Rough Diamond: An Extension of Equivalence-based Rewriting

Matt Kaufmann and J Strother Moore

Dept. of Computer Science, University of Texas, Austin, TX, USA
`{kaufmann,moore}@cs.utexas.edu` `http://www.cs.utexas.edu`

Abstract. Previous work by the authors generalized conditional rewriting from the use of equalities to the use of arbitrary equivalence relations. Such (classic) *equivalence-based rewriting* automates the replacement of one subterm by another that may not be strictly equal to it, but is equivalent to it, where this equivalence is determined *automatically* to be sufficient at that subterm occurrence. We extend that capability by introducing *patterned* congruence rules in the ACL2 theorem prover, to provide more control over the occurrences where such a replacement may be made. This extension enables additional automation of the rewriting process, which is important in industrial-scale applications. However, because this feature is so new (introduced January, 2014), we do not yet have industrial applications to verify its utility, so we present a small example that illustrates how it supports scaling to large proof efforts.

Keywords: ACL2, rewriting, congruence, equivalence relation

1 Introduction

A conditional rewrite rule, $(P_1 \wedge \ldots \wedge P_n \rightarrow L = R)$, directs an instance of the term L to be rewritten to the corresponding instance of the term R, provided the corresponding instances hold for hypotheses P_1 through P_n. In previous work [1] we showed how to generalize conditional rewrite rules to allow an arbitrary equivalence relation, $\sim$, in place of $=$, thus: $(P_1 \wedge \ldots \wedge P_n \rightarrow L \sim R)$. Key is the use of proved *congruence rules* and *refinement rules*[1] to associate, *automatically*, an equivalence relation with each call of the rewriter, such that it is sound to replace a subterm by one that is equivalent. The above generalized rewrite may then be used when $\sim$ is a refinement of that equivalence relation. This capability is implemented in the ACL2 theorem prover [7, 6] and has seen substantial use: as of ACL2 Version 6.4, the community distribution of ACL2 input files [10] contains more than 1800 instances of congruence rules.

We give a preliminary report on a generalization, *patterned congruence rules*, introduced into ACL2 in Version 6.4, January, 2014. At this stage we can only guess at uptake of this capability by the ACL2 community, although we do expect it to be used at least by the requester of this feature at Centaur Technology [9].

[1] Refinement rules work essentially the same way in this new setting as they did before. We do not mention them further in this paper.

Our (existing and updated) approach to equivalence-based rewriting differs from approaches based on the use of quotient structures in higher-order logic, for example in HOL [3], Isabelle [4], and Coq [2]. To the best of our knowledge, our approach to first-order equivalence-based rewriting (without quotients) is the only one that automates the tracking of which equivalences are sufficient to preserve in a given context.

We begin in Section 2 by presenting a self-contained example to illustrate our previous work [1]. Section 3 then builds on that example to introduce our extension to patterned congruence rules, followed by a sketch of the relevant algorithm and theory in Section 4. We conclude with a few reflections.

The online ACL2 User's Manual [7] provides user-level introductions to equivalence-based rewriting. See topics EQUIVALENCE, CONGRUENCE, and (for this new work) PATTERNED-CONGRUENCE.

2 Previous work

The example below uses traditional syntax. Complete ACL2 input is online [5].

The following recursively-defined equivalence relation holds for two binary trees when one can be transformed to the other by some sequence of "flips": switching left and right children.

```
t1 ∼ t2 ≜ IF leaf-p(t1) ∨ leaf-p(t2) THEN t1=t2
          ELSE (left(t1) ∼ left(t2) ∧ right(t1) ∼ right(t2)) ∨
               (left(t1) ∼ right(t2) ∧ right(t1) ∼ left(t2))
```

When provided a suitable induction scheme, ACL2 automatically proves and stores the theorem that ∼ is an equivalence relation. We now define a function that swaps every pair of children in a binary tree (`cons` is the pairing operation).

```
mirror(tree) ≜ IF leaf-p(tree) THEN tree
               ELSE cons(mirror(right(tree)), mirror(left(tree)))
```

