Maintaining the ACL2 Theorem Proving System

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

ESCoR Workshop, FLoC, Seattle, Aug 21, 2006

Introduction

Topic of this talk:

What sorts of challenges do we face in making our theorem prover useful in practice?

- Automated reasoning talks often focus on topics such as algorithms, logics, and applications.
- This talk will focus on the **pragmatics** of **maintenance**.

For more information:

- See the full paper.
- Try out ACL2 (with assistance of documentation and `acl2-help` email list): `http://www.cs.utexas.edu/users/moore/acl2/`.

Introduction

Topic of this talk:

What sorts of challenges do we face in making our theorem prover useful in practice?

- Automated reasoning talks often focus on topics such as algorithms, logics, and applications.
- This talk will focus on the **pragmatics** of **maintenance**.

For more information:

- See the full paper.
- Try out ACL2 (with assistance of documentation and `acl2-help` email list): `http://www.cs.utexas.edu/users/moore/acl2/`.

Introduction

Topic of this talk:

What sorts of challenges do we face in making our theorem prover useful in practice?

- Automated reasoning talks often focus on topics such as algorithms, logics, and applications.
- This talk will focus on the **pragmatics** of **maintenance**.

For more information:

- See the full paper.
- Try out ACL2 (with assistance of documentation and `acl2-help` email list): `http://www.cs.utexas.edu/users/moore/acl2/`.

Outline

Imagine that we're going to lunch to have a chat about what we're doing to make automated reasoning useful in practice.

- Before We Eat:
  Some general background on ACL2
- Main Course:
  A selection of recent enhancements to ACL2
- Dessert:
  Discussion

I invite questions and comments throughout the talk, not only during dessert. In particular, it would be interesting to compare experiences in maintaining automated reasoning systems.

Outline

Imagine that we're going to lunch to have a chat about what we're doing to make automated reasoning useful in practice.

- Before We Eat:
  Some general background on ACL2
- Main Course:
  A selection of recent enhancements to ACL2
- Dessert:
  Discussion

I invite questions and comments throughout the talk, not only during dessert. In particular, it would be interesting to compare experiences in maintaining automated reasoning systems.

Outline

Imagine that we're going to lunch to have a chat about what we're doing to make automated reasoning useful in practice.

- Before We Eat:
  Some general background on ACL2
- Main Course:
  A selection of recent enhancements to ACL2
- Dessert:
  Discussion

I invite questions and comments throughout the talk, not only during dessert. In particular, it would be interesting to compare experiences in maintaining automated reasoning systems.

Outline

Imagine that we're going to lunch to have a chat about what we're doing to make automated reasoning useful in practice.

- Before We Eat:
  Some general background on ACL2
- Main Course:
  A selection of recent enhancements to ACL2
- Dessert:
  Discussion

I invite questions and comments throughout the talk, not only during dessert. In particular, it would be interesting to compare experiences in maintaining automated reasoning systems.

Outline

Imagine that we're going to lunch to have a chat about what we're doing to make automated reasoning useful in practice.

- Before We Eat:
  Some general background on ACL2
- Main Course:
  A selection of recent enhancements to ACL2
- Dessert:
  Discussion

I invite questions and comments throughout the talk, not only during dessert. In particular, it would be interesting to compare experiences in maintaining automated reasoning systems.

Before We Eat:
Some general background on ACL2

- Introduction to ACL2
- Some milestones
- The user's view of ACL2: A small example
- Summary of some useful ACL2 features

Introduction to ACL2

- ACL2 (ACL^2 = ACLACL): A Computational Logic for Applicative Common Lisp
- Programmed primarily in itself: forces attention to sufficient language features and efficiency; functional language facilitates maintenance (vs. Nqthm, e.g.)
- Has evolved with user feedback — to see applications, follow “Books and Papers” link from ACL2 home page and then follow “Quick Summary” link
- Source files total 8.4M (Version 3.0.1)
- 256 release note items strictly after March, 2004 release (2.8); much more waiting on the “to do” list

