Proof Search Debugging Tools in ACL2

Matt Kaufmann[1] and J Strother Moore[2]

[1] Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA,
kaufmann@cs.utexas.edu,
WWW home page: http://cs.utexas.edu/users/kaufmann
[2] Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA,
moore@cs.utexas.edu
WWW home page: http://cs.utexas.edu/users/moore

Abstract. Any mechanized theorem prover with interesting automatic search features should provide the user some feedback on the search process so that failed proof attempts can be debugged. Proof search automation is a central theme of the ACL2 system: once the prover is started the user can do nothing to guide it. The prover's behavior is determined largely by previously proved theorems in its data base and user advice supplied with the goal conjecture. This design allows ACL2 to be configured to prove a class of problems in a given domain "automatically," which supports the industrial "replay" of big proofs about slightly modified designs. But it means that proofs often fail and the user is left wondering why. ACL2 provides a wide variety of search debugging tools to allow the user to answer the questions "what is happening?" and "what went wrong?" We discuss those tools.

1 Background

Suppose the ACL2 user wishes to prove that reversing a list three times is equivalent to reversing it once, a theorem called `triple-rev` below.

```
(defthm triple-rev
  (equal (rev (rev (rev a)))
         (rev a)))
```

The ACL2 user might think through the argument informally as follows. (We follow ACL2 syntax by using a semicolon (';') to start a comment up to the end of the line.)

> Reversing a list twice is an identity. So `(rev (rev a))` rewrites to `a` and we're done.
>
> To formalize and mechanize this, I'll prove the key lemma, `double-rev`, `(rev (rev x))` = `x`, and have ACL2 use that as a rewrite rule.
>
> Then, when I ask the system to prove `triple-rev`, it will automatically apply the rewrite rule to the underlined subterm below and reduce the left-hand side of `triple-rev` to the right-hand side:

$$\begin{array}{lll} & \texttt{(rev }\underline{\texttt{(rev (rev a))}}\texttt{)} & \texttt{;}\ \mathit{lhs}\text{ of }\texttt{triple-rev} \\ = & & \\ & \texttt{(rev }\underline{\texttt{a}}\texttt{)} & \texttt{;}\ \mathit{rewritten\ lhs} \\ = & & \\ & \texttt{(rev a)} & \texttt{;}\ \mathit{rhs}\text{ of }\texttt{triple-rev} \end{array}$$

Q.E.D.

Note that this hypothetical user invents `double-rev` and then thinks about how to make ACL2 use it automatically. An alternative would be for ACL2 to require the user to provide a detailed proof.

The basic philosophy behind the ACL2 system is that it should be as automatic as possible, guided by rules derived from definitions and theorems that have already been certified by the system. This is empowering for several reasons. The user is not responsible for soundness: rules in the data base are vetted by the system using already vetted rules. The user is relieved of much tedious detail: terms and patterns in the goal conjecture trigger the application of vetted rules and the system does this quickly and accurately. In essence, the ACL2 user strives to develop a general and powerful collection of rules for manipulating the concepts in a given class of problems. This gives the system the flexibility often to tolerate minor changes in previously checked work. For example, if ACL2 can prove some property of function `f` and then the definition of `f` is undone and modified in some slight way so that the property still holds, ACL2 is often able to discover the proof "again." (We put the word in quotes because the two proofs are different because the two `f`s are different.) Because models of hardware and software are often modified to reflect changing designs, this ability to "replay" proofs automatically is important.

Returning to the `triple-rev` example, a problem with the ACL2 approach is revealed when we realize that `double-rev` is not a theorem! One can read the prover's output to discover that `(rev (rev x))` is not `x` if `x` is a list structure that terminates with a non-`nil` value. The formula imagined by our hypothetical user is not valid; for example, if `x` is `7`, `(rev (rev x))` is `nil`, not `7`. An accurate formalization of `double-rev` is

```
(defthm double-rev
  (implies (true-listp x)
           (equal (rev (rev x))
                  x))).
```

While this particular mistake could be avoided in a strongly typed system, it illustrates a very common class of mistakes: missing hypotheses. Having discovered this correction to `double-rev` we ought to rethink its intended use in the bigger proof we are constructing.

But people are sloppy. Working top down to prove some main theorem (e.g., `triple-rev`) we invent or remember "lemmas" (e.g., the faulty version of `double-rev`) without getting them exactly right. We imagine the operation of the rules derived from these faulty lemmas and design proof sketches based on these powerful rules. Then we set out to formalize our thinking. In interaction with a theorem prover, we debug statements of our lemmas and catch all the little mistakes. But in the heat of the chase, we forget to carry what we learn back to our main proof sketch. When we finally try to check the main proof, the actual rules carry us astray. The proof fails.

Furthermore, since each newly conjectured lemma requires a proof, the general state of affairs is that the user is in a deep recursive descent through a tree of conjectures rooted in an imagined proof of the main one. Many theorems have been accurately stated and proved while many others remain to be. At any one time, the user is working on the next lemma or theorem and is expecting its formal proof to follow the previously imagined sketch.

In a case like `triple-rev` – where only one new lemma is involved – it might be easy to remember to modify the main theorem or its proof. But in interesting proofs about hardware and software, hundreds of crucial properties and relations might be involved and it is simply impossible to keep them all in mind.

It is illusory to think that the problem is avoided by working bottom up or with a goal manager. Formalization and symbolic manipulation are inevitably carried out late in whatever intellectual process occurs as we think about a given formal challenge. The early stage of proof discovery is necessarily informal since often we must invent concepts that are nowhere formalized or even expressed. There will often be a gap between this early informal stage, where mistakes often occur, and its subsequent mechanized formalization.