The equivalence-based rewrite rule below directs the replacement of any instance of the term `mirror(x)` by the corresponding instance of the term `x`, in contexts for which it suffices to preserve equivalence with respect to ∼. Of course, the ordinary rewrite rule `mirror(x) = x` is not a theorem!

```
REWRITE RULE: tree-equiv-mirror
mirror(x) ∼ x
```

The following function returns the product of the numeric elements of the fringe of a tree. It provides an example for sound replacement of `mirror(x)` by `x`: ACL2 proves the congruence rule below, stating that the return values are equal for equivalent inputs of the function `tree-product`. In general, a congruence rule states that the return values of a function call are equal (or more generally, suitably equivalent) when replacing a given argument by one that is equivalent.

```
tree-product(tree) ≜
```

```
IF [tree is a number] THEN tree
ELSE IF leaf-p(tree) THEN 1
ELSE tree-product(left(tree)) * tree-product(right(tree))
```

```
CONGRUENCE RULE: tree-equiv-->-equal-tree-product
x ∼ y → tree-product(x) = tree-product(y)
```

ACL2 can now prove the following theorem automatically by applying rewrite rule `tree-equiv-mirror` to the term `mirror(x)`. The congruence rule immediately above justifies this rewrite. When that rule is instead a rewrite rule, ACL2 is not able to use either it or `tree-equiv-mirror` to prove the theorem below.

```
THEOREM: tree-product-mirror
tree-product(mirror(y)) = tree-product(y)
```

This particular theorem is easy for ACL2 to prove automatically even without congruence rules or the rewrite rule `tree-equiv-mirror` (though induction would then be required). But to see the scalability of this approach, imagine that there are k_1 functions like `mirror` and k_2 like `tree-product`. If we then prove k_1 rewrite rules like `tree-equiv-mirror` and k_2 congruence rules like `tree-equiv--implies-equal-tree-product`, then these $k_1 + k_2$ rules set us up to perform automatically all $k_1 * k_2$ rewrites like `tree-product-mirror`.

3 Patterned congruence rules

A congruence rule, as discussed above, specifies when a given argument of a function call may be replaced by one that is suitably equivalent. A *patterned congruence rule* generalizes this idea by allowing a specified subterm of that call, which is not necessarily a top-level argument, to be replaced by one that is suitably equivalent. The following example is discussed further below.

PATTERNED CONGRUENCE RULE: `tree-equiv-->-equal-first-tree-data`
$x \sim y \rightarrow$ `first(tree-data(`x`))` = `first(tree-data(`y`))`

Notice that unlike a "classic" congruence rule, where the replacement of an equivalent subterm is specified at a specific argument of a function call, here x is to be replaced by y at a deeper position: a subterm of a subterm of the call. Indeed, the conclusion of the rule can be an equivalence between complex patterns, for example: $x \sim_1 y \rightarrow f(3, h(u,x), g(u)) \sim_2 f(3, h(u,y), g(u))$. That rule justifies replacement of a term x by a term $y \sim_1 x$, within any term of the form $f(3, h(u,x), g(u))$ that occurs where it suffices to preserve $\sim_2$.

A *patterned congruence rule* is thus a formula of the form $x \sim_{\texttt{inner}} y \rightarrow L \sim_{\texttt{outer}} R$, subject to the following requirements. Function symbols $\sim_{\texttt{inner}}$ and $\sim_{\texttt{outer}}$ have been proved to be equivalence relations. L and R are function calls such that x occurs in L, y occurs in R, and these are the only occurrences of x and y in the rule. Finally, R is the result of substituting y for x in L.

This rule enables the automatic rewrite of a subterm of L at the position of x to a term that is $\sim_{\texttt{inner}}$-equivalent to x, in any context where it suffices to preserve $\sim_{\texttt{outer}}$. We illustrate this process by continuing the example of the preceding section, this time defining a function that sweeps a tree to collect a *list* of results, whose first element is the product of the numeric leaves (as before). We omit some details; function `combine-tree-data(t1,t2)` returns a list whose first element is the product of the first elements from the recursive calls.

```
tree-data(tr) ≜
IF [tr is a number] THEN [tr, ...]
ELSE IF leaf-p(tr) THEN [1, ...]
ELSE combine-tree-data(tree-data(left(tr)), tree-data(right(tr)))
```