Introduction to ACL2

- ACL2 (ACL^2 = ACLACL): A Computational Logic for Applicative Common Lisp
- Programmed primarily in itself: forces attention to sufficient language features and efficiency; functional language facilitates maintenance (vs. Nqthm, e.g.)
- Has evolved with user feedback — to see applications, follow "Books and Papers" link from ACL2 home page and then follow "Quick Summary" link
- Source files total 8.4M (Version 3.0.1)
- 256 release note items strictly after March, 2004 release (2.8); much more waiting on the "to do" list

Introduction to ACL2

- ACL2 (ACL^2 = ACLACL): A Computational Logic for Applicative Common Lisp
- Programmed primarily in itself: forces attention to sufficient language features and efficiency; functional language facilitates maintenance (vs. Nqthm, e.g.)
- Has evolved with user feedback — to see applications, follow "Books and Papers" link from ACL2 home page and then follow "Quick Summary" link
- Source files total 8.4M (Version 3.0.1)
- 256 release note items strictly after March, 2004 release (2.8); much more waiting on the "to do" list

Introduction to ACL2

- ACL2 (ACL^2 = ACLACL): A Computational Logic for Applicative Common Lisp
- Programmed primarily in itself: forces attention to sufficient language features and efficiency; functional language facilitates maintenance (vs. Nqthm, e.g.)
- Has evolved with user feedback — to see applications, follow “Books and Papers” link from ACL2 home page and then follow “Quick Summary” link
- Source files total 8.4M (Version 3.0.1)
- 256 release note items strictly after March, 2004 release (2.8); much more waiting on the “to do” list

Introduction to ACL2

- ACL2 (ACL^2 = ACLACL): A Computational Logic for Applicative Common Lisp
- Programmed primarily in itself: forces attention to sufficient language features and efficiency; functional language facilitates maintenance (vs. Nqthm, e.g.)
- Has evolved with user feedback — to see applications, follow “Books and Papers” link from ACL2 home page and then follow “Quick Summary” link
- Source files total 8.4M (Version 3.0.1)
- 256 release note items strictly after March, 2004 release (2.8); much more waiting on the “to do” list

Some milestones

- 1971-73: Boyer/Moore "Edinburgh Pure Lisp Theorem Prover"
- 1979: Boyer and Moore, *A Computational Logic*
- 1986: Kaufmann joins Boyer/Moore project
- 1988: Boyer and Moore, *A Computational Logic Handbook*
- 1989: Boyer and Moore begin ACL2 (but continue to maintain Nqthm)
- 1992: Final release of Boyer-Moore "Nqthm" prover
- 1993: Kaufmann formally added as a co-author of ACL2
- 1999: First of six (so far) ACL2 workshops
- 2000: *Computer-Aided Reasoning: An Approach* published
- 2006: Boyer, Kaufmann, and Moore win 2005 ACM Software System Award for Boyer-Moore family of provers

The user's view of ACL2: A small example

Example: the length is unchanged when a list is reversed.

But first, a summary of ACL2 interaction (thanks, Robert Krug):

- ACL2 is entirely automatic once you start it.
- It has considerable built-in knowledge about predicate logic, linear arithmetic, equality reasoning, etc. — but this is rarely enough.
- Users can examine output of failed proofs to learn what needs to be done to succeed.
- In the example below, we see the typical activity of creating and proving a rewrite rule based on that output.
- Such helper lemmas, when well designed, can be reused automatically.
- There is a large body of proved theorems ("books") distributed with ACL2, containing just such rules.

The user's view of ACL2: A small example

Example: the length is unchanged when a list is reversed.
But first, a summary of ACL2 interaction (thanks, Robert Krug):

