Double Rewriting for Equivalential Reasoning in ACL2

Matt Kaufmann
University of Texas at Austin
kaufmann@cs.utexas.edu

J Strother Moore
University of Texas at Austin
moore@cs.utexas.edu

ABSTRACT

Several users have had problems using equivalence-based rewriting in ACL2 because the ACL2 rewriter caches its results. We describe this problem in some detail, together with a partial solution first implemented in ACL2 Version 2.9.4. This partial solution consists of a new primitive, `double--rewrite`, together with a new warning to suggest possible use of this primitive.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*computational logic, mechanical theorem proving*

General Terms

Verification

Keywords

Verification, formal methods, rewriting, congruences, equivalence relations, double-rewrite

1. INTRODUCTION

Several users have had problems using equivalence-based rewriting in ACL2 because the ACL2 rewriter caches its results. We describe this problem in some detail, together with a partial solution first implemented in ACL2 Version 2.9.4 that employs a new primitive, `double-rewrite`, together with a new warning to suggest possible use of this primitive.

Section 2 begins with a very brief review of congruence-based rewriting that is intended to make this paper reasonably self-contained. It then presents a simple example that illustrates the problem and shows a solution using the new primitive, `double-rewrite`. Logically, this is the identity function: thus `(double-rewrite x)` is equal to `x`. However, the ACL2 rewriter treats a call `(double-rewrite u)` in a special manner: it first rewrites `u` in the current context, and then it rewrites the result. We explain this point in more detail with an example in aforementioned Section 2 and with a careful specification of `double-rewrite` in Section 3.

It is asking a lot of the user to identify when it is necessary to decorate terms in rewrite rules with `double-rewrite`. Therefore, ACL2 provides warnings when such calls may be indicated. We explain and illustrate this point in Section 4. We then show in Section 5 how to use warnings to discover missing congruence rules.

We conclude in Section 6 with discussion that suggests possible future work.

Above, we refer to this work as a *partial* solution. Ideally, ACL2 would automatically insert calls of `double-rewrite`, either explicitly or implicitly, without input from the user. An example below, in the concluding section, shows the potential inefficiency of inserting every `double-rewrite` call mentioned in a warning. One purpose of this paper is to encourage the ACL2 community to keep in mind the problem of automating the insertion of `double-rewrite` calls as they go about their own work, and to communicate to us any ideas towards a solution.

2. THE PROBLEM

The ACL2 congruence-based rewriter takes a term α, a substitution σ, an equivalence relation *equiv*, and some assumptions γ (arising for example from hypotheses of the goal and from governing assumptions from the `if`-structure of the surrounding term). It returns a term β such that

`(implies` γ `(`$equiv\ \alpha/\sigma\ \beta$`))`

is an ACL2 theorem. The rewriter *maintains* a set of known equivalence relations, *equiv*, for which it suffices to replace the input term with one that is *equiv* to it. Any of these may serve as the equivalence relation *equiv* above.

Further background on ACL2's congruence-based rewriting is worked into explanation of the example below. See also [1] or the ACL2 documentation [2] for more leisurely and thorough introductions to congruence-based rewriting in ACL2. Some ACL2 users find congruence-based rewriting to be an essential capability for their use of ACL2.

The following example, based on one sent to us by Dave Greve, is explained below.

```
(defun equiv (x y)
  (equal x y))
(defequiv equiv)
(defund pred (x) (equal x 17))
```

```
(defcong equiv iff (pred x) 1)
(encapsulate ((f (x) t) (g (x) t) (h (x) t))
 (set-ignore-ok t)
 (set-irrelevant-formals-ok t)
 (local (defun f (x) (pred x)))
 (local (defun g (x) 17))
 (local (defun h (x) 17))
 (defthm pred-h (pred (h x)))
 (defthm g-to-h (equiv (g x) (h x)))
 (defthm pred-implies-f
   (implies (pred x) (f x))))
; Fails!
(defthm f-of-g
 (f (g y)))
```

One might expect the proof `f-of-g` to succeed as follows. The rewriter first attempts to rewrite `(g y)` but leaves this term unchanged: rule `g-to-h` does not apply because we do not know that it suffices to maintain `equiv` in the first argument of `f`. Then one might expect rewriting of `(f (g y))` to complete as follows.

