RP-Rewriter: An Optimized Rewriter for Large Terms in ACL2

Mertcan Temel
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas, USA
mert@utexas.edu

RP-Rewriter (*Retain-Property*) is a verified clause processor that can use some of the existing ACL2 rewrite rules to prove conjectures through term rewriting. Optimized for conjectures that can expand into large terms, the rewriter tries to mimic some of the ACL2 rewriting heuristics but also adds some extra features. It can attach side-conditions to terms that help the rewriter retain properties about them and prevent possibly some very expensive backchaining. The rewriter supports user-defined complex meta rules that can return a special structure to prevent redundant rewriting. Additionally, it can store fast alists even when values are not quoted. RP-Rewriter is utilized for two applications, multiplier design proofs and SVEX simplification, which involve very large terms.

1 Introduction

During the development process of a new proof method for correctness of multiplier designs, which is based on term-rewriting, we have stressed ACL2's built-in rewriting system for very large terms. Even though it is an intricate program that can prove various conjectures with a variety of rules, we have found that some conjectures that can expand into very large terms such as multiplier designs may pose a challenge to the built-in rewriter. In order to optimize the performance, we came up with some additional rewriter features including support for side-conditions (defined below) that can prevent exhaustive backchaining, a mechanism to prevent redundant rewriting of terms returned from meta functions, and an ability for rewrite rules to create fast-alists when keys are quoted and values are not. Throughout the paper, we refer to ACL2+Books Documentation for certain topics with the note "see :DOC doc-name", and they are available online [1].

We introduce a rewriting mechanism where terms can have certain properties attached as side-conditions in order to help relieve hypotheses efficiently. For example, `integerp` can be attached to a term indicating that it satisfies `integerp`. Retaining such properties about terms can provide some performance benefits as well as convenience. For example, consider the basic *logop* (see :DOC logops) and *4vec* (see :DOC sv::4vec) functions. The *log* operations, such as `logapp`, `loghead` and `logand`, work with integers, which may represent two-valued bit vectors, for bit-level operations. Most of the logops have a corresponding 4vec function that performs the same operation for an extended domain (i.e., four-valued bit-vectors). Assume that we have a library of rewrite rules that efficiently simplifies large terms composed of 4vec functions. If we want to apply the same simplification algorithm for a term composed of logops, then we could either copy and prove the same rules for corresponding logops, or we could more easily rewrite logops to their corresponding 4vec functions. The latter is a more practical approach; however, as we rewrite logops to 4vec functions, we lose the information that the rewritten term satisfies `integerp`, granted the logops in question always return an integer. With the built-in rewriter, if we are working with very large terms, then we would have to backchain to the innermost terms to show

G. Passmore, R. Gamboa (Eds.): ACL2 Workshop 2020
EPTCS ??, 2020, pp. 1–14, doi:10.4204/EPTCS.??.??

that the term satisfies `integerp`. However, if we can attach `integerp` as a side-condition to logop terms as we rewrite them to 4vec functions, then we can preserve the property of the term that it satisfies `integerp` no matter what it or its subterms become after rewriting. Such a system can help create a unitary library of rewrite rules for both function families with a lower maintenance cost.

Regular meta rules in ACL2 may cause redundant rewriting of terms that may entail performance issues when the terms in question are very large. ACL2's built-in rewriter works in an inside-out manner when reasoning about conjectures. Whenever a rewrite rule is applied, the system is smart enough not to rewrite the already processed inner terms. It can accomplish that because rewrite rules have a well defined pattern; and it is possible for the rewriter to tell apart the already processed terms from the newly introduced terms. However, when a meta rule is applied, the associated meta function can change the term in any way, which is not visible to the rewriter. Therefore, terms returned from a meta rule are rewritten completely as if everything is new. In cases where meta rules work with large terms, this redundancy may cause some performance issues.

For terms that include `hons-acons` and `hons-get`, the built-in rewriter uses logical definitions when at least one argument is not quoted. However, it is possible to develop a system that can create a fast-alist whose keys are quoted and values are not. For conjectures that create fast-alists, having such an ability may improve proof-time performance.

With the goal of addressing the aforementioned three problems, we have developed a customized rewriter, RP-Rewriter (*Retain-Property*), which we have published in the ACL2 Community books (projects/rp-rewriter/) as a verified clause processor. RP-Rewriter uses meta-extract [2] to retrieve and utilize a subset of existing ACL2 rewrite rules. Rewriting is done in a fashion similar to the built-in rewriter with much simpler heuristics, and we add user-friendly features to overcome these three problems. In Section 2, we outline the steps taken when rewriting occurs, and we discuss the implementation details that set RP-Rewriter apart from other rewriters. In Section 3, we briefly describe two applications where RP-Rewriter is utilized. In Section 4, we state the supported rewrite rules by RP-Rewriter. Finally, we briefly discuss some important details of the verification of RP-Rewriter in Section 5.