It is also illusory to think the problem is solved by printing out everything the system is doing. To give the reader a feel for the size of the problem, consider Liu's M6 model of the Java Virtual Machine [7]. The model is about 160 pages of ACL2 and includes all JVM data types except floats. It models multi-threading, dynamic class loading, class initialization, and synchronization via monitors. We have a data base of about 5,000 rules designed primarily to allow us to construct a proof about an invariant on the class table under dynamic class loading. But the rules are also sufficient to allow us to prove the correctness of simple JVM bytecode programs.

The system can "automatically" prove the correctness of the bytecode produced by the Sun Microsystems `javac` compiler for

```
public static int fact(int n){
    if (n>0)
      {return n*fact(n-1);}
    else return 1;
  }.
```

The statement of correctness is that when the program runs on a non-negative 32-bit Java `int`, n, it terminates and returns the (possibly negative) integer represented in twos-complement notation by the low-order 32-bits of $n!$. Here, $n!$ is the true mathematical value of factorial. To prove this, the system does an induction on n and attempts to apply the 5,000 named rules in its data base. It tries a total of about 63,215 applications to terms in the problem. Only about half are useful. Many of these applications are on branches of the proof search that are ultimately pruned (e.g., simplification done while backchaining on unsuccessful paths). The actual proof involved 180 different rules. The entire search takes 2 seconds on a 2 GHz Intel Core Duo running ACL2 built on SBCL Common Lisp. Furthermore, the system can tolerate minor correctness-preserving changes to the bytecode.

ACL2 prints a description of the evolving proof attempt, but the printed steps are very large. For example, one formula, which takes 67 lines to print, is reported in real time to have simplified to the conjunction of 11 formulas. The 11 formulas take a total of 6,886 lines to print – over 100 pages. This single simplification step is justified by the combined application of 162 different rules. When ACL2 fails or ceases printing, the user can get a rough idea of how things went by navigating through this proof output using text processing tools like Emacs.

The Method described in [4] for debugging failed proofs involves inspecting certain critical subgoals, called *checkpoints*, and looking for subterms within them that should have been simplified by known rules or that suggest the need for new rules, keeping in mind the informal proof sketch for the goal.

It is not surprising that with so many rule applications and such large formulas, it can be difficult for the user to debug failed proofs. Proof search debugging tools are necessary.

2 Organization of this Paper

In this paper we describe many of ACL2's proof search debugging tools. First, we describe the basic structure of the ACL2 prover in Section 3. As the system operates, it prints a running description of a "proof." This text will actually describe a proof if the system is successful in finding one. Otherwise, it will describe a failed proof attempt. We describe our prover output in Section 4. Then we describe a useful navigational aid for exploring failed proofs (Section 5), our use of verbose warnings and error messages (Section 6), a facility for monitoring the attempted applications of a given rewrite rule (Section 7), a dynamic display of all rule applications (Section 8), a profiling tool for rule applications (Section 9), and a goal manager used to explore proofs of particular subgoals (Section 10). We briefly mention other facilities in Section 11 and then conclude.

3 Architecture of the System

The ACL2 theorem prover can be thought of as a series of *proof techniques* arrayed around a set of subgoals called the *pool*. Every formula in the pool must be proved. The user starts the proof process by putting the goal formula into the pool. The proof process ends successfully when the pool is emptied.

The techniques are (1) application of user supplied hints, (2) preprocessing, (3) simplification, (4) destructor elimination, (5) heuristic use of equality,

(6) generalization, (7) elimination of irrelevance, and (8) induction. Each proof technique takes a formula ϕ as input and signals a *hit*, a *miss* or (to abort the proof) a *fail*. When a technique hits, it produces a set of new formulas $\phi_1 \ldots \phi_k$ such that such that if each ϕ_i is provable then so is ϕ. Of course, if $k = 0$, the technique proved ϕ, but generally the techniques just reduce ϕ to some heuristically chosen subgoals.

Techniques (1–3) are thought of as routine and are all, loosely speaking, just equivalence preserving transformations. Techniques (4–7) are somewhat unusual. In fact, (5–7) may generalize the problem, producing unprovable but sufficient subgoals. We apply induction (8) only to the simplest, most general formulas we can find.

Thus, processing proceeds as follows. While the pool is non-empty, the system removes a formula ϕ from the pool and feeds it to the first technique. If the technique hits, the generated subgoals are deposited into the pool. If the technique misses, ϕ is passed to the next technique. If any technique fails or all of them miss, the proof process stops with failure. This process is called *the waterfall*.

Technique (3), the simplifier, is the core of the ACL2 prover and the rewriter is the core of the simplifier. We expand on both.

The simplifier operates on formulas and the rewriter operates on terms and takes as an additional argument a set of assumptions governing the occurrence of the term being rewritten. All these concepts are natural descendents of those concepts described in Boyer and Moore's *A Computational Logic* [1] and are probably familiar to most readers of this volume. As described in [3], the original algorithms have been extensively (if naturally) elaborated to include various decision procedures and rule interpreters. We review the basics quickly just to establish terminology.

By "formula" here we mean a clause. In ACL2, each literal of a clause is just a term, where the value `nil` denotes false and any other value denotes true. For our purposes, the "assumptions" argument of the rewriter is just a set of (possibly negated) terms indicating what is known true and false; this is actually represented as a very useful data structure called a *type alist* that maps terms to types and with respect to which the types of other terms can be quickly determined. (This typing algorithm is itself sensitive to rules derived from theorems in the data base and is sort of a fast, cheap, abstract rewriter, called `type-set`.)

