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

Mertcan Temel

University of Texas at Austin

mert@utexas.edu

May 28, 2020

Introduction

- RP-Rewriter = **R**etain-**P**roperty Rewriter
- Goal: simplify and prove conjectures using regular ACL2 rewrite/meta rules.
- A verified clause processor.
- Developed for verification of very large multipliers using a rewrite-based method; however, it is a generic rewriter.
- Can be faster than ACL2's rewriter for conjectures with very large terms.
- We introduce a notion of side-conditions: certain properties can be attached to terms and hypotheses can be relieved without any backchaining.

Overview

1. Mechanism of Side-Conditions
2. Experiments and Examples
3. Verification of RP-Rewriter
4. Applications

Side-Conditions

Side-conditions can **r**etain **p**roperties about terms. For example:

```
(rp 'integerp (f1 x (rp 'booleanp (f2 y))))
```

- `rp` is an identity function where `(rp prop term)` = `term`
- The rewriter knows that `(f2 y)` and `(f1 x (f2 y))` satisfy `booleanp` and `integerp`, respectively. These are called side-conditions.
- `rp` instances can be introduced using rewrite or meta (for more advanced users) rules.
- Side-conditions can prevent excessive backchaining.

Attaching Side-Conditions

Consider the `logand` and `4vec-bitand` functions. `logand` can be rewritten to `4vec-bitand`.

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

After the events above, RP-Rewriter will have this rewrite rule:

```
(implies (and (integerp x) (integerp y))
         (equal (logand x y)
                (rp 'integerp (4vec-bitand x y))))
```

Users may never introduce an `rp` instance explicitly in a rewrite rule.

Relieving Hypotheses with Side-Conditions

Assume we have a tree of `logand` calls to be rewritten with `logand-to-4vec-bitand`:

```
(logand (logand (iget 0 e) (iget 1 e))
        (logand (iget 2 e) (iget 3 e)))
```

where `(iget i e)` = `(ifix (cdr (assoc i e)))`.

ACL2 will rewrite:

- the inner two `logand` instances to `(4vec-bitand (iget ...) ...)` (`integerp-of-iget` will be used 4 times.);
- then, the outer `logand` instance to `(4vec-bitand (4vec-bitand ...) (4vec-bitand ...))` (It will backchain and use `integerp-of-4vec-bitand` 2 times and `integerp-of-iget` 4 times again.).

Relieving Hypotheses with Side-Conditions (cntd.)

On the other hand, RP-Rewriter will rewrite:

- the inner two `logand` instances to

  `(rp 'integerp (4vec-bitand (iget ...) ...)`

  (`integerp-of-iget` will be used 4 times);
- then, the outer `logand` instance to

  `(rp 'integerp (4vec-bitand (rp 'integerp ...) ...))`

  (It will **not** backchain. Instead, it will use the attached side conditions.).

The final term from RP-Rewriter:

```
(rp 'integerp
    (4vec-bitand (rp 'integerp (4vec-bitand (iget 0 e)
                                            (iget 1 e)))
                 (rp 'integerp (4vec-bitand (iget 2 e)
                                            (iget 3 e)))))
```