2 Implementation

When the clause processor function for RP-Rewriter is called, it first parses and gathers designated rewrite rules using meta-extract [2]. Then the conjecture and parsed rules are passed to our main rewriter function, `rp-rw`. Upon calling `rp-rw`, we follow the steps below:

- Check if rewriting should terminate with *dont-rw* (described in the subsection below).
- If the rewriter is in if-and-only-if context, try to reduce the term to `'t` by checking existing side-conditions.
- When applicable, update the term with the current context (i.e., a list of known facts that may come from hypotheses or `if` expressions).
- If the term is of the form `(if test then else)`, rewrite the arguments accordingly by expanding the context; in other cases, rewrite the arguments (i.e., subterms) by making recursive calls to `rp-rw`.
- Try applying executable-counterpart. For the current term, if the arguments are quoted and the executable counterpart of the function is enabled, then run the function using `magic-ev-fncall` [2].

- Try applying meta rules. If the term is changed, recursively call `rp-rw` on the resulting term.
- Try applying rewrite rules. If the term is changed, recursively call `rp-rw` on the resulting term; otherwise, terminate rewriting and return the current term.

These steps define the outline of our rewriting heuristics. In the subsections below, we describe the features and implementation of our rewriter that sets it apart from most rewriters. We start with the *dont-rw* structure that determines what terms should be rewritten and when rewriting should terminate. Secondly, we describe the notion of side-conditions with a few examples to show how it can be used to prove a conjecture when the built-in rewriter struggles, and how it can be faster than the built-in rewriter. Then we briefly talk about the fast-alist support as well as the meta functions that can be used with RP-Rewriter.

2.1 Controlled Rewriting with "dont-rw" Structure

We use a special data structure, called *dont-rw*, that we pass to our rewriter, `rp-rw`, along with the term in order to control rewriting and terminate when necessary. We traverse this data structure the same way as the main term being rewritten, that is we call `car` and `cdr` on the term and *dont-rw* at the same time. Whenever *dont-rw* is an atom and non-nil, the rewriter stops and returns the current term.

Example 1. *An example term being rewritten and a corresponding dont-rw. None of the f3 terms will be rewritten because their corresponding entry in dont-rw is a non-nil atom (i.e., `y` and `z`).*

```
(f1 (f2 a (f3 b c)) (f4 (f3 b c)))
(f1 (f2 x y) (f4 z))
```

We update *dont-rw* whenever a rule is applied and the term changes. When an applicable rewrite rule, which has the form `(implies hyp (equal lhs rhs))`, is found, we unify the term with `lhs`. We create a new term by applying the bindings from unification to `rhs`; and we pass the rule's `rhs` as *dont-rw*. We do the same for `hyp` when reliving hypotheses. We give meta functions the option to return *dont-rw* so that they can attain control over what terms should be rewritten. When meta functions do not return *dont-rw*, then it is set to `nil` and everything is rewritten by default. This *dont-rw* structure is invisible to the users expect for when they are creating meta functions.

2.2 Side-conditions

We develop a system to attach properties (side-conditions) to terms that may be useful while relieving hypotheses. For that purpose, we define an identity function `rp` with signature `(rp prop term)`, where the first argument `prop` is ignored and the second argument `term` is returned unchanged. When a term has a side-condition, it is wrapped by this `rp` function with its property to be retained. This property is a quoted function symbol with an arity of 1 (e.g., `'integerp` implying that the term is an integer). We maintain an invariant during the rewriting process that the attached side-condition is correct. Below we describe how side-conditions for terms are created with a few simple examples; and another example showing how side-conditions can be instrumental in proving some conjectures.

Terms might attain side-conditions through meta rules or rewrite rules. In meta functions, users might simply create an `rp` instance, correctness of which users would have to prove. This may be challenging in some cases and it is tailored more for advanced users. A more user-friendly to attach side-conditions can be done through rewrite rules. After deciding the properties to retain for certain terms on the right hand side, users can simply prove a lemma for those side-conditions; and attach that lemma to the rewrite rule

with a simple utility. That way, whenever such a rewrite rule is applied, designated terms can soundly retain the given side-conditions.

Example 2 shows how a side-condition can be attached to a rewrite rule. When we rewrite `logand` to `4vec-bitand`, we lose the immediate knowledge that the rewritten term can satisfy `integerp`. After rewriting `logand` terms with the first rule in this example, we would have to use the second rule to show that the resulting `4vec-bitand` term satisfies `integerp`, and backchain (again), which can be very expensive if arguments are very large terms. Instead, we attach this property as a side-condition to the first rule with `rp-attach-sc`. When RP-Rewriter starts to process the rules, it merges the two rules into one, and replaces all the instances of `(4vec-bitand x y)` with `(rp 'integerp (4vec-bitand x y))`. If the rewriter later runs into a term of the form `(integerp (rp 'integerp ...)`, then this term is replaced with `'t` if the conjecture is in if-and-only-if context. It is required that the conjuncts from the hypotheses of the side-condition lemmas are identical or a subset of the conjuncts from the hypotheses of the main rule. Note `def-rp-rule` is a macro that first calls `defthm` and then saves the rule to be used by RP-Rewriter.