ACL2 can now automatically prove the patterned congruence rule displayed at the start of this section, `tree-equiv-->-equal-first-tree-data`. ACL2 then proves the theorem below as follows, much as it proves Theorem `tree--product-mirror` in the preceding section. First, the patterned congruence rule informs the rewriter that it suffices to preserve $\sim$ when rewriting `mirror(y)`. Hence, the rewrite rule `tree-equiv-mirror` (from the preceding section) is used to replace `mirror(y)` by `y`. Also as before, this small example suggests the importance of (patterned) congruences for scalability, where $k_1 + k_2$ rules set us up to perform automatically $k_1 * k_2$ different rewrites.

```
THEOREM: first-tree-data-mirror
first(tree-data(mirror(y))) = first(tree-data(y))
```

4 Algorithm correctness and patterned equivalence relations

In this section we outline briefly the algorithm implemented in ACL2 for using pattern-based congruence rules, and we touch on why it is correct. More details, including discussion of efficiency tricks and addressing of subtle issues (e.g., an example showing that arguments cannot be rewritten in parallel), are provided in a long comment in the ACL2 source code [8]. Of special concern is that ACL2 procedures that manipulate terms must quickly determine the available equivalences on-the-fly and tend to sweep the terms left-to-right, innermost first.

ACL2 implements classic equivalence-based rewriting by maintaining a *generated equivalence relation*, or *geneqv*: a finite list of function symbols that have each been proved to be an equivalence relation, representing the smallest equivalence relation containing them all. Rewriting is inside out, so to rewrite a function call, the rewriter first rewrites each argument of that call. Congruence rules are employed to compute the geneqv for rewriting each argument.

We have incorporated patterned congruence rules into that algorithm without changing its basic structure or efficiency (based on timing the ACL2 regression suite [10]). The key idea is to pass around a list representing so-called *patterned equivalences*, or *pequivs* for short, as defined below. We show how this list is updated as the rewriter dives into subterms, ultimately giving rise to equivalences to add to the current geneqv.

A pequiv is an equivalence relation corresponding to a term L that is a function call, a variable x that occurs uniquely in L, an equivalence relation $\sim$, and a substitution s. The pequiv *based on* L, x, $\sim$, and s is the smallest equivalence relation containing the following relation: $a \approx b$ if and only if there exist substitutions s_1 and s_2 extending s that agree on all variables except perhaps x such that $a = L/s_1$, $b = L/s_2$, and $s_1(x) \sim s_2(x)$.

For a natural number k and function call $C = f(t_1, \ldots, t_k, \ldots, t_n)$, the following notation is useful: $\mathit{pre}(C)$ is the list $(t_1, \ldots, t_{k-1})$, $@(C)$ is t_k, and $\mathit{post}(C)$ is the list $(t_{k+1}, \ldots, t_n)$. Now consider the pequiv based on L, x, $\sim$, and s, and let u be a term $f(u_1, \ldots, u_k, \ldots, u_n)$, where f is the function symbol of L and x occurs in the k^{th} argument of L. We define the *next equiv* as follows when for some substitution s' extending s, $\mathit{pre}(u)$ is $\mathit{pre}(L)/s'$ and $\mathit{post}(u)$ is $\mathit{post}(L)/s'$.[2] Let s' be the minimal such substitution. There are two cases. If x is an argument of L then the next equiv is the equivalence relation, $\sim$. Otherwise the next equiv is the pequiv based on $@(L)$, x, $\sim$, and s'.

The ACL2 rewriter maintains a list of pequivs and a geneqv (list of equivalence relations). Here we outline how those lists change when the rewriter, which is inside-out, calls itself recursively on a subterm. As before [1], classic congruence rules are applied to create a geneqv for the subterm; here we focus on how the list of pequivs contributes to the pequivs and geneqv for the subterm. Consider a pequiv p based on L, x, $\sim$, and s, among the list of pequivs maintained as we are rewriting the term $f(u_1, \ldots, u_k, \ldots, u_n)$, and consider the rewrite of u_k. There are three cases. If the next equiv for p (for position k) is $\sim$, then $\sim$ is added to the geneqv for rewriting u_k. If the next equiv for p is a pequiv p', then p' is added to the list of pequivs for rewriting u_k. Otherwise the next equiv for p does not exist, and p is ignored when rewriting u_k.

