified worlds. However, theorem provers of this latter kind will not be able to serve the purposes we have mentioned, for which a wide range of traditional mathematical techniques must be supported.

In this paper we will focus on the IMPS *proof script* mechanism; we intend to illustrate why it aids in developing the large bodies of mathematical theory that are needed for these goals. As such, the paper is intended as a pragmatic one

* Supported by the MITRE-Sponsored Research program. To appear in the proceedings of the 12th International Conference on Automated Deduction (Nancy, France, June 28–July 1, 1994), *Lecture Notes in Computer Science*, Springer-Verlag.

rather than a theoretical one. We aim to emphasize concrete, practical ways of getting proofs done. We believe that the IMPS proof script mechanism aids the user in carrying out proofs, in tailoring and reusing previously developed proof ideas, and in conveying the essential content and structure of proofs.

Our pragmatic approach derives from our view that, in mechanized theorem proving, what happens on the outside may be more important than what happens on the inside. For example, a user may be more sensitive to the time it takes him to do some simple data entry tasks, or to find a clever encoding of a mathematical idea, than he is to the time it takes the machine to explore deduction steps. Frequently, one should be less concerned with what it is possible to do and more concerned with what can be done conveniently.

In particular, our aim is to provide flexible and convenient ways of manipulating and reusing proofs. The operations that will serve this purpose cannot be determined from proof theory, but primarily from experience. Given our aim, it is not necessary to adopt an object logic in which proofs themselves are first class objects, because we do not aim to prove things about proofs. On the contrary, it is more important to make it very easy for a user to get his hands on his proofs and (especially) partial proof attempts; to make it very cheap for him to restart proofs and reexecute portions; and to encourage an experimental attitude to proof construction.

Mechanizing mathematics is widely acknowledged to be hard work. It forces us to a more formal level, and at the same time to a more concrete representational level, than we would normally adopt in standard mathematical practice. Thus, offsetting the benefit that we gain confidence in the correctness of our proofs, there are the extra burdens that most everyone who has dealt with a mechanized theorem prover has surely experienced. Effective styles of usage are needed to mitigate these burdens and to provide new methods exploiting the more concrete structure we have at our disposal. Below, we will describe some techniques we have found so far.

IMPS is self-consciously an interactive system; however, we believe that current mechanized theorem provers intended for mathematics are all interactive in one sense or another. The kinds of interaction can vary from the crafting of an appropriate sequence of lemmas to reach the theorem, to the setting of various switches before control is passed to the machine, to the user supplying most of the proof by hand. Perhaps a more interesting distinction than the familiar but flawed one of "degree of autonomy" is the distinction between systems in which the user interacts only between attempts to construct a proof, and those in which he interacts during the process of constructing a proof. IMPS is emphatically of the second kind. We find that, in the first kind of system, the human helps the machine to prove the theorem, while in the second kind, the machine can help the human. Partly for this reason, IMPS provides relatively large proof steps in many of its proof commands, which aids the user in correlating the steps in constructing an IMPS proof with the successive portions of an intuitive proof sketch.

In order to encourage a substantial community of users to adopt IMPS, we want to describe a range of styles of effective usage. These styles of usage are best refined only after substantial experience has been gained in using the system for a particular type of problem. Over time, these styles of usage evolve in tandem with improvements or adaptations in the theorem prover itself. In this paper we convey some aspects of a successful style for using the IMPS facilities for interactive and script-based proof.

The paper is organized as follows. Section 2 introduces special procedures for applying theorems called *macetes*, which play a fundamental role in the IMPS proof system. Sections 3–5 describe the IMPS proof script mechanism and ways it can be put to use. Section 6 briefly compares IMPS proof scripts and macetes with traditional tactics. And Section 7 contains a conclusion.

2 Macetes

Macetes supplement the IMPS simplifier [4, 5] in order to provide more flexibility to the user. The simplifier applies universally quantified equalities as rewrite rules in a manner which is usually beyond the user's control. In particular, it is not possible for the user to direct the simplifier to apply only those theorems that belong to a specified set of theorems. This rigidity of rewrite rule application clearly clashes with normal mathematical practice, where theorems are usually applied, individually or in groups, in a way which is dependent on content and ultimately determined by the mathematics practitioner.

