Rough Diamond: An Extension of Equivalence-based Rewriting

Matt Kaufmann (speaker) and J Strother Moore

The University of Texas at Austin

July, 2014
OUTLINE

Introduction

Examples

Conclusion
OUTLINE

Introduction

Examples

Conclusion
OVERVIEW
OVERVIEW

QUESTION: Out of hundreds of improvements made to ACL2 since its inception in 1989, why are we reporting on this one?
**OVERVIEW**

**QUESTION:** Out of hundreds of improvements made to ACL2 since its inception in 1989, why are we reporting on this one?

**ANSWER:** This one is a bit less specific to ACL2 than most.
OVERVIEW

QUESTION: Out of hundreds of improvements made to ACL2 since its inception in 1989, why are we reporting on this one?

ANSWER: This one is a bit less specific to ACL2 than most.

- Previous work rewrites with equivalences,
OVERVIEW

QUESTION: Out of hundreds of improvements made to ACL2 since its inception in 1989, why are we reporting on this one?

ANSWER: This one is a bit less specific to ACL2 than most.

- Previous work rewrites with equivalences, not just equalities,
OVERVIEW

QUESTION: Out of hundreds of improvements made to ACL2 since its inception in 1989, why are we reporting on this one?

ANSWER: This one is a bit less specific to ACL2 than most.

- Previous work rewrites with equivalences, not just equalities, and does so efficiently and automatically.
OVERVIEW

QUESTION: Out of hundreds of improvements made to ACL2 since its inception in 1989, why are we reporting on this one?

ANSWER: This one is a bit less specific to ACL2 than most.

- Previous work rewrites with equivalences, not just equalities, and does so efficiently and automatically.

- Today we’ll discuss an extension of that work.
ACL2 AND REWRITING

- ACL2:
  A Computational Logic for Applicative Common Lisp
ACL2 AND REWRITING

- ACL2:
  A Computational Logic for Applicative Common Lisp
- Under continuous development since 1989
ACL2 AND REWRITING

- ACL2: A Computational Logic for Applicative Common Lisp
- Under continuous development since 1989
- In regular use in industry (AMD, Centaur Technology, Intel, Oracle, and Rockwell Collins)
ACL2 AND REWRITING

- ACL2:
  A Computational Logic for Applicative Common Lisp
- Under continuous development since 1989
- In regular use in industry (AMD, Centaur Technology, Intel, Oracle, and Rockwell Collins)
- Sophisticated system (> 14 MB of source) supporting programming and proof
ACL2 AND REWRITING

- ACL2: *A Computational Logic for Applicative Common Lisp*
- Under continuous development since 1989
- In regular use in industry (AMD, Centaur Technology, Intel, Oracle, and Rockwell Collins)
- Sophisticated system (> 14 MB of source) supporting programming and proof
- Built-in automated induction, integrated decision procedures for linear arithmetic and Boolean logic, and many heuristics
ACL2 AND REWRITING

- **ACL2:**
  A Computational Logic for Applicative Common Lisp
- Under continuous development since 1989
- In regular use in industry (AMD, Centaur Technology, Intel, Oracle, and Rockwell Collins)
- Sophisticated system (> 14 MB of source) supporting programming and proof
- Built-in automated induction, integrated decision procedures for linear arithmetic and Boolean logic, and many heuristics
  - But the key proof technique is conditional rewriting:
    Theorem. $H \rightarrow L = R$
    suggests replacement of an instance $L/s$ of $L$ by a corresponding instance $R/s$ of $R$, if instance $H/s$ is provable.
EQUIVALENCE-BASED REWRITING

**Question:** Instead of

\[ H \rightarrow L \equiv R \quad \text{(or, } L \equiv R) \]

can we preserve mere equivalence instead?

\[ H \rightarrow L \sim R \quad \text{(or, } L \sim R) \]
EQUIVALENCE-BASED REWRITING

Question: Instead of
\[ H \rightarrow L=R \quad \text{(or, } L=R) \]
can we preserve mere equivalence instead?
\[ H \rightarrow L\sim R \quad \text{(or, } L\sim R) \]

Answer: depends on the context, i.e., the position in the surrounding term.
**EQUIVALENCE-BASED REWRITING**

**Question:** Instead of

\[ H \rightarrow L \equiv R \]  \hspace{1cm} (or, \( L \equiv R \))

can we preserve mere equivalence instead?

\[ H \rightarrow L \sim R \]  \hspace{1cm} (or, \( L \sim R \))