The simplifier is responsible for transforming a goal formula ϕ into an equivalent set of subgoals $\{\phi_1, \ldots, \phi_k\}$. It does this by rewriting each literal of ϕ under the assumptions that the other literals are all false. Then it splits out the if-then-else expressions contained in the rewritten literal to generate clauses. For example, if the literal `(P` α`)` rewrites to `(P (IF` β_1 β_2 β_3`))` under the assumptions ψ, then the simplifier would generate the clauses corresponding to $(\psi \wedge \beta_1) \rightarrow$ `(P` β_2`)` and $(\psi \wedge \neg\beta_1) \rightarrow$ `(P` β_3`)`.

The rewriter operates on a target term and under some assumptions.[3] The target term is actually specified by a pair of arguments: a term (called the *matrix*), α, and a substitution, σ. The target term being rewritten is α/σ, the instantiation of α by σ. The rewriter takes many additional arguments but one important additional argument is an equivalence relation to be maintained by the rewriter. If the matrix α with instantiation σ (in the context of assumptions ψ and equivalence relation $\cong$) rewrites to β, then it is a theorem that $\psi \rightarrow (\alpha/\sigma \cong \beta)$. The rewriter may use a multitude of rules to make this step and those rules are reported in other outputs of the rewriter.

When the simplifier calls the rewriter, the equivalence relation is propositional equivalence $\leftrightarrow$, but as the rewriter descends through function symbols it uses *congruence* rules in its data base to switch to other equivalences. For example, a congruence rule might tell it that while rewriting `(member` δ α`)` to maintain propositional equivalence it can rewrite α to maintain the `permutation` relation, which is also an equivalence relation. Under the `permutation` equivalence, a correct `sort` function is the identity, so `(permutation (sort x) x)` may be used as a rewrite rule to replace instances of `(sort x)` by `x`, provided we are maintaining `permutation`. Then, for example, the rewriter will transform `(member a (sort x))` to `(member a x)` by rewriting `(sort x)` to `x` under the equivalence relation of `permutation`.

The rewriter recursively descends through the matrix. When it encounters a variable it just looks it up in the substitution σ. We maintain the invariant that the substitution always binds variables to already rewritten terms. When the rewriter encounters an if-then-else, it rewrites the test and then, depending on the result, rewrites the true and/or false branches under appropriate modifications of the assumptions. When the rewriter encounters a non-`IF` function application in the matrix, `(`$f\ \alpha_1 \ldots \alpha_n$`)`, it recursively rewrites the α_i under the same assumptions and the equivalence relations specified by known congruence rules, obtaining α'_i. Then it looks for conditional rewrite rules, preserving the appropriate equivalence relations (namely, $\cong$ and its refinements as revealed by *refinement* rules, including equality), about `(`$f\ \alpha'_1 \ldots \alpha'_n$`)`.

The general form of a rewrite rule is suggested by:

$$\eta_1 \wedge \ldots \wedge \eta_m \rightarrow\ (f\ \gamma_1 \ldots \gamma_n)\ \cong\ \beta.$$

To apply such a rule to the target term `(`$f\ \alpha'_1 \ldots \alpha'_n$`)` the rewriter attempts to match `(`$f\ \gamma_1 \ldots \gamma_n$`)` with the target. If the match is successful, a new substitution ρ binding some of the variables of the rule to already rewritten terms is produced. The rewriter then rewrites the η_i with substitution ρ, maintaining propositional equivalence. Care is taken to extend ρ to ρ' to deal with so-called *free variables* in the η_i not bound in ρ. If all the η_i are rewritten to true (i.e., terms whose types are known to be non-`nil`), the system replaces the target term by the equivalent (modulo $\cong$) result of rewriting β with the substitution ρ'. If the match fails or any η_i fails to rewrite to true, the attempt to apply the rule is abandoned.

[3] The rewriter is implemented as ACL2 source function `rewrite`. The ACL2 source code is publicly available from [5]. See also [6] for more about ACL2's congruence-based rewriter.

This description omits entirely our use of decision procedures, computation on constants, special handling of certain function symbols, other rule interpreters like our use of verified user-supplied simplifiers called *metafunctions*, and other proof techniques mentioned in [4].

There are two take-home messages for the reader. First, when a rule in the data base is considered and applied, it may spawn a lot of work to rewrite the η_i and β. Furthermore, this work may be unnecessary. The work done to rewrite η_1 to true is irrelevant if η_2 fails to be rewritten to true. Second, in the context of huge terms and thousands of rules, it can be extremely difficult for the user to understand what is happening. Some kind of proof search debugging tools are required.

We describe a variety of such tools below. But this paper is not intended to serve as a user's manual. Our online user's manual [5] provides an extensive discussion of how to use the tools. Because we are not writing for users, we skip over a very common "wart" of most of the tools discussed below: they have to be activated before they can be used. Debugging tools slow the system down and most of the time users do not want them around! Thus, most are deactivated by default. For details on how to activate each tool, see the user's manual.

