How Can I Do That with ACL2?
Recent Enhancements to ACL2

Matt Kaufmann

Dept. of Computer Science,
University of Texas at Austin
kaufmann@cs.utexas.edu

J Strother Moore

Dept. of Computer Science,
University of Texas at Austin
moore@cs.utexas.edu

The last several years have seen major enhancements to ACL2 functionality, largely driven by requests from its user community, including utilities now in common use such as `make-event`, `mbe`, and trust tags. In this paper we provide user-level summaries of some ACL2 enhancements introduced after the release of Version 3.5 (in May, 2009, at about the time of the 2009 ACL2 workshop) up through the release of Version 4.3 in July, 2011, roughly a couple of years later. Many of these features are not particularly well known yet, but most ACL2 users could take advantage of at least some of them. Some of the changes could affect existing proof efforts, such as a change that treats pairs of functions such as `member` and `member-equal` as the same function.

1 Introduction

This paper discusses ACL2 enhancements made in ACL2 Versions 3.6 through 4.3, that is, during the 2+ years that have passed since Version 3.5 was released around the time of the preceding (2009) ACL2 workshop. These enhancements primarily concern programming, proof control, and system infrastructure — as opposed to improved proof procedures, sophisticated logical extensions, and theoretical studies. Readers from outside the ACL2 community — should there be any! — may find this pragmatic stance surprising. But ACL2's total integration of programming, proof, and system issues is one of the reasons ACL2 finds industrial application and we believe that the research issues raised by our commitment to integration are at least as important as more conventional theorem proving work.

Even though ACL2 is typically modified in response to user requests, still we suspect that most recent enhancements are unknown to most ACL2 users. Perhaps that is because the release notes for the above versions of ACL2 list about 300 improvements, hence serving as a large, rather flat and complete reference document that one may prefer not to read carefully. Our goal here is to raise awareness of the most important of these enhancements.

Our focus is on the user level, both here and in the release notes. This paper thus includes many examples. We do not claim that this paper covers every interesting enhancement. A more complete summary of changes can be found in the release notes and various documentation topics. Indeed, as our goal is to bring awareness of recent ACL2 changes to the community, even the topics that we do cover in this paper are sometimes dispatched with no more than pointers to relevant documentation topics. Thus, what we say in this paper is in the spirit of past ACL2 Workshop talks on "What's New".

We highlight documentation topics with the marker "see :DOC"; for example, see :DOC release-notes and its subtopics (e.g., see :DOC note-3-6 and see :DOC note-4-3 for changes introduced in ACL2 Versions 3.6 and 4.3, respectively). We also refer to documentation topics implicitly using underlining, for example: the topic `acl2-tutorial` is much improved. For both kinds of references to documentation topics (explicit and implicit), online copies of this paper have hyperlinks to the documentation topic.

Hardin and Schmaltz (Eds): ACL2 2011
EPTCS 70, 2011, pp. 46–60, doi:10.4204/EPTCS.70.4

Those interested in implementation details are invited to see the source code, which is extensively commented (see Subsection 4.3), available from the ACL2 home page [5]. In particular, each `deflabel` form for a release note has Lisp comments typically at a lower level than the user documentation.

We present each enhancement by way of a question that we believe might be asked by some ACL2 users, which is followed by an answer. These enhancements break naturally into categories. We begin in Section 2, which focuses on new programming features. Next, Section 3 discusses enhancements pertaining to doing proofs. Finally, Section 4 addresses changes at the system level. We conclude with brief reflections.

2 Programming Features

In this section we describe several recent ACL2 enhancements that are of particular use when programming.