**Answer:** depends on the *context*, i.e., the position in the surrounding term.  
(Note: Not \( \equiv \) context, etc.)
**EQUIVALENCE-BASED REWRITING**

**Question:** Instead of

\[ H \rightarrow L= R \quad (\text{or, } L= R) \]

can we preserve mere equivalence instead?

\[ H \rightarrow L\sim R \quad (\text{or, } L\sim R) \]

**Answer:** depends on the **context**, i.e., the position in the surrounding term.  
(Note: Not **IF** context, etc.)

**Example.** Let \( \sim \) be bag-equivalence (two lists have the same members) and consider this **equivalence-based** rewrite rule:

\[ \text{remove-duplicates}(x) \sim x \]

**EQUIVALENCE-BASED REWRITING**

**Question**: Instead of

\[ H \rightarrow L = R \quad \text{(or,} \quad L = R \text{)} \]

can we preserve mere equivalence instead?

\[ H \rightarrow L \sim R \quad \text{(or,} \quad L \sim R \text{)} \]

**Answer**: depends on the context, i.e., the position in the surrounding term.  
*(Note: Not IF context, etc.)*

**Example**. Let \( \sim \) be bag-equivalence (two lists have the same members) and consider this *equivalence-based* rewrite rule:

\[
\text{remove-duplicates}(x) \sim x
\]

**Bad**: \( \text{length} (\text{remove-duplicates}(x)) = \text{length}(x) \).
**EQUIVALENCE-BASED REWRITING**

**Question:** Instead of

\[ H \rightarrow L = R \quad \text{(or, } L = R) \]

can we preserve mere equivalence instead?

\[ H \rightarrow L \sim R \quad \text{(or, } L \sim R) \]

**Answer:** depends on the *context*, i.e., the position in the surrounding term. *(Note: Not IF context, etc.)*

**Example.** Let \( \sim \) be bag-equivalence (two lists have the same members) and consider this *equivalence-based* rewrite rule:

\[
\text{remove-duplicates}(x) \sim x
\]

**Bad:** \( \text{length}(\text{remove-duplicates}(x)) = \text{length}(x) \).

**Good:** \( (a \in \text{remove-duplicates}(x)) = (a \in x) \).
CONTRIBUTION

Previously:

CONTRIBUTION

Previously:


- Equivalence-based rewriting
CONTRIBUTION

Previously:

- Equivalence-based rewriting
- Automatic tracking of equivalence relations sufficient to preserve in a given context
CONTRIBUTION

Previously:

- Equivalence-based rewriting
- Automatic tracking of equivalence relations sufficient to preserve in a given context
- Tracking is based on user-defined congruence rules
CONTRIBUTION

Previously:

- Equivalence-based rewriting
- Automatic tracking of equivalence relations sufficient to preserve in a given context
- Tracking is based on user-defined congruence rules
- > 1800 congruence rules in ACL2 Community Books
CONTRIBUTION

Previously:

- Equivalence-based rewriting
- Automatic tracking of equivalence relations sufficient to preserve in a given context
- Tracking is based on user-defined congruence rules
- > 1800 congruence rules in ACL2 Community Books

NEW:

*Patterned congruence rules* provide finer-grained specification of contexts for preserving equivalence relations.
CONTRIBUTION

Previously:

- Equivalence-based rewriting
- Automatic tracking of equivalence relations sufficient to preserve in a given context
- Tracking is based on user-defined congruence rules
- > 1800 congruence rules in ACL2 Community Books

NEW:

*Patterned congruence rules* provide finer-grained specification of contexts for preserving equivalence relations.

“Rough Diamond”: Patterned congruence rules are too new (released 01/2014) to have seen widespread use.
OUTLINE

Introduction

Examples

Conclusion
EXAMPLES

This talk will present examples from the paper.
This talk will present examples from the paper.

- Introduce some functions on trees.
**EXAMPLES**

This talk will present examples from the paper.

- Introduce some functions on trees.
- Review previous equivalence-based rewriting.
EXAMPLES

This talk will present examples from the paper.

- Introduce some functions on trees.
- Review previous equivalence-based rewriting.
- Illustrate the new extension.
EXAMPLES

This talk will present examples from the paper.

- Introduce some functions on trees.
- Review previous equivalence-based rewriting.
- Illustrate the new extension.

Our examples are based on binary trees.
DEFINITIONS

See the paper for the recursive definitions of the following notions.
DEFINITIONS

See the paper for the recursive definitions of the following notions.

- $t_1 \sim t_2$:
  Obtain $t_2$ from $t_1$ by a sequence of swaps of node children.
DEFINITIONS

See the paper for the recursive definitions of the following notions.