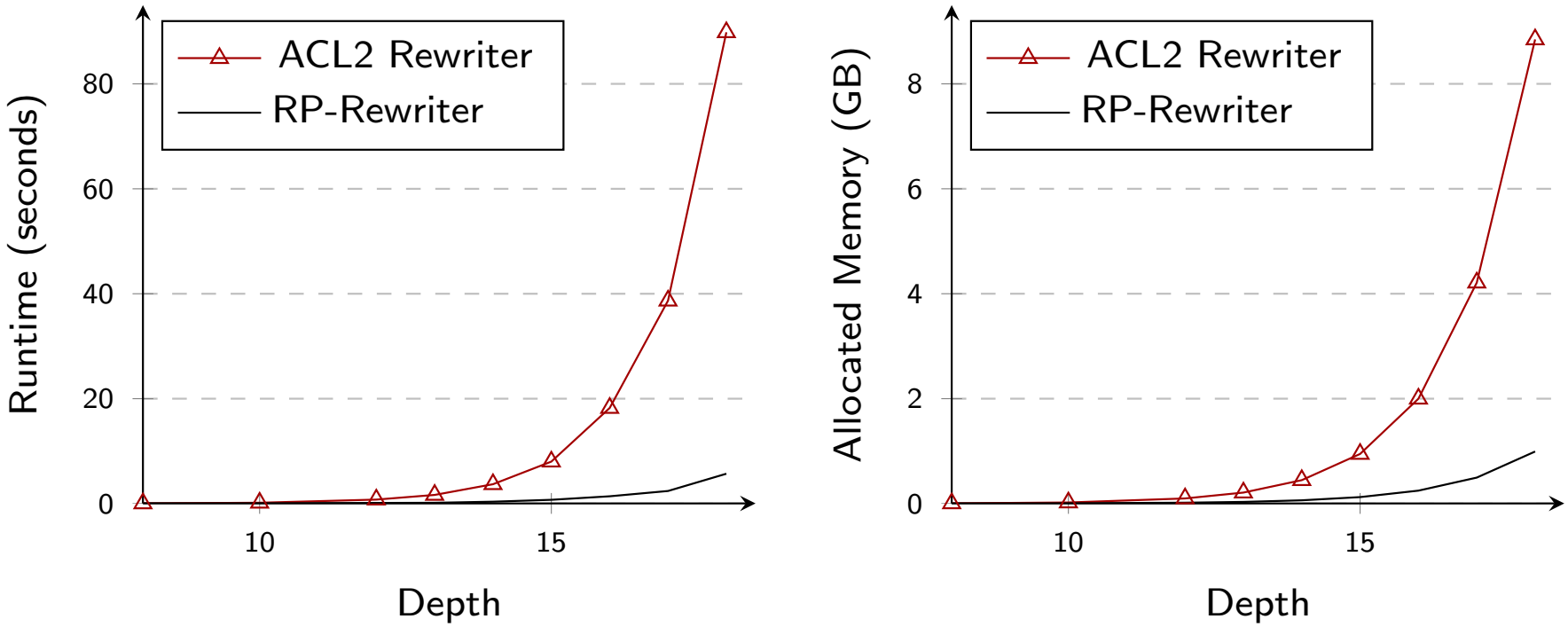
Experiments with Side-Conditions

Let's test ACL2's rewriter and RP-Rewriter on such `logand` trees and prove:

```
(thm
 (equal (4vec-bitand (4vec-bitand (4vec-bitand (iget 0 e) ...)
                                  (4vec-bitand (iget 2 e) ...))
                     ...)
        (logand ... ...)))
```

Increase the tree depth and compare the results.

Experiments with Side-Conditions (cntd.)



Performance comparison of ACL2's built-in rewriter and RP-Rewriter on a conjecture with a term tree of 4vec-bitand and logapp functions.

Round-to-Even Example

Side-conditions can help prove conjectures that the built-in rewriter fails. Consider the following events.

```
(defund d2      (x) (/ x 2))
(defund f2      (x) (floor x 2))
(defund neg-m2 (x) (- (mod x 2)))
(def-rp-rule d2-is-f2-when-even
  (implies (evenp x)
           (equal (d2 x) (f2 x))))
(defun round-to-even (a)  ;; e.g., (round-to-even 93/10) = 8
  (+ a (neg-m2 a)))
(add-rp-rule round-to-even) ;; RP-Rewriter saves the def. rule
(defthmd round-to-even-is-even
  (evenp (+ a (neg-m2 a))))
(rp-attach-sc round-to-even
              round-to-even-is-even)
```

Also assume there are rules about commutativity and associativity of `+`.

Round-to-Even Example (cntd.)

Submitting the event below will fail

```
(defthm three-round-to-evens
  (equal (d2 (+ (round-to-even a) (round-to-even b)))
         (f2 (+ (neg-m2 a) (neg-m2 b) a b))))
```

because:

1. ACL2 will rewrite the `round-to-even` instances, and the LHS will become:

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

2. Using associativity and commutativity of `+`, this will become:

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

3. The lemma `d2-is-f2-when-even` will fail because ACL2 cannot prove that this argument of `d2` is `evenp`.

Round-to-Even Example (cntd.)

On the other hand, submitting the event below will succeed

```
(defthm three-round-to-evens
  (equal (d2 (+ (round-to-even a) (round-to-even b)))
         (f2 (+ (neg-m2 a) (neg-m2 b) a b)))
  :hints (("Goal" :clause-processor (rp-rewriter clause ...))))
```

because:

1. RP-Rewriter will rewrite the `round-to-even` instances, and the LHS will become:

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

2. Using associativity and commutativity of `+`, this will become:

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

3. The lemma `d2-is-f2-when-even` will apply because RP-Rewriter can prove that this argument of `d2` is `evenp`.

Verification of RP-Rewriter

Main RP-Rewriter functions have proofs with the following functions.

- A generic evaluator `(rp-evl term a)` (created with `defevaluator`)
- A special validity evaluator for side-conditions: `(valid-sc term a)`.
  - Returns `t` or `nil`
  - Checks for `rp` (designates a side-condition) and `if` (designates a context change) instances.
- A special syntax checker `(rp-termp term)`
  - Similar to `pseudo-termp` but has more constraints (e.g., lambda expressions are not allowed)
  - Helps some of the proofs.
  - Defines some invariants.

Also, `meta-extract` functions are used to retrieve the rewrite rules from ACL2's world, and run executable-counterparts.

Applications

1. A rewrite-based integer multiplier verification tool.
   - Uses RP-Rewriter, and rewrites and simplifies large multiplier designs.
   - Depends on the side-condition feature similar to the `round-to-even` problem.
   - Verifies various 64x64 multipliers automatically in 2 seconds, and 1024x1024 multipliers in 10 minutes.
2. `svex-simplify`.
   - `SVEX` is a special expression type for Verilog designs as parsed by books/centaur/sv.
   - RP-Rewriter is called as a regular function to simplify SVEX expressions using regular rewrite rules.

Conclusion

- A verified rewriter with a new side-conditions feature is introduced. (books/projects/rp-rewriter)
- Side-conditions feature can have performance benefits, or help prove conjectures that ACL2's built-in rewriter may fail.
- Future work:
  - Support for lambda expressions.
  - Support for outside-to-inside rewriting on demand.

```
(equal (bitmask very-large-term mask)
       much-smaller-term)
```

The End