2.1 Equality variants

Question: *How can I avoid proving separate sets of rules for pairs of functions such as* `member` *and* `member-equal`*, which have logically equivalent definitions but use different equality tests in their definitions?*

To understand the question, recall that `equal`, `eql` and `eq` are logically equivalent functions with different guards and different runtime efficiencies. The user is expected to choose the variant that provides the most appropriate tradeoff between proof obligations and runtime performance. The variants of `member` differ only by the equality test used; hence they may be proved equivalent but are not defined identically. Now consider the following sequence of two theorems, both proved automatically by ACL2. Because the function `reverse` is defined in terms of `revappend`, ACL2 can automatically apply the first theorem (as a rewrite rule) to prove the second theorem (without using induction).

```
(defthm member-revappend
  (iff (member a (revappend x y))
       (or (member a x)
           (member a y)))
  :hints (("Goal" :induct (revappend x y))))
(defthm member-reverse
  (iff (member a (reverse x))
       (member a x)))
```

But the corresponding theorem about `member-equal`, just below, fails to be proved in ACL2 versions preceding 4.3, where `member` and `member-equal` were essentially different functions: their recursive definitions were similar but differed in the equality test used (`eql` or `equal`, respectively).

```
(defthm member-equal-reverse
  (iff (member-equal a (reverse x))
       (member-equal a x)))
```

However, after Version 4.2, the proof succeeds for `member-equal-reverse`. Indeed, if `member-reverse` is proved first, then it is applied in the proof of `member-equal-reverse`. The upshot is that we no longer need to create separate libraries of rules for `member` and `member-equal`.

Briefly put, the change is that `member` is a macro that generates a call of `member-equal` in the logic. Here is a log showing in some detail the macroexpansion of calls of `member-eq` and `member`, using `:trans1`.

```
ACL2 !>:trans1 (member-eq a x)
 (MEMBER A X :TEST 'EQ)
ACL2 !>:trans1 (member a x :test 'eq)
 (LET-MBE ((X A) (L X))
          :LOGIC (MEMBER-EQUAL X L)
          :EXEC (MEMBER-EQ-EXEC X L))
ACL2 !>:trans1 (let-mbe ((x a) (l x))
                        :logic (member-equal x l)
                        :exec (member-eq-exec x l))
 (LET ((X A) (L X))
      (MBE :LOGIC (MEMBER-EQUAL X L)
           :EXEC (MEMBER-EQ-EXEC X L)))
ACL2 !>
```

As seen above, calls of macros `member-eq` and `member` ultimately generate calls of the function `member-equal` within the `:logic` component of an `mbe` call. Many parts of the ACL2 reasoning engine reduce an `mbe` call to its `:logic` component; so it is fair to say that the ACL2 prover treats a call of `member-eq` or `member` as a corresponding call of the function `member-equal`. Indeed, as part of this change we extended such reduction of `mbe` calls to additional contexts (for more on this, search the documentation for references to "guard holder").

For more information about uniform treatment of functions whose definitions differ only on the equality predicates used, including a full listing of such functions, see :DOC equality-variants.

2.2 Defattach

Question: *How can I execute encapsulated functions, modify certain built-in function behavior, or program using refinements?*

The `defattach` utility [3, 4] provides all of the above. Consider for example the following sequence of events, which introduces a "fold" function that applies a given associative-commutative function to successive members of a list.

```
(encapsulate
 (((ac-fn * *) => *
   :formals (x y)
   :guard (and (acl2-numberp x)
               (acl2-numberp y))))
 (local (defun ac-fn (x y)
          (+ x y)))
 (defthm ac-fn-commutative
   (equal (ac-fn x y)
          (ac-fn y x)))
 (defthm ac-fn-associative
   (equal (ac-fn (ac-fn x y) z)
          (ac-fn x (ac-fn y z)))))
(defun fold (lst root)
  (cond ((endp lst) root)
        (t (fold (cdr lst)
                 (ac-fn (car lst) root)))))
```

At this point, evaluation of `(fold '(2 3 4 5) 1)` fails, because `fold` calls `ac-fn`, which is not defined. But if we *attach* the built-in ACL2 multiplication function to `ac-fn` we can do such evaluation, as shown below. Indeed, we can use evaluation to explore conjectures, such as whether the value returned by a call of `fold` is unchanged if its first argument is reversed. We omit the output from the call of `defattach`, which shows proof obligations being discharged.

```
ACL2 !>(defattach ac-fn binary-*)
[[.. output omitted..]]
ACL2 !>(fold '(2 3 4 5) 1)
120
ACL2 !>(fold (reverse '(2 3 4 5)) 1)
120
ACL2 !>
```

Note however that attachments are not invoked during proofs. Continuing with the example above, the proof fails for `(thm (equal (fold '(2 3 4 5) 1) 120))`. Indeed, because attachments can be overwritten with new attachments it is important that they are turned off not only for proofs but also for other logical contexts, such as the evaluation of `defconst` forms.

