RP-Rewriter: A Customized, Knowledge Preserving Rewriter to Reduce Backchaining

Mertcan Temel

03/29/2019

Outline

1. Introduction and Features
2. Demo
3. Correctness Proofs of the Rewriter
4. Dynamic Proofs for Clause Processor
5. Test Results

Introduction

- This project started because backchaining was becoming a huge problem in multiplier verification

- Maintaining the type information of some terms would solve that problem but ACL2's rewriter does not support that.

- Designed as a clause processor, rp-rewriter uses the rewrite rules in ACL2's world to reduce terms that represent theorems to 't.

- Rp-rewriter is targeted to be used for fast verification of theorems involving big terms such as multipliers, and it has 2 extra features: *side-conditions* and *fast-alist support*.

Features: Side Conditions

User can attach type information to terms (called *side conditions*)

- We use an identity function **rp** to store side conditions

  Logically (rp ‘prop x) = x, but an invariance (prop x) is maintained

- Side conditions can be introduced by rewrite rules. For example:

```
(defthm binary-and-is-*
    (implies (and (bitp x) (bitp y))
             (equal (binary-and x y)
                    (rp 'bitp (* x y)))))
```

- All the terms introduced by the above rewrite rule will preserve the knowledge that `(bitp (* x y))`, and won’t try to backchain if a lemma requires that knowledge in its hypotheses.

- The rewriter will store the term with the rp wrapper but ignore it when performing rule LHS - term matching

Features: Side Conditions (continued)

Theorems involving side-conditions can be submitted using defthm-rp

```
(defthm-rp binary-and-is-*
    (implies (and (bitp x) (bitp y))
             (equal (binary-and x y)
                    (rp 'bitp (* x y)))))
```

The above term translates to:

```
(progn (defthm binary-and-is-*-rp-side-cond
                (implies (and (bitp x) (bitp y))
                         (and (bitp (* x y)))))
        (defthm binary-and-is-*
                (implies (and (bitp x) (bitp y))
                         (equal (binary-and x y)
                                (rp 'bitp (* x y))))))
```

Features: Fast-alists

If a theorem is expected to have big alists, user can hint the customized rewriter to store them as fast-alists, which can improve the performance significantly.

- We use an identity function **falist** to store the corresponding fast-list.

  Logically (falist fast term) = term, but an invariance (falist-consistent fast term) is maintained.

- “fast” in (falist fast term) is a quoted fast alist. For example:

```
fast = '(('a . x)
         ('b . (fn1 y z)))
```

```
term = (cons (cons 'a x)
             (cons (cons 'b (fn1 y z))
                   'nil))
```

- If user defines their alists using hans-acons in their theorems, the customized rewriter will know to store and look them up using this feature.

Features: Fast-alists (continued)

For example, assume we defined this function:

```
(defun append-to-alist (keys vals alist)
     (if (atom keys)
          alist
       (cons (cons (car keys)
                   (if (consp vals) (car vals) nil))
             (append-to-alist (cdr keys) (cdr vals)
                              alist))))
```

We can have this rule to trigger this special feature of the rewriter:

```
(defthm append-to-alist-def-hons
     (equal (append-to-alist (cons a b) vals alist)
            (hons-acons a (car vals)
                        (append-to-alist b (cdr vals)
                                         alist))))
```

Demo

I will only show and example for the side-conditions feature, but not for the fast-alist feature.

… Now switching to Emacs

Correctness Proofs

Correctness proofs of the rewriter are complete with the following final theorem:

```
(defthm rp-rw-aux-is-correct
    (implies (and (valid-term-syntaxp term)
                  (symbol-alistp exc-rules)
                  (alistp a)
                  (rp-evl-meta-extract-global-facts :state state)
                  (valid-rules-alistp rules-alist))
             (iff (rp-evl
                   (mv-nth 0 (rp-rw-aux term rules-alist exc-rules rp-stat state)) a)
                  (rp-evl term a))))
```

where 1. `term` is the term being rewritten

2. `exc-rules` is the list of names for enabled executable counter-part rules (it is a fast-alist for efficient look-up with ignorable entries)

3. `rules-alist` is a fast-alist for rewrite rules (keys are function names, entries are rule lists for those function names)

4. `rp-stat` is used to collect statistics and irrelevant to the correctness proofs

Correctness Proofs (cntd.)