1. Match with rule `pred-implies-f`. So it suffices to relieve (i.e., rewrite to `t`) the instantiated hypothesis `(pred (g y))`.
2. The `defcong` rule above allows rewriting of `(g y)` in a context where it suffices to maintain `equiv`. So rule `g-to-h` rewrites `(g y)` to `(h y)`.
3. Rewrite `(pred (h y))` to `t` using rule `pred-h`.

So, why does the proof of `f-of-g` fail?

The problem is that we have ignored the substitution argument of the rewriter. Here is a more accurate description of the rewriting of `(f (g y))`.

1. Match with rule `pred-implies-f`. So it suffices to relieve the hypothesis `(pred x)` with `x` bound to `(g y)`.
2. The `defcong` rule above allows us to rewrite `x`, with `x` bound to `(g y)`, in a context where it suffices to maintain `equiv`. Since we are rewriting a variable (i.e., `x`), the result is obtained by looking up the binding of that variable, which yields `(g y)`.
3. Attempt to apply lemmas to complete the rewrite of `(pred (g y))`. None apply; return `(pred (g y))`, thus failing to relieve the hypothesis of `pred-implies-f`.

In brief, the problem is that `(g y)` was originally left unchanged by the rewriter in a context where `equiv` was *not* being maintained, and this result was cached by binding `x` to `(g y)`. This sort of caching is important in general for efficiency of the rewriter. Unfortunately, in this case subsequent rewriting was just a matter of looking up `x` to get `(g y)`, without any further rewrite of `(g y)`.

A solution is provided by using function `double-rewrite` to create an improved version of rewrite rule `pred-implies-f`:

```
(defthm pred-implies-f-better
  (implies (pred (double-rewrite x)) (f x)))
```

With this rule, we do better than Step 2 above.

1. Match with rule `pred-implies-f-better`. So it suffices to relieve the hypothesis `(pred (double-rewrite x))` with `x` bound to `(g y)`.
2. The `defcong` rule above allows us to rewrite `(double-rewrite x)`, with `x` bound to `(g y)`, in a context where it suffices to maintain the equivalence relation `equiv`.
   (a) Rewrite `x`, with `x` bound to `(g y)`, obtaining `(g y)` as before.
   (b) The `double-rewrite` call now directs us to rewrite that result, `(g y)`, in the empty binding environment, still maintaining `equiv`. Hence rule `g-to-h` applies, yielding `(h y)`.
3. Rewrite `(pred (h y))` to `t` using rule `pred-h`.

We invite the interested reader to execute the form

```
(trace$ (rewrite :entry (take 2 arglist))
        (rewrite-with-lemma
         :entry
         (list (car arglist)
               (cadr (access rewrite-rule
                             (cadr arglist)
                             :rune)))))
```

in the ACL2 loop, before proving `(f (g y))`. This will show the steps described above, where the first argument of `rewrite` and of `rewrite-with-lemma` is the term to be rewritten, the second argument of `rewrite` is the binding context, and the second argument shown for `rewrite-with--lemma` is the name of the lemma being applied.[1] `Rewrite` returns a structure `(mv rewritten-term ttree)` and `rewrite-with-lemma` returns a structure `(mv success-flag rewritten-term ttree)`, where `ttree` is a structure that records the rewrite rules used.