4 Proof Output

ACL2 provides several layers of output describing proofs. We describe the most verbose first. Any technique that hits causes the system to assign a unique subgoal name to its input formula ϕ and then print ϕ, along with a text message describing the transformation wrought on ϕ, the rules used, and the subgoals $\phi_1, \ldots, \phi_k$ produced. Each subgoal is printed when it is pulled out of the pool. The result is a linear textual description of a hierarchical proof. Here is a schematic example of verbose output for the formula ϕ.

```
This simplifies, using rules_0 to 3 subgoals:


Subgoal 3
φ_3
This simplifies to T using rules_3.


Subgoal 2
φ_2
This simplifies using rules_2 to 1 subgoal:
Subgoal 2'
φ_2,1
Destructor elimination with rules_2,1 produces 1 subgoal:
Subgoal 2''
φ_2,2
This simplifies using rules_2,2 to 1 subgoal:
Subgoal 2'''
```

```
$\phi_{2,3}$
This simplifies to T using $rules_{2,3}$.


Subgoal 1
$\phi_1$
This simplifies to T using $rules_1$.
Q.E.D.
```

The first simplification transformed ϕ, using the set of rules $rules_0$, into the subgoals ϕ_3, ϕ_2, and ϕ_1. The first and last of these were proved by further simplification; the middle one was transformed further by simplification, then by destructor elimination and then finally finished by two successive simplifications.

The most desirable ACL2 proofs are either pure simplification (in the loose sense of techniques (1–3)), or inductions followed by such simplifications of each case to T. Such proofs depend on fewer heuristics and are easier to "maintain." Therefore, when a failed proof attempt takes a step out of this class, i.e., the first time any technique other than (1-3) hits along some branch of the proof tree, the user should take note. These points in the proof are called *key checkpoints*. In the example above, `Subgoal 2'` is a key checkpoint. The formula $\phi_{2,1}$ at that position is *stable under simplification* because simplification was attempted before destructor elimination and must have missed.

Upon seeing a key checkpoint the user ought to inspect the associated formula, $\phi_{2,1}$ above, and look further simplifications. A typical situation is that the user sees some combination of terms that can be rewritten into a simpler form, or sees that some subset of the hypotheses are contradictory. Since $\phi_{2,1}$ is stable under simplification, new lemmas expressing these observations can be formulated from subterms that appear in $\phi_{2,1}$ and when the main proof is tried again, the simplification step that previously missed will encounter those subterms and hit. Alternatively, the user might recognize that some existing rule should have applied but did not; see Sections 7 and 10.

To help the user focus on the key checkpoints, ACL2 (as of Version 3.3) prints the first few key checkpoints at the end of any failed proof. This leaves those key checkpoints on the user's screen when a proof fails.

ACL2 provides a mechanism for inhibiting all output or just output of a specified type, such as proof output. But even if proof output is enabled, it is possible to "gag" ACL2 so it is less verbose. Several levels of verbosity are possible. The most severe, achieved by executing `(set-gag-mode t)`, causes ACL2 to print only key checkpoints that cannot be proved without induction and inductions it does. Thus, in this mode, ACL2 prints nothing to describe the proof above.

A slightly more verbose setting, `(set-gag-mode :goals)` causes ACL2 to print additionally the names of the subgoals (but not the formulas, descriptions, or rules used) as they are generated. In this mode, the schematic proof above would become:

```
Subgoal 3
Subgoal 2
Subgoal 2'
Subgoal 2''
Subgoal 2'''
Subgoal 1
Q.E.D.
```

After a proof, whether successful or not, the user can use the *print saved output* command, `:pso`, to print out the verbose output. Experienced ACL2 users rarely look at successful proofs, and they often avoid looking at more than the key checkpoints of failed proofs rather than perusing all of the verbose output. However, verbose output can be useful after a failed proof when a key checkpoint has a surprising form, to understand something about how it was generated. For example, if a forgotten rewrite rule moved the proof in a surprising direction, then mention of that rule in the verbose output could be useful information that leads the user to disable (turn off) that rule in subsequent proof attempts.

5 Proof Trees

ACL2 provides dynamically printed *proof trees*. The display is a series of textual snapshots, each of which captures the state of the system after a hit and shows only the active branch of the proof. This display is printed into a window different from that in which the proof output is spooled. The effect is an animation of the evolution of the proof search.

Consider the moment when the system has simplified `Subgoal 2''` above. By that time, the `Goal` ϕ has been split into three subgoals, `Subgoal 3` has been proved and `Subgoal 1` has not yet been attacked. The proof tree display would look like this:

```
   3 Goal simp
   1 |  Subgoal 2 simp
c  1 |  |  Subgoal 2' ELIM
   1 |  |  |  Subgoal 2'' simp
     |  <1 more subgoal>
```

The "c" indicates that `Subgoal 2'` is a checkpoint. When `Subgoal 2'''` is simplified to `T`, completing the proof of `Subgoal 2`, the display rolls back and `Subgoal 1` appears where `Subgoal 2` is shown above, immediately under `Goal`. The lower fringe of the proof tree tends to flicker since that is where the action is.

The proof tree display is connected to the verbose proof output; the former may be used to navigate through the latter. For example, if in Emacs one positions the cursor on a given line of the proof tree and hits a certain key, one is transferred to the corresponding position in the proof output. Other Emacs proof tree commands transfer to the next checkpoint, suspend the proof tree display, resume the display, etc. The proof tree display is also incorporated nicely into ACL2s [2], the *ACL2 Sedan* used in undergraduate courses.

6 Warnings and Error Messages

The user essentially programs ACL2 by defining functions and proving theorems that are translated into rules that guide ACL2's behavior. Sometimes it is possible to detect that the user may misunderstand the operational behavior induced by a definition or theorem. This happens even for expert users because of the complexity of ACL2's heuristics and the complexity of the models such users build. So when we detect possible misunderstandings, we warn the user. We give two examples.

ACL2 has many heuristics for controlling the expansion of defined functions. Most of those heuristics were designed for controlling recursively defined functions, i.e., those whose expansion introduces another call of the same function symbol. But often users make so-called *non-recursive* definitions[4] like

```
(defun sq (n) (* n n))
```

Here we are defining `(sq` n`)` to be n^2.

