Maintaining the ACL2 Theorem Proving System

Matt Kaufmann and J Strother Moore
University of Texas
{kaufmann,moore}@cs.utexas.edu

Abstract

This talk will provide a view into the task of improving the ACL2 theorem prover to meet users' needs.

1 Introduction

The goal of this talk is to provide a sense of some possible challenges to making a theorem proving system useful in practice. Specifically, we will draw from our experiences maintaining the ACL2 theorem proving system [KMM00b, KM04a]. Although ACL2 incorporates years of ongoing research in automated reasoning, the focus of this talk is on the engineering required to make the system useful.

Imagine you're at lunch, where two guys are blabbing to you about their work. Fortunately, this blabbing might be of some interest, since they are talking about how they make their theorem proving system easier to use. They definitely want you to ask questions and share your own related experiences, though you may find it hard to interrupt them after they get started.

But even before we eat, they insist on providing some general background on their system so that when they get going, at least they might make some sense. By the time dessert arrives, these two guys will be eager for questions and comments, if for no other reason than that they can eat!

Thus, this talk consists of three parts.

First, *Before We Eat*: While we're walking to lunch you'll get some general background on ACL2. Then while we're waiting for a table, we'll see a small example that gives a sense of how ACL2 is used. After we're seated, we'll take a quick look at a list of features that we'll come across while we eat.

Next, for the *Main Course*, we will focus on some selected items taken from recent ACL2 release notes (minimally edited in a few cases). These selected items should give a sense of what can be involved in maintaining an empirically successful automated reasoning system.

Finally, we will have a give-and-take during *Dessert*. With any luck, the ensuing questions and comments will be stimulating and informative for all of us.

We will use footnotes for material that we will make a point of skipping during the talk, due to time constraints.[1]

[1] A related talk in 1991 [Kau91] gives a large list of aspects of mechanized reasoning systems and

2 Before We Eat — Some general background on ACL2

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." The ACL2 system is the latest in the Boyer-Moore family of provers, and is joint work of Matt Kaufmann and J Moore, with substantial early contributions from Bob Boyer and ongoing contributions from many. The paper [KM04b] provides a reasonably self-contained introduction to ACL2, including relevant background and how to use the system. Quoting from that paper:

> "ACL2" is the name of a functional programming language (based on Common Lisp), a first-order mathematical logic, and a mechanical theorem prover. ACL2, which is sometimes called an "industrial strength version of the Boyer-Moore system," is the product of Kaufmann and Moore, with many early design contributions by Boyer. It has been used for a variety of important formal methods projects of industrial and commercial interest, including....

A long but incomplete list of applications is then given, including various algorithms, software, and hardware designs.

ACL2 is freely available, distributed under the GPL [gpl]. It can be obtained from the ACL2 home page [KM04a], which also has links to many papers that describe ACL2 applications, as well other useful links (mailing lists, tours, demos, documentation, workshops, and installation instructions).

Here is some relevant data.

- Some milestones:
  - 1973: The Boyer/Moore "Edinburgh Pure Lisp Theorem Prover"
  - 1979: Boyer and Moore, A Computational Logic [BM79]
  - 1986: Kaufmann joins Boyer/Moore project
  - 1988: Boyer and Moore, A Computational Logic Handbook (1997, 2nd ed. [BM97])
  - 1989: Boyer and Moore begin ACL2
  - 1992: Final release of Boyer-Moore "Nqthm" prover
  - 1993: Kaufmann formally added as a co-author of ACL2
  - 2000: Kaufmann, Manolios, and Moore write a book on ACL2, Computer-Aided Reasoning: An Approach [KMM00b], and edit proceedings of the first ACL2 workshop, Computer-Aided Reasoning: ACL2 Case Studies [KMM00a]

comments on them. The focus here is narrower: What sorts of things must be done to make such a system useful? Our focus is actually still narrower, as for example the following are critical but not addressed directly here: fundamental reasoning algorithms, execution efficiency, logical foundations, system architecture, and trust. Rather, we present here selections from release notes that give a flavor of the maintenance required on a particular mature system, ACL2, in order to make it a system that (some) people want to use. That is, the point here is not to give an overview of what ACL2 provides, but to focus on maintenance.