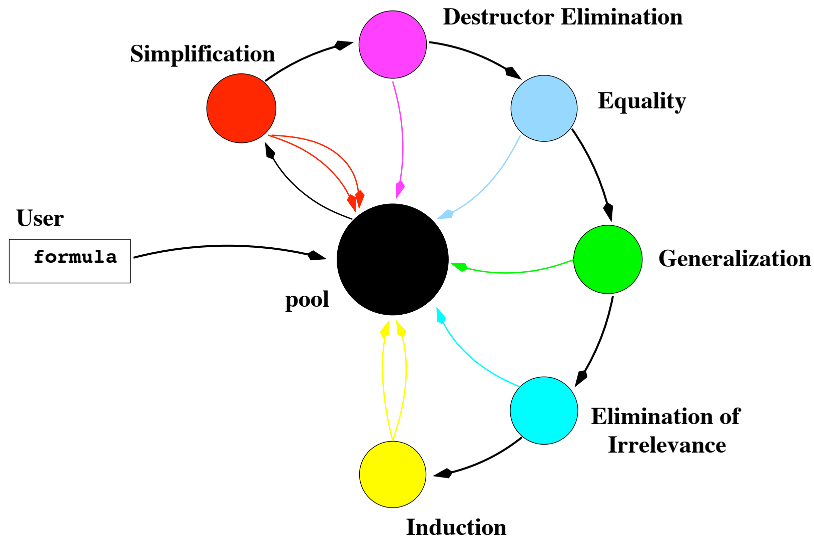
- ACL2 is entirely automatic once you start it.
- It has considerable built-in knowledge about predicate logic, linear arithmetic, equality reasoning, etc. — but this is rarely enough.
- Users can examine output of failed proofs to learn what needs to be done to succeed.
- In the example below, we see the typical activity of creating and proving a rewrite rule based on that output.
- Such helper lemmas, when well designed, can be reused automatically.
- There is a large body of proved theorems ("books") distributed with ACL2, containing just such rules.

The user's view of ACL2: A small example

Example: the length is unchanged when a list is reversed.
But first, a summary of ACL2 interaction (thanks, Robert Krug):

- ACL2 is entirely automatic once you start it.
- It has considerable built-in knowledge about predicate logic, linear arithmetic, equality reasoning, etc. — but this is rarely enough.
- Users can examine output of failed proofs to learn what needs to be done to succeed.
- In the example below, we see the typical activity of creating and proving a rewrite rule based on that output.
- Such helper lemmas, when well designed, can be reused automatically.
- There is a large body of proved theorems ("books") distributed with ACL2, containing just such rules.

The Waterfall



I'll discuss this partial log:

```
ACL2 !>(defthm len-reverse
         (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.


This simplifies, using the :definition REVERSE, to the following two
conjectures.

Subgoal 2
(IMPLIES (STRINGP X)
         (EQUAL (LEN (COERCE (REVAPPEND (COERCE X 'LIST) NIL)
                             'STRING))
                (LEN X))).

But simplification reduces this to T, using the :definition LEN, the
:executable-counterpart of EQUAL and primitive type reasoning.

Subgoal 1
(IMPLIES (NOT (STRINGP X))
         (EQUAL (LEN (REVAPPEND X NIL))
                (LEN X))).

Name the formula above *1.

...............

******** FAILED ********  See :DOC failure  ******** FAILED ********
```

The failed goal:

```
(IMPLIES (NOT (STRINGP X))
         (EQUAL (LEN (REVAPPEND X NIL))
                (LEN X)))
```

With a little thought we submit this rewrite rule, which will complete the proof of `len-reverse`.

```
(DEFTHM LEN-REVAPPEND
  (EQUAL (LEN (REVAPPEND X Y))
         (+ (LEN X) (LEN Y))))
```

The failed goal:

```
(IMPLIES (NOT (STRINGP X))
         (EQUAL (LEN (REVAPPEND X NIL))
                (LEN X)))
```

With a little thought we submit this rewrite rule, which will complete the proof of `len-reverse`.

```
(DEFTHM LEN-REVAPPEND
  (EQUAL (LEN (REVAPPEND X Y))
         (+ (LEN X) (LEN Y))))
```

```
ACL2 !>(defthm len-revappend
         (equal (len (revappend x y))
                (+ (len x) (len y))))

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are suggested by
this conjecture.  These merge into two derived induction schemes.  However, one
of these is flawed and so we are left with one viable candidate.

...............

Time:  0.01 seconds (prove: 0.01, print: 0.00, other: 0.00)
 LEN-REVAPPEND
ACL2 !>(defthm len-reverse
         (equal (len (reverse x)) (len x)))

...............

Summary
Form:  ( DEFTHM LEN-REVERSE ...)
Rules: ((:DEFINITION LEN)

...............

        (:REWRITE LEN-REVAPPEND)
        (:TYPE-PRESCRIPTION LEN))
Warnings:  Non-rec
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 LEN-REVERSE
ACL2 !>
```

```
ACL2 !>:pl len

Rune:        (:REWRITE LEN-REVERSE)
Status:      Enabled
Lhs:         (LEN (REVERSE X))
Rhs:         (LEN X)
Hyps:        T
Equiv:       EQUAL
Backchain-limit-lst:    NIL
Subclass:   ABBREVIATION

Rune:        (:REWRITE LEN-REVAPPEND)
Status:      Enabled

...............

ACL2 !>:pl (len (reverse (cons a b)))

1. LEN-REVERSE
  New term: (LEN (CONS A B))
  Hypotheses: <none>
  Equiv: EQUAL
  Substitution: ((X CONS A B))
```