In almost all cases, ACL2's heuristics will expand calls of non-recursive functions. Furthermore, ACL2's simplifier proceeds to rewrite from the inside out. Thus, in attacking the formula `(p (g (sq c) c))`, ACL2's simplifier will transform it to `(p (g (* c c) c))` *before* considering rules to rewrite the `g` term. So a rewrite rule with the left-hand side `(g (* x x) x)` would match and might be applied, but a rule with left-hand side `(g (sq x) x)` would not.

Thus, when the user formulates a theorem that will generate a rewrite rule in which a non-recursive function, like `sq`, appears on the left-hand side, an elaborate warning message is printed and the user is referred to the hypertext documentation that explains the details.

The second example illustrates even more complicated heuristics in ACL2. While most rewrite systems use only equalities as rewrite rules, ACL2 can rewrite using any relation that has been proved to be an equivalence relation. It switches from one equivalence relation to another using congruence rules.

Because of the details of how ACL2 rewrites terms and binds variables during pattern matching, it is possible for a variable to be bound to a term that was rewritten preserving one sense of equivalence (e.g., identity or `equal`) and used in a context admitting a more generous equivalence (e.g., `permutation`). When this happens, ACL2 misses the opportunity to rewrite the term under the more generous sense of equivalence. This unexpected behavior can be triggered by a user-supplied rule that uses a variable in two different equivalential contexts. ACL2 includes code that detects this when lemmas are posed to the system and prints an elaborate warning message that alerts the user to the "unexpected" behavior and explains how to get the "expected" behavior (at some cost in performance).

ACL2 can produce about 70 such warnings even though the detected conditions do not cause unsound behavior.

[4] We use the term in a strictly syntactic sense. All concretely defined functions in ACL2 are computable.

ACL2 also can produce about 1000 distinct error messages. Unlike warnings, errors actually abort the processing and force the user to deal with the offending definition or lemma. Furthermore, we detect such errors as early as possible rather than after significant expenditure of proof effort. For example, suppose the user poses a lemma to the prover and indicates that, once proved, it is to be used as a conditional rewrite rule. But imagine that the lemma cannot actually be so used. We detect that before we try to prove it, rather than after the proof.

7 Monitors

Recall our earlier example

```
(defthm triple-rev
  (equal (rev (rev (rev a)))
         (rev a)))
```

and our proof sketch for it involving rewriting `(rev (rev a))` to `a` using the flawed version of `double-rev` omitting the hypothesis.

```
(defthm double-rev
  (implies (true-listp x)
           (equal (rev (rev x))
                  x)))
```

Suppose the user has proved the corrected version of `double-rev` and then asks the system to prove `triple-rev`. What happens?

The proof fails, after trying to prove `triple-rev` by induction on `a`. In particular, the proof did not follow our expectation that the `(rev (rev a))` would rewrite to `a`.

What can the user do? A tool provides the ability to set breakpoints (or "monitors") on rules. In particular, the user may install a monitor on `double-rev` with the `:monitor` command. Replaying the attempt to prove `triple-rev` would cause the system to enter an interactive break when `double-rev` is considered. At that time, the user could inspect the context of the attempted application using a variety of commands like:

`:target` - the term being rewritten.

`:path` - a description of the rewriter's call stack. At its most trivial, the path reveals where the target appears in the top-level goal, i.e., it may be the first argument of the first argument of the concluding equality, as `(rev (rev a))` is in the goal `(equal (rev (rev (rev a))) (rev a))`. But more generally, the path may involve other rules. For example, the target may be the first argument of the third hypothesis of a rule being applied to the conclusion of the top-level goal.

`:lhs` - the left-hand side of the rewrite rule.

`:hyp` n - hypothesis n of the monitored rule.

`:unify-subst` - the substitution that instantiates the left-hand side of the monitored rewrite rule to make it identical to the target.

`:type-alist` - the assumptions governing this occurrence of the target.

In addition, other commands direct the rewriter to proceed in various ways and perhaps the most common is

`:eval` - attempt to apply the rule and re-enter the interactive break afterwards.

Below is a session log showing the first attempt to apply `double-rev` and the use of a few of the break commands described above. User input is underlined.

```
(1 Breaking (:REWRITE DOUBLE-REV) on (REV (REV A)):
1 ACL2 >:path
1. Simplifying the clause
     ((EQUAL (REV (REV #)) (REV A)))
2. Rewriting (to simplify) the atom of the first literal,
     (EQUAL (REV (REV (REV A))) (REV A)),
3. Rewriting (to simplify) the first argument,
     (REV (REV (REV A))),
4. Rewriting (to simplify) the first argument,
     (REV (REV A)),
5. Attempting to apply (:REWRITE DOUBLE-REV) to
     (REV (REV A))
1 ACL2 >:lhs
(REV (REV X))
1 ACL2 >:unify-subst
     X : A
1 ACL2 >:eval
1x (:REWRITE DOUBLE-REV) failed because :HYP 1 rewrote to
   (TRUE-LISTP A).
1 ACL2 >:hyp 1
(TRUE-LISTP X)
1 ACL2 >:ok
1)
```

From the explanation of the rule's failure it is clear we cannot expect the rule to apply: we must show that `a` is a `nil`-terminated list.[5] But `a` is a universally quantified variable about which we know nothing. We must change the goal theorem if we expect `double-rev` to apply here. We could, for example, weaken `triple-rev` by adding the hypothesis `(true-listp a)`.