  - 2006: Boyer, Kaufmann, and Moore win ACM Software System Award for Boyer-Moore family of provers

- Version 3.0 source files size: 8.3M of Common Lisp (including source code, documentation strings, and comments)
- There are 229 release note items strictly after Version 2.8 (March, 2004) as follows (but we'll look at just a few of these):
  - 63 in Version 2.9, October, 2004
  - 19 in Version 2.9.1, December, 2004
  - 30 in Version 2.9.2, April, 2005
  - 31 in Version 2.9.3, August, 2005
  - 53 in Version 2.9.4, February, 2006
  - 33 in Version 3.0, June, 2006

An interesting aspect of ACL2 is that it is written in its own formal language (with the exception of a small amount of Common Lisp code mainly of a bootstrapping nature). This has forced us to make ACL2's formal programming language, which is a carefully-crafted extension of an applicative subset of Common Lisp, a language that is both efficient and convenient to use. As a result, ACL2 users often employ ACL2's programming environment to write tools.

2.1 The user's view of ACL2: A small example

The following example will provide a sense of ACL2. The first thing to notice is that the syntax is Lisp's prefix syntax, so for example we write `(+ 3 4)` and `(len x)` rather than more traditional notation such as `3+4` and `len(x)`, respectively. The syntax is case-insensitive.

Suppose that we want to prove that the length does not change when we reverse a list. Lists and some list-processing functions, including `reverse` and `len` for reverse and length of a list, are built into ACL2. So let us submit a theorem named `len-reverse`, stating that for any list `x`, the length of the reverse of `x` is equal to the length of `x`:[2]

```
ACL2 !>(defthm len-reverse
         (implies (true-listp x)
                  (equal (len (reverse x)) (len x))))

ACL2 Warning [Non-rec] in ( DEFTHM LEN-REVERSE ...):  A :REWRITE rule
generated from LEN-REVERSE will be triggered only by terms containing
the non-recursive function symbol REVERSE.  Unless this function is
disabled, this rule is unlikely ever to be used.
```

[2]The warning below is not too important here. Think of it as a reminder that after the proof is complete, we should *disable* the definition of `reverse` so that the equality can be used as a left-to-right rewrite rule by ACL2's inside-out rewriter. If `reverse` were left enabled, the rewriter would first replace a term of the form `(len (reverse x))` by expanding the definition of `reverse`, after which that term would no longer match `(len (reverse x))`.

```
This simplifies, using the :definition REVERSE, to

Goal'
(IMPLIES (TRUE-LISTP X)
         (EQUAL (LEN (REVAPPEND X NIL))
                (LEN X))).

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  Subsumption reduces that number to two.
These merge into one derived induction scheme.
```

The proof ultimately fails. But since `Goal'` above did not simplify, we will take a look at it in a moment. As an aside, notice that "`using the :definition REVERSE`" the prover has replaced the original call of `reverse` with a call of `revappend`, which makes sense if we use a "print event" utility:

```
ACL2 !>:pe reverse
 V     -477  (DEFUN REVERSE (X)
                    "Documentation available via :doc"
                    (DECLARE (XARGS :GUARD (OR (TRUE-LISTP X) (STRINGP X))))
                    (COND ((STRINGP X)
                           (COERCE (REVAPPEND (COERCE X 'LIST) NIL)
                                   'STRING))
                          (T (REVAPPEND X NIL))))
```

Of course, if `reverse` hadn't been built in, we could have defined it exactly as shown above, using the `defun` command.

Looking again at `Goal'`, we realize that a suitable rewrite rule could simplify the term `(LEN (REVAPPEND X NIL))`. First let us look at the definition of `revappend`, this time using a "print formula" query.

```
ACL2 !>:pf revappend
(EQUAL (REVAPPEND X Y)
       (IF (CONSP X)
           (REVAPPEND (CDR X) (CONS (CAR X) Y))
           Y))
ACL2 !>
```

Thus, if we don't know it already, we now see that `(revappend x y)` pushes successive elements of `x` onto `y`. The following lemma, aptly named `len-revappend`, says that the length of a `revappend` call is the sum of the lengths of the arguments. ACL2 proves this by induction automatically (in less than 1/100 second).[3]

[3]The reader may notice that the lemma `len-revappend` is about `(revappend x y)` rather than the original term, `(revappend x nil)`. We have generalized by hand to produce a lemma whose proof seems amenable to induction. In this talk we do not consider research in performing such generalizations automatically; in ACL2 as it is today, the user is responsible for such generalization, though occasionally ACL2 makes useful generalizations on the fly.

```
(defthm len-revappend
  (equal (len (revappend x y))
         (+ (len x) (len y))))
```

Theorems are stored by default as (conditional) rewrite rules; so now, any instance of `(len (revappend x y))` encountered during a proof will be replaced by the corresponding instance of `(+ (len x) (len y))`. Thus, our original theorem, `len-reverse`, is now proved automatically and immediately.