In IMPS the macete mechanism is designed to provide a simple facility to extend the simplifier in straightforward ways (or build simple simplifiers from scratch) so that the user has more control over what theorems get applied. Macetes are of two basic kinds: atomic and compound. For instance, when a theorem is installed (after it has been proved), a corresponding atomic macete (called a theorem macete) is automatically created. These theorem macetes do a variety of kinds of conditional rewriting depending on the syntactic form of the underlying theorem. Theorem macetes are created for all theorems, even those that are not conditional equalities [5]. A theorem may be applied as a macete using ordinary matching or using *translation matching*, an inter-theory form of expression matching which allows a theorem to be applied outside of its home theory [3]. Atomic macetes also include simplification and beta-reduction.

Compound macetes are specified using an extremely simple language for determining control of the process of applying atomic macetes. This language provides a few simple constructors for sequencing and iteration of arbitrary macetes.

Macetes are applied via special proof commands that add at most one inference to the deduction graph. Since the number of macetes that are loaded into the system may be large (1000 macetes is typical), the facility would not be too practical if the user had to guess the right one out of the blue. To deal with this IMPS provides a special menu—the "macete menu"—to tell the user which macetes may be applicable to a given subgoal. In situations where over 1000 macetes are available, there are rarely more than 10 macetes presented to the user. Consequently, this menu can be used to provide guidance on what to do next. It provides information to the novice about what is available in the theory library, and it provides feedback to the expert about the essential content of his subgoal.

3 Proof Scripts in IMPS

A logical deduction is represented in IMPS as a kind of directed graph called a *deduction graph* [4]. An IMPS user initially sees proof as an interactive process—frequently with much trial and error—acting on a deduction graph. In this view, developing the proof means issuing a sequence of commands, with the IMPS user interface supplying a good deal of information about what steps may be useful. Each command is applied to a specific node in the deduction graph and produces (zero or more) additional nodes. Roughly speaking, the graph structure represents the relation of entailment between the nodes. When an inference supports a node with subgoal nodes, and all of the subgoals are recognized to be true ("grounded"), then the node is also grounded. A proof is complete when its original goal node is grounded. Throughout the proof, the user has a current node, which may be freely changed between commands. After each command, the system selects a current node. The next command will apply to this newly selected node unless the user explicitly changes the current node. When a command has added new unsupported subgoals, the new current node is generally the leftmost; when a command has grounded its node, the system generally chooses the leftmost unsupported descendent of the nearest ungrounded ancestor.[1] IMPS enforces a goal directed style of reasoning, in which the proof is constructed from the conclusion backwards using a sequent-based system of rules.

A *proof script* (or *script* for short) is a sequence of certain s-expressions that, when executed, applies a sequence of commands to a deduction graph. An example of a proof script is shown in Figure 1. The structure of the deduction graph and the default way of selecting a new current node determine how these commands are applied to the deduction graph. An intimate knowledge of the syntax for proof scripts is unimportant: no one types scripts. Instead, the user interacts with the system through its interface, usually selecting from menus generated on the fly, to carry out the commands. At any point in the process, the user can ask IMPS to create a proof script that is a transcript of the proof or partial proof. In practice, all proof scripts are created from these basic transcripts of interactive execution, by editing them to introduce control structures.

The s-expressions in a proof script are of several kinds:

– Command forms do the ultimate work of adding new nodes to the deduction graph. In the example shown in Figure 1, applying macetes such as `tr%subseteq-antisymmetry` and `indicator-facts-macete` and doing the direct inferences are command forms.

[1] A more elaborate scheme is used when the nearby portion of the deduction graph is not tree-like.

– Node motion forms cause script execution to continue at a node other than the natural continuation node; for instance, `(jump-to-node top)`.
– Assignment forms may define local macetes or invocable scripts (see below) which can be referenced at other places within the script. This is especially useful in certain kinds of proofs, such as proofs by symmetry, which involve two or more instances of the same argument. `(label-node top)` is a different kind of assignment form.
– Conditionals subordinate the execution of a portion of a script to the validity of a certain condition. Typical conditions are that the assertion of the goal sequent matches a given expression or that a particular subscript succeeds in adding inferences to the proof.
– Iteration forms provide for execution of a subscript while a specified condition holds, or over a specified range of nodes. In Figure 1, `for-nodes` begins an iteration over the set of unsupported descendents of the target node.
– Block and comment forms provide structure and documentation.

```
((label-node top)
 (apply-macete-with-minor-premises tr%subseteq-antisymmetry)
 (script-comment
   "Replace equation with two inclusions.")
 direct-inference
 (jump-to-node top)
 (for-nodes
   (unsupported-descendents)
   (block
     insistent-direct-inference
     (apply-macete-with-minor-premises indicator-facts-macete)
     beta-reduce-repeatedly))))
```

