New Rewriter Features in FGL

Sol Swords
Centaur Technology, Inc.
sswords@centtech.com

FGL is a successor to GL, a proof procedure for ACL2 that allows complicated finitary conjectures to be translated into efficient Boolean function representations and proved using SAT solvers. A primary focus of FGL is to allow greater programmability using rewrite rules. While the FGL rewriter is modeled on ACL2's rewriter, we have added several features in order to make rewrite rules more powerful. A particular focus is to make it more convenient for rewrite rules to use information from the syntactic domain, allowing them to replace built-in primitives and meta rules in many cases. Since it is easier to write, maintain, and prove the soundness of rewrite rules than to do the same for rules programmed at the syntactic level, these features help make it feasible for users to precisely program the behavior or the rewriter. We describe the new features that FGL's rewriter implements, discuss the solutions to some technical problems that we encountered in their implementation, and assess the feasibility of adding these features to the ACL2 rewriter.

1 Introduction

FGL is a bitblasting framework for ACL2 and a successor to GL [6,8]. Its aims are similar to those of GL: to allow finitary propositions written in idiomatic ACL2 to be solved using Boolean reasoning techniques, including state-of-the-art SAT solving and circuit-based simplification. While GL approached this goal largely by including specialized routines for symbolically simulating a large set of primitive functions, FGL allows more user customization of its behavior. FGL still allows such specialized routines in the form of metafunctions, but it has replaced many of these with rewrite rules, which are easier for users to create, enable, and disable than the specialized primitive functions of GL. FGL also supports the use of incremental SAT to solve sequences of satisfiability queries while preserving the lemmas learned by the SAT solver from previous queries. A further design goal of FGL is to allow sophisticated reasoning routines built around incremental SAT to be programmed using rewrite rules.

With these goals in mind, FGL's rewriter includes several new features that are not present in either the GL rewriter or ACL2's rewriter. These new features allow rewrite rules to program the rewriter more accurately and efficiently without resorting to the use of complicated meta rules. We are assured of the soundness of these techniques because we have proved FGL's correctness as a verified clause processor [4]. The full FGL sources, including its soundness proof, are available in the ACL2 community books [7]. This paper describes these new rewriter features, comparing them with existing features of the ACL2 rewriter.

We begin by reviewing conditional rewriting as implemented in ACL2 in Section 2. In Section 3 we describe the new rewriter features introduced in FGL. In Section 4 we discuss two technical problems that we encountered in adding these features, and their solutions. In Section 5 we assess the practical feasibility of adding these features to the ACL2 rewriter.

G. Passmore, R. Gamboa (Eds.): ACL2 Workshop 2020
EPTCS 327, 2020, pp. 32–46, doi:10.4204/EPTCS.327.3

2 Basic Rewriting

ACL2, GL, and FGL each use an inside-out rewriter that takes as input a term and a subsitution alist. ACL2's rewriter returns a new term, whereas GL's and FGL's return a *symbolic object* of a hybrid format that incorporates both termlike constructs (variables and function calls) and references to Boolean function objects. We use *result objects* to refer generically to terms for the ACL2 rewriter and symbolic objects for GL/FGL. The input substitution alist σ maps variables to result objects that are considered not to need further rewriting. Each of the rewriters operates basically as follows, eliding many important details:

- If the input term is a quote, then return the quotation of its value as a result object.
- If a variable, return its binding from the substitution alist.
- If a lambda application, recursively rewrite the actuals, then create a new substitution by pairing the formals with the results from rewriting the actuals and recursively rewrite the body with that substitution.
- Otherwise, a function call: recursively rewrite the arguments and then create the result object representing the call of the function on the rewritten arguments. Apply rewrite rules to this object; if any rule succeeds, return its result, otherwise return the function call object.

The correctness contract of such a rewriter is essentially the following equation:

$$\mathrm{Ev}\,(in \setminus \sigma, env) \equiv \mathrm{Ev}\,(out, env)$$

Here Ev is an evaluator for result objects—terms for the ACL2 rewriter, symbolic objects for the GL/FGL rewriters. The notation $x\setminus\sigma$ denotes applying substitution σ, a mapping from variables to result objects, to a term x, producing a result object. The equivalence relation $\equiv$ is an auxiliary input to each rewriter and is modified according to congruence rules as the rewriters recur over terms.