Example 2. *An example of how a side-condition can be attached to a rule.*

```
(def-rp-rule logand-to-4vec-bitand
  (implies (and (integerp x)
                (integerp y))
           (equal (logand x y)
                  (4vec-bitand x y))))
(defthm logand-to-4vec-bitand-side-cond
  (implies (and (integerp x)
                (integerp y))
           (integerp (4vec-bitand x y))))
(rp-attach-sc logand-to-4vec-bitand
              logand-to-4vec-bitand-side-cond)
```

The process of attaching side-conditions is invisible to users. During unification, terms inside of `rp` functions are extracted in order to make side-conditions invisible to the unification process. Therefore, users do not need to consider side-conditions when creating rewrite rules; in fact, users are not allowed to have an `rp` call in a rewrite rule supplied to `def-rp-rule`. Such rules would be rejected by RP-Rewriter. The only times users need to be aware of this side-condition mechanism are when writing meta functions and using `syntaxp`.

Side-conditions may be useful in proving conjectures that can expand into very large terms. For example, as discussed in the introduction, we may want to utilize a simplification library for 4vec functions when working with logops. We can simply rewrite all the logops to their corresponding 4vec functions while remembering that all such terms satisfy `integerp`. This can help with the performance when trying to relieve `integerp` of very large terms. Example 3 shows a term that is rewritten with *logand-to-4vec-bitand* from Example 2. In this example, every 4vec-bitand instance is known to be an integer; and if we need to prove that this term is an integer, we can do that without any backchaining. When such terms are very large, this can reduce memory use and improve proof-time performance.

Example 3. *Two terms before and after we rewrite with logand-to-4vec-bitand from Example 2.*

```
(logand (logand x y)
        (logand a b))

(rp 'integerp
    (4vec-bitand (rp 'integerp (4vec-bitand x y))
                 (rp 'integerp (4vec-bitand a b))))
```

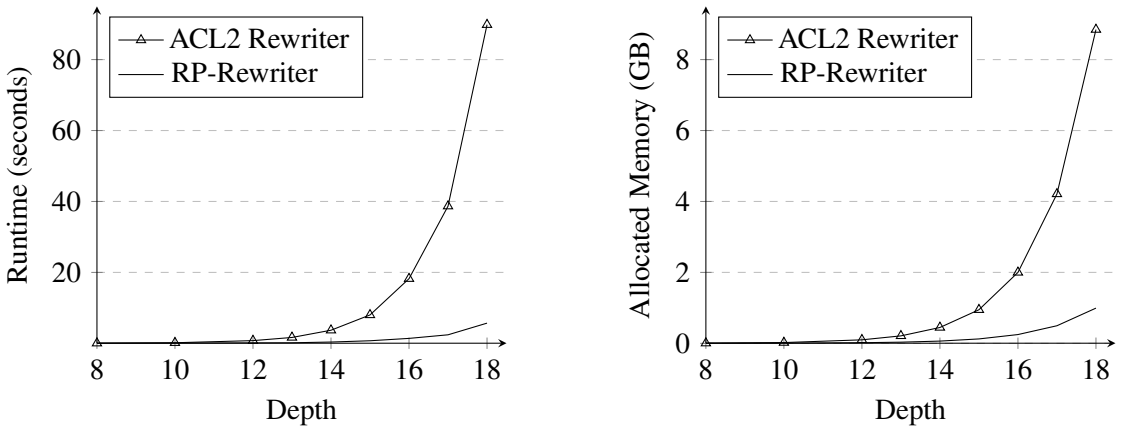

Figure 1: Performance comparison of ACL2's built-in rewriter and RP-Rewriter on a conjecture with a term tree of 4vec-bitand and logapp functions only using the rewrite rules from Example 2.

Figure 1 shows a performance comparison for the built-in rewriter and RP-Rewriter on the same conjectures, which highlights the performance benefits of side-conditions. We created dummy conjectures that have a binary tree of `4vec-bitand` functions on the left hand side, and an equivalent binary tree of `logand` on the right hand side. For example, if the tree depth is 3, then the left hand size would have 7 $(1+2+4)$ `4vec-bitand` instances nested within each other as seen below:

```
(thm
 (equal (4vec-bitand (4vec-bitand (4vec-bitand (iassoc 0 env) ...)
                                  (4vec-bitand (iassoc 2 env) ...))
                     ...)
        (logand ... ...)))
```

The innermost elements are of the form `(iassoc x env)`, where `iassoc` is known to return an integer, `x` is a unique constant, and `env` is the only variable in the whole conjecture. Before starting the proofs, we set the theory with `minimal-theory` (see :DOC minimal-theory) for both rewriters, and enabled only the rewrite rules from Example 2 and `integerp` of `iassoc`. We used `thm` and `rp-thm` (see :DOC rp-thm) to prevent the additional cost of saving rules. We did not provide any hints, or made any attachments to the built-in rewriter. We gradually increased the tree depth of the conjectures and measured the overall time and allocated memory of each event. As seen in the figure, the built-in rewriter has a much sharper incline in resource allocation in terms of both time and memory. This is mainly because the built-in rewriter has to backchain to innermost elements every time it needs to apply `logand-to-4vec-bitand` whereas RP-Rewriter uses the attached side-conditions and does not backchain. We have run these experiments with SBCL 2.0.2 on a machine with an Intel(R) Xeon(R) CPU E1270 @ 3.50GHz.