The discussion above shows how `defattach` supports execution of encapsulated function calls and gives a hint about refinement. But a third use is the modification of built-in function behavior, towards opening up the architecture of ACL2. Certain ACL2 prover functions are now implemented with `defattach` (see source file `boot-strap-pass-2.lisp`), permitting the user to customize some heuristics by attaching other functions to them. We invite the user community to request more such support. One example is the built-in function `ancestors-check`, which implements a rewriting heuristic. Robert Krug requested that this function be attachable, and we thank him for that; actually he went further and provided the necessary proof support.

There is much more to know about `defattach`, but our goal in this paper is simply to provide an introduction to it. To learn more see :DOC defattach. For logical foundations and (significant) implementation subtleties, see a comment in the ACL2 source code entitled “Essay on Defattach”, which explains the subtle logical foundations of `defattach`, and will ultimately be incorporated into a comprehensive treatment [4].[1]

2.3 Return-last

Question: *How can I arrange that my macros have raw-Lisp side effects, like time$?*

Recall that `(time$ form)` is semantically the same as `form`, except that timing information is printed to the terminal after evaluation is complete. To see how this works, we consider the macroexpan-

[1] We thank César Muñoz and Shankar for useful email discussions on relationships between the `defattach` feature of ACL2 and the PVS features of `defattach`, theory interpretations, and theory parameters with assumptions. We expect to explore these relationships in the aforementioned paper.

sion of a call of `time$`. Note: the interpretation of `(list 0 nil nil nil nil)` is not important for this explanation.

```
ACL2 !>:trans1 (time$ (foo 3 4))
 (TIME$1 (LIST 0 NIL NIL NIL NIL)
         (FOO 3 4))
ACL2 !>:trans1 (time$1 (list 0 nil nil nil nil)
                        (foo 3 4))
 (RETURN-LAST 'TIME$1-RAW
              (LIST 0 NIL NIL NIL NIL)
              (FOO 3 4))
ACL2 !>
```

In the logic, `return-last` is a function that returns its last argument (as its name suggests). But in raw Lisp, `return-last` is a macro. In essence, it expands to a call of the (unquoted) first argument on the remaining two arguments, which should return the value of the last argument.

```
ACL2 !>:q
Exiting the ACL2 read-eval-print loop.  To re-enter, execute (LP).
? [RAW LISP] (macroexpand-1
              '(return-last 'time$1-raw
                            (list 0 nil nil nil nil)
                            (foo 3 4)))
(TIME$1-RAW (LIST 0 NIL NIL NIL NIL) (FOO 3 4))
T
? [RAW LISP]
```

The raw Lisp macro `time$1-raw` is what actually carries out the timing of the indicated call of `foo` above.

Note that there must be an active trust tag (see :DOC defttag) in order to extend the special treatment of `return-last` to additional values of its first argument. It is the user's responsibility, when making such an extension, to ensure that any call of the value of the first argument does indeed return the value of the last argument — which brings us back to the original question, above, which is how to make such an extension.