Summary of some useful ACL2 features (1)

Note: This partial list is intended to set the stage for our discussion of maintenance.

- **Documentation**: over 1000 topics, organized hierarchically with hyperlinks (or 1200+ pages)
- **Error messages** and **warnings**
- **Macros**, and more via `make-event`
- **Top-level read-eval-print loop** provides interactive testing
- Efficient **evaluation** of ground terms in top-level loop and proofs, via guards, compilation, single-threaded objects, and `mbe` ("must be equal")
- **Proof techniques**: congruence-based conditional rewriting, many others
- **Namespaces** via Lisp packages

Summary of some useful ACL2 features (2)

- **Books** (files) of **events** (top-level forms): definitions (`defun`), and theorems (`defthm`), etc.
  - **Certification** and subsequent **`include-book`**
  - **`Local`** events present a logical and maintenance challenge but are very useful for scoping
  - About 1600 books in over 270 directories, contributed by many; useful for regression testing
- **`Encapsulate`** scoping mechanism also provides modularity; and it provides partial definitions
- A **functional instantiation** utility for a kind of "second-order" reasoning

Summary of some useful ACL2 features (3)

- **User-installed simplifiers**: meta-rules
- **Proof control**: rule classes, hints (explicit, computed, default), theories
- **Database control**: undo and undo-the-undo commands
- Interactive **proof-checker**: a goal manager providing the feel of tactic-based proof assistants but with ACL2 automation available
- **Proof debug**: proof-checker (above), inspection utility for rewriter loops, `break-rewrite` rewrite debugger, and proof-tree display for proof log navigation

Main Course:
A selection of recent enhancements to ACL2

- In the next few slides, we'll discuss a variety of examples that illustrate a range of maintenance tasks, largely in response to user feedback.

- I'll keep the slides somewhat terse but provide verbal elaboration. Details may be found in the paper.

Subgoal counting

Ouch – “six constraints generated” yet only five subgoals!

```
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
....
```

After a fix:

```
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.
```

A rough edge in theory control

```
(include-book "ordinals/ordinals" :dir :system)
(defund fact (n)  ; defun, then disable
  (if (zp n) 1 (* n (fact (1- n)))))
(thm (equal (fact 65534) 0) ; silly example
     :hints
     (("Goal"
       :in-theory
       (disable (:executable-counterpart fact)))))
```

Stack overflow!! The problem: evaluation of `(O<= (fact 65534) 0)` in spite of the above `disable`, from the following forward-chaining rule:

```
(IMPLIES (AND (NOT (EQUAL A B))
              (O<= A B)
              (O-P A) (O-P B))
         (O< A B))
```

Solution: Evaluator that comprehends which rules are enabled.

A rough edge in theory control

```
(include-book "ordinals/ordinals" :dir :system)
(defund fact (n)  ; defun, then disable
  (if (zp n) 1 (* n (fact (1- n)))))
(thm (equal (fact 65534) 0) ; silly example
     :hints
     (("Goal"
       :in-theory
       (disable (:executable-counterpart fact)))))
```

Stack overflow!! The problem: evaluation of `(O<= (fact 65534) 0)` in spite of the above `disable`, from the following forward-chaining rule:

```
(IMPLIES (AND (NOT (EQUAL A B))
              (O<= A B)
              (O-P A) (O-P B))
         (O< A B))
```

Solution: Evaluator that comprehends which rules are enabled.