- **t1 ~ t2:**
  Obtain t2 from t1 by a sequence of swaps of node children.

- **mirror(tree):**
  Swap all left and right children.
**Definitions**

See the paper for the recursive definitions of the following notions.

- **$t_1 \sim t_2$:**
  Obtain $t_2$ from $t_1$ by a sequence of swaps of node children.

- **mirror(tree):**
  Swap *all* left and right children.

- **tree-product(tree):**
  Multiply the leaves of a tree.
CONGRUENCE RULES AND REWRITING

Left-to-right rewrite rule that is *legal in only some contexts*:

\[ \text{mirror}(x) \sim x \]
CONGRUENCE RULES AND REWRITING

Left-to-right rewrite rule that is *legal in only some contexts*: 
\[ \text{mirror}(x) \sim x \]

Congruence Rule (inner equivalence \(\sim\), outer equivalence \(=\)):
\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]
CONGRUENCE RULES AND REWRITING

Left-to-right rewrite rule that is *legal in only some contexts*:

\[ \text{mirror}(x) \sim x \]

Congruence Rule (inner equivalence \( \sim \), outer equivalence \( = \)):

\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]

Rewriting example:
**CONGRUENCE RULES AND REWRITING**

Left-to-right rewrite rule that is *legal in only some contexts*:

\[ \text{mirror}(x) \sim x \]

Congruence Rule (inner equivalence \(\sim\), outer equivalence \(=\)):

\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]

Rewriting example:

\[ \text{tree-product}(\text{mirror}(a)) \]
**CONGRUENCE RULES AND REWRITING**

Left-to-right rewrite rule that is *legal in only some contexts*:

\[ \text{mirror}(x) \sim x \]

Congruence Rule (inner equivalence \( \sim \), outer equivalence \( = \)):

\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]

Rewriting example:

\[ \text{tree-product}(\text{mirror}(a)) \]

*Congruence rule makes it OK to preserve \( \sim \):*
**CONGRUENCE RULES AND REWRITING**

Left-to-right rewrite rule that is *legal in only some contexts*:

\[ \text{mirror}(x) \sim x \]

Congruence Rule (inner equivalence \( \sim \), outer equivalence \( = \)):

\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]

Rewriting example:

\[ \text{tree-product}(\text{mirror}(a)) \]

*Congruence rule makes it OK to preserve \( \sim \):*

\[ \text{tree-product}(\text{mirror}(a)) \]
CONGRUENCE RULES AND REWRITING

Left-to-right rewrite rule that is *legal in only some contexts*: 
\[ \text{mirror}(x) \sim x \]

Congruence Rule (inner equivalence \(\sim\), outer equivalence \(=\)): 
\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]

Rewriting example:
\[ \text{tree-product}(\text{mirror}(a)) \]

*Congruence rule makes it OK to preserve \(\sim\):*
\[ \text{tree-product}(\text{mirror}(a)) = \]
CONGRUENCE RULES AND REWRITING

Left-to-right rewrite rule that is \textit{legal in only some contexts}:
\[
mirror(x) \sim x
\]

Congruence Rule (inner equivalence $\sim$, outer equivalence $=$):
\[
x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y)
\]

Rewriting example:
\[
\text{tree-product}(\mirror(a))
\]

\textit{Congruence rule makes it OK to preserve $\sim$:}
\[
\text{tree-product}(\mirror(a)) =
\]

\textit{So, we can rewrite with $\mirror(x) \sim x$:}
CONGRUENCE RULES AND REWRITING

Left-to-right rewrite rule that is *legal in only some contexts*:

\[
\text{mirror}(x) \sim x
\]

Congruence Rule (inner equivalence \(\sim\), outer equivalence \(=\)):

\[
x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y)
\]

Rewriting example:

\[
\text{tree-product(mirror(a))}
\]

*Congruence rule makes it OK to preserve \(\sim\):*

\[
\text{tree-product(mirror(a))} =
\]

*So, we can rewrite with \(\text{mirror(x)} \sim x\):*

\[
\text{tree-product(a)}
\]
**CONGRUENCE RULES AND REWRITING**

Left-to-right rewrite rule that is *legal in only some contexts*:
\[ \text{mirror}(x) \sim x \]

Congruence Rule (inner equivalence \(\sim\), outer equivalence \(=\)):
\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]

Rewriting example:
\[
\text{tree-product}(\text{mirror}(a)) = \text{tree-product}(\text{mirror}(a))
\]

*Congruence rule makes it OK to preserve \(\sim\):*
\[
\text{tree-product}(\text{mirror}(a)) = \text{tree-product}(\text{mirror}(a))
\]