In the rest of this paper we'll sometimes abuse notation by eliding the evaluation operator and environment in equations like the one above. That is, if we say $x \equiv y$, where contextually x and y are either both result objects or both terms, we really mean for all env, $\mathrm{Ev}\,(x, env) \equiv \mathrm{Ev}\,(y, env)$, where Ev is a result object evaluator or term evaluator as appropriate. In this notation we can state our rewriter correctness contract as simply

$$in \setminus \sigma \equiv out\,.$$

The rewrite rules applied in the process described above are justified by theorems of the form

$$hyps \Rightarrow lhs \equiv_R rhs$$

where *hyps*, *lhs*, and *rhs* are terms and $\equiv_R$ is some equivalence relation. Applying such a rule to a result object x is done using the following steps:

- Check that the equivalence $\equiv_R$ of the rule is a refinement of the current equivalence context $\equiv$ of the rewriter; that is, $a \equiv_R b$ implies $a \equiv b$.
- Try to find a substitution σ for which $x = lhs \setminus \sigma$. This is done using a unification algorithm such as ACL2's `one-way-unify`.
- Check that the hypotheses can be proved for the substitution σ under the current set of assumptions. Usually this check is done by backchaining, that is, recursively rewriting each of the hypotheses under the substitution.

- Rewrite *rhs* under substitution σ and equivalence relation $\equiv$ to obtain a result *res* for which $res \equiv rhs\backslash\sigma$, and return *res* as the replacement for *x*.

This last step is justified by the fact that, if all the previous steps were successful, then instantiation of the rewrite rule theorem by σ produces $x \equiv_R rhs\backslash\sigma$, and since $rhs\backslash\sigma \equiv res$, therefore $x \equiv res$.

Note that the substitution σ is simply derived from unifying the left-hand side of the rule with the target term. However, this will only bind variables that appear in *lhs*. For any other variables, the substitution could be extended with any assignment as long as it makes the hypotheses true. ACL2 takes advantage of this with its `bind-free` feature, which FGL replicates. Furthermore, FGL's most powerful rewriting features are extensions of this capability, allowing free variables to be bound in more contexts and with more flexible semantics.

Meta rules [2] are another form of rule that can be used for rewriting in ACL2. A meta rule names a function, called a *metafunction*, that can be used to syntactically transform a term to another equivalent term, and optionally a *hypothesis metafunction* that generates hypotheses that must be relieved in order to apply that metafunction. FGL supports meta rules as well, but its metafunctions are slightly different in that instead of returning only a new term, they return a term and a substitution. FGL's meta rules do not yet support hypothesis metafunctions.

3 FGL Rewriter Features

We first introduce two relatively simple features of the FGL rewriter, and then discuss a more complicated and powerful feature involving the binding of free variables.

3.1 `Unequiv` context

FGL's rewriter supports congruence-based rewriting much like ACL2's rewriter, using congruence rules to determine the equivalence relations that must be maintained on subterms when recurring through the term to be rewritten. (FGL only supports simple congruences, not patterned congruences [3].) A simple extension to congruence-based reasoning is special recognition of the trivial equivalence relation under which all objects are equivalent, which we call `unequiv`[1]. This equivalence relation is special because when rewriting any function or lambda call under `unequiv`, it is sound to also rewrite the arguments under `unequiv`—this is also true of `equal`, but not true of other equivalence relations. There are only two rewriter changes necessary to support `unequiv`: to automatically propagate the `unequiv` context into function and lambda arguments, and to recognize that all equivalence relations are refinements of `unequiv`.

Under an `unequiv` context, it is permissible to replace any term with any result. A rewrite rule with `unequiv` as its equivalence relation effectively has no proof obligation, and as such can be used to program arbitrary routines into the FGL rewriter. FGL also allows certain special features under an `unequiv` context that would be unsound otherwise:

- `syntax-interp` evaluates its argument term as in ACL2's `syntaxp` or `bind-free`, that is, under the assignment of each variable to the result object bound to it in the current substitution.
- `fgl-interp-obj` rewrites its argument term as usual, then if its result is the quotation of a term, it calls the rewriter on that term.