A rough edge in theory control

```
(include-book "ordinals/ordinals" :dir :system)
(defund fact (n)  ; defun, then disable
  (if (zp n) 1 (* n (fact (1- n)))))
(thm (equal (fact 65534) 0) ; silly example
     :hints
     (("Goal"
       :in-theory
       (disable (:executable-counterpart fact)))))
```

Stack overflow!! The problem: evaluation of `(O<= (fact 65534) 0)` in spite of the above `disable`, from the following forward-chaining rule:

```
(IMPLIES (AND (NOT (EQUAL A B))
              (O<= A B)
              (O-P A) (O-P B))
         (O< A B))
```

Solution: Evaluator that comprehends which rules are enabled.

Prover heuristics tweaks

These improvements came out of user feedback and were regression tested:

- Avoid certain infinite loops during *destructor elimination*
- Avoid forward-chaining from a rewritten term
- Avoid certain infinite loops due to interaction of equality reasoning with opening up of recursive functions
- Limit subsumption checks (to 1,000,000 matcher calls)

The above are all documented in the source code, which contains over 36,000 lines of comments (not including the applicative source code itself, which also serves as documentation!).

Prover heuristics tweaks

These improvements came out of user feedback and were regression tested:

- Avoid certain infinite loops during *destructor elimination*
- Avoid forward-chaining from a rewritten term
- Avoid certain infinite loops due to interaction of equality reasoning with opening up of recursive functions
- Limit subsumption checks (to 1,000,000 matcher calls)

The above are all documented in the source code, which contains over 36,000 lines of comments (not including the applicative source code itself, which also serves as documentation!).

Prover heuristics tweaks

These improvements came out of user feedback and were regression tested:

- Avoid certain infinite loops during *destructor elimination*
- Avoid forward-chaining from a rewritten term
- Avoid certain infinite loops due to interaction of equality reasoning with opening up of recursive functions
- Limit subsumption checks (to 1,000,000 matcher calls)

The above are all documented in the source code, which contains over 36,000 lines of comments (not including the applicative source code itself, which also serves as documentation!).

Prover heuristics tweaks

These improvements came out of user feedback and were regression tested:

- Avoid certain infinite loops during *destructor elimination*
- Avoid forward-chaining from a rewritten term
- Avoid certain infinite loops due to interaction of equality reasoning with opening up of recursive functions
- Limit subsumption checks (to 1,000,000 matcher calls)

The above are all documented in the source code, which contains over 36,000 lines of comments (not including the applicative source code itself, which also serves as documentation!).

Prover heuristics tweaks

These improvements came out of user feedback and were regression tested:

- Avoid certain infinite loops during *destructor elimination*
- Avoid forward-chaining from a rewritten term
- Avoid certain infinite loops due to interaction of equality reasoning with opening up of recursive functions
- Limit subsumption checks (to 1,000,000 matcher calls)

The above are all documented in the source code, which contains over 36,000 lines of comments (not including the applicative source code itself, which also serves as documentation!).

Prover heuristics tweaks

These improvements came out of user feedback and were regression tested:

- Avoid certain infinite loops during *destructor elimination*
- Avoid forward-chaining from a rewritten term
- Avoid certain infinite loops due to interaction of equality reasoning with opening up of recursive functions
- Limit subsumption checks (to 1,000,000 matcher calls)

The above are all documented in the source code, which contains over 36,000 lines of comments (not including the applicative source code itself, which also serves as documentation!).

A library improvement using `MBE` (1)

- Many distributed *books* undergo improvements
- One such set of books is known as the *rtl library*
- At AMD, we needed more efficient execution

A library improvement using MBE (2)

Old definition of `(bits x i j)` used floor, mod, and exponentiation:

```
(if (or (not (integerp i)) (not (integerp j)))
    0
  (fl (/ (mod x (expt 2 (1+ i)))
         (expt 2 j))))
```

Now, `bits` executes using bitwise-and and shift:

```
(mbe :logic [[as above]]
     :exec ; generates proof obligation
     (if (< i j)
         0
       (logand (ash x (- j))
               (1- (ash 1 (1+ (- i j))))))))
```

Avoided the need to modify existing proofs!