There may be other cases where having the side-condition system can provide much more convenience than only performance, and it may prove conjectures that ACL2's rewriter cannot. Example 4 shows the events to create such a case. With these events (also published in ACL2 Community Books in /projects/rp-rewriter/demo.lsp), we try to prove some conjectures such as `three-round-to-evens`.

Example 4. *A set of events to create a case where RP-Rewriter can prove a conjecture with side-conditions when ACL2's built-in rewriter fails.*

```
(encapsulate
  (((d2 *) => *) ((f2 *) => *) ((neg-m2 *) => *))
  (local (include-book "arithmetic-5/top" :dir :system))
  (local (defun d2 (x) (/ x 2)))
  (local (defun f2 (x) (floor x 2)))
  (local (defun neg-m2 (x) (- (mod x 2))))
  ;; syntaxp is necessary because rp-rewriter does not support loop-stopper.
  (def-rp-rule +-comm
    (implies (syntaxp (and (not (lexorder y x))
                           (or (atom x)
                               (not (equal (car x) 'binary-+)))))
             (and (equal (+ y x) (+ x y))
                  (equal (+ y x z) (+ x y z)))))
  (def-rp-rule my+-assoc
    (equal (+ (+ a b) c) (+ a b c)))
  (progn
    ;; a lemma for RP-Rewriter to maintain evenness
    (def-rp-rule my+-assoc-for-evens
      (implies (and (evenp (+ a b)) (evenp c))
               (equal (+ (+ a b) c) (+ a b c))))
    (defthmd my+-assoc-for-evens-side-cond
      (implies (and (evenp (+ a b)) (evenp c))
               (evenp (+ a b c))))
    (rp-attach-sc my+-assoc-for-evens
                  my+-assoc-for-evens-side-cond))
  (def-rp-rule d2-is-f2-when-even
    (implies (evenp x)
             (equal (d2 x) (f2 x))))
  ;; a function that rounds down a number to an even value
  ;; e.g., (round-to-even 93/10) = 8
  (defun round-to-even (a)
    (+ a (neg-m2 a)))
  ;; add definition rule to rp-rewriter's rule-set
  (add-rp-rule round-to-even)
  ;; rhs of the definition rule
  (defthmd round-to-even-is-even
    (evenp (+ a (neg-m2 a))))
  (rp-attach-sc round-to-even
                round-to-even-is-even))
(acl2::must-fail
 (defthm three-round-to-evens
   (equal (d2 (+ (round-to-even a) (round-to-even b) (round-to-even c)))
          (f2 (+ (neg-m2 a) (neg-m2 b) (neg-m2 c) a b c)))))
(acl2::must-succeed
 (defthmrp three-round-to-evens-use-rp ;; use RP-Rewriter as a clause-processor
   (equal (d2 (+ (round-to-even a) (round-to-even b) (round-to-even c)))
          (f2 (+ (neg-m2 a) (neg-m2 b) (neg-m2 c) a b c)))))
(acl2::must-succeed
 (defthmrp four-round-to-evens ;; use RP-Rewriter as a clause-processor
   (equal (d2 (+ (round-to-even a) (round-to-even b)
                 (round-to-even c) (round-to-even d)))
          (f2 (+ (neg-m2 a) (neg-m2 b) (neg-m2 c) (neg-m2 d) a b c d)))))
```

When the ACL2's built-in rewriter works on the conjecture given in `three-round-to-evens`, it first expands the definitions of `round-to-even` instances. Then the left-hand-side becomes:

```
(d2 (+ (+ a (neg-m2 a)) (+ b (neg-m2 b)) (+ c (neg-m2 c))))}
```

Then using commutativity and associativity of +, this term becomes:

```
(d2 (+ a b c (neg-m2 a) (neg-m2 b) (neg-m2 c))).
```

The attempt to apply `d2-is-f2-when-even` fails because it cannot prove that the argument is `evenp`. The individual groups of terms that make this argument even are distributed across the summation. For this specific problem, users might derive various solutions; for example, it is possible to prove a lemma that terms of this form satisfy `evenp`, or alternatively we can set `neg-m2` as an invisible function for + for the loop-stopper algorithm (this keeps `a` and `neg-m2` next to each other) and prove some simple lemmas to show evenness. However, for cases where terms are very large and the arguments are rewritten to different terms, which is the case for our multiplier design proofs [3], then neither of these solutions would work. In fact for our multiplier proofs, we have not been able to find a feasible solution using the built-in rewriter, other than possibly creating a complex meta rule to show evenness, which would likely be very tedious to implement and costly for proof-time performance.