[1] We chose this name as both an abbreviation for "universal equivalence" and because if two objects are said to be *unequiv*, we think it suggests that they're not necessarily equivalent, but not necessarily inequivalent either.

- `assume` rewrites its first argument, then assumes that result to be true while rewriting its second argument.

Users can prove congruence rules that induce an `unequiv` context on any function argument that is irrelevant to the value of that function. For example, we define `(fgl-prog2 x y)` = `y`, and provide a congruence rule that induces an `unequiv` context on the first argument of `fgl-prog2`. This can be used to perform extralogical analyses and print results as a side effect. We also provide a utility `(bind-var x y)` which must occur with *x* a free variable; this rewrites *y* under an `unequiv` context and binds *x* to its result. This is often used in combination with `syntax-interp` to obtain `bind-free`-like functionality that can be used in the midst of a rule's right-hand side as well as in the hypotheses.

3.2 `abort-rewrite`

It is always sound to decide not to apply a rule. FGL supports a special identity function `abort-rewrite` which causes the rewriter to abort the current rule attempt whenever it is encountered. This allows rules to be programmed such that while rewriting the right-hand side, some condition may cause the rule not to be applied after all. The logical definition of `abort-rewrite` is an identity function so that authors of such rules may wrap it around whatever term is most convenient for proving the rule correct. The wrapped term will not be rewritten when applying the rule; instead, the application of the rewrite rule will be aborted.

An example of the use of `abort-rewrite` is shown in Listing 1, which shows a rewrite rule that can be used for resolving many calls of `equal`. In this rule, functions such as `check-integerp` are binder functions, discussed in Section 3.3, and calls such as `(check-integerp x-intp x)` return true if `x` is syntactically known to be an integer. This rule uses `abort-rewrite` in several situations where it doesn't know how to resolve the equality of the inputs. In such cases, application of this rule fails, but the target object may still be rewritten by other rules.

3.3 Free Variable Binding

The choice of bindings for variables not present in the left-hand side of a rewrite rule is a powerful tool. Since free variables can be bound to anything, they can be bound based on extralogical considerations such as the syntax of the term being rewritten. This flexibility can help give rewrite rules the power of metafunctions without needing to program them wholly at the syntactic level.

Like the ACL2 rewriter, the FGL rewriter supports `bind-free` hypotheses [2]. These allow arbitrary bindings to computed based on term syntax and added to the substitution that will be used in applying the rule. Both rewriters also support free variable bindings based on equivalence hypotheses `(equiv var term)`, described in the ACL2 documentation topic `free-variables` [1]. FGL also adds a new way of binding free variables using *binder rules*. Binder rules may be used to bind free variables in the right-hand side of rules as well as the hypotheses. They provide wide flexibility in the strategies used to choose the bindings, from rewriting (as in equivalence-based binding hypotheses) to syntactic interpretation as in `bind-free`. In fact, the `bind-var` utility described in Section 3.1 is implemented using a binder rule. Finally, they also allow facts about how the variable will be bound to be used in the logical justification for the rewrite rules in which they occur.

The rule `fgl-equal` shown in Listing 1 provides a basic example of the kind of rewriter programming that may be done with binder rules. Each of the functions prefixed `check-` is a binder function which effectively checks whether the argument `x` or `y` satisfies some syntactic criteria; for example,

Listing 1: Example rule using `abort-rewrite` and binder functions

```
(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))))))
```

`(check-integerp x-intp x)` returns `t` if `x` is syntactically known to be an integer. This is accomplished by binding the free variable `x-intp` to the result of the syntactic check. To do this in a rule written for the ACL2 rewriter, one would need to perform all of these free variable bindings using `bind-free` hypotheses, rather than as needed in the right-hand side. It is also known in the logic that `(check-integerp x-intp x)` implies that `x` is an integer. In contrast, `bind-free` doesn't provide any information in the logic about the free variables that it binds, so the result of a syntactic check that `x` was an integer would need to be paired with a symbolic check such as `(integerp x)`, resolved by further rewriting.