To a first approximation, it's fair to say that ACL2 users work as illustrated above. That is, they prove collections of rewrite rules, discovering missing rules by looking at output from the prover. Our intention is that users can make good progress with the system without having to understand automated reasoning structures and concepts, as might be necessary in order to program tactics in tactic-based provers or set parameters in resolution-based provers.

2.2 Summary of some useful ACL2 features

As we wait for our food, we'll take a very brief look at a smattering of ACL2 features that we'll see while devouring the Main Course. Occasionally these features interact in unexpected ways, leading to maintenance tasks. Many improvements in these and other features are the direct result of user requests, which are very important to the evolution of the system.

There is no intention here to be complete. ACL2 is a large system not to be explored thoroughly over the course of a meal! Our goal is just to give a sense of the kind of support provided for one empirically successful automated reasoning system.[4]

- *Release Notes* consist of brief notes alerting the experienced user to important differences between one release and the next. They make frequent citations into the documentation (below) and are not intended to be self-contained or to be read by the new user.

- *Documentation* consists of over 1000 topics organized hierarchically. The documentation source consists of strings in the ACL2 source code that are liberally sprinkled with hyperlink annotations, which is processed to create HTML, Emacs Info, and (generally not used) printed views. The documentation is extensive; for example, the HTML version of the Version 3.0 documentation is about 3.3 megabytes. Documentation for a new release is intended to be complete and accurate for that release; for example, the HTML has grown about 300 kilobytes since Version 2.9, released less than two years ago. The documentation sometimes takes about as much time to write as the code to implement a feature or change, but our impression from users is that it's worth the effort.

- *Error messages* and *warnings* provide critical feedback, often pointing to documentation topics. We put a lot of care into these!

[4] In order to save time, during the talk we'll run through these very quickly, just to give a sense of what is coming.

- *Macros* make it easy to extend the syntax, but they have limitations (addressed by a new feature, `make-event`).
- A *book* is a collection of legal *embedded event forms* (*events*), in particular definitions (`defun` events) and theorems (`defthm` events), that have been *admitted*: syntax has been checked, theorems have been proved, and termination has been proved for recursive definitions.
  - *Certification* of a book creates a *certificate* witnessing the successful processing of the book.
  - The command `(include-book "foo")` will load events from `foo.lisp` into the current session.
  - However, `local` events, e.g., `(local (defun foo ...))`, are not exported by `include-book`. A logical story [KM01] involving conservativity justifies the dropping of `local` events.
  - About 850 books in about 70 directories, mostly contributed by users rather than the developers, are distributed with ACL2, with over 700 more in over 200 directories available from supporting materials for the first five ACL2 workshops (not including the one this year, 2006). Thus there are over 1500 books in our regression suite. We rely heavily on that test suite to test purported improvements to the prover's heuristics.
- Like books, `encapsulate` provides a modular structuring mechanism. `Encapsulate` events can be used to provide partial definitions for functions: that is, functions are total but may have incomplete axiomatizations.
- The `defevaluator` macro generates events that define an *evaluator*, against which one can prove *meta-rules* [BM81, HKK+05] that, in essence, augment the simplifier with formally verified user-defined functions.
- *Proof control* includes `in-theory` events and *hints*, which *disable* (turn off) or *enable* (turn on) specified rules. Supported are not only `in-theory` hints but others, for example directing induction, function expansion, or the use of previously-proved theorems. These can be attached to specific named goals or can be generated by code ("computed hints"), which can be specified globally ("default hints").
- *Database control* includes undo and undo-the-undo (`oops`) commands.
- An interactive *proof-checker* loop is a goal manager that has the feel of tactic-based prover interfaces, allowing a range of commands, from individual rewrites to calls of the full prover.
- *Proof debug* is supported by the above-mentioned proof-checker and also by a utility for inspecting apparent rewriter loops, a `break-rewrite` debugger for the rewriter, and *proof-tree* displays for navigating proof output.
- A *top-level read-eval-print loop* allows for interactive testing of one's functions. Such testing is typically relatively slow unless one issues a compilation command.