*So, we can rewrite with \(\text{mirror}(x) \sim x\):*
\[
\text{tree-product}(a)
\]

Complexity: \(k_1 + k_2\) instead of \(k_1 \times k_2\) for:

- \(k_1\) functions like \text{mirror};
- \(k_2\) functions like \text{tree-product}. 
Consider a function \texttt{tree-data} that returns two values (as is common in ACL2 programming), with this \textit{patterned congruence rule}:

\[
\begin{align*}
x \sim y & \rightarrow \text{first} (\text{tree-data}(x)) = \\
& \text{first} (\text{tree-data}(y))
\end{align*}
\]
Consider a function `tree-data` that returns two values (as is common in ACL2 programming), with this patterned congruence rule:

\[ x \sim y \rightarrow \text{first}(\text{tree-data}(x)) = \text{first}(\text{tree-data}(y)) \]

**NOTE**: Classic congruence rules specified the context as an argument position of a single function symbol, e.g.:

\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]
Patterned congruence rules

Consider a function tree-data that returns two values (as is common in ACL2 programming), with this patterned congruence rule:

\[ x \sim y \rightarrow \text{first}(\text{tree-data}(x)) = \text{first}(\text{tree-data}(y)) \]

**NOTE:** Classic congruence rules specified the context as an argument position of a single function symbol, e.g.:

\[ x \sim y \rightarrow \text{tree-product}(x) = \text{tree-product}(y) \]

Compare with this patterned congruence rule:

\[ x \sim_1 y \rightarrow f(3, h(u, x), g(u)) \sim_2 f(3, h(u, y), g(u)) \]
PATTERNED CONGRUENCE RULES (CONTINUED)

Rewrite rule, unchanged from first example:

\[ \text{mirror}(x) \sim x \]
Patterned Congruence Rules (continued)

Rewrite rule, unchanged from first example:

\[ \text{mirror}(x) \sim x \]

Our patterned congruence rule, again:

\[ x \sim y \rightarrow \]

\[ \text{first}(\text{tree-data}(x)) = \text{first}(\text{tree-data}(y)) \]
Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

\[ \text{mirror}(x) \sim x \]

Our patterned congruence rule, again:

\[ x \sim y \rightarrow \text{first(tree-data}(x)) = \text{first(tree-data}(y)) \]

Modified rewriting example:
**Patterned Congruence Rules (Continued)**

Rewrite rule, unchanged from first example:

\[
\text{mirror}(x) \sim x
\]

Our patterned congruence rule, again:

\[
x \sim y \rightarrow \\
\text{first(tree-data}(x)) = \text{first(tree-data}(y))
\]

Modified rewriting example:

\[
\text{first(tree-data(mirror}(a)))
\]
**Patterned Congruence Rules (continued)**

Rewrite rule, unchanged from first example:

\[
\text{mirror}(x) \sim x
\]

Our patterned congruence rule, again:

\[
x \sim y \rightarrow \\
\text{first(tree-data}(x)) = \text{first(tree-data}(y))
\]

Modified rewriting example:

\[
\text{first(tree-data}(\text{mirror}(a)))
\]

*Patterned congruence rule provides context:*
**Patterned Congruence Rules (Continued)**

Rewrite rule, unchanged from first example:

\[
mirror(x) \sim x
\]

Our patterned congruence rule, again:

\[
x \sim y \rightarrow \text{first(tree-data}(x)) = \text{first(tree-data}(y))
\]

Modified rewriting example:

\[
\text{first(tree-data(mirror(a)))}
\]

*Patterned congruence rule provides context:*

\[
\text{first(tree-data(mirror(a)))}
\]
Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

\[ \text{mirror}(x) \sim x \]

Our patterned congruence rule, again:

\[ x \sim y \rightarrow \]
\[ \text{first}(\text{tree-data}(x)) = \text{first}(\text{tree-data}(y)) \]

Modified rewriting example:

\[ \text{first}(\text{tree-data(\text{mirror}(a)))} \]
*Patterned congruence rule provides context:*
\[ \text{first}(\text{tree-data(\text{mirror}(a)))} = \]
Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

\[ \text{mirror}(x) \sim x \]

Our patterned congruence rule, again:

\[ x \sim y \rightarrow \]
\[ \text{first} (\text{tree-data}(x)) = \text{first} (\text{tree-data}(y)) \]

Modified rewriting example:

\[ \text{first} (\text{tree-data}(\text{mirror}(a))) \]