Fig. 1. Script for proving sets are equal

While a script is executing, it maintains two distinguished nodes in the deduction graph. First, there is the *head* node, which remains fixed. Second, there is the *current* node, which starts off as the head node, and as execution progresses, evolves according to the default selection rules described above or the dictates of explicit node motion forms included in the script.

We regard a script execution as an attempt to provide a proof, or part of a proof, for a particular subgoal node, namely the one selected as the head node. For this reason if, part way through execution, the head node should become grounded, the remainder of the script is discarded. There is nothing more for it usefully to do. We have found that this principle greatly improves the robustness and predictability of the script mechanism. Without it, there is the risk that "overachieving" scripts will carry out meaningless proof steps in some adjacent portion of the deduction, with the consequence that later proof commands may by default apply to the wrong subgoals. As a special case, a *block* is an anonymous script procedure with no parameters. In effect, it merely introduces a head node and discards any of the nested commands that may remain after the head node is grounded.

These remarks describe the possible "moves" in interacting with IMPS. Later we will we describe some of the "strategies" we employ in playing the game.

3.1 Invocable Scripts

The most basic user level inference steps are given by built-in IMPS *proof commands*; these are Lisp procedures[2] which call *primitive inferences* in useful patterns. Assuming that the primitive inferences, several of which carry out sophisticated reasoning steps, are correctly implemented, proof commands are guaranteed only to make sound inferences, because they modify deduction graphs only by calling primitive inferences. There are approximately 60 built-in proof commands in IMPS. A single command will, in general, add several nodes, and often several levels of nested subgoals, to the deduction graph. Moreover, the same command, issued in different contexts, may add different numbers of nodes or levels.

In theory, new proof commands can be added to IMPS by directly writing new Lisp procedures, although writing such procedures is usually difficult. An *invocable script* is a new proof command created by the user from a proof script. It can be invoked—either interactively or in other scripts—just like the basic IMPS proof commands implemented in the underlying Lisp. When requested, IMPS tells the user which proof commands, whether built-in or user-defined, are possibly applicable to a given subgoal.

The parameters to an invocable script are untyped, and referenced positionally by positive integers. Their actual values may be:

- integers;
- strings, used to represent expressions; or
- symbols, used to represent theorems, macetes, and commands primarily.

An invocable script may be defined at the top level, as a globally available command, using the `def-script` form as in Figure 2. Alternatively, it may be purely local to an encompassing proof script. For instance, the local invocable script in Figure 3 contraposes against the assumption matching the pattern given as argument, after which it uses a group of theorems about the algebra of fractions as a macete, before finally calling the simplifier. This local invocable script is then used three times in the proof fragment that follows.

Invocable scripts may take other invocable scripts or other commands as arguments, as was illustrated by the example in Figure 2. It repeatedly applies direct inferences (sequent calculus right introduction rules) and antecedent inferences (sequent calculus left introduction rules) backwards to generate a set of subgoals, before finally applying the command given by its actual parameter

[2] Yale's T dialect of Scheme [10] is in fact the implementation language.

```
(def-script command-on-direct-descendents 1
  ((label-node compound)
   direct-and-antecedent-inference-strategy
   (jump-to-node compound)
   (for-nodes
    (unsupported-descendents)
    $1)))
```

Fig. 2. Invocable script with a command parameter

```
(let-script
 contrapose-denom-remove 1
 ;; The arg is the pattern to contrapose on
 ;;
 ((contrapose $1)
  (apply-macete-with-minor-premises
   fractional-expression-manipulation)
  simplify))
($contrapose-denom-remove "with(r:rr,r<0);")
($contrapose-denom-remove "with(r:rr,r=0);")
($contrapose-denom-remove "with(r:rr,r=1);")
```

Fig. 3. Local invocable script and its application

to every leaf node introduced in this process. This invocable script is frequently called with the parameter `simplify`, although the example of Figure 1 could be rewritten by passing the contents of the block as its argument.