On the other hand, with the retained side-conditions, RP-Rewriter can prove the same conjecture as seen in `three-round-to-evens-use-rp`, where `defthmrp` is a macro that creates a `defthm` event that calls RP-Rewriter as a clause processor. When RP-Rewriter works on the same term, the left-hand-side becomes:

```
(d2 (+ (rp 'evenp (+ a (neg-m2 a)))
       (rp 'evenp (+ b (neg-m2 b)))
       (rp 'evenp (+ c (neg-m2 c)))))
```

With the given lemmas, the terms in summations are ordered while maintaining evenness, and we get:

```
(d2 (rp 'evenp (+ a b c (neg-m2 a) (neg-m2 b) (neg-m2 c)))).
```

Then we can apply `d2-is-f2-when-even` because the evenness of the argument is maintained and hypothesis is relieved without using any lemma. The same system is sufficient for other similar conjectures such as `four-round-to-evens`. Due to readability, we cannot show a case for large terms where subterms might change drastically; however, we use the same system for cases where there may be millions of nodes under subterms.

2.3 Fast-alist Support for Non-quoted Terms

For conjectures that expand into terms with big association lists, RP-Rewriter implements a system to support creation of fast-alists when keys are quoted and values are not. When a rewrite rule introduces a term with `hons-acons` with a quoted key, instead of using the logical definition of `hons-acons`, we create a *shadowing fast-alist*. When an instance of `hons-get` is introduced for the same association list, we retrieve entries from the shadowing fast-alist.

The described fast-alist mechanism is a part of the verified rewriter. We create an identity function `falist` with a signature `(falist fast-alist term)` where the first argument is ignored and the second one is returned. The first argument of this function is always quoted and it is the shadowing fast-alist that corresponds to the term in the second argument. This mechanism is also invisible to users, and `falist` instances are created and updated with `hons-acons` and `fast-alist-free`. We maintain an invariant about `falist` instances, that is, the first and second arguments are association lists in correct syntactic shape and always have matching entries. Similar to `rp` instances, users are not allowed to introduce `falist` instances through rewrite rules.

Assume that we introduce three `hons-acons` instances on an empty alist (`'nil`) with three different quoted keys and distinct values. The complete term is:

```
(hons-acons 'key1 val1
            (hons-acons 'key2 val2
                        (hons-acons 'key3 val3 'nil)))
```

RP-Rewriter would create a `falist` instance and would rewrite this term to:

```
(falist '((key1 . val1) (key2 . val2) (key3 . val3))
        (cons (cons 'key1 val1)
              (cons (cons 'key2 val2)
                    (cons (cons 'key3 val3) 'nil))))
```

This first argument of this term is our shadowing fast-alist, and the second argument is the logical equivalent of the input term.

This support for fast-alist can provide significant benefits for certain applications. We have tested this feature with an earlier version of our multiplier proof library where we could enable and disable the fast-alist feature easily. Our multiplier correctness proofs create a large alist when expanding the definition of multiplier designs to store the values of internal wires. When the fast-alist feature is enabled, the described mechanism is deployed and `hons-get` retrieves elements from that alist; when it is disabled, another meta rule parses the term representing the alist to retrieve elements. Table 1 shows these experimental results for some Wallace and Dadda tree multipliers with simple partial products and simple final stage adder. As seen on this table, the effect of our fast-alist support can be very significant with larger circuits. We have run these experiments with SBCL 2.0.2 on a machine with an Intel(R) Xeon(R) CPU E1270 @ 3.50GHz.

Table 1: Runtime performance of the fast-alist feature on some multiplier correctness proofs (seconds)

Size	Multiplier Type	Fast-Alist Feature Enabled	Fast-Alist Feature Disabled
32x32	Dadda	0.58	0.75
	Wallace	0.80	0.92
64x64	Dadda	3.24	5.75
	Wallace	5.24	7.82
128x128	Dadda	27.33	83.26
	Wallace	39.05	101.03

2.4 Meta Functions for RP-Rewriter

Meta functions for RP-Rewriter may be implemented the same way as regular ACL2 meta functions excluding the support for *stobj*s. When proving the correctness of meta functions, users may not use an arbitrary evaluator but have to use our evaluator `rp-evlt`, which we used to prove RP-Rewriter correct. Since `rp-evlt` might not recognize the functions that users reason about in their meta functions, we implement a mechanism to help this evaluator recognize new functions. We adapt a system from FGL libraries from ACL2 Community books (/centaur/fgl), and used the utility `def-formula-checks`, which creates an executable *formula-checks* function with a single argument, state. Execution of the formula-checks function should always evaluate to `t` unless some function is redefined in ACL2. This utility also creates some rewrite rules for `rp-evlt` to know how to evaluate given new functions. Suppose that we want to reason about a newly defined function `(foo arg1 arg2)`, then the generated rewrite rule for `foo` would have the following form:

```
(defthm rp-evlt-of-foo-when-example-formula-checks
  (implies (and (example-formula-checks state)
                (rp-evl-meta-extract-global-facts))
           (equal (rp-evlt (list 'foo arg1 arg2) a)
                  (foo (rp-evlt arg1 a)
                       (rp-evlt arg2 a)))))
```