- Efficient *execution* is supported for ground terms not only in the top-level loop, but also during proofs. Efficient execution also relies on single-threaded objects [BM02], or *stobjs*, including the ACL2 `state` objects.
- *Guards* provide a powerful, flexible analogue of types, and help support efficient execution by way of a connection to the underlying Common Lisp. The `mbe` ("must be equal") feature allows one to attach efficient code to logically elegant functions [GKM+].
- Lisp *packages* provide namespaces.
- While the main *proof technique* is conditional rewriting, there are certainly others (for example, integrated decision procedures for ground equality and linear arithmetic). And, rewriting is actually congruence-based, i.e., can be used to replace a term with one that is suitably equivalent even if not actually equal.
- A *functional instantiation* utility [BGKM91, KM01] allows deriving a theorem $\varphi(g)$ from a corresponding theorem $\varphi(f)$ provided the function g satisfies all constraints on the function f.

3 Main Course — A selection of recent enhancements to ACL2

We now present a selection of items from recent ACL2 release notes, annotated with explanations and discussion about implications for system maintenance. We introduce each item very briefly, then display the Emacs Info version of the relevant release note, and finally explain the issues if necessary.

3.1 Subgoal counting

This item illustrates the effort we put into prover output. Here, "`:functional-instance`" refers to ACL2's functional instantiation utility, mentioned above; but the main point here is about output format, not functional instantiation.

```
.....................................................................
  Fixed a bug that was causing proof output on behalf of
  :functional-instance to be confusing, because it failed to mention that
  the number of constraints may be different from the number of subgoals
  generated.  Thanks to Robert Krug for pointing out this confusing
  output.  The fix also causes the reporting of rules used when silently
  simplifying the constraints to create the subgoals.
.....................................................................
```

Here is output from a proof attempt using ACL2 Version 2.9.3 that illustrates the problem. Notice that "six constraints" doesn't match up with the subgoal numbering, which counts down from 5 to 1. (We count down to give the user a real-time sense of how much work remains as the output scrolls by.) The old output was confusing, and thus potentially undermined the user's confidence in his understanding of what ACL2 is doing and in his belief in ACL2's correctness.

```
We now augment the goal above by adding the hypothesis indicated by
the :USE hint.  This produces a propositional tautology.  The hypothesis
can be derived from AC-FN-LIST-REV via functional instantiation, provided
we can establish the six constraints generated.

Subgoal 5
(EQUAL (TIMES-LIST X)
       (IF (ATOM X)
           1 (* (CAR X) (TIMES-LIST (CDR X))))).

But simplification reduces this to T, using the :definitions ATOM and
TIMES-LIST and primitive type reasoning.

Subgoal 4
....
```

Here is the corresponding output (suitably elided) from ACL2 3.0.

```
We now augment .... provided
we can establish the six constraints generated.  By the simple :rewrite
rules ASSOCIATIVITY-OF-* and UNICITY-OF-1 we reduce the six constraints
to five subgoals.
```

[...and so on, as before]

3.2 A rough edge in theory control

ACL2 uses evaluation as part of its proof strategy, but it allows the user to disable evaluation of calls of a function `f` by disabling the so-called *executable-counterpart* rule for `f`. For a particular type of conditional rule, a *forward-chaining* rule, evaluation of ground hypotheses had taken place without regard to which executable-counterpart rules are disabled, thus severely impacting efficiency in at least one user's experience.

```
.....................................................................
  Fixed a long-standing bug in forward-chaining, where variable-free
  hypotheses were being evaluated even if the executable-counterparts of
  their function symbols had been disabled.  Thanks to Eric Smith for
  bringing this bug to our attention by sending a simple example that
  exhibited the problem.
.....................................................................
```

