Rough Diamond: An Extension of Equivalence-based Rewriting

Matt Kaufmann (speaker) and
J Strother Moore

The University of Texas at Austin

July, 2014

OUTLINE

OUTLINE

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 **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp

ACL2 AND REWRITING

- ACL2:
  A **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp
- Under continuous development since 1989

ACL2 and rewriting

- ACL2:
  A Computational **L**ogic for **A**pplicative **C**ommon **L**isp
- Under continuous development since 1989
- In regular use in industry (AMD, Centaur Technology, Intel, Oracle, and Rockwell Collins)

ACL2 AND REWRITING

- ACL2:
  A Computational **L**ogic for **A**pplicative **C**ommon **L**isp
- 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 **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp
- 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 **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp
- 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

$H \rightarrow L{=}R$ (or, $L{=}R$)

can we preserve mere equivalence instead?

$H \rightarrow L{\sim}R$ (or, $L{\sim}R$)

EQUIVALENCE-BASED REWRITING

Question: Instead of

$H \rightarrow L{=}R$ (or, $L{=}R$)

can we preserve mere equivalence instead?

$H \rightarrow L{\sim}R$ (or, $L{\sim}R$)

Answer: depends on the *context*, i.e., the position in the surrounding term.

EQUIVALENCE-BASED REWRITING

Question: Instead of

$H \rightarrow L{=}R$ (or, $L{=}R$)

can we preserve mere equivalence instead?

$H \rightarrow L{\sim}R$ (or, $L{\sim}R$)

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

EQUIVALENCE-BASED REWRITING

Question: Instead of

$H \rightarrow L{=}R$ (or, $L{=}R$)

can we preserve mere equivalence instead?

$H \rightarrow L{\sim}R$ (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:

`remove-duplicates(x)` $\sim$ `x`

Equivalence-based rewriting

Question: Instead of

$H \rightarrow L{=}R$ (or, $L{=}R$)

can we preserve mere equivalence instead?

$H \rightarrow L{\sim}R$ (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:

`remove-duplicates(x)` $\sim$ `x`

Bad: `length(remove-duplicates(x))` = `length(x)`.

Equivalence-based rewriting

Question: Instead of

$H \rightarrow L{=}R$ (or, $L{=}R$)

can we preserve mere equivalence instead?

$H \rightarrow L{\sim}R$ (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:

`remove-duplicates(x)` $\sim$ `x`

Bad: `length(remove-duplicates(x)) = length(x)`.

Good: `(a ∈ remove-duplicates(x)) = (a ∈ x)`.

CONTRIBUTION

Previously:

Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning* 40 (2008), pp. 293-306.

CONTRIBUTION

Previously:

Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning* 40 (2008), pp. 293-306.

- Equivalence-based rewriting

Contribution

Previously:

Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning* 40 (2008), pp. 293-306.

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

CONTRIBUTION

Previously:

Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning* 40 (2008), pp. 293-306.

- 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:

Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning* 40 (2008), pp. 293-306.

- 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:

Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning* 40 (2008), pp. 293-306.

- 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:

Bishop Brock, Matt Kaufmann, and J Strother Moore. Rewriting with Equivalence Relations in ACL2. *Journal of Automated Reasoning* 40 (2008), pp. 293-306.

- 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

Examples

This talk will present examples from the paper.

EXAMPLES

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.

- **t1** $\sim$ **t2**:
  Obtain **t2** from **t1** by a sequence of swaps of node children.

DEFINITIONS

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

- **t1** $\sim$ **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.

- **t1** $\sim$ **t2**:
  Obtain **t2** from **t1** 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:

`mirror(x)` $\sim$ `x`

Congruence rules and rewriting

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

`mirror(x)` $\sim$ `x`

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

`x` $\sim$ `y` $\rightarrow$ `tree-product(x) = tree-product(y)`

Congruence rules and rewriting

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

`mirror(x)` $\sim$ `x`

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

`x` $\sim$ `y` $\rightarrow$ `tree-product(x) = tree-product(y)`

Rewriting example:

Congruence rules and rewriting

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

`mirror(x) ∼ x`

Congruence Rule (inner equivalence ∼, outer equivalence =):

`x ∼ y → tree-product(x) = tree-product(y)`

Rewriting example:

`tree-product(mirror(a))`

Congruence rules and rewriting

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

```
mirror(x) ∼ x
```

Congruence Rule (inner equivalence ∼, outer equivalence =):

```
x ∼ y → tree-product(x) = tree-product(y)
```

Rewriting example:

```
   tree-product(mirror(a))
Congruence rule makes it OK to preserve ∼:
```

Congruence rules and rewriting

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

```
mirror(x) ~ x
```

Congruence Rule (inner equivalence ~, outer equivalence =):

```
x ~ y → tree-product(x) = tree-product(y)
```

Rewriting example:

```
   tree-product(mirror(a))
Congruence rule makes it OK to preserve ~:
   tree-product(mirror(a))
```