The fact that `bind-free` and the related utility `syntaxp` give no information in the logic about the syntactic computation performed leads to awkward usage patterns. Often we know that a term satisfies some property, either by a syntax check or by construction, but we still must add a hypothesis checking that property or we won't be able to prove the rule. Here are two examples from the "rtl/rel9" library of the ACL2 community books [1, 5], preceded by the definition of `power2p` used therein:

```
(defund power2p (x)
  (declare (xargs ...))
  (cond ((or (not (rationalp x))
             (<= x 0))
         nil)
        ((< x 1) (power2p (* 2 x)))
        ((<= 2 x) (power2p (* 1/2 x)))
        ((equal x 1) t)
        (t nil) ;got a number in the doubly-open interval (1,2)
        ))

(defthm power2p-shift-2
  (implies (and (syntaxp (power2-syntaxp y))
                ;this should be true if the syntaxp hyp is satisfied
                (force (power2p y)))
           (equal (power2p (* x y))
                  (power2p x))))

(defthm expo-shift-general
  (implies (and (bind-free (bind-k-to-common-expt-factors x) (k))
                ...
                (force (power2p k))
                ...
                )
           (equal (expo x)
                  (+ (expo k) (expo (* (/ k) x))))))
```

In both theorems the `(force (power2p` v`))` hypothesis is checking something that is already known. The syntax check `power2-syntaxp` provably is only true of terms whose evaluation satisfies `power2p`, and the binding function `bind-k-to-common-expt-factors` provably will only bind the variable `k` to a term satisfying `power2-syntaxp`. However, the author of these rules couldn't use these facts when proving the theorems justifying them, so the forced `power2p` hypotheses were used instead. When these rules are applied, these redundant hypotheses must be relieved using rewriting.

Arguably, such a redundant check is a small price to pay to be able to use syntactic checks and arbitrary free variable bindings. In many cases the checks can likely be optimized so that they only need to repeat a small amount of work. However, even the single rule `power2p-shift-2` above demonstrates that blowups are possible: note that if this rule is used to prove `power2p` of a product of size n, then `power2-syntaxp` will run $\Theta(n^2)$ times since each top-level call recurs through the whole product.

Listing 2: Binder function examples

```
;; No restrictions
(defun bind-var (var x)
  var)

;; True only if x is true
(defun syntactically-true (var x)
  (and x var))

;; Upper bound for (integer-length x) if nonnil
(defun integer-length-bound (var x)
  (and (integerp var)
       (<= (integer-length x) var)
       var))

;; All elements of x are in either the first or second return value,
;; and all elements of the first return value are in y
(defun-nx split-list-by-membership (var x y)
   (mv-let (part1 part2) var
       (if (and (set-equiv (append part1 part2) x)
                (subsetp-equal part1 y))
           (mv part1 part2)
         (mv nil x))))
```

Without the redundant hypothesis, only one linear check would be necessary.

To help avoid the need for these redundant checks, FGL adds *binder rewrite* and *binder meta* rules. A binder rule of either kind acts on a *binder function*. The binder function expresses the properties that the free variable's eventual binding must satisfy, and the binder rule dictates how the variable will be bound. The binder function's first argument (per our convention) is the free variable, which the function fixes so that it satisfies some property that may depend on the rest of the arguments. That is, if the first argument satisfies the desired properties, it is passed through unchanged; otherwise, some other value that does satisfy the properties is returned instead.

Listing 2 shows several examples of binder functions without their binder rule implementations. First, `bind-var` simply returns the first argument unchanged, so it places no restriction on the binding. The next, `syntactically-true`, could be implemented as a check that `x` is either a constant nonnil value or a call of some function known not to return `nil`—or, at the most conservative, the implementation could always bind the variable to `nil`. Third, `integer-length-bound` produces an upper bound for the integer-length of `x`, or `nil` if none can be found. The final, and most complex, `split-list-by-membership` returns two lists which must satisfy two criteria: the two lists together must be set-equivalent to the input list `x`, and elements of the first list must be members of the second input list `y`. Crucially, the use of a free variable allows the split not to be a function of `x` and `y`. The implementing binder rule could use various levels of effort to check whether elements of `x` are verifiable members of `y` so that it can put them in the first list.