4 Proof by Emacs

A crucial advantage of the IMPS script language is that simple textual manipulations of the scripts allow a user to reuse proofs or portions of proofs in a highly predictable way. In many cases, a very superficial understanding of many portions of a proof is enough to enable a user to transform it into a proof of another theorem.

As an example, consider Figure 4, which was created while developing parts of freshman calculus using nonstandard analysis. It establishes the theorem that the limit of a sum equals the sum of the individual limits. The proof script contained within it (below the word `proof`) is given almost as it appears in our files; we have added only the marker `&` appearing at the right of some of the lines. It is not necessary to understand the script completely. That is part of the point.

Suppose that we now want to prove the analogous theorem about the limit of a product. This theorem about products does not follow from the theorem

```
(def-theorem sum-of-limits
  "forall(f,g:[rr,rr], c:rr, #(lim(f,c)) and #(lim(g,c)) implies
     lim(lambda(x:rr,f(x)+g(x)),c)=lim(f,c)+lim(g,c))"          &
  (theory nsa-theory)
  (proof
   (direct-and-antecedent-inference-strategy
    (apply-macete-with-minor-premises
     iota-free-characterization-of-lim)
    direct-and-antecedent-inference-strategy
    (apply-macete-with-minor-premises ast-composition-binary)
    beta-reduce-repeatedly
    (force-substitution "ast(+)" "++" (0))                      &
    (move-to-sibling 1)
    simplify
    extensionality
    (unfold-single-defined-constant (0) ++)                     &
    (apply-macete-with-minor-premises additivity-of-st)         &
    (apply-macete-with-minor-premises
     iota-free-characterization-of-lim-existence)
    (unfold-single-defined-constant-globally ++)                &
    (apply-macete-with-minor-premises ast-extends-compound)
    (apply-macete-with-minor-premises
     lim-existence-implies-finite-on-monad)
    direct-and-antecedent-inference-strategy
    (apply-macete-with-minor-premises
     lim-existence-implies-finite-on-monad)
    direct-and-antecedent-inference-strategy)))
```

Fig. 4. Limit of a sum

about sums. Moreover, in most standard treatments the proofs are considerably different. However, in our nonstandard treatment there is a proof of the theorem for products that is very much like the theorem for sums. The extra work that one would expect to have to do is encapsulated in a lemma corresponding to the lemma `additivity-of-st`.

A user more or less familiar with what was going on, but who did not necessarily follow the proof completely, can recognize that only the lines with a `&` to the right seem to have anything to do with addition in particular. The other lines are more or less generic with respect to the issue here. We now change just those lines to the corresponding ones for multiplication, generally using global-replace or query-replace in Emacs; hence the description "proof by Emacs." In this case, $+$ is replaced with $*$ and `additivity` is replaced by `multiplicativity`, which suffices to produce both the theorem to be proved and also the proof script. When we run this new script on the new theorem, we see that a complete proof is obtained. One might ask how the user is to know the "corresponding" lines. Although ultimately this is a matter of mathematical understanding, IMPS can provide some assistance, as our next example will illustrate.

Continuing our excursion through freshman calculus, consider the analogous theorem on the limit of a quotient. Suppose we try exactly the same approach. Imagine we have changed the addition to division, as we changed the addition to multiplication above. We might mistakenly assume that the analogue of `multiplicativity-of-st` is called `divisibility-of-st`, and make the change accordingly. We now run the script, but when it tries to execute

```
(apply-macete-with-minor-premises divisibility-of-st)
```

it returns the error message that there is no such macete. IMPS will help us find the correct name. We then jump back into the script at the point this was attempted, or rerun the portion of the script up to that point. Now, the correct choice `st-of-quotient` will pop up in the menu of applicable macetes, and it can be inserted. One might argue that our naming convention was not very consistent, but how much consistency should you expect, especially when theory libraries are developed by different users?