3.3 Prover heuristic tweaks

Sometimes we find improvements to ACL2's prover heuristics. All three items below describe changes that were carefully made in response to user feedback, and tested with our regression suite to gain confidence that our heuristic changes would not severely impact users. These changes are only necessary because ACL2 attempts to provide significant automation.

```
.....................................................................
  We fixed an infinite loop that could occur during destructor elimination
```

```
  (see *Note ELIM::).  Thanks to Sol Swords for bringing this to our
  attention and sending a nice example, and to Doug Harper for sending a
  second example that we also found useful.
.....................................................................
  The simplifier has been changed slightly in order to avoid using
  forward-chaining facts derived from a literal (essentially, a top-level
  hypothesis or conclusion) that has been rewritten.  As a practical
  matter, this may mean that the user should not expect forward-chaining
  to take place on a term that can be rewritten for any reason (generally
  function expansion or application of rewrite rules).  Formerly, the
  restriction was less severe: forward-chaining facts from a hypothesis
  could be used as long as the hypothesis was not rewritten to t.  Thanks
  to Art Flatau for providing an example that led us to make this change;
  see the comments in source function rewrite-clause for details.
.....................................................................
  We modified the rewriter to avoid certain infinite loops caused by an
  interaction of the opening of recursive functions with equality
  reasoning.  (This change is documented in detail in the source code, in
  particular functions rewrite-fncall and fnstack-term-member.)  Thanks to
  Fares Fraij for sending us an example that led us to make this change.
.....................................................................
```

There are over 36,000 lines of comments in the source code, some of which survived multiple translations from the earliest version of the Boyer-Moore system. The comments are largely intended to be a record, for the implementors, of why things are the way they are. This is important in a software project of 35 years duration. Sometimes the comments show how we used to do something and why and when we changed it. The comments also sometimes contain interesting examples and counterexamples illustrating supposed properties of the code. Despite the original intention of the implementors to use comments as a way of recording the design decisions and history, many ACL2 users read the source code. Since ACL2 is written in ACL2, this is straightforward and sort of represents a second, more detailed, level of documentation.

3.4 A library improvement using `MBE`

The following release note item illustrates one maintenance aspect: we update the distributed books (libraries of definitions and proved theorems), often in consultation with users.

```
.....................................................................
  Several interesting new definitions and lemmas have been added to the
  rtl library developed at AMD, and incorporated into books/rtl/rel4/lib/.
  Other book changes include a change to lemma truncate-rem-elim in
  books/ihs/quotient-remainder-lemmas.lisp, as suggested by Jared Davis.
.....................................................................
```

But buried in this item is a change that we find particularly interesting.

We mentioned *guards* earlier as a flexible analogue of types, and we mentioned `mbe` as a way to attach executable counterparts efficiently.

At AMD, we found a need for more efficient execution of bit-vector operations. Through Version 2.9.1, the *rtl library*, `books/rtl/rel4/lib/`, contained the following definition of the bit-slice operation that returns bits `i` down to `j` of a natural number `x`. (Here, `defund` is a define-then-disable command, implemented in response to a user's request.)

```
(defund bits (x i j)
  (declare (xargs :guard (rationalp x)))
  (if (or (not (integerp i))
          (not (integerp j)))
      0
    (fl (/ (mod x (expt 2 (1+ i))) (expt 2 j)))))
```

However, we found this definition in terms of floor, modulo, and exponentiation operations painfully slow to execute. We really wanted a definition that uses bitwise-and and shift operations instead:

```
(defund bits (x i j)
  (if (< i j)
      0
    (logand (ash x (- j)) (1- (ash 1 (1+ (- i j)))))))
```

Fortunately, we were able to change the definition of `bits` for purposes of execution without changing its logical definition, which saved us from having to rework our proofs of any lemmas! The `mbe` ("must be equal") call below says to use the form after `:logic` as the body, with a proof obligation that the `:guard` (that `x`, `i`, and `j` are natural numbers) implies the equality of the `:logic` and `:exec` forms. The `:guard` must also imply certain formulas generated for the calls in the `:exec` form; for example the call `(ash x (- j))` carries a guard-related obligation that `x` and `(- j)` be integers, which is trivial from the guard assumptions that `x` and `j` are natural numbers. Then calls of `bits` on natural numbers will be executed directly in Common Lisp using the `:exec` form as the definition.

```
(defund bits (x i j)
  (declare (xargs :guard (and (natp x)
                              (natp i)
                              (natp j))))
  (mbe :logic (if (or (not (integerp i))
                      (not (integerp j)))
                  0
                (fl (/ (mod x (expt 2 (1+ i))) (expt 2 j))))
       :exec  (if (< i j)
                  0
                (logand (ash x (- j)) (1- (ash 1 (1+ (- i j))))))))
```