A macro `defmacro-last` makes it a rather simple undertaking to make such an extension. The distributed book `books/misc/profiling.lisp` illustrates how this works by defining an ACL2 macro, `with-profiling`, together with a raw-Lisp macro `with-profiling-raw` that causes the desired side effects. For more explanation of this example, and of `defmacro-last` and `return-last`, see :DOC return-last.

```
; A trust tag is needed for progn!; see :DOC defttag.
(defttag :profiling)
(progn!
 (set-raw-mode t)
 (load (concatenate 'string
                    (cbd)
                    "profiling-raw.lsp")))
(defmacro-last with-profiling)
```

Additional examples show the flexibility of `defmacro-last`. Sol Swords and Jared Davis have used `defmacro-last` to create a macro `with-fast-alist` for the HONS version of ACL2, which is defined and documented in the book `centaur/misc/hons-extra.lisp` distributed in the acl2-books svn repository [1]. David Rager has used `defmacro-last` to create a timing utility that shows garbage collection information, distributed with ACL2 as `books/tools/time-dollar-with-gc.lisp`. Both books, as well as the book `profiling.lisp` mentioned above, come with associated raw Lisp files that implement the desired side effects.

2.4 Avoiding guard violations

Question: *How can I avoid errors on ill-guarded calls, even in raw Lisp, and even for* `:program` *mode functions?*

See :DOC with-guard-checking and see :DOC ec-call. The latter was introduced in ACL2 Version 3.4, and replaces a call with its so-called "executable-counterpart". But `with-guard-checking` is newer (introduced in Version 4.0), and can be used to suppress guard checking for executable counterparts. The following example illustrates how these two work together to answer the above question.

```
ACL2 !>(defun foo (x)
         (declare (xargs :mode :program))
         (with-guard-checking nil (ec-call (car x))))
Summary
Form:  ( DEFUN FOO ...)
Rules: NIL
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 FOO
ACL2 !>(foo 3)
NIL
ACL2 !>
```

Note that the use of `ec-call` is necessary in order to avoid calling `car` on `3` in raw Lisp. If instead `foo` were defined in `:logic` mode, then the use of `ec-call` would not be necessary above because the executable-counterpart of `car` would be called on `3`.

For background about how guards and evaluation work, see :DOC guard and its subtopics; in particular see :DOC guards-and-evaluation and see :DOC guard-evaluation-table.

2.5 Printing without state

Question: *I'm doing printing without accessing state. How can I avoid producing messages when proof output is turned off? More generally, how best can I print without actually reading or writing the state?*

Macros `observation-cw` and `warning$-cw` are analogues of macros `observation` and `warning$` that, however, do not access `state`. We strongly suggest using these in place of the macro `cw` in functions called during a proof, for example during evaluation of clause-processors or computed hints, so that users can turn off such messages by using `set-inhibit-output-lst`.

Remarks. (1) The above two macros are implemented using wormholes, which were given an improved implementation in Version 4.0 and later. (2) There are now many utilities with the suffix

“-cmp”. These traffic in so-called “context-message pairs”, rather than state, as described in the “Essay on Context-message Pairs” in the ACL2 source code (file translate.lisp).

Question: *How can I create a string using formatted printing functions, without printing out the string and preferably without accessing the ACL2 state?*

See :DOC printing-to-strings for analogues of functions like fmt that return strings and do not access state. For example:

```
ACL2 !>(fmt1-to-string "Hello, ~x0"
                       (list (cons #\0 'world))
                       0)
(12 "Hello, WORLD")
ACL2 !>
```

Also see :DOC io for a discussion of how to open a channel that connects to a string, along with an associated utility for retrieving the string printed to that channel, get-output-stream-string$ (which however does access the ACL2 state).

Remark for system developers. If you are willing to use trust tags (see :DOC defttag), then see :DOC with-local-state for a potentially unsound utility that allows you to create a temporary ACL2 state object out of thin air!

2.6 Parallel evaluation

Question: *How can I build an application that evaluates code in parallel?*

ACL2(p) is an experimental extension of ACL2 that incorporates research and code from David Rager [7, 9]. Recent additions include a macro spec-mv-let, which allows speculative evaluation in parallel. See :DOC parallelism. Later below we discuss parallel proofs of subgoals.

2.7 Other recent programming support

Question: *How can I get around some syntactic restrictions imposed by the use of multiple values?*

The macros mv? and mv-let? are analogues of mv and mv-let which, however, may return or bind just one variable (respectively).