Binder rules resolve calls of binder functions by producing a term that can be consistently used simultaneously as the new binding for the free variable and the replacement for the binder function call. A binder rewrite rule is justified by a theorem of the form

$$hyps \wedge (var \equiv_H form) \Rightarrow f(var, args) \equiv_C var$$

where *hyps* and *form* are terms, *args* is an argument list of zero or more terms, *var* is a variable not present in any of those terms, and f is the target binder function. Applying such a rule to a call $f(v, args')$, with v a free variable, is done using the following steps:

- Check that the equivalence $\equiv_C$ of the rule is a refinement of the rewriter's current equivalence context $\equiv$.
- Try to find a substitution σ for which $args' = args\backslash\sigma$, using one-way unification.
- Relieve the hypotheses under substitution σ by backchaining.
- Rewrite *form* under substitution σ and equivalence $\equiv_H$ to obtain a result *res* for which $res \equiv_H form\backslash\sigma$. Add *res* as the binding for v and return it as the replacement for the call of f.

The last step is justified because, if all the previous steps were successful, then instantiation of the binder rule theorem by $\{var \leftarrow v\} \cup \sigma$ produces

$$(v \equiv_H form\backslash\sigma) \Rightarrow f(v, args') \equiv_C v$$

(using the assumption that *var* does not appear in *hyps*, *form*, or *args* as noted above). Since v is a free variable we may bind it however we want; binding it to *res* ensures that the antecedent of the above implication holds, so that we may replace the call of f with *res* as well.

For the binder function examples above, we'll now describe binder rules that implement the intended checks. The binder rule for `bind-var` is shown in Listing 3; also important in its implementation is a congruence rule which allows rewriting its second argument under `unequiv`. When a `bind-var` form is encountered with its first argument a free variable, the second argument is first rewritten under `unequiv` due to the congruence rule, and the variable is then bound to that result due to the binder rule.

We can implement `syntactically-true` using a pair of binder rewrite rules, as listed in Listing 4. The second rule will be tried before the first, assuming they are submitted in that order. The second handles the case where `x` is known to be true, assigning `var` the value `t`. The first rule applies if the second fails, in which case `var` is assigned `nil`.

For `integer-length-bound`, we could use several rewrite rules to handle different cases like we did with `syntactically-true`. Instead, we show in Listing 5 an implementation that uses one rewrite rule, programmed in a more explicit style. The rule checks three possible cases, based on whether `x` is syntactically a symbolic integer, a concrete (quoted) value, or neither. If it is neither, then it produces `nil` as the new binding. If it is a concrete value, then it returns its exact integer length. For symbolic integers, it checks whether `(int-endp x)` is known to be true—that is, `x` is a non-integer, 0, or -1. If so, then its integer-length is 0. Otherwise, we bound the integer-length of the `logcdr` (right-shift by 1) of `x`, and add 1 to the result if it is non-nil.

This rule could potentially be improved by using a more specialized test for `int-endp`; as is, a full expression for `int-endp` must be computed for each tail of `x`. Another approach to implementing `integer-length-bound` is to use a binder meta rule. A binder meta rule allows a certain metafunction to be used to generate a binding for a binder function call. The metafunction takes the binder function name f and argument objects *args* as input and returns *form* and σ. The theorem justifying a meta rule says that evaluation of these terms always satisfies the binder rewrite rule formula; that is, $form\backslash\sigma$ is always an object that is preserved by the application of f with the given arguments. For `integer-length-bound`, the metafunction could count the bits in the symbolic integer representation directly, rather than doing it by iterative rewriting.