3. SPECIFICATION OF DOUBLE REWRITING IN ACL2

Logically, `double-rewrite` is the identity function, i.e., `(double-rewrite x)` is equal to `x`. However, the ACL2 rewriter treats calls of `double-rewrite` in the following special manner. When it encounters a term `(double-rewrite u)`, it first rewrites `u` in the current environment (with the same equivalence relations being maintained and the same bindings). Then, the rewriter rewrites the result in the empty binding environment (but again with the same equivalence relations being maintained).

ACL2 handles a common case automatically, without the need to call `double-rewrite`: namely, the term is a variable that is either the entire hypothesis or, more generally, a branch of the top-level `IF` structure of a hypothesis. The following example illustrates this point: `foo-holds` applies to prove the final theorem, even without a call of `double-rewrite` in the hypothesis of `foo-holds`. Note also that there is no "`double-rewrite`" warning when submitting `foo-holds`.

```
(encapsulate (((foo *) => *) ((bar *) => *))
 (local (defun foo (x) (declare (ignore x)) t))
 (local (defun bar (x) (declare (ignore x)) t))
 (defthm foo-holds
   (implies x (equal (foo x) t)))
```

[1]Some calls may be missing because of tail recursion elimination by the compiler.

```
 (defthm bar-holds-propositionally
   (iff (bar x) t)))
(thm (foo (bar y)))
```

4. WARNINGS

In this section we attempt to provide a deeper understanding of the need for `double-rewrite`, by explaining when ACL2 produces corresponding warnings.

Recall the following lemma from Section 2, which we ultimately improved by replacing its hypothesis with `(pred (double-rewrite x))`.

```
 (defthm pred-implies-f
   (implies (pred x) (f x)))
```

This lemma illustrates a problem with our solution: How does the user know to insert a call of `double-rewrite`? It seems unreasonable to expect this need to be obvious to the user, so it seems critical to report such situations. In the example above, therefore, we see the following warning when ACL2 processes the event `pred-implies-f`, and we see "double-rewrite" in the summary "Warnings" string at the end of the surrounding `encapsulate` event.[2]

```
ACL2 Warning [Double-rewrite] in ( DEFTHM
PRED-IMPLIES-F ...): In a :REWRITE rule generated
from PRED-IMPLIES-F, equivalence relation EQUIV is
maintained at one problematic occurrence of
variable X in the first hypothesis, but not at any
binding occurrence of X.  Consider replacing that
occurrence of X in the first hypothesis with
(DOUBLE-REWRITE X).  See :doc double-rewrite for
more information on this issue.
```

This warning suggests exactly the call of `double-rewrite` that we added. How did ACL2 figure this out? It considered the occurrence of `x` that will be bound when rewriting with the above rule, and noted that it is in a context where equality is (of course) maintained but `equiv` is not. Yet, `equiv` is maintained in the occurrence of `x` in the hypothesis, because of the `defcong` event shown in the example above.

ACL2 warns on missing calls of `double-rewrite` for variables occurring in hypotheses of rewrite rules and linear rules. When does it produce such warnings?

In general, ACL2 warns when it finds a non-binding occurrence of a bound variable in a context that is maintaining a known equivalence relation, such that no binding occurrence of that variable is in such a context. Binding variable occurrences are initially those in the left-hand side of a rewrite rule or a maximal (trigger) term of a linear rule. The only binding variable occurrences in hypotheses are as follows. First, a hypothesis of the form `(equal var term)` binds the variable `var` if it is not already bound and all variables occurring free in `term` are already bound. Second, a hypothesis of the form `(equiv var (double-rewrite term))` binds the variable `var` if `equiv` is a known equivalence relation, `var` is not yet bound, and all variables occurring free in `term` are already bound. We require the `double-rewrite` call in the latter case for the sake of backward compatibility. For example, if we treat all calls of other equivalence relations like calls of `equal`, then the proof fails for this event from `:mini-proveall`: `(defcong perm iff (mem x y) 2)`.

Again, such warnings are avoided in the situation described at the end of the preceding section.