*Patterned congruence rule provides context:*
\[ \text{first} (\text{tree-data}(\text{mirror}(a))) = \text{first} (\text{tree-data}(\text{mirror}(a))) \]

So, we can rewrite with \( \text{mirror}(x) \sim x \):
**PATTERNED CONGRUENCE RULES (CONTINUED)**

Rewrite rule, unchanged from first example:

\[
mirror(x) \sim x
\]

Our patterned congruence rule, again:

\[
x \sim y \rightarrow \text{first(tree-data}(x)) = \text{first(tree-data}(y))
\]

Modified rewriting example:

\[
\text{first(tree-data}(\text{mirror}(a)))
\]

*Patterned congruence rule provides context:*

\[
\text{first(tree-data}(\text{mirror}(a))) = \text{first(tree-data}(a))
\]

*So, we can rewrite with \( mirror(x) \sim x: \)*

\[
\text{first(tree-data}(a))
\]
**Patterned Congruence Rules (Continued)**

Rewrite rule, unchanged from first example:

\[
\text{mirror}(x) \sim x
\]

Our patterned congruence rule, again:

\[
x \sim y \rightarrow \\
\text{first(tree-data}(x)) = \text{first(tree-data}(y))
\]

Modified rewriting example:

\[
\text{first(tree-data(mirror(a)))}
\]

*Patterned congruence rule provides context:* \[
\text{first(tree-data(mirror(a)))} = \\
\text{So, we can rewrite with mirror}(x) \sim x:
\text{first(tree-data}(a))
\]

(Same complexity argument as before: \(k_1 + k_2\), not \(k_1 \times k_2\))
CONCLUSION

Not covered in this talk:
CONCLUSION

Not covered in this talk:

- General form of patterned congruence rules
CONCLUSION

Not covered in this talk:

- General form of patterned congruence rules
- Theory, e.g., how patterned congruence rules induce equivalence relations
**CONCLUSION**

Not covered in this talk:

- General form of patterned congruence rules
- Theory, e.g., how patterned congruence rules induce equivalence relations
- Algorithm for tracking equivalence relations to maintain
CONCLUSION

Not covered in this talk:

- General form of patterned congruence rules
- Theory, e.g., how patterned congruence rules induce equivalence relations
- Algorithm for tracking equivalence relations to maintain

The algorithm was challenging to implement, as the ACL2 rewriter has:
CONCLUSION

Not covered in this talk:

► General form of patterned congruence rules
► Theory, e.g., how patterned congruence rules induce equivalence relations
► Algorithm for tracking equivalence relations to maintain

The algorithm was challenging to implement, as the ACL2 rewriter has:

► 47 mutually recursive functions, which call many other functions;
CONCLUSION

Not covered in this talk:

- General form of patterned congruence rules
- Theory, e.g., how patterned congruence rules induce equivalence relations
- Algorithm for tracking equivalence relations to maintain

The algorithm was challenging to implement, as the ACL2 rewriter has:

- 47 mutually recursive functions, which call many other functions;
- 18 arguments in top-level rewrite function; and
CONCLUSION

Not covered in this talk:

- General form of patterned congruence rules
- Theory, e.g., how patterned congruence rules induce equivalence relations
- Algorithm for tracking equivalence relations to maintain

The algorithm was challenging to implement, as the ACL2 rewriter has:

- 47 mutually recursive functions, which call many other functions;
- 18 arguments in top-level \texttt{rewrite} function; and
- structured arguments; one has 18 fields.

See a 400-line comment in the ACL2 source code.
CONCLUSION (CONTINUED)

But we think this work will find use, especially since many ACL2 functions return multiple values.
CONCLUSION (CONTINUED)

But we think this work will find use, especially since many ACL2 functions return multiple values.
— so we believe that the effort was worthwhile!
But we think this work will find use, especially since many ACL2 functions return multiple values. — so we believe that the effort was worthwhile!

As with many recent ACL2 enhancements, this was driven by a request from an industrial user. Quoting Sol Swords:

Those are pretty simple examples, but I think they show one very useful application of patterned congruences, which is that you can have some structured object that has different congruences on different fields accessed/updated by nth/update-nth or g/s.
CONCLUSION (CONTINUED)

But we think this work will find use, especially since many ACL2 functions return multiple values. — so we believe that the effort was worthwhile!

As with many recent ACL2 enhancements, this was driven by a request from an industrial user. Quoting Sol Swords:

Those are pretty simple examples, but I think they show one very useful application of patterned congruences, which is that you can have some structured object that has different congruences on different fields accessed/updated by nth/update-nth or g/s.

Thank you for your attention.