New Rewriter Features in FGL

Sol Swords
Centaur Technology, Inc.

ACL2 Workshop 2020

Paper: http://acl2-2020.info/papers/new-rewriter-features.pdf

What is FGL?

Bit-blasting framework, successor to GL. Core of FGL is a rewriter inspired by ACL2's.

Compared to GL:

- Uses rewriter as its core; basic behavior is coded as rewrite/meta rules rather than built in
- Supports incremental SAT
- Improved debugging features

Compared to ACL2 rewriter:

- No linear, nonlinear, typeset reasoning
- Bit blasting emphasis: creates Boolean variables from unresolved IF tests

Verified clause processor, modulo trust tags for (e.g.) external SAT solver integrations.

Challenge

Replace hand-coded, meta-level “primitives” with rewrite rules

Make it easy to code custom behavior for new functions and types

Make it easy for rewrite rules to use syntactic and heuristic information

→ Make rewrite rules as powerful as meta rules!

The kind of rewrite rule I want

```
(equal (foo x y)
       (let ((sauce (bar x)))
         (if (syntactically-true-p sauce)
             (fuz y)
           (let ((crust (biz x y)))
             (prog2$
               (cw “Crust term: ~x0~%” (syntactic-term crust))
               (if (syntactically-posp (deliciosity crust))
                   (buz crust y)
                 (do-not-apply-this-rule-after-all)))))))
```

Reasons we can't do this in the ACL2 rewriter

- Can't examine term syntax from the RHS of a rule
- Can't abort a rule application from within its RHS
- Can't derive semantic information from syntactic checks
  - Need `(syntactically-posp x)` to imply `(posp x)`
  - Need `(syntactically-true-p x)` to imply `x`

Outline: Two tweaks and one feature

- Tweak 1: `unequiv` -- “anything goes” equivalence relation
- Tweak 2: `abort-rewrite` -- like it says
- Feature: Binder rules
    - Short summary here, details in the paper

`unequiv`

```
(defun unequiv (x y)
  (declare (xargs :guard t))
  t)
```

- This is an equivalence relation!
- All objects are `unequiv` to each other
- *You can do anything you want* when rewriting under an `unequiv` context

Entering an `unequiv` context

Congruence rules can induce an `unequiv` context on irrelevant arguments:

```
(defun fgl-prog2 (x y) y)
(defcong unequiv equal (fgl-prog2 x y) 1)
```

Sneak peek ahead to binder rules:

```
(defun bind-var (x y) x)
(defcong unequiv equal (bind-var x y) 2)
```

Binds free variable `x` to the result of rewriting `y` under `unequiv` context.

Propagating the `unequiv` context

When a function or lambda call is in an `unequiv` context, its arguments are too.

As if we had

```
(defcong unequiv unequiv (f …) n)
```

For all functions `f`, argument indices `n` (plus special support for lambdas).

This is the only special support needed in the rewriter. Would likely be easy to add to ACL2.

Allowed under `unequiv` context

- Rewrite anything to anything else.
  - Using plain rewrite or meta rules with `unequiv` as the equivalence relation -- trivial proof obligation
  - Application: Arbitrary, extralogical debugging/exploration routines
- `Syntax-interp`: access syntax of terms as in `syntaxp`/`bind-free`
  - Print/analyze/debug results from rewriting
- `Fgl-interp-obj`: if argument rewrites to a quoted term, then rewrite that term
- Other future special forms?

abort-rewrite

Just like the name. Logically an identity function, to make it easy to prove rules that use it. Typical usage:

```
(equiv lhs
       (cond (test1 rhs1)
             (test2 rhs2)
             (t     (abort-rewrite lhs))))
```

Binder rules (sketch)

- When applying a rewrite rule, free variables may be soundly bound to anything.
  - Due to the fact that you can instantiate a theorem with any substitution.
  - `(bind-var x y)` returns x, binds (free variable) x to result of rewriting y under `unequiv` context
- Problem: when proving the theorem justifying the rule, free variables are just free variables
  - Similar to how `(syntaxp (syntactically-integerp x))` doesn't imply `(integerp x)`
  - `(bind-var x (make-constant-integer))` is still logically just `x`, about which we know nothing

Solution

E.g.: Instead of `bind-var`, use `(bind-positive-int x y)`

- Logically: x is a free variable, `(bind-positive-int x y)` defined to be `(pos-fix x)`
- *Binder rule* determines how x will be bound when rewriting
- Rule shows that we can select an x that `bind-positive-int` will return unchanged:

```
(implies (equal x (choose-positive-int y))
         (equal (bind-positive-int x y)
                x)
```

Effectively: *If we choose* x *to be the result of rewriting* `(choose-positive-int y)`, *then* `bind-positive-int` *will return* x *unchanged*.

Original example, in FGL

```
(equal (foo x y)
       (let ((sauce (bar x)))
         (if (bind-syntac-true-p sauce-provable sauce)
             (fuz y)
           (let ((crust (biz x y)))
             (fgl-prog2
               (syntax-interp (cw “Crust term: ~x0~%” crust))
               (if (bind-syntac-posp crust-delishp (deliciosity crust))
                   (buz crust y)
                 (abort-rewrite (foo x y))))))))
```

```
(def-fgl-rewrite fgl-equal
  (equal (equal x y)
         (cond ((check-integerp x-intp x)
                (cond ((check-integerp y-intp y)
                       (and (iff (intcar x) (intcar y))
                            (or (and (check-int-endp x-endp x)
                                     (check-int-endp y-endp y))
                                (equal (intcdr x) (intcdr y)))))
                      ((check-non-integerp y-non-intp y) nil)
                      (t (abort-rewrite (equal x y)))))
               ((check-booleanp x-boolp x)
                (cond ((check-booleanp y-boolp y)
                       (iff x y))
                      ((check-non-booleanp y-non-boolp y) nil)
                      (t (abort-rewrite (equal x y)))))
               ((check-consp x-consp x)
                (cond ((check-consp y-consp y)
                       (and (equal (car x) (car y))
                            (equal (cdr x) (cdr y))))
                      ((check-non-consp y-non-consp y) nil)
                      (t (abort-rewrite (equal x y)))))
               ((and (check-integerp y-intp y)
                     (check-non-integerp x-non-intp x)) nil)
               ((and (check-booleanp y-boolp y)
                     (check-non-booleanp x-non-boolp x)) nil)
               ((and (check-consp y-consp y)
                     (check-non-consp x-non-consp x)) nil)
               (t (abort-rewrite (equal x y))))))
```

In the paper

- More explanation of binder rules
- Implementation problems and solutions
- What would it take to port these features to the ACL2 rewriter?