We produce "double-rewrite" warnings for the right-hand side of the conclusion of a rewrite rule, as well as for the conclusion of a linear rule, in analogy to how we produce warnings for the hypotheses.[3] Consider the following example. If the `double-rewrite` call is omitted in `rule1`, then we will get a warning and the proof will fail for the final `thm`.

```
(skip-proofs
 (progn
   (defstub equiv1 (x y) t)
   (defequiv equiv1)
   (defstub c (x) t)
   (defstub e (x) t)
   (defstub f (x) t)
   (defstub g (x) t)
   (defstub h (x) t)
   (defstub i (x) t)
   (defthm rule1 (equiv1 (e x) (double-rewrite x)))
   (defthm rule2 (equiv1 (f x) (g x)))
   (defcong equiv1 equal (h x) 1)
   (defthm rule3
     (implies (h (double-rewrite x)) (c x)))
   (defthm rule4 (h (g a)))))
(thm (c (e (f a))))
```

Let us look carefully at the proof of the final `thm` above, understanding that when we are relieving hypotheses, we make just one pass through the rewriter.

1. Match `(c (e (f a)))` with `rule3`, binding `x` to `(e (f a))`.
2. Attempt to rewrite the hypothesis `(h (double-rewrite x))` of `rule3` to `t`, where `x` is bound to `(e (f a))`.
3. Rewrite `(double-rewrite x)` with `x` bound to `(e (f a))`, in a context where it suffices to maintain `equiv1` because we dove into the argument of `h`. The result is a term u obtained as follows:
   (a) Look up `x` and then apply `double-rewrite` to rewrite `(e (f a))`, still maintaining `equiv1`. The argument `(f a)` is left unchanged by the rewriter (note that `equiv1` is not being maintained because of the surrounding call of `e`).
   (b) Apply `rule1` to rewrite `(e x)` with `x` bound to `(f a)`, maintaining `equiv1`.
   (c) If `rule1` had a right-hand side of `x`, then the result u would be `(f a)`, obtained by looking up the binding of `x`. But the `double-rewrite` call invokes the rewriter on `(f a)`, still maintaining `equiv1`, to yield $u =$ `(g a)` by `rule2`.
4. It remains then to rewrite `(h` u`)`. Since u is `(g a)`, `rule4` applies to yield `t`. But if u had been `(f a)` as discussed above, then `rule4` would not apply and the hypothesis of `rule3` would ultimately rewrite to `(h (f a))`, not `t`.

[2]The warning is slightly different in ACL2 Version 2.9.4, but has been implemented to appear as shown in subsequent versions.

[3]This extra functionality is not present in ACL2 Version 2.9.4, but has been implemented for subsequent versions.

Note that there is a way to get the `thm` proved without putting a `double-rewrite` on the right-hand side of `rule1`. The way is to put *two* `double-rewrite`s in `rule3`, i.e., turn it into:

```
(defthm rule3
  (implies (h (double-rewrite (double-rewrite x)))
           (c x)))
```

But you can't predict how many nested `double-rewrite`s you'll need, so we view the "fault" as lying with `rule1`. Here is the warning we get with `rule1` if the `double-rewrite` call is omitted there.

```
ACL2 Warning [Double-rewrite] in ( DEFTHM RULE1
...): In a :REWRITE rule generated from RULE1,
equivalence relation EQUIV1 is maintained at one
problematic occurrence of variable X in the
right-hand side, but not at any binding occurrence
of X.  Consider replacing that occurrence of X in
the right-hand side with (DOUBLE-REWRITE X).  See
:doc double-rewrite for more information on this
issue.
```

5. DISCOVERING CONGRUENCE RULES

The "Double-rewrite" warnings described above provide feedback that is useful for the discovery of congruence rules. We illustrate how this works using an example pulled from supporting materials file `mini-proveall-plus.lisp`, which has a few others as well.