The ability to monitor rewrite rules and to interact with the rewriter during every attempted application is a critically important facility. We should note that sometimes a rule may be tried so often that the user simply cannot cope with the plethora of interactive breaks. When one installs a monitor on a rule it is possible to provide a test to evaluate in the context of the application to

[5] We do not distinguish in this paper between upper and lower case ASCII text in ACL2 formulas.

decide whether an interactive break should occur, and if not, then whether trace information should nevertheless be printed. Because ACL2 is simultaneously a mathematical logic and a functional programming language, it is convenient to express these tests in ACL2. The fact that the ACL2 prover is implemented in that same functional language makes it easy to allow the test to access much of the theorem prover's state.

Upon figuring out why the `double-rev` fails to apply here, the user could abort the proof. But another option is to let the rewriter proceed to the failure we know will happen. That is the option the user takes above, by typing `:ok`. Note that the entire interactive session is surrounded by `(1 ... 1)` to allow easy mechanized inspection of the proof script with text editing tools like Emacs.

In the proof attempt in question, another break occurs:

```
(1 Breaking (:REWRITE DOUBLE-REV) on (REV (REV (REV A))):
1 ACL2 >:eval
1x (:REWRITE DOUBLE-REV) failed because :HYP 1 rewrote to
(TRUE-LISTP (REV A)).
1 ACL2 >:ok
1)
```

This break on `double-rev` alerts us to the fact that there is a second opportunity to apply the rule and that attempt fails because we cannot prove that `(rev a)` is a `nil`-terminated list.

However, it turns out that `(rev a)` *is always* a `nil`-terminated list, even if `a` is not. The system cannot prove that fact with rewrite rules alone, as the rewriter must in order to relieve a hypothesis in a conditional rewrite rule. But if the user realizes `(true-listp (rev a))` is in fact true, that fact could be formulated as a lemma and made available as a rewrite rule (implicitly).

```
(defthm true-listp-rev
  (true-listp (rev a)))
```

For our standard definition of `rev`, the above lemma is proved fully automatically with two inductions (one to unwind `rev` and the other to unwind its subroutine `append`).

Below is a subsequent attempt to prove `triple-rev`, now with both `double-rev` and `true-listp-rev` in the data base. In the proof below we have removed the monitor on `double-rev`.

```
(defthm triple-rev
  (equal (rev (rev (rev a)))
         (rev a)))
But simplification reduces this to T, using primitive type reasoning
and the :rewrite rules DOUBLE-REV and TRUE-LISTP-REV.
Q.E.D.
Summary
Form:  ( DEFTHM TRIPLE-REV ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE DOUBLE-REV)
        (:REWRITE TRUE-LISTP-REV))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 TRIPLE-REV
```

Note that we have discovered a way to prove the original version of `triple-rev` without weakening it with the `(true-listp a)` hypothesis, thanks in part to the use of a breakpoint on the corrected `double-rev`.

8 Dynamic Display of the Rewrite Stack

ACL2 provides a dynamic display of the rewriter's activation stack. Each line of the display is similar to the lines presented by the `:path` command. The stack grows downward, so that the top line is the top-level entry into the rewriter and, roughly speaking, each subsequent line corresponds to an invocation of the rewriter. Lines appear and disappear as the rewriter is entered and exited. This stack is displayed in a window separate from the window in which the proof is being spooled. We call this the DMR (dynamically monitored rewrites) window.

If one took a snapshot of the DMR window and read every line one would understand what the rewriter is doing at that moment. But one would need to take a snapshot! The lower lines of the window are flickering so fast they cannot be read. That flickering is one of the reasons the DMR display is so valuable. Often by reading the stable lines just above the flicker one can see a subterm that takes a long time to simplify, or one can see a rule (e.g., a backchaining step) that has spawned a lot of work. Furthermore, the user who has an expectation as to how a successful proof will proceed will often realize these subterms and rules are simply irrelevant to the successful proof and can modify the system's strategy so as to avoid them.

Here is a brief example of a snapshot of the DMR window. The lines without a leading '`*`' are those that have remained unchanged since the last time the screen was refreshed.

```
 0. (DEFTHM . WP-ZCOEF-G-MULTIPLIES)
 1. SIMPLIFY-CLAUSE
   2. Rewriting (to simplify) the atom of literal 18; argument(s) 1
   4. Rewriting (to simplify) the expansion; argument(s) 3|2
   7. Applying (:DEFINITION WP-ZCOEF-G)
*  8. Rewriting (to simplify) the rewritten body; argument(s) 2|1|2|2
*  13. Applying (:REWRITE MOD-ZERO . 2)
*    14. Rewriting (to establish) the atom of hypothesis 4
*    15. Applying (:META META-INTEGERP-CORRECT)
```

9 Profiling Rule Applications

We also provide a profiling tool that gives a static summary of the recent rule behavior of the rewriter. This tool often finds application when the system is performing sluggishly. Every time a rule, r, is tried, a *frame* containing r is pushed on a stack and that frame persists on the stack until the attempt to apply r has either succeeded or failed, when it is popped off the stack. Rules tried during the attempt on r are pushed "on top" of that frame. Thus, a measure of the work "caused" by r is the number of frames pushed during its attempted application. When frames are popped off the stack, the amount of work they caused is recorded in the frame below, along with a note as to whether the attempt succeeded or failed.

We allow the user to print and/or reset the accumulated data (see the topic `accumulated-persistence` in our documentation). Using this tool it is possible to obtain statistics on rule use during a given session, series of proofs, failed proof attempt, etc.