The function mv-list converts multiple values to a single value that is a list, for example as follows.

```
ACL2 !>(mv-list 3 (mv 5 6 7))
(5 6 7)
ACL2 !>(cdr (mv-list 3 (mv 5 6 7)))
(6 7)
ACL2 !>
```

Question: *How can I redefine system functions and macros inside the ACL2 loop?*

A utility for this purpose, redef+, is now an embedded event form (i.e., it can go in books). Note that the counterpart of :redef+, :redef-, now turns off redefinition (it formerly had not done so).

Question: *What support is provided for tracing function calls inside the ACL2 loop?*

See :DOC trace$. Although this utility has been around for many years, it has benefited from recent improvements.

Question: *What other recent ACL2 programming enhancements might I be missing?*

See :DOC release-notes. Useful new features include the following.

- The macro `time$` is now user-customizable (thanks to an initial implementation contributed by Jared Davis).
- The function `pkg-imports` returns the list of symbols imported into a specified package.
- `(File-write-date$ filename state)` returns the Common Lisp file-write-date of the given filename.
- The macro `append` no longer requires two or more arguments: now `(append)` expands to `nil`, and `(append X)` expands to `X`.

3 Proof Debug, Control, and Reporting

This section addresses recent ACL2 improvements in user interaction with the ACL2 prover.

3.1 Hints

The hints mechanism continues to become more flexible and better documented. In Subsection 3.2 we discuss one major improvement, the use of the `:instructions` keyword in hints; but first we point out several other advances in hints.

The first two new features mentioned below, override-hints and backtrack hints, have been used to integrate testing with the ACL2 prover [2].

Question: *How can I provide default hints that are not ignored when I give explicit hints to goals?*

See :DOC override-hints.

Question: *The hints mechanism has always confused me a bit; for example, some hints are inherited by subgoals and others are not. How can I better understand the "flow" of hints?*

See :DOC hints-and-the-waterfall for a detailed explanation of how hints are processed. Also, some helpful examples may be found in distributed book `books/hints/basic-tests.lisp`.

Question: *How can I write a computed hint that can backtrack if 'undesirable' subgoals are created?*

See :DOC hints for a discussion of `:backtrack` hints.

Question: *How can I program up fancy computed hints that do not keep announcing "thanks" each time one is applied?*

See :DOC hints for a discussion of `:no-thanks` hints.

Question: *I know how to limit backtracking in the rewriter by using* `set-backchain-limit`, *but how can I do this at the level of hints?*

See :DOC hints for a discussion of `:backchain-limit-rw`.

3.2 Proof-checker enhancements

Question: *How can I better employ the proof-checker to create proper ACL2 events?*

See :DOC proof-checker for a (long-standing) utility for conducting proofs interactively. Probably the most common use of the proof-checker is to invoke `verify` to explore the proof of a conjecture whose automated attempt has failed. But sometimes it is convenient to save a proof-checker proof using its `:exit` command, creating an event by pasting that proof as the value of an `:instructions` keyword. An example is given below.

The proof-checker's use in the creation of events has recently been made more flexible in two ways.

- User-defined macro commands (see :DOC define-pc-macro) are now legal for `:instructions`.
- The use of an `:instructions` keyword is now supported inside `:hints`, in particular at the subgoal level.