We are not done yet, as the observant reader will have noticed, because this "theorem" is not true. When we run the new script, we no longer get an error message, but after all the commands have been executed, the goal node is not grounded. The user now examines the ungrounded nodes and considers what IMPS was unable to prove. The user might examine, for example, the default current node. A quick look, and perhaps an IMPS simplification, will make it quite clear that the problem is that the hypothesis saying that the limit of the denominator is nonzero is missing. Without this hypothesis, there is no way to discharge the proof obligation to show that the denominator is nonzero or that the quotient is defined. Amending the hypothesis, rerunning the script, and doing a few additional steps will ground the corrected theorem. Naturally, the process will not always proceed as smoothly, but it would be unrealistic to expect that it would.

In extreme cases of proof by Emacs, no editing whatever is needed to reuse a script. In proving that a finite set S with n elements has 2^n subsets, a key ingredient for the induction is that if you remove any element x from S, then every subset of S is either a subset of $S \setminus \{x\}$ or of the form $A \cup \{x\}$, where A is such a subset. Figure 5 contains a proof script for the lemma that asserts this. There is an analogous theorem that states that if a set S has n elements, then for $k \leq n$, S has $\binom{n}{k}$ subsets of cardinality k. The inductive proof of that theorem depends on an analogue of the above lemma for decomposing the k element subsets. It turns out that the script displayed in Figure 5 will also ground this theorem, even though the proof in the strict sense (for instance, represented as a deduction graph) is entirely different.

Of course, not all IMPS proofs are, or could be, "done by Emacs" in any significant way. On the other hand, however, it is often the case in mathematics that a claim is made that the proof of theorem B is similar to the proof of theorem A, with the implicit or sometimes explicit suggestion that the details are left to the reader. Often the reader does not check the details, and more significantly, neither does the author. Not infrequently, what looks quite similar

```
(def-theorem power-decomposition
  "forall(s:sets[ind_1], x:ind_1, n:nn, x in s implies
     power(s)=
     power(s diff singleton{x}) union power(s) inters filter(x))"
  (theory generic-theory-1)
  (proof
   (direct-and-antecedent-inference-strategy
    unfold-defined-constants
    set-equality-script
    direct-and-antecedent-inference-strategy
    (contrapose "with(p:prop,not(p));")
    simplify-insistently
    (contrapose "with(p:prop,not(p));")
    simplify
    direct-and-antecedent-inference-strategy
    set-containment-script
    direct-and-antecedent-inference-strategy
    (incorporate-antecedent
     "with(f,x:sets[ind_1],x subseteq f)")
    simplify-insistently )))
```

Fig. 5. Power set decomposition lemma

on a superficial level is quite different when all the details need to be supplied. This sometimes leads to errors.

In such cases, "proof by Emacs" can be extremely useful, particularly because it is cheap and easy. A user can try a number of scripts quickly without taking much time or effort, and especially, without requiring much hard thinking, which is especially important at the end of a long day. When a successful proof is found, one can then reflect upon it at one's convenience, to absorb the mathematical content. Although one might waste one's time in vain attempts, the underlying soundness of IMPS insures that if one does ground the theorem, then it has really been proved, even if the user does not yet understand exactly why it worked.

The relatively high level of IMPS scripts is needed: it preserves the similarity on the superficial level, which often breaks down if one must descend to greater detail. Recall that a single command in a script will often generate many steps in the underlying proof, and the same command may generate different proof steps in different contexts. Consequently, by using scripts we can take advantage of similarity on the more schematic level. IMPS automatically supplies the necessary differences in detail.

"Proof by Emacs" may prove a useful technique in cooperative mathematics research and mathematics education. It may be possible to borrow proofs which might involve techniques or concepts outside the expertise of the borrower, but which he could then adapt for his purposes. This often happens informally today, but with far less assurance that the adaptation is legitimate. Similarly, instructors can give students worked examples to adapt, allowing the students to verify their adaptations for themselves. Of course, this sort of thing is also done now, except that either details are omitted or students do not get timely reinforcement. This would also allow the assignment of much larger, more realistic problems, in place of the much shorter ones that are customary today, and yet would no doubt save a great deal of wear and tear on the instructor.

"Proof by Emacs" is one among a range of techniques that IMPS supports to allow the mathematician to do what he would like to do anyway, but in a more convenient and reliable way.