Here are the first few lines of the output of the profiling tool after the successful correctness proof for the JVM factorial method, `fact`. Over a thousand lines are produced, describing the attempted application of 235 rewrite rules.

```
Accumulated Persistence
(63215 :tries useful, 34040 :tries not useful)
   :frames   :tries    :ratio  rune
   --------------------------------
    215390       28 ( 7692.50) (:REWRITE SIMPLE-RUN-OPENER-J)
    215390       28    [useful]
         0        0    [useless]
   --------------------------------
     75483      783 (   96.40) (:REWRITE
                                MEM-BASE-TYPES-IMPLIES-NOT-EQUAL)
         0        0    [useful]
     75483      783    [useless]
   --------------------------------
     70502        4 (17625.50) (:REWRITE SIMPLE-RUN-C+)
     70502        4    [useful]
         0        0    [useless]
   --------------------------------
     59158       34 ( 1739.94) (:REWRITE DO-INST-OPENER)
     57618       18    [useful]
      1540       16    [useless]
   --------------------------------
     43175     1800 (   23.98) (:DEFINITION MEM)
         0        0    [useful]
     43175     1800    [useless]
   --------------------------------
     ...
```

Note that from the above profile we can see that the rule `MEM-BASE-TYPES--IMPLIES-NOT-EQUAL` causes an enormous amount of useless work, spawning a total of 75,483 frames to be built, all to no avail. By simply disabling this rule we can avoid that useless work and speed up the proof. In fact, if we disable that one rule, the proof time is reduced from 1.93 seconds to 0.89 seconds. Reductions of such magnitude are important in industrial settings where it may take an hour or more to replay a proof.

10 Interactive Goal Management

In Section 7 we used an example to illustrate a monitoring capability for proof debugging with ACL2. In this section we use the same example to illustrate how ACL2's goal manager, typically called its *proof-checker*, can instead be used for such proof debugging. This utility will be familiar to users of interactive provers, in particular for higher-order logics.

We begin by submitting our goal to the proof-checker as follows.

```
(verify
 (equal (rev (rev (rev a)))
        (rev a)))
```

Recall that the following rewrite rule is at our disposal:

```
(defthm double-rev
  (implies (true-listp x)
           (equal (rev (rev x))
                  x))).
```

As in our previous discussion of monitoring, we might first think of applying this rule to the subterm `(rev (rev a))`. The following log shows our attempt to dive to that subterm (Emacs support is provided for the user who wants to avoid typing numbers) and consider applying that rewrite rule. We underline user input.

```
->: p ;  Show current subterm — initially, the top-level goal.
(EQUAL (REV (REV (REV A))) (REV A))
->: (dv 1 1) ;  Dive to the first argument of the first argument.
->: p
(REV (REV A))
->: sr ;  Show applicable rewrite rules.
1. DOUBLE-REV
  New term: A
  Hypotheses: ((TRUE-LISTP A))
  Equiv: EQUAL
2. REV
  New term: (AND (CONSP (REV A))
                 (APPEND (REV (CDR (REV A)))
                         (LIST (CAR (REV A)))))
  Hypotheses: <none>
  Equiv: EQUAL
->:
```

It appears that we want to apply the rule `DOUBLE-REV`. (The second rule is the defintion of `REV`, which seems unhelpful in this context.) However, we see that application of `DOUBLE-REV` will generate an instantiated hypothesis of `(TRUE-LISTP A)`, and we realize that we have no way to prove that. So we move up one level and try again.

```
->: up ;  Move up the goal term one level.
->: p
(REV (REV (REV A)))
->: sr
1. DOUBLE-REV
  New term: (REV A)
  Hypotheses: ((TRUE-LISTP (REV A)))
  Equiv: EQUAL
2. REV
  New term: (AND (CONSP (REV (REV A)))
                 (APPEND (REV (CDR (REV (REV A))))
                         (LIST (CAR (REV (REV A))))))
  Hypotheses: <none>
  Equiv: EQUAL
->:
```

AHA! We see now that if only we can prove `(TRUE-LISTP (REV A))`, we will be done. Let us put that off for the moment and apply the rule.

```
->: (r 1) ;  Rewrite with the first rule that was shown.
Rewriting with DOUBLE-REV.
Creating one new goal:  (MAIN . 1).
->: p ;  Here we see that the rule was indeed applied.
(REV A)
->: top ;  Move back up to the top level of the full term.
->: p
(EQUAL (REV A) (REV A))
->: s ;  This is trivial to prove using a simplify command!
The proof of the current goal, MAIN, has been completed.  However,
the following subgoals remain to be proved:
  (MAIN . 1).
Now proving (MAIN . 1).
->: p ;  Return to consider the goal we created for the true-listp hypothesis.
(TRUE-LISTP (REV A))
->: exit ;  OK; now prove a lemma that rev returns a true list.
Exiting....
 NIL
ACL2 !>
```

We have discovered the lemma `true-listp-rev` shown in Section 7. We may now prove that lemma and then `triple-rev`, as before.

Of course, here we have given a very simple application of the proof-checker. A more realistic example might require backchaining through several rule applications until the problematic hypothesis is discovered. One can apply a rule and then call the prover (or just the simplifier) on each instantiated hypothesis, to discover one that is problematic, and then iterate this process on that hypothesis until finally backchaining to a goal that fails to simplify. At this point the user often gains the insight needed to fix the proof.