A library improvement using MBE (2)

Old definition of (bits x i j) used floor, mod, and exponentiation:

```
(if (or (not (integerp i)) (not (integerp j)))
    0
  (fl (/ (mod x (expt 2 (1+ i)))
         (expt 2 j))))
```

Now, bits executes using bitwise-and and shift:

```
(mbe :logic [[as above]]
     :exec ; generates proof obligation
     (if (< i j)
         0
       (logand (ash x (- j))
               (1- (ash 1 (1+ (- i j))))))))
```

Avoided the need to modify existing proofs!

A library improvement using `MBE` (2)

Old definition of `(bits x i j)` used floor, mod, and exponentiation:

```
(if (or (not (integerp i)) (not (integerp j)))
    0
  (fl (/ (mod x (expt 2 (1+ i)))
         (expt 2 j))))
```

Now, `bits` executes using bitwise-and and shift:

```
(mbe :logic [[as above]]
     :exec ; generates proof obligation
     (if (< i j)
         0
       (logand (ash x (- j))
               (1- (ash 1 (1+ (- i j))))))))
```

Avoided the need to modify existing proofs!

Some convenience features

- Rewriter debug command `cw-gstack` takes an argument for the number of frames to display
- `Set-enforce-redundancy` allows user to enforce a style of book management, where proofs are kept in subsidiary books
- `Disabledp`, like other commands, allows a macro to be an alias for a function
- `Compilation` is fully supported at the book level

Some convenience features

- Rewriter debug command `cw-gstack` takes an argument for the number of frames to display
- `Set-enforce-redundancy` allows user to enforce a style of book management, where proofs are kept in subsidiary books
- `Disabledp`, like other commands, allows a macro to be an alias for a function
- `Compilation` is fully supported at the book level

Some convenience features

- Rewriter debug command `cw-gstack` takes an argument for the number of frames to display
- `Set-enforce-redundancy` allows user to enforce a style of book management, where proofs are kept in subsidiary books
- `Disabledp`, like other commands, allows a macro to be an alias for a function
- `Compilation` is fully supported at the book level

Some convenience features

- Rewriter debug command `cw-gstack` takes an argument for the number of frames to display
- `Set-enforce-redundancy` allows user to enforce a style of book management, where proofs are kept in subsidiary books
- `Disabledp`, like other commands, allows a macro to be an alias for a function
- `Compilation` is fully supported at the book level

Some convenience features

- Rewriter debug command `cw-gstack` takes an argument for the number of frames to display
- `Set-enforce-redundancy` allows user to enforce a style of book management, where proofs are kept in subsidiary books
- `Disabledp`, like other commands, allows a macro to be an alias for a function
- `Compilation` is fully supported at the book level

Portability

We support all major Common Lisp implementations of which we are aware:

- Gnu Common Lisp (GCL)
- OpenMCL
- Allegro Common Lisp
- SBCL
- CMU Common LIsp
- CLISP
- Lispworks

Why support so many platforms?

- Catch bugs
- User choice (e.g., profilers vary)
- Support experimentation (e.g., parallelism in OpenMCL and SBCL (Rager), hash-cons in GCL and OpenMCL (Boyer/Hunt))

Namespace control

Lisp (and ACL2) *packages* provide namespace control: e.g., the same string `"ABC"` can name a symbol in two different packages. There have been several bugs involving packages:

- Disallow `"LISP"` package (exists already in some implementations, not others)
- Require uppercase package names — at least one Lisp gets this wrong
- Hand-coded function to execute `pkg-witness`: bug recently found in its default behavior for non-strings

Some other recent improvements

- Improved reporting in rewriter's debugger (`break-rewrite`) for free variables
- Several bugs related to `local` — presents a serious challenge to implementing logic!
- `Double-rewrite` utility to override the rewriter's caching of results in the presence of congruences (see recent ACL2 workshop paper)
- Miscellaneous bug fixes, e.g., in timing utility and handling of Lisp type specs (`satisfies`)
- Recent performance enhancements, especially for theories: Critical for users at Rockwell Collins
- Prover time limits
- Context recovery for failed proofs (`redo-flat`)