5 Proof by Symmetry

Various essentially different kinds of reasoning are lumped together under the term "proof by symmetry." In some cases, a proof by symmetry may be formalized by constructing a theory interpretation from an axiomatic theory $\mathcal{T}$ into itself; for instance, the right cancellation law in groups follows from the left cancellation law, using the "symmetry" (interpretation) that maps the group operation $\cdot$ to $\lambda x, y \,.\, y \cdot x$. The IMPS mechanisms supporting this form of reasoning are discussed in [4, 3]. In this section, we will instead focus on cases which do not easily fit that pattern, but in which portions of a proof are symmetrical with each other. The formula shown in Figure 6 involving the floor function[3] supplies a very simple example: In proving the right hand side from the left hand side, the two halves of the nested biconditional are symmetrical. In fact, in proving the left hand side from the right hand side, we also create two symmetrical subgoals, namely to prove that $\mathrm{floor}(x) \leq \mathrm{floor}(y)$ and that $\mathrm{floor}(y) \leq \mathrm{floor}(x)$, but these are handled trivially by the IMPS simplifier.

$$\boxed{\forall x, y : \mathbf{R} \quad \mathrm{floor}(x) = \mathrm{floor}(y) \iff (\forall m : \mathbf{Z} \quad m \leq x \iff m \leq y)}$$

Fig. 6. Floor equality criterion

The user starts the proof by breaking apart the logical connectives, after which he confronts, as his current subgoal, the task of showing $m \leq y$ assuming

$$m \leq x \quad \text{and} \quad \mathrm{floor}(x) = \mathrm{floor}(y).$$

To do so, he instantiates the first theorem shown in Figure 7 with the arguments x and m, and instantiates the second theorem with y. Simplification completes this case. Since the second case is obviously true for the same reason, he can request that IMPS print the text of his proof so far, as shown in Figure 8. The user

[3] The symbol "floor" is a constant defined in the IMPS theory of real arithmetic as that function which for every real x returns the unique integer n such that $n \leq x < n+1$.

$$\forall x : \mathbf{R}, n : \mathbf{Z} \quad n \leq x \supset n \leq \text{floor}(x)$$

$$\forall x : \mathbf{R} \quad \text{floor}(x) \leq x$$

Fig. 7. Lemmas used in the proof

```
direct-and-antecedent-inference-strategy
(instantiate-theorem floor-not-much-below-arg ("x" "m"))
(instantiate-theorem floor-below-arg ("y"))
simplify
```

Fig. 8. Transcript of first case

may then edit this text to construct a locally defined script, shown in Figure 9. Finally, the user types and executes the proof command `($do-case "y" "x")`, which carries out the same inferences with the roles of x and y interchanged. The full proof, as it is recorded in an IMPS theory file, is given in Figure 10. This final presentation has the advantage that the symmetry between the two subgoals is made explicit for a later reader by the two `$do-case` forms. Thus, our approach to proof by symmetry eases the user's burden in the course of developing the proof, and makes the structure of the proof easier to read off its final form.

We have used this technique in many examples, where the individual subgoals may be far more demanding. Another frequent source of symmetrical subgoals—apart from biconditionals in the goal—is the instances of the anti-symmetry of $\leq$. Instances of the trichotomy of $<$ also frequently furnish two symmetrical cases (the strict inequalities) as well as the nonsymmetrical, and frequently quite easy, case with the equality.

6 Comparison with Tactics

Tactic-based theorem proving [8] has been a major area of research in automated reasoning since the development of Edinburgh LCF [6]. In fact, the ML programming language was invented for writing LCF tactics. Today tactics are used in several major theorem proving systems, including HOL [7], Isabelle [9], and Nuprl [1].

Tactics are procedures that automate part of the proof process. They come in many flavors. For example, a *refinement tactic* is a procedure which generates a list of subgoals from a given goal in a deduction, and a *transformational tactic* is a procedure that constructs a new deduction from an old deduction [1]. The former notion of a tactic is fairly restrictive, while the latter notion is quite broad. Although the notion of a tactic varies widely, all tactics are constructed

```
(let-script
 do-case 2
 ((instantiate-theorem floor-not-much-below-arg ((% " ~a " $1) "m"))
  (instantiate-theorem floor-below-arg ((% " ~a " $2)))
  simplify))
```

Fig. 9. Script for the symmetrical cases

```
direct-and-antecedent-inference-strategy
(let-script
 do-case 2
 ((instantiate-theorem floor-not-much-below-arg ((% " ~a " $1) "m"))
  (instantiate-theorem floor-below-arg ((% " ~a " $2)))
  simplify))
($do-case "x" "y")
($do-case "y" "x")
(apply-macete-with-minor-premises <=-anti-symmetry)
(apply-macete-with-minor-premises floor-not-much-below-arg)
(command-on-direct-descendents simplify)
```

Fig. 10. Final proof script

or applied in a special way so they are guaranteed to be sound with respect to the proof system being used.