The following two theorems (relative to an implicit first-order theory) justify this algorithm. The first explains why a congruence rule justifies the sufficiency of maintaining the corresponding pequiv. The second explains why it suffices to maintain the next pequiv when rewriting a subterm.

Theorem 1. *For a provable patterned congruence rule* $x \sim_{\texttt{inner}} y \rightarrow L \sim_{\texttt{outer}} R$*, let* $\sim$ *be the pequiv based on* L*,* x*,* $\sim_{\texttt{inner}}$*, and the empty substitution. Then* $\sim$ *refines* $\sim_{\texttt{outer}}$*,* i.e.*, the following is a theorem:* $x \sim y \rightarrow x \sim_{\texttt{outer}} y$*.*

Theorem 2. *Let* $\sim_1$ *be a pequiv, let* u *be a term, and assume that the next equiv,* $\sim_2$*, exists for* $\sim_1$ *and* k*. Let* arg *be the* k^{th} *argument of* u*, let* arg' *be a term, and let* u' *be the result of replacing the* k^{th} *argument of* u *by* arg'*. Then the following is a theorem:* $\mathit{arg} \sim_2 \mathit{arg}' \rightarrow u \sim_1 u'$*.*

5 Reflections

ACL2 development began in 1989. Recent years have seen an increase in industrial application, with regular use at Advanced Micro Devices, Centaur Tech-

[2] We are simplifying the actual condition here, because the rewriter applies to both a term and a substitution, and this substitution must be applied to $\mathit{post}(u)$.

nology, Intel, Oracle, and Rockwell Collins, as well as academia and the U.S. Government. In order to support these users, we have been continuously improving ACL2; in particular, after the December 2012 release of Version 6.0 through the January 2014 release of Version 6.4, 129 distinct improvements have been reported in RELEASE-NOTES topics of the online ACL2 User's Manual [7].

While some of these improvements may present topics of interest to the ITP community, most are technical and specific to ACL2, as the focus has largely been on direct support for the user community, in particular industrial users. While few of these topics are likely candidates for traditional academic publication, patterned congruence rules seem to us an exception: any modern ITP system might benefit from them, if it is important to perform rewriting efficiently at the scale of industrial projects.

Acknowledgments

We thank the reviewers for helpful remarks. This research was supported by DARPA under Contract No. N66001-10-2-4087 and by ForrestHunt, Inc.

References

1. Brock, B., Kaufmann, M., Moore, J: Rewriting with equivalence relations in ACL2. Journal of Automated Reasoning 40(4), 293–306 (2008), `http://dx.doi.org/10.1007/s10817-007-9095-9`
2. Cohen, C.: Pragmatic quotient types in Coq. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving, Lecture Notes in Computer Science, vol. 7998, pp. 213–228. Springer Berlin Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-39634-2_17`
3. Homeier, P.: A design structure for higher order quotients. In: Hurd, J., Melham, T. (eds.) Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 3603, pp. 130–146. Springer Berlin Heidelberg (2005), `http://dx.doi.org/10.1007/11541868_9`
4. Huffman, B., Kunar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) Certified Programs and Proofs, Lecture Notes in Computer Science, vol. 8307, pp. 131–146. Springer International Publishing (2013), `http://dx.doi.org/10.1007/978-3-319-03545-1_9`
5. Kaufmann, M.: ACL2 demo of (patterned) congruences, see URL `https://acl2-books.googlecode.com/svn/trunk/demos/patterned-congruences.lisp`
6. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Boston, MA (Jun 2000)
7. Kaufmann, M., Moore, J S.: ACL2 home page, see URL `http://www.cs.utexas.edu/users/moore/acl2`
8. Kaufmann, M., Moore, J S.: Essay on Patterned Congruences and Equivalences, in ACL2 source file `rewrite.lisp`; see URL `https://acl2-devel.googlecode.com/svn/trunk/rewrite.lisp`
9. Swords, S.: Personal communication
10. ACL2 Community Books, see URL `http://acl2-books.googlecode.com/`