3.5 Some convenience features

The following three items all make life easier for the user, as we explain below each one.

```
.....................................................................
  Improved cw-gstack to allow a :frames argument to specify a range of one
  or more frames to be printed.  See *Note CW-GSTACK::.
.....................................................................
```

ACL2 makes very few restrictions on how users introduce rewrite rules to program the rewriter. This freedom, however, makes it possible to introduce infinite loops. When that occurs, ACL2 aborts cleanly (a major advance starting with Version 2.8 — previously it sometimes seg faulted!) and suggests use of the tool `cw-gstack`, which shows the rewrite stack. Unfortunately, the entire rewrite stack is large, so there was interest in being able to limit the number of frames printed.

```
.....................................................................
  A new event, set-enforce-redundancy, enforces a restriction that all
  defthms, defuns, and most other events are redundant.  See *Note
  SET-ENFORCE-REDUNDANCY::.
.....................................................................
```

AMD's rtl library (mentioned above) employed a methodology in which the proof work was restricted to books in an auxiliary directory. It seemed desirable to enforce this methodology, so that the main directory was kept clean and the auxiliary directory could be modified as desired. Here is how that works.

Suppose we have a file `top.lisp` that we want to certify as a book.

```
(local (include-book "work/book-1"))
(defthm result-1 ...)
...
```

Here, imagine that `result-1` is proved in file `work/book-1.lisp`. The `local` annotation guarantees that additional theorems proved in `work/book-1.lisp` will ultimately disappear, except for `result-1`, which (as seen above) we have made explicit. When we are certifying the present book, we expect that `result-1` will be *redundant* because it already appears in `work/book-1.lisp`. But suppose we accidentally delete `result-1` in `work/book-1.lisp`. ACL2 would then try to prove `result-1`, but we may prefer that ACL2 instead fail immediately with a clear complaint that it didn't find that `result-1` has already been proved.

The item above provides a solution. We simply start `top.lisp` with the form `(set-enforce-redundancy t)`.

One thing we've found is that nothing is ever simple! So for example, certain kinds of events called `deflabel` events are not allowed to be redundant. So even with `set-enforce-redundancy`, we need to allow non-redundant `deflabel` events.

```
.....................................................................
  The function disabledp can now be given a macro name that has a
  corresponding function; see *Note MACRO-ALIASES-TABLE::.  Also,
  disabledp now has a guard of t but causes a hard error on an
  inappropriate argument.
.....................................................................
```

For example, in ACL2 `append` is a macro, because functions must take a fixed number of arguments but we want to be able to apply `append` to an arbitrary number of arguments. We can see how this works by using ACL2's `:trans1` command to perform a single-step macroexpansion.

```
ACL2 !>:trans1 (append x y z)
 (BINARY-APPEND X (BINARY-APPEND Y Z))
ACL2 !>
```

ACL2, however, is kind enough to print terms using `append` rather than using the corresponding function, `binary-append`. Thus, novice users might not even realize that `append` is not a function.

ACL2 has a notion of *table* events that allows maintenance of information of interest, and once such table associates `append` with `binary-append`. The user then may refer to `append` in contexts where a function symbol is expected, for example when disabling a definition, for example:

```
(in-theory (disable append))
```

```
.....................................................................
  The macro comp is now an event, so it may be placed in books.
.....................................................................
```

The above item simply allows a compilation directive to be placed in books. It's a simple thing to provide and we wish we had done it sooner in order to save users some annoyance!

3.6 Common Lisp compatibility: Packages

Namespace control is provided by Common Lisp *packages*. Each *symbol* is in essence a pair of strings: a package name and a symbol name. But getting this exactly right is quite tricky. A rather elaborate fix was made in Version 2.8, not shown here, to deal with an unsoundness that could result from a subtle use, carefully employing `local`, of two different packages with the same name. (See ACL2's documentation topic "hidden-death-package" if you want to learn more about this issue. And it points to a very elaborate comment in the source code that gives even more of an idea of how nasty this issue really is.)