Some of the prominent lemmas for the final correctness proofs pertain:

- Invariance for syntax: All functions should return terms that satisfy:

```
(and (pseudo-termp2 term)
     (rp-syntaxp term)
     (all-falist-consistent term))
```

- Invariance for side-conditions: All functions should return terms that satisfy:

```
(valid-sc term a)
```

- The function `rp-match-lhs` should return valid bindings such that when applied to the LHS of the rule, the result should be equivalent to the term.

This part of the proof was particularly challenging because of the case of rules with repeating variables. Due to the side-conditions feature, terms for the same variable should be under the same context and be equivalent with a special relation `rp-equal`

Dynamic Proofs for Clause Processor

- We need to know that the rules that will be applied with their side-conditions are indeed correct, so we need to validate them every time we need to use the clause processor of the rewriter.

- Main function of the rewriter is `rp-rw-aux`. The correctness lemma of this function is:

```
(defthm rp-rw-aux-is-correct
    (implies (and (valid-term-syntaxp term)
                  (symbol-alistp exc-rules)
                  (alistp a)
                  (rp-evl-meta-extract-global-facts :state state)
                  (valid-rules-alistp rules-alist))
             (iff (rp-evl
                   (mv-nth 0 (rp-rw-aux term rules-alist exc-rules rp-stat state)) a)
                  (rp-evl term a))))
```

It is not trivial to relieve **`(valid-rules-alistp rules-alist)`**

Dynamic Proofs for Clause Processor (cntd.)

- We need to relieve **`(valid-rules-alistp rules-alist)`**
- `valid-rules-alistp` calls `valid-rulep-sk` for every given rule.

```
(defun-sk valid-rulep-sk (rule)
  (forall a
          (implies (rp-evl (rp-hyp rule) a)
                   (and (if (rp-iff-flag rule)
                            (iff (rp-evl (rp-lhs rule) a)
                                 (rp-evl (rp-rhs rule) a))
                          (equal (rp-evl (rp-lhs rule) a)
                                 (rp-evl (rp-rhs rule) a)))
                        (implies (include-fnc (rp-rhs rule) 'rp)
                                 (valid-sc (rp-rhs rule) a))))))
```

Problem: The evaluator `rp-evl` may not cover all the function symbols in the rules.
Solution: 1. Create a new evaluator when the clause processor is about to be used.
2. Define copies of functions to use the new evaluator.
3. Using functional instantiation, prove a copy of `rp-rw-aux-is-correct` for the new evaluator.

Dynamic Proofs for Clause Processor (cntd.)

Problem:
Defining a new evaluator and validating the correctness of the rules from scracth can be costly if done every time the clause processor for the rp-rewriter is used.

Solution:
- Remember the most recent defined evaluator and validated rules (that can be store in a user-managed table.)

- Define a new evaluator only when necessary, try to validate only the new rules since last time.

Test Results

- This rewriter will be first used for verification of fast integer multipliers. For theorems some of which ACL2's rewriter cannot even finish, rp-rewriter gives favorable results:

I/O bits	Runtime (seconds)	
	ACL2's rewriter	RP-Rewriter
64/128 (v1)	2.92	0.83
128/256 (v1)	33.33	6.04
64/128 (v2)	-	0.80
128/256 (v2)	-	5.80

- These results are for radix-4 Booth encoded Dadda Multipliers (on SBCL on my laptop).
- **v2** is a multiplier verification mechanism that supports multipliers with parallel prefix *adders* where **v1** does not.
- **v2** takes advantage of the side-conditions feature of the rp-rewriter

Conclusion & Future Work

- Correctness are complete and the mechanism for the dynamic proofs of clause processor will be done in a week.

- The customized rewriter is working better than expected for the multiplier proofs. Will look for other use cases that show its advantage.

- More basic features can be added to the rewriter as needed such as some heuristics for definition rules or basic type-reasoning

- Will create a documentation and put the rewriter in ACL2 books.