An IMPS proof script is a kind of tactic which serves the same purposes as other kinds of tactics:

- To create new proof commands (rules of inference).
- To represent executable proof sketches.
- To store proofs in a compact, replayable form.

A proof script is more general than a refinement tactic since the behavior of a proof script is dependent on the structure, contents, and current node of the deduction graph to which it is applied. On the other hand, a proof script is more restrictive than a transformational tactic since the script programming language is restricted and since scripts can change the structure of a deduction graph only by adding new nodes. From our point of view, proof scripts have just about the right level of generality: they are powerful enough to do useful things, but controlled enough to be easily manipulated as text. Proof scripts also have several idiosyncrasies that set them apart from other kinds of tactics:

- The systematic use of simplification.
- The application of theorems via macetes.
- The use of the "current node" idea to linearize the execution of scripts.
- The use of "blocks" to make scripts more robust.

A macete is also a kind of tactic, but it has a very limited purpose: to apply and retrieve theorems (or organized collections of theorems). Macetes play an extremely important role in the IMPS proof process. Like proof scripts, macetes are idiosyncratic:

- Macetes apply a theorem or collection of theorems at any location in (the assertion of) a goal, even deeply within it.
- As we mentioned in Section 2, macetes use both ordinary expression matching and translation matching so that theorems can be applied both inside and outside of their home theories.
- Theorem macetes (see Section 2) are automatically created by IMPS when theorems are installed.

Even though the means to program macetes is quite restricted in comparison to tactics and IMPS proof scripts, in practice it is very easy for the IMPS user to build useful macetes.

7 Conclusion

For any proof system, we may distinguish the knowledge explicitly formalized in it from the body of more procedural knowledge which is not formalized, but which a person must grasp to use it effectively. Proving interesting theorems, whether with pencil and paper or with mechanized theorem provers, requires a great deal of each.

In IMPS, the explicitly formalized knowledge required is primarily codified in the theory library, a large and open-ended collection of axiomatic theories and theorems. Theory interpretations serve as links to interconnect the theories in the library. The current theory library contains about 50 named theories and over 1100 replayable proofs. The theories include formalizations of the real number system and objects like sets and sequences; theories of abstract mathematical structures such as groups and metric spaces; and theories to support specific applications of IMPS in computer science. Several of the theorems proved reach about the level of the fundamental theorem of calculus. The most developed area of mathematics in the theory library is abstract mathematical analysis.

The more implicit, procedural knowledge is encoded in several ways. Compound macetes provide one way, as many of them amount to rudimentary procedures for (logically sound) symbolic computation. Proof scripts are intended as another repository of this sort of practical knowledge. User-defined commands introduced by `def-script` encapsulate knowledge of how to perform some conceptually unified, higher-level manipulation on a goal. In "proof by Emacs," the proof of one theorem serves as a template indicating how to prove related theorems. One form of proof by symmetry uses the procedural knowledge accumulated in proving one case to codify what is common between the cases. More generally, we consider the proof scripts in the theory library as a mine of ideas and techniques for getting a wide variety of substantial theorems rigorously proved with IMPS.

In this paper we have tried to convey a portion of this procedural knowledge for the case of IMPS. We think that the developers and users of various proof systems can make valuable contributions to our common goals by formulating and exchanging this kind of information, in addition to the more usual information about explicit logical techniques.

Acknowledgments

We are grateful for the suggestions received from the referees.

References

1. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
2. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: System description. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1992.
3. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.
4. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.
5. W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M93B-138, The MITRE Corporation, Bedford, MA, November 1993.
6. M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh* LCF*: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
7. M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Surahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 73–128. Kluwer, 1987.
8. R. Milner. The use of machines to assist in rigorous proof. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 77–88. Prentice/Hall International, 1985.
9. L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–368. Academic Press, 1990.
10. J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual*. Computer Science Department, Yale University, fifth edition, 1988.

This article was processed using the LaTeX macro package with LLNCS style