Below are three package issues solved more recently than that one, and not nearly as complex. They show how we sometimes need to work hard to ensure compatibility with the host Common Lisp.

```
.....................................................................
  We fixed a soundness hole due to the fact that the "LISP" package does
  not exist in OpenMCL.  We now explicitly disallow this package name as
  an argument to defpkg.  Thanks to Bob Boyer and Warren Hunt for bringing
  an issue to our attention that led to this fix.
.....................................................................
  ACL2 now requires all package names to consist of standard characters
  (see *Note STANDARD-CHAR-P::, none of which is lower case.  The reason
```

```
  is that we have seen at least one lisp implementation that does not
  handle lower case package names correctly.  Consider for example the
  following raw lisp log (some newlines omitted).

       >(make-package "foo")
       #<"foo" package>
       >(package-name (symbol-package 'FOO::A))
       "foo"
       >(package-name (symbol-package '|FOO|::A))
       "foo"
       >
.....................................................................
  (GCL only) A bug in symbol-package-name has been fixed that could be
  exploited to prove nil, and hence is a soundness bug.  Thanks to Dave
  Greve for sending us an example of a problem with defcong (see below)
  that led us to this discovery.
.....................................................................
```

3.7 Portability, and help from others

ACL2 can be built on most (all?) stable Common Lisp implementations, including GCL, OpenMCL, Allegro CL, SBCL, CMUCL, CLISP, and Lispworks. The most recent addition is SBCL. There are at least two reasons for porting to all of these Lisps, in spite of a certain amount of low-level Lisp-specific code we need to write and maintain. One is that we sometimes find bugs in our code that are in some sense "forgiven" by most, but not all, Lisps. The other is that we want users to be able to build on whatever Lisp platform they happen to have. Perhaps a third reason is to support each Lisp's development by providing a non-trivial test suite.

```
.....................................................................
  Added SBCL support.  Thanks to Juho Snellman for significant assistance
  with the port.  Thanks to Bob Boyer for suggesting the use of feature
  :acl2-mv-as-values with SBCL, which can allow thread-level parallelism
  in the underlying lisp; we have done so when feature :sb-thread is
  present.
.....................................................................
```

3.8 User-level debug support

ACL2 has a *break-rewrite* utility that allows the user to put a breakpoint upon the application of a specified rewrite rule, optionally under specified conditions. The situation becomes complicated when there are so-called *free variables in hypotheses*. For example, consider the conditional rewrite rule saying that if predicate `p2` holds of `x` and `y`, and predicate `p3` holds of `y`, then predicate `p1` holds of `x`:

```
(implies (and (p2 x y)
              (p3 y))
         (equal (p1 x) t))
```

Now suppose the rewriter encounters the term `(p1 (foo a))`. So, `x` is bound to `(foo a)` when we apply the above rule. But how can we rewrite the first hypothesis (to *true*) if we do not have a binding for the *free variable* `y`?

In this case, ACL2 simply looks in its current context for some term α for which `(p1 (foo a)` α`)` is known to be true. When it finds such an α then it binds `y` to α and goes on to the next hypothesis. So it will now be "thinking about" `(p3` α`)`. If the rewriter cannot prove this is true, it will backtrack and look for another value of `y` in place of α for the first hypothesis.

The above information can be critical to a user who is trying to understand why a rule is failing to be applied, especially when there is a complex set of available rules operating on the hypotheses. The following item describes an improvement that provides convenient display of such information.

```
.....................................................................
  Improved reporting by the break-rewrite utility upon failure to relieve
  hypotheses in the presence of free variables, so that information is
  shown about the attempting bindings.  See *Note
  FREE-VARIABLES-EXAMPLES-REWRITE::.  Thanks to Eric Smith for requesting
  this improvement.  Also improved the break-rewrite loop so that terms,
  in particular from unifying substitutions, are printed without hiding
  subterms by default.  The user can control such hiding ("evisceration");
  see *Note SET-BRR-TERM-EVISC-TUPLE::.
.....................................................................
```