Congruence rules and rewriting

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

```
mirror(x) ∼ x
```

Congruence Rule (inner equivalence ∼, outer equivalence =):

```
x ∼ y → tree-product(x) = tree-product(y)
```

Rewriting example:

```
   tree-product(mirror(a))
Congruence rule makes it OK to preserve ∼:
   tree-product(mirror(a))   =
```

Congruence rules and rewriting

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

```
mirror(x) ∼ x
```

Congruence Rule (inner equivalence ∼, outer equivalence =):

```
x ∼ y → tree-product(x) = tree-product(y)
```

Rewriting example:

```
   tree-product(mirror(a))
Congruence rule makes it OK to preserve ∼:
   tree-product(mirror(a))   =
So, we can rewrite with mirror(x) ∼ x:
```

Congruence rules and rewriting

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

```
mirror(x) ∼ x
```

Congruence Rule (inner equivalence ∼, outer equivalence =):

```
x ∼ y → tree-product(x) = tree-product(y)
```

Rewriting example:

```
   tree-product(mirror(a))
Congruence rule makes it OK to preserve ∼:
   tree-product(mirror(a))   =
So, we can rewrite with mirror(x) ∼ x:
   tree-product(a)
```

Congruence rules and rewriting

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

```
mirror(x) ∼ x
```

Congruence Rule (inner equivalence ∼, outer equivalence =):

```
x ∼ y → tree-product(x) = tree-product(y)
```

Rewriting example:

```
   tree-product(mirror(a))
Congruence rule makes it OK to preserve ∼:
   tree-product(mirror(a))   =
So, we can rewrite with mirror(x) ∼ x:
   tree-product(a)
```

Complexity: $k_1 + k_2$ instead of $k_1 * k_2$ for:

- k_1 functions like `mirror`;
- k_2 functions like `tree-product`.

Patterned congruence rules

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

```
x ~ y → first(tree-data(x)) =
        first(tree-data(y))
```

PATTERNED CONGRUENCE RULES

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

$$x \sim y \rightarrow \texttt{first(tree-data(x))} = \texttt{first(tree-data(y))}$$

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

$$x \sim y \rightarrow \texttt{tree-product(x)} = \texttt{tree-product(y)}$$

PATTERNED CONGRUENCE RULES

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

```
x ~ y → first(tree-data(x)) =
        first(tree-data(y))
```

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

```
x ~ y → tree-product(x) = 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:

```
mirror(x) ∼ x
```

Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

```
mirror(x) ∼ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$
`first(tree-data(`x`)) = first(tree-data(`y`))`

Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

```
mirror(x) ∼ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$
`first(tree-data(`x`))  =  first(tree-data(`y`))`

Modified rewriting example:

PATTERNED CONGRUENCE RULES (CONTINUED)

Rewrite rule, unchanged from first example:

```
mirror(x) ~ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$
`first(tree-data(`x`)) = first(tree-data(`y`))`

Modified rewriting example:

```
first(tree-data(mirror(a)))
```

Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

```
mirror(x) ∼ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$

`first(tree-data(`x`)) = first(tree-data(`y`))`

Modified rewriting example:

```
   first(tree-data(mirror(a)))
Patterned congruence rule provides context:
```

Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

```
mirror(x) ∼ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$
`first(tree-data(`x`))` = `first(tree-data(`y`))`

Modified rewriting example:

```
   first(tree-data(mirror(a)))
Patterned congruence rule provides context:
   first(tree-data(mirror(a)))
```

Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

```
mirror(x) ~ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$
`first(tree-data(`x`)) = first(tree-data(`y`))`

Modified rewriting example:

```
   first(tree-data(mirror(a)))
Patterned congruence rule provides context:
   first(tree-data(mirror(a)))  =
```

Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

```
mirror(x) ∼ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$
first(tree-data(x)) = first(tree-data(y))

Modified rewriting example:

```
   first(tree-data(mirror(a)))
Patterned congruence rule provides context:
   first(tree-data(mirror(a)))  =
So, we can rewrite with mirror(x) ∼ x:
```

Patterned congruence rules (continued)

Rewrite rule, unchanged from first example:

```
mirror(x) ∼ x
```

Our patterned congruence rule, again:

$x \sim y \longrightarrow$
first(tree-data(x)) = first(tree-data(y))

Modified rewriting example:

```
   first(tree-data(mirror(a)))
Patterned congruence rule provides context:
   first(tree-data(mirror(a)))   =
So, we can rewrite with mirror(x) ∼ x:
   first(tree-data(a))
```

PATTERNED CONGRUENCE RULES (CONTINUED)

Rewrite rule, unchanged from first example:

```
mirror(x) ∼ x
```

Our patterned congruence rule, again:

$x \sim y \rightarrow$
`first(tree-data(`x`))  =  first(tree-data(`y`))`

Modified rewriting example:

```
   first(tree-data(mirror(a)))
Patterned congruence rule provides context:
   first(tree-data(mirror(a)))  =
So, we can rewrite with mirror(x) ∼ x:
   first(tree-data(a))
```

(Same complexity argument as before: $k_1 + k_2$, not $k_1 * k_2$)

Outline

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 `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!

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.*

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.