11 A Miscellany of Other Features

For users wishing to see the system's progress through the waterfall and some of the major subroutines of the prover, it is possible to have a trace printed in real time. Here is a snippet of the trace output during the factorial proof.

```
CP2> PREPROCESS-CLAUSE
  CP3> FORWARD-CHAIN1
  CP3< FORWARD-CHAIN1 [0.00 seconds]
CP2< PREPROCESS-CLAUSE [0.001 seconds]
CP2> SIMPLIFY-CLAUSE
  CP3> FORWARD-CHAIN1
  CP3< FORWARD-CHAIN1 [0.00 seconds]
  CP3> SETUP-SIMPLIFY-CLAUSE-POT-LST
  CP3< SETUP-SIMPLIFY-CLAUSE-POT-LST [0.00 seconds]
  CP3> PROCESS-EQUATIONAL-POLYS
  CP3< PROCESS-EQUATIONAL-POLYS [0.00 seconds]
  CP3> REWRITE-ATM
  CP3< REWRITE-ATM [0.00 seconds]
  CP3> CLAUSIFY
    CP4> STRIP-BRANCHES
    CP4< STRIP-BRANCHES [0.00 seconds]
    CP4> SUBSUMPTION-REPLACEMENT-LOOP
    CP4< SUBSUMPTION-REPLACEMENT-LOOP [0.00 seconds]
  CP3< CLAUSIFY [0.00 seconds]
```

When the system aborts or is interrupted, it prints a summary of the active branch of such a trace to see where the system was. See the documentation for `pstack`.

Sometimes the data base of rules will put the rewriter into an infinite loop.[6] Unless one can guess the likely rules involved in the loop, the monitoring facilities will not help. Instead, we insure that the record of the rewriter's activity can be preserved after a stack overflow. We provide a facility similar to the `:path`

[6] Actually, ACL2 provides a way for the user to limit the depth of recursion in the rewriter. When the limit is reached, the proof aborts. The default depth is 1000. Often users will abort earlier by causing an interrupt.

command (which, recall, prints the rewriter's path from the top-level goal to the current subterm, including any rules involved) that will describe exactly how the rewriter got to a stack overflow. Often, reading this path description will make it obvious which rules are causing the loop.

We touch here on a few other ways to debug ACL2 proof failures. Users can write their own *computed hints* to direct the theorem prover, and these can write debugging information to the screen; see for example Jared Davis's directory `books/finite-set-theory/osets/`, distributed with ACL2. The distribution also includes tools for symbolic simulation in `books/misc/expander.lisp`.

But more routine than symbolic simulation is a ubiquitous debugging tool — direct evaluation — and the ability to trace it that comes with perhaps every Common Lisp implementation. For example, recall the necessary `true-listp` hypothesis for the theorem `double-rev`. The following log suggests that one could discover the need for that hypothesis by evaluation.

```
ACL2 !>(rev (rev (cons 1 (cons 2 7))))
(1 2)
ACL2 !>(trace$ rev) ;  Try to get more information ....
NIL
ACL2 !>(rev (rev (cons 1 (cons 2 7))))
  1> (ACL2_*1*_ACL2::REV (1 2 . 7))>
    2> (REV (1 2 . 7))>
      3> (REV (2 . 7))>
        4> (REV 7)>
        <4 (REV NIL)>
      <3 (REV (2))>
    <2 (REV (2 1))>
  <1 (ACL2_*1*_ACL2::REV (2 1))>
  1> (ACL2_*1*_ACL2::REV (2 1))>
    2> (REV (2 1))>
      3> (REV (1))>
        4> (REV NIL)>
        <4 (REV NIL)>
      <3 (REV (1))>
    <2 (REV (1 2))>
  <1 (ACL2_*1*_ACL2::REV (1 2))>
(1 2)
ACL2 !>
```

Finally, we note that users can find ways to use ACL2 prover capabilities to do proof debugging. For example, consider the ACL2 function `force`. Although `force` is semantically the identify function, the ACL2 rewriter treats it specially: when encountering a top-level call of `force` in considering (an instance of) a rewrite rule's hypothesis that it is unable to rewrite to `T`, the rewriter nevertheless discharges the hypothesis while making a note to reconsider it at the end of the proof. At that point the full theorem prover can be brought to bear on that hypothesis, not merely the rewriter. Sandip Ray (personal communication) has told us that he uses `force` as a proof debugging tool. The idea is that when a rewrite rule fails to be applied as expected, then instead of using monitoring (Section 7) or a goal manager (Section 10), one uses ACL2's *redefinition* utility to prove a new version of the problem rule that has `force` applied to each hypothesis. Then at the end of a new attempt at proving the theorem of interest, one gets insight from seeing the unproved, forced hypotheses.

12 Conclusion

We have described a number of debugging tools that allow the user of the ACL2 theorem prover to see what the system is doing both in real time and in static summaries.

We often call ACL2 an interactive automatic theorem prover. By that we mean to indicate that even though it is automatic once it has been given a conjecture to prove, nevertheless its search behavior is significantly influenced by the user. The power of interactive automatic inference tools like ACL2 is that with them humans and machines can accomplish reasoning feats that are far beyond what either humans or machines alone could accomplish with comparable levels of precision and correctness.

Until our theorem provers are so good that they are no long interactive, the user must help with suggestions, often using insight and creativity. But given the number of steps explored, the user often needs to be told what the system is doing at just the right level of detail, with abstractions of its operation that are accurate enough to be predictive.

We see the development of proof debugging tools as necessary for the industrial success of mechanized formal methods.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
2. P. Dillinger, P. Manolios, J S. Moore, and D. Vroon. ACL2s: "The ACL2 Sedan". *Theoretical Computer Science*, 174(2):3–18, 2006.
3. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.
4. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
5. M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2008.
6. M. Kaufmann and J S. Moore. Rewriting with equivalence relations in ACL2. *Journal of Automated Reasoning*, to appear. available online from SpringerLink.
7. H. Liu and J S. Moore. Java program verification via a JVM deep embedding in ACL2. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *17th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 184–200. Springer, 2004.