Below is an example, inspired by the event `not-equal-intern-in-package-of-symbol-nil` from the distributed book `books/coi/gensym/gensym.lisp`. You'll see that we exit the proof-checker when we've gotten past the sticky bit, and that we use the new capability for putting `:instructions` inside `:hints` (though that's not actually needed for this example).

```
ACL2 !>(verify
        (implies
         (and (stringp string)
              (symbolp symbol)
              (equal (intern-in-package-of-symbol string symbol)
                     nil))
         (equal string "NIL")))
->: bash
[[.. output omitted; simplified to one goal ..]]
->: th  ; show current goal's hypotheses and conclusion
*** Top-level hypotheses:
1. (STRINGP STRING)
2. (SYMBOLP SYMBOL)
3. (NOT (INTERN-IN-PACKAGE-OF-SYMBOL STRING SYMBOL))
The current subterm is:
(EQUAL STRING "NIL")
->: (casesplit  ; split into two goals, by cases
     (not  ; using the negation makes example more interesting
      (equal (symbol-name
              (intern-in-package-of-symbol string symbol))
             string)))
Creating one new goal:  ((MAIN . 1) . 1).
->: prove
[[.. output omitted; the proof fails ..]]
->: th
*** Top-level hypotheses:
1. (STRINGP STRING)
```

```
2. (SYMBOLP SYMBOL)
3. (NOT (INTERN-IN-PACKAGE-OF-SYMBOL STRING SYMBOL))
4. (NOT (EQUAL (SYMBOL-NAME (INTERN-IN-PACKAGE-OF-SYMBOL STRING
                                                         SYMBOL))
               STRING))
The current subterm is:
(EQUAL STRING "NIL")
->: (drop 3)  ; Drop the third hypothesis.
; Hypothesis 4 is false, but hypothesis 3 gets in the way.
->: prove
***** Now entering the theorem prover *****
But simplification reduces this to T, using primitive type
reasoning and the :rewrite rule
SYMBOL-NAME-INTERN-IN-PACKAGE-OF-SYMBOL.
Q.E.D.
The proof of the current goal, (MAIN . 1), has been completed.
However, the following subgoals remain to be proved:
  ((MAIN . 1) . 1).
Now proving ((MAIN . 1) . 1).
->: (exit t)
Not exiting, as there remain unproved goals:  ((MAIN . 1) . 1).
The original goal is:
    (IMPLIES (AND (STRINGP STRING)
                  (SYMBOLP SYMBOL)
                  (EQUAL (INTERN-IN-PACKAGE-OF-SYMBOL STRING
                                                      SYMBOL)
                         NIL))
             (EQUAL STRING "NIL"))
  Here is the current instruction list, starting with the first:
    (:BASH
     (:CASESPLIT
        (NOT (EQUAL (SYMBOL-NAME
                     (INTERN-IN-PACKAGE-OF-SYMBOL STRING SYMBOL))
                    STRING)))
     (:DROP 3)
     :PROVE)
->: exit
Exiting....
 NIL
; Now we can paste in the above instructions and prove the theorem.
ACL2 !>(thm
        (implies
         (and (stringp string)
              (symbolp symbol)
              (equal (intern-in-package-of-symbol string symbol)
                     nil))
         (equal string "NIL"))
        :hints
        (("Goal"
          :instructions
          (:BASH
            (:CASESPLIT
              (NOT (EQUAL (SYMBOL-NAME (INTERN-IN-PACKAGE-OF-SYMBOL
                                        STRING SYMBOL))
                          STRING)))
            (:DROP 3)
            :PROVE))))
[Note:  A hint was supplied for our processing of the goal above.
Thanks!]
We now apply the trusted :CLAUSE-PROCESSOR function
PROOF-CHECKER-CL-PROC to produce one new subgoal.
Goal'
(IMPLIES
     (AND (STRINGP STRING)
          (SYMBOLP SYMBOL)
          (NOT (INTERN-IN-PACKAGE-OF-SYMBOL STRING SYMBOL))
          (EQUAL (SYMBOL-NAME (INTERN-IN-PACKAGE-OF-SYMBOL STRING
                                                           SYMBOL))
                 STRING))
     (EQUAL STRING "NIL")).
But simplification reduces this to T, using the
:executable-counterpart of SYMBOL-NAME.
Q.E.D.
Summary
Form:  ( THM ...)
Rules: ((:EXECUTABLE-COUNTERPART SYMBOL-NAME))
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
Prover steps counted:  69
Proof succeeded.
ACL2 !>
```

3.3 Parallelism in proofs

Question: *Can I speed up proofs by having subgoals proved in parallel?*

Yes, if you build the experimental extension for parallelism that incorporates David Rager's dissertation work [8]. See :DOC parallelism.