The ACL2 documentation topic "`free-variables-examples-rewrite`" describes how all this works. We'll just show a piece of that documentation here in order to give a visual cue of what we provide.

```
(1 Breaking (:REWRITE LEMMA-1) on (PROP U0):
1 ACL2 >:eval

1x (:REWRITE LEMMA-1) failed because :HYP 1 contains free variables.
The following display summarizes the attempts to relieve hypotheses
by binding free variables; see :DOC free-variables and see :DOC set-
brr-term-evisc-tuple.

    [1] X : X1
Failed because :HYP 3 contains free variables Y and Z, for which no
suitable bindings were found.
    [1] X : X2
Failed because :HYP 2 rewrote to (BAD X2).
    [1] X : X3
        [3] Z : Z1
            Y : Y1
Failed because :HYP 6 rewrote to (FOO X3 Y1).
        [3] Z : Z1
            Y : Y3
Failed because :HYP 6 rewrote to (FOO X3 Y3).

1 ACL2 >
```

3.9 Some other release note items of interest

- Several bugs have been fixed that are related to `local`. It seems somewhat difficult to anticipate all interactions of other aspects of the system and logic with `local`.

- Two very different kinds of hints for `defthm` events are generally incompatible: `:hints` to direct the automatic prover, and `:instructions` to direct the replay of commands saved during a session with the *proof-checker*, an interactive goal-directed proof management tool. We quite sensibly caused an error if both `:hints` and `:instructions` were present for the same `defthm` event. But we added a notion of default hints without noticing that we needed to allow them with `:instructions`, in which case the default hints should apply to any individual instruction that calls the full prover. (This has been fixed.)

- Users can undo events and they can even undo the undo. But some heavy users are hitting memory limitations, so we now provide the option of trading the "undo the undo" capability with the reclamation of space.

- A feature new to Version 3.0, of excitement to some experienced ACL2 users, is a capability, `make-event`, that is similar to macros but which is sensitive to the environment (e.g., the ACL2 `state` object). The main idea is that expansions that might otherwise depend on the environment, which is illegal for macros,[5] are saved in the book's certificate. But there were lots of complications to solve (for example, what if the `make-event` is submitted interactively before certification is begun).

- Users can specify a limit on backchaining through rewrite rules, and they can specify syntactic checks to control the application of a rewrite rule [HKK+05]. But until a user requested it, these features were not available with conditional meta-rules.

- ACL2 supports rewriting with congruences, where the original and rewritten term are equivalent but not necessarily equal. ACL2 also caches rewrite results, for efficiency. There are occasions when the cached result is from an equality rewrite, but we need to rewrite with an equivalence, which could produce a stronger result. If we always ignore the cache in such cases, efficiency becomes a problem. But after receiving a user request, we instituted a compromise where we give special handling in some cases when the equivalence relation is Boolean equivalence. More recently [KM06], we have provided the user a means to handle this situation for other equivalence relations, together with warnings that bring this situation to the user's attention.

[5] It would take us too far afield to explain in detail why it is illegal for macros to depend on the current state. But it's not hard to imagine that otherwise, a macro might expand to give one definition of a function as a book is certified, but a different definition of the same function when the book is later included. Besides, ACL2 compiles its books, and the Common Lisp specification disallows dependence of macros on the state.

4 Dessert

I intend to leave time for audience members to share related observations from their own experiences, and to ask further questions.

Acknowledgments

We thank Robert Krug and Sandip Ray for useful comments on a draft of this paper. This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591.

References

[BGKM91] R.S. Boyer, D.M. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BM97] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.

[BM02] R. S. Boyer and J S. Moore. Single-threaded objects in ACL2. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *PADL 2002, LNCS 2257*, pages 9–27, 2002.

[GKM+] D. A. Greve, M. Kaufmann, P. Manolios, J S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. Submitted.

[gpl] `http://www.gnu.org/copyleft/gpl.html`.

[HKK+05] W. A. Hunt, Jr, M. Kaufmann, R. B. Krug, J S. Moore, and E. W. Smith. Meta reasoning in ACL2. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*. Springer, August 2005.

[Kau91] M. Kaufmann. An informal discussion of issues in mechanically-assisted reasoning. In M. Archer, J. J. Joyce, K. N. Levitt, and P. H. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 318–337, Los Alamitos, CA, 1991. IEEE Computer Society Press.

[KM01] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[KM04a] 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, 2004.

[KM04b] M. Kaufmann and J S. Moore. How to prove theorems formally. In *http://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/*. Department of Computer Sciences, University of Texas at Austin, 2004.

[KM06] M. Kaufmann and J S. Moore. Double rewriting for equivalential reasoning in ACL2. In *Proceedings of ACL2 Workshop 2006*, August 2006.

[KMM00a] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.