Having created a formula-checks function `example-formula-checks`, users can prove the final correctness theorem for evaluation of meta functions:

```
(defthm example-meta-fnc-correct
  (implies (and (example-formula-checks state)
                (rp-evl-meta-extract-global-facts)
                (valid-sc term a)
                (rp-termp term))
           (equal (rp-evlt (example-meta-fnc term) a)
                  (rp-evlt term a))))
```

Before running meta functions, we run the generated formula-checks functions to relieve this extra hypothesis.

Functions `valid-sc` and `rp-termp` check the correctness of side-conditions and syntactic coherence of terms, respectively. In addition to term evaluation with `rp-evlt`, users also have to prove another lemma stating that returned terms from meta functions satisfy `valid-sc`. On the other hand, proving that meta functions return `rp-termp` is optional but recommended. If that proof is not provided by the user, RP-Rewriter will run `rp-termp` for all the returned terms, which may slow down the a proof significantly. A more detailed discussion of `rp-termp` and `valid-sc` is given in Section 5.

A meta function may or may not return *dont-rw* structure. If the user chooses to generate *dont-rw*, then the return signature of the meta function should be `(mv term dont-rw)`. It should be noted that *dont-rw* structure is not the only way to prevent redundant rewriting of the terms returned from meta functions. Users might also wrap the terms that should not be rewritten (upon returning from a meta function) with `hide`. With a rule rewriting `(hide x)` to `x`, instances of `hide` will be removed and rewriting on selected terms will be avoided. This alternative method can also be used for meta functions for ACL2's rewriter. In RP-Rewriter, users may choose either method to control which terms should be rewritten upon returning from their meta functions. Depending on the application, using the `hide` method might be more costly than *dont-rw* because creating `hide` instances and applying a rewrite rule to remove them can cause more memory to be allocated than creating a *dont-rw*. This memory gain might not be significant (for our multiplier proofs, we observe to use around 2-3% less memory); however, we find the *dont-rw* method to be more convenient. Inserting numerous `hide` instances inside terms in meta functions makes them less readable. Deciding later which terms should or should not be rewritten with the *dont-rw* structure facilitates an easier development process.

3 Applications

RP-Rewriter has been utilized for various applications that may involve dealing with very large terms. In this section, we briefly describe the use cases for multiplier design verification, where RP-Rewriter is used as a clause processor, and simplification of SV expressions, where RP-Rewriter is used as a logic-mode function.

3.1 Multiplier Proofs

We use RP-rewriter to implement our verification method for multiplier designs, which is based completely on term rewriting [3]. We make extensive use of performance benefits of side-conditions and fast-alist support. We have tested the rewriter for very large multipliers (i.e., 1024x1024), whose proofs create terms with millions of nodes but finish in less than 10 minutes.

Our multiplier verification method is based on rewriting one-bit adder modules (e.g., full/half adders) in terms of `mod` and `floor` functions with rewrite rules that have hypotheses that all the inputs satisfy `bitp`. After these rewrites, `mod` and `floor` functions are simplified with some other rewrite rules and meta functions, and the circuit semantics are rewritten to the design specification. During this arithmetic simplification, the terms change their shapes so much that the hypotheses in our rewrite rules become too difficult to relieve similarly to the scenario given in Example 4. Therefore we attach side-conditions to terms and eliminate the need for backchaining when relieving hypotheses that may otherwise be very expensive.

We have two libraries implementing the same multiplier verification algorithm with different implementation techniques. The first one is a mix of rewrite rules and meta functions, whereas the second implements the majority of the algorithm with a meta function for better time and memory performance. They are published in ACL2 Community books (/books/projects/rp-rewriter/lib) (see :DOC rp::multiplier-proofs and :DOC rp::multiplier-proofs-2). We do not have working libraries for ACL2's built-in rewriter since these proofs depend on the side-conditions feature.

3.2 SVEX Simplify

A Symbolic Vector Expression (SVEX) is a core data type defined with a fixed set of functions to describe the behavior of translated Verilog modules (see :DOC sv::svex). In some cases, these expressions can be too large and complex, and may need to be simplified. We implement a flexible program with RP-Rewriter to take these expressions saved as a constant, rewrite and simplify them, and return an equivalent expression. We call this program `svex-simplify` (see :DOC svl::svex-simplify).