3.4 Limiting proof effort

Question: *I like using* `with-prover-time-limit` *to limit proof effort, but is there something similar that is platform-independent?*

See :DOC with-prover-step-limit. Also see :DOC set-prover-step-limit, which lets you set the default limit for the current environment (whether it be at the top level, or in an `encapsulate`, `progn`, `make-event`, `certify-book`, etc.).

Note that `with-prover-step-limit` may be used to form events in books. The same is now true (but had not been in the past) for `with-prover-time-limit`.

3.5 Proof debugging

Question: *I formerly used* `accumulated-persistence`*, but its output seemed too limited. Are there any new options that could make it more useful?*

By default, `show-accumulated-persistence` now breaks down the statistics by “useful” and “useless” applications of the rules. If you enable the feature with `(accumulated-persistence :all)`, then statistics are further broken down by rule hypothesis and conclusion.

Also, see :DOC accumulated-persistence for a discussion of the `:runes` option for obtaining a raw, alphabetical listing.

Question: *A* `defthm` *failed in the middle of an* `encapsulate` *or* `certify-book`*. How can I get into a state where I can work on the failed proof?*

`Redo-flat` has been around since Version 3.0.1, but among the latest improvements is that now it works for `certify-book`.

Question: *How can I get debug-level information on what’s going on with forward-chaining?*

See :DOC forward-chaining-reports.

Question: *How do I control all the noise I get from proofs?*

Starting with Version 4.0, you can inhibit specified parts of the Summary printed at the conclusion of an event; see :DOC set-inhibited-summary-types. For example, ACL2 developers sometimes evaluate the form

```
(set-inhibited-summary-types ’(time))
```

to compare proof output from two runs without the distraction of time differences.

But the most important recent such development is a bit older, introduced in Version 3.3: gag-mode. Gag-mode allows you to turn off all but key prover output, so that you can focus on key checkpoints (see :DOC the-method and see :DOC introduction-to-the-theorem-prover). We may well make gag-mode the default at some point in the future.

Two improvements to gag-mode were introduced with Version 4.3. (1) The printing of induction schemes is suppressed in gag-mode. (2) You can now limit the printing of subgoal names when using `:set-gag-mode :goals`; see :DOC set-print-clause-ids.

3.6 New heuristics

Question: *Are there any new developments in proof heuristics?*

There are two significant new heuristics in Version 4.3.

ACL2 now caches information for failed applications of rewrite rules. We have seen a speedup of 11% on the ACL2 regression suite, but in some cases the speedup is significantly higher. See :DOC set-rw-cache-state for information about controlling this feature, including information on how to turn it off in the very unlikely case that it makes a proof fail.

Our description of the second heuristic relies on an understanding of free variables in hypotheses of rules; see :DOC free-variables. Since Version 2.2 (November, 2002), ACL2 has by default considered every match from the current context for free variables in a hypothesis of a rewrite, linear, or forward-chaining rule, until finding a match for which the rule's hypotheses are all discharged. Now, that behavior is also the default for type-prescription rules; see :DOC free-variables-type-prescription.

Question: *I'd like to learn more about how to use the ACL2 prover effectively, and I'm willing to do some reading about that. But where should I start?*

The acl2-tutorial :DOC topic has been significantly expanded and improved. It contains pointers to different materials that you may choose to read, depending on your learning style. In particular, see :DOC introduction-to-the-theorem-prover for a tutorial on how to use the ACL2 prover effectively.

4 System-level Enhancements

Here we discuss a few infrastructural improvements other than direct support of programming and proofs. Most experienced ACL2 users consider system infrastructure an important component of ACL2's usability.

4.1 Two-run certification to avoid trust tags

The first question below is only likely to be asked by system builders.

Question: *How can I certify a book that uses unverified proof tools whose solutions I know how to check — without making the book depend on a trust tag?*

See :DOC set-write-acl2x.