Some other recent improvements

- Improved reporting in rewriter's debugger (`break-rewrite`) for free variables
- Several bugs related to `local` — presents a serious challenge to implementing logic!
- `Double-rewrite` utility to override the rewriter's caching of results in the presence of congruences (see recent ACL2 workshop paper)
- Miscellaneous bug fixes, e.g., in timing utility and handling of Lisp type specs (`satisfies`)
- Recent performance enhancements, especially for theories: Critical for users at Rockwell Collins
- Prover time limits
- Context recovery for failed proofs (`redo-flat`)

Some other recent improvements

- Improved reporting in rewriter's debugger (`break-rewrite`) for free variables
- Several bugs related to `local` — presents a serious challenge to implementing logic!
- `Double-rewrite` utility to override the rewriter's caching of results in the presence of congruences (see recent ACL2 workshop paper)
- Miscellaneous bug fixes, e.g., in timing utility and handling of Lisp type specs (`satisfies`)
- Recent performance enhancements, especially for theories: Critical for users at Rockwell Collins
- Prover time limits
- Context recovery for failed proofs (`redo-flat`)

Some other recent improvements

- Improved reporting in rewriter's debugger (`break-rewrite`) for free variables
- Several bugs related to `local` — presents a serious challenge to implementing logic!
- `Double-rewrite` utility to override the rewriter's caching of results in the presence of congruences (see recent ACL2 workshop paper)
- Miscellaneous bug fixes, e.g., in timing utility and handling of Lisp type specs (`satisfies`)
- Recent performance enhancements, especially for theories: Critical for users at Rockwell Collins
- Prover time limits
- Context recovery for failed proofs (`redo-flat`)

Some other recent improvements

- Improved reporting in rewriter's debugger (`break-rewrite`) for free variables
- Several bugs related to `local` — presents a serious challenge to implementing logic!
- `Double-rewrite` utility to override the rewriter's caching of results in the presence of congruences (see recent ACL2 workshop paper)
- Miscellaneous bug fixes, e.g., in timing utility and handling of Lisp type specs (`satisfies`)
- Recent performance enhancements, especially for theories: Critical for users at Rockwell Collins
- Prover time limits
- Context recovery for failed proofs (`redo-flat`)

Some other recent improvements

- Improved reporting in rewriter's debugger (`break-rewrite`) for free variables
- Several bugs related to `local` — presents a serious challenge to implementing logic!
- `Double-rewrite` utility to override the rewriter's caching of results in the presence of congruences (see recent ACL2 workshop paper)
- Miscellaneous bug fixes, e.g., in timing utility and handling of Lisp type specs (`satisfies`)
- Recent performance enhancements, especially for theories: Critical for users at Rockwell Collins
- Prover time limits
- Context recovery for failed proofs (`redo-flat`)

Some other recent improvements

- Improved reporting in rewriter's debugger (`break-rewrite`) for free variables
- Several bugs related to `local` — presents a serious challenge to implementing logic!
- `Double-rewrite` utility to override the rewriter's caching of results in the presence of congruences (see recent ACL2 workshop paper)
- Miscellaneous bug fixes, e.g., in timing utility and handling of Lisp type specs (`satisfies`)
- Recent performance enhancements, especially for theories: Critical for users at Rockwell Collins
- Prover time limits
- Context recovery for failed proofs (`redo-flat`)

Dessert: Discussion

A final note: Throughout maintenance we make some effort to maintain backward compatibility:

- Support for user community
- Regression suite is critical

Fed up yet?

Had your fill?

If not, then let's chat some more. Comments? Questions?

Dessert: Discussion

A final note: Throughout maintenance we make some effort to maintain backward compatibility:

- Support for user community
- Regression suite is critical

Fed up yet?

Had your fill?

If not, then let's chat some more. Comments? Questions?

Dessert: Discussion

A final note: Throughout maintenance we make some effort to maintain backward compatibility:

- Support for user community
- Regression suite is critical

Fed up yet?

Had your fill?

If not, then let's chat some more. Comments? Questions?

Dessert: Discussion

A final note: Throughout maintenance we make some effort to maintain backward compatibility:

- Support for user community
- Regression suite is critical

Fed up yet?

Had your fill?

If not, then let's chat some more. Comments? Questions?