We convert *SVEX*es to regular ACL2 expressions using its evaluator `svex-eval` with our rewriter. The resulting terms are composed of *4vec* functions, and they are rewritten and simplified with proved rewrite rules. Then the resulting terms are converted back to *SVEX*es. The `svex-simplify` function with a simplification library of rewrite rules are available in ACL2 Community books (/books/centaur/svl/). In this system, we use the main rewriter function, `rp-rw`, as a logic-mode function instead of a clause processor because we do not try to prove a conjecture but take in a constant and return another. Users may change the simplification method by disabling or adding rewrite rules to the system. All the functions are in logic mode and guard-verified.

4 Supported Rewrite Rules

RP-Rewriter supports only some of the rewrite rules. If a rule of any other class, such as type-prescription, is supplied to RP-rewriter, then it would be treated as a rewrite rule. A rewrite rule should match the following criteria:

- Hypotheses, LHS and RHS should satisfy `rp-termp`, which defines the syntax of our terms (see Section 5).
- Hypotheses, LHS and RHS cannot contain an instance of `rp` or `falist`, which can be introduced only internally or by meta-functions.

- Hypotheses and RHS cannot contain a variable that is not present in LHS (i.e., free variables and `bind-free` are not allowed)
- LHS cannot have an instance of `if` due to some difficulties observed during verification of RP-Rewriter with regard to side-conditions.

If a rewritten term has an instance of the form `(if test then else)`, the context (i.e., known facts) is expanded with `test` and `(not test)` when the rewriter dives into `then` and `else` terms, respectively. The side-conditions under `then` and `else` might be valid with these expanded contexts. When terms with side-conditions are unified with the LHS of a rewrite rule, we add those side-conditions to the context as well. Keeping these in mind, consider that we want to rewrite this term:

```
(foo (if (p2 x) (rp 'p1 x) y) x)
```

Only one of the `x` instances has a side-condition, and adding `(p1 x)` to the context would be unsound because it might be true only when `(p2 x)` is true. We would have to develop a special mechanism for such cases. Even though it is possible, we decided not to complicate our rewriting mechanism and verification of RP-Rewriter. We do not expect to see many rewrite rules of this form, and may start supporting it if the need rises.

It should be noted that conjectures and rewrite rules that contain *lambda* expressions are *beta-reduced* before rewriting starts. It is due to some difficulties experienced when verifying the side-condition feature (they present a completely different class of terms, which is very difficult to integrate), and we leave the support for lambda expressions as a future work. For cases where users have to maintain *lambda* structure in rewrite rules to prevent repeated rewriting of terms, our utility *defthm-lambda* can be used while proving the rewrite rule in ACL2. This utility creates different functions for every lambda expression, and divides the RHS of the rule into multiple rewrite rules. This removes all the immediate lambda expressions from RHS but maintain the same functionality. Example 5 shows how this utility works.

Example 5. *The defthm-lambda utility replacing a rewrite rule with a lambda expression on its RHS with an equivalent rewrite rule without lambda expressions.*

```
(defthm-lambda foo-redef
  (implies (p x)
           (equal (foo x)
                  (let* ((a (f1 x))
                         (b (f2 x)))
                    (f4 a a b)))))
;; The above event is translated into this:
(encapsulate
  (((foo-redef_lambda-fnc_1 * *) => *)
   ((foo-redef_lambda-fnc_0 * *) => *))
  (local (defun-nx foo-redef_lambda-fnc_1 (b a)
           (f4 a a b)))
  (local (defun-nx foo-redef_lambda-fnc_0 (a x)
           (foo-redef_lambda-fnc_1 (f2 x) a)))
  (def-rp-rule foo-redef_lambda-opener
    (and (equal (foo-redef_lambda-fnc_1 b a)
                (f4 a a b))
         (equal (foo-redef_lambda-fnc_0 a x)
                (foo-redef_lambda-fnc_1 (f2 x) a))))
  (def-rp-rule foo-redef
    (implies (p x)
             (equal (foo x)
                    (foo-redef_lambda-fnc_0 (f1 x) x)))))
```

5 Verification of RP-Rewriter

RP-Rewriter is a verified clause processor. We accomplished this by creating a proof scheme with various invariants throughout our rewriter functions. These invariants are:

- Evaluation of terms with our evaluator `(rp-evlt term a)` should remain the same.
- Evaluation for validity of side-conditions with `(valid-sc term a)` should always return `t` .
- The syntax of terms should satisfy `rp-termp`.

In this section, we describe these invariants, and how meta-extract is used to prove RP-Rewriter correct.

5.1 Evaluation of Terms

Every verified meta function and clause processor must be proved correct with an evaluator generated by `defevaluator`, which is a generic utility distributed with ACL2. RP-Rewriter is not an exception, and we created an evaluator `rp-evl` using `defevaluator`, and we use an extension of that evaluator, namely `rp-evlt` that processes terms before evaluating. For every function that changes a term, we prove that the returned term evaluates to the same value as the input.