4.2 Certifying a subset of the distributed books

Question: *How can I better control book certification? In particular, I'd like to avoid certifying all the distributed books, since I only intend to include some of them.*

See :DOC book-makefiles for answers to such infrastructural questions. In particular, see the discussion there of environment variable `ACL2_BOOK_DIRS`.

It has been the case for some time that by default, no acl2-customization file is loaded during '`make regression`'. The above documentation topic also mentions the new name for an environment variable, now `ACL2_CUSTOMIZATION`, and explains how it can be used to override that default behavior.

4.3 Size and breakdown of ACL2 source code

Question: *How big is the ACL2 source code?*

We now distribute a file `doc/acl2-code-size.txt`. Feel free to poke around in the `doc/` directory, or email the authors of ACL2, if you want to use the same tools we use to compute size. As of this writing, here are the contents of the above file.

```
CODE LINES:
   97465 lines,   4270000 characters
COMMENT LINES:
   51917 lines,   3062889 characters
BLANK LINES (excluding documentation):
   22884 lines,     24404 characters
DOCUMENTATION LINES:
   79543 lines,   3550075 characters
TOTAL:
  251809 lines,  10907368 characters
```

4.4 An mbe restriction lifted

Question: *Is there any way to call* `mbe` *in the body of a definition within an* `encapsulate` *that has a non-empty signature?*

Yes. Some such restriction is necessary (see :DOC note-3-4). However, this restriction is now lifted provided you declare the definition to be non-executable (typically by using `defun-nx`).

4.5 Aborting just one ld level

Question: *How can I avoid popping all the way back to the top level when I merely want to exit the* `:brr` *"break-rewrite" loop?*

Use `:p!` instead of `:a!`. This same trick works if you are in a nested call of `ld`.

5 Concluding remarks

We believe that one of ACL2's greatest strengths is its integration of programming and proof — with due regard for both efficiency and soundness. The ACL2 system continues to evolve through feedback from the ACL2 user community. Many of the enhancements discussed here came about in response to such feedback; see :DOC release-notes to find specific individuals associated with enhancement requests. We very much appreciate the opportunity to improve ACL2 in useful ways, and thus we strongly encourage ACL2 users to let us know how we can make the system more effective for them.

Acknowledgements

We are grateful to the ACL2 user community for being primary drivers of the development of ACL2. We also thank Sandip Ray for helpful feedback on a draft of this paper.

This material is based upon work supported the National Science Foundation under Grant Nos. CCF-0945316 and CNS-0910913.

References

[1] The ACL2 community. The acl2-books svn repository. See http://acl2-books.googlecode.com/.

[2] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating Testing and Interactive Theorem Proving. In: ACL2 ’11: Proceedings of the ninth international workshop on the ACL2 theorem prover and its applications. November 2011, Austin, Texas.

[3] M. Kaufmann. Trusted Extension of ACL2 System Code: Towards an Open Architecture. In: Workshop on Trusted Extensions of Interactive Theorem Provers, Cambridge, UK, August 11-12, 2010, http://www.cs.utexas.edu/users/kaufmann/itp-trusted-extensions-aug-2010/.

[4] M. Kaufmann and J S. Moore. Defattach: A utility for formally verified refinement and evaluation. In preparation.

[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.

[6] M. Kaufmann and J S. Moore. ACL2 User’s Manual, http://www.cs.utexas.edu/users/moore/acl2/current/acl2-doc.html#User's-Manual.

[7] D. L. Rager. Adding Parallelism Capabilities in ACL2. In: ACL2 ’06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications. 2006, pp. 90–94, Seattle, Washington. ACM, New York, New York, USA. doi:10.1145/1217975.1217994

[8] D. L. Rager. Ph.D. Dissertation, University of Texas at Austin. In preparation.

[9] D. L. Rager and and W. A. Hunt, Jr. Implementing a Parallelism Library for a Functional Subset of Lisp. In: Proceedings of the 2009 International Lisp Conference, 2009, Cambridge, Massachusetts, pp. 18–30. Association of Lisp Users, Sterling, Virginia, USA.

[10] G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.