Consider the following congruence rule and its associated expansion. It says that the `perm`-equivalence class of a `cons` term is preserved when replacing the term's second argument, `Y`, with a `perm`-equivalent argument, `Y-EQUIV`.

```
ACL2 !>:trans1 (defcong perm perm (cons x y) 2)
 (DEFTHM PERM-IMPLIES-PERM-CONS-2
         (IMPLIES (PERM Y Y-EQUIV)
                  (PERM (CONS X Y)
                        (CONS X Y-EQUIV)))
         :RULE-CLASSES (:CONGRUENCE))
```

Now suppose we submit the following event.

```
(defthm insert-is-cons
  (perm (insert a x) (cons a x)))
```

Note that `perm` is maintained at `x` in `(cons a x)` because of the above congruence rule. However, with no similar congruence rule for `perm` and `insert`, we get the following warning.

```
ACL2 Warning [Double-rewrite] in ( DEFTHM
INSERT-IS-CONS ...): In a :REWRITE rule generated
from INSERT-IS-CONS, equivalence relation PERM is
maintained at one problematic occurrence of
variable X in the right-hand side, but not at any
binding occurrence of X.  Consider replacing that
occurrence of X in the right-hand side with
(DOUBLE-REWRITE X).  See :doc double-rewrite for
more information on this issue.
```

The warning is suggesting that when the rule is applied, then the instantiated argument `x` of `insert` will have been rewritten while maintaining only equality, so the resulting occurrence of that `x` in the `cons` term generated from the right-hand side might miss legal rewrites, using rules with equivalence relation `perm`. The warning suggests a solution: insert `double-rewrite` around `x` on the right-hand side so that such `perm`-based rewrites will be done on `x`. But a different solution is to arrange that the second (`x`) argument of `insert` had already been rewritten maintaining `perm`. The second form below then no longer produces a "Double-rewrite" warning, because now `perm` is maintained at both occurrences of `x`, not just the one on the right-hand side.

```
(defcong perm perm (insert a x) 2)
(defthm insert-is-cons[again-no-warn]
  (perm (insert a x) (cons a x)))
```

6. CONCLUSION

The current ACL2 regression suite was developed before the implementation of `double-rewrite` and associated warnings. Over 1300 "`double-rewrite`" warnings have occurred in the regression suite, which may seem to suggest that one can ignore such warnings. However, this work was motivated by a need for the support provided by `double-rewrite` in the work done by Dave Greve at Rockwell Collins.

The associated supporting materials provide several examples that illustrate "double-rewrite" warnings and the use of `double-rewrite`.

At this point we leave it to the user to decide, based on experimentation, whether or not to insert calls of `double--rewrite` suggested by associated warnings. A little experimentation of our own convinced us that it is can be too expensive to do double rewriting automatically. For example, consider the following commands, where the first is abbreviated from the `defpkg` command in `books/finite--set-theory/set-theory.acl2`.

```
(defpkg "S" ...)
(ld "finite-set-theory/set-theory.lisp"
    :dir :system :ld-skip-proofsp t)
(in-package "S")
:ubt! tl-pair
(acl2::time$ (defthm tl-pair
               (= (tl (pair x y)) y)))
```

The `defthm` event takes more than 100 times as long (73 seconds vs. less than 0.6 seconds on the same machine) after loading a patch that automatically inserts calls of `double--rewrite` in rewrite rules wherever ACL2 warns that such calls may be appropriate (see Section 4).

We welcome suggestions for how to automate the insertion of calls of `double-rewrite` without unacceptably slowing down proofs.

Acknowledgments

We thank Dave Greve of Rockwell Collins, Inc. for many useful discussions that led us to the development of this work. We also appreciate discussions on this topic with others members of the ACL2 community over time, including Vernon Austel and Eric Smith. This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591.

7. REFERENCES

[1] B. Brock, M. Kaufmann, and J S. Moore. Rewriting with equivalence relations in ACL2. In preparation.

[2] M. Kaufmann and J S. Moore. The ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2/`.