`(rp-evlt term a)` is equivalent to `(rp-evl (rp-trans term) a)`. The terms we rewrite are not regular, translated ACL2 terms, but we allow a sequence of `cons` terms ending with `nil` to be stored and rewritten as `list` instances. For example, the term `(cons a (cons b (cons c 'nil)))` may be stored as `(list a b c)`. The function `rp-trans` translates the instances of `list` to its equivalent `cons` form. This is an experimental feature, and such untranslated terms can only be created and modified in meta functions. The purpose of this feature is to reduce memory allocation when very large lists are present.

5.2 Evaluation of Side-conditions

The regular evaluator is expectedly not capable of recognizing side-conditions in our terms. Therefore, we create a special evaluator `(valid-sc term a)` that validates the correctness of side-conditions that might be present in terms. Whenever there is a proof of evaluation with `rp-evlt` for a function, we also have a proof with `valid-sc`. These proofs are in the following form:

```
(defthm some-rp-rw-fnc-valid-sc
  (implies (and (valid-sc term a)
                <other-hypotheses>)
           (valid-sc (some-rp-rw-fnc term <other-args>) a)))
```

`valid-sc` is mutually-recursive with `valid-sc-subterms`, and they traverse a term and its subterms for side-conditions. `(valid-sc term a)` returns `t` when `term` is an atom or quoted. If it is matches the form `(if test then else)`, then it evaluates to:

```
(and (valid-sc test a)
     (if (rp-evlt test a)
         (valid-sc then a)
       (valid-sc else a)))
```

This is because we test the correctness under the current context as updated by such `if` calls. If `term` is an instance of the form `(rp 'prop x)`, then it evaluates:

```
(and (rp-evlt `(prop ,x) a)
     (valid-sc x a))
```

For all the other function calls, `valid-sc` runs `valid-sc-subterms` on the arguments. That function in return runs `valid-sc` on each argument.

5.3 Syntax of Terms

Similar to `pseudo-termp`, we have `rp-termp` that set a standard and rules for the syntax of our terms. We prove a similar property about `rp-termp` for each of our functions as `valid-sc`. `(rp-termp term)` evaluates to `t` when `term` satisfies these conditions:

- Innermost elements should each either be a non-nil symbol or quoted.
- Calls for `rp` should have two arguments, and the first argument should be a quoted non-nil symbol.
- Calls for `falist` should have two arguments and the first argument should be a shadowing fast-alist of the second argument as described in Section 2.3. This is how we maintain the invariant for the fast-alists feature.
- Function names must be a symbol and non-nil. In other words, lambda expressions are not allowed.

Guaranteeing that terms retain this syntax can help prove other lemmas correct because it defines the shape of terms and helps avoid dealing with unexpected cases.

5.4 Meta-extract

Meta-extract extends the capabilities of meta functions or clause processors to retrieve more facts from the `state` [2]. The utility `def-meta-extract` (see :DOC def-meta-extract), creates a compact macro for evaluators that can be added to the hypotheses when proving a meta-function/clause processor correct. For our evaluator `rp-evl`, this utility creates `(rp-evl-meta-extract-global-facts)`. When we add this call to our hypotheses, we can trust `magic-ev-fncall` (used for running executable-counterparts and meta-functions added to RP-Rewriter), `meta-extract-formula` (used for extracting rewrite/definition rules from ACL2) and other meta-extract functions to return correct values. Working with meta-extract has been fairly easy, and it did not complicate our proof procedure after integration.

After establishing the correct invariants and integrating meta-extract, we could achieve a low-maintenance and versatile proof scheme that can hold up well when changes are made or new features are added to the rewriter.

6 Conclusion

In this paper, we have introduced a rewriter customized specifically for conjectures that may expand into very large terms. We have proved the correctness of this rewriter and saved it as a verified clause-processor in ACL2. RP-Rewriter has the distinctive capability to retain properties about terms that we call side-conditions. With other features including fast-alist support and the *dont-rw* structure, we have utilized RP-Rewriter for applications that involved very large terms such as multiplier design verification and SVEX simplification. Besides these use cases, users might also use rp-rewriter as a clause processor on its own, or as a part of their meta functions in order to perform term rewriting.

Acknowledgments

We would like to specially thank Matt Kaufmann for detailed discussions that help build the foundations of RP-Rewriter, J Strother Moore and Warren A. Hunt, Jr. for feedback, and Sol Swords for his suggestions to improve the system. This material is based upon work supported in part by DARPA under Contract No. FA8650- 17-1-7704.

References

[1] ACL2 Community (Accessed May 1, 2020): *ACL2+Books Documentation*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[2] Matt Kaufmann & Sol Swords (2017): *Meta-extract: Using Existing Facts in Meta-reasoning*. In: *Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23, 2017*, pp. 47–60. Available at `https://doi.org/10.4204/EPTCS.249.4`.

[3] Mertcan Temel, Anna Slobodova & Warren A. Hunt Jr. (2020): *Automated and Scalable Verification of Integer Multipliers*. In: *CAV2020 (to appear)*.