Finally, an implementation of `split-list-by-membership` is shown in Listing 6. This is conceptually similar to the `integer-length-bound` rule, recurring down a list and conservatively crafting a

Listing 3: Bind-var implementation

```
(defcong unequiv equal (bind-var var x) 2)
  (add-fgl-congruence unequiv-implies-equal-bind-var-2)

(def-fgl-brewrite bind-var-binder-rule
   (implies (equal var x)
            (equal (bind-var var x) var)))
```

Listing 4: Syntactically-true implementation

```
(def-fgl-brewrite syntactically-true-binder-rewrite-false
  (implies (equal var nil)
           (equal (syntactically-true var x) var)))

(def-fgl-brewrite syntactically-true-binder-rewrite-true
  (implies (and x (equal var t))
           (equal (syntactically-true var x) var)))
```

Listing 5: Integer-length-bound implementation

```
(def-fgl-brewrite integer-length-bound-binder-rw
  (implies
    (equal var (cond ((bind-var symbolic (syntax-interp
                                          (fgl-object-case x :g-integer)))
                      (if (syntactically-true known-int-endp (int-endp x))
                          0
                        (let ((rest-bound (integer-length-bound
                                           rest-bound (logcdr x))))
                          (and rest-bound (+ 1 rest-bound)))))
                     ((bind-var concrete (syntax-interp
                                          (fgl-object-case x :g-concrete)))
                      (integer-length x))
                     (t nil)))
    (equal (integer-length-bound var x) var)))
```

Listing 6: split-list-by-membership implementation

```
(def-fgl-brewrite split-list-by-membership-binder-rule
  (implies (equal var (if (syntactically-true known-consp (consp x))
                          (mv-let (rest1 rest2)
                            (split-list-by-membership rest-call (cdr x) y)
                            (if (syntactically-true known-member
                                                    (member-equal (car x) y))
                                (mv (cons (car x) rest1) rest2)
                              (mv rest1 (cons (car x) rest2))))
                        (mv nil x)))
           (equal (split-list-by-membership var x y) var)))
```

result that satisfies the restriction imposed by the binder function; we include it to illustrate the variety of types of constraints that can be handled by these rules.

4 Technical Problems and Solutions

The free variable binding features of FGL led to two technical problems. We describe those problems and their solutions in this section.

4.1 Free Variable Binding with Lambdas

One of the design goals for the FGL rewriter was to allow free variables to be bound anywhere in a rewrite rule. The `bind-var` and binder rule features may bind variables in the midst of rewriting a term such as a hypothesis or the right-hand side. Logically, we just require that the variable was not previously bound, and that the binding site of the variable was its first use. However, ACL2's handling of `let`/`lambda` expressions poses a problem. A well-formed lambda term in ACL2 must bind all the free variables of its body. For example, the translation of `(let ((b (b-expr))) (f a b))` will be a call of a lambda that has both `a` and `b` as formals, such as `((lambda (a b) (f a b)) a (b-expr))`. What happens, then, if we want to bind a free variable inside a lambda body? For example, suppose `f` in the `let` expression above was a binder function, and `a` a free variable. Unfortunately, `a` appears in the lambda arguments, which means the rewriter would first encounter it as an argument to the lambda call, not at its binding site.

To work around this problem, we use a different strategy for handling lambdas than the ACL2 rewriter or the usual form of ACL2 evaluator. The usual way is to first process (rewrite or evaluate) the actuals of the lambda call under the current variable bindings (call them σ_0), then pair the lambda formals with the results of processing the actuals to create a new set of variable bindings σ_1, and use these bindings to process the body of the lambda. Instead, before we begin rewriting the actuals, we strip out any self-pairings from the formal/actual pairs, such as the pairing of `a` with itself in the example above. This allows FGL's rewriter to avoid encountering free variables such as `a` before their intended binding sites. We rewrite the remaining actuals under σ_0 and create a variable binding alist σ_2 by pairing the remaining formals with the results. However, instead of using σ_2 by itself when processing the lambda body, we append it to the existing variable bindings and use the combined bindings $\sigma_2 :: \sigma_0$, where the bindings of σ_2 shadow those of σ_0. The critical fact showing the semantic equivalence of these two strategies is that each variable present in the lambda body has the same binding in σ_1 as in $\sigma_2 :: \sigma_0$. In particular, a variable that was formerly self-paired in the lambda call will not be present in σ_2, so its binding in $\sigma_2 :: \sigma_0$ is its binding from σ_0. Its binding in σ_1 is derived by evaluating or rewriting the variable itself under σ_0. Since evaluating or rewriting a variable is done by looking it up in the bindings, these are equivalent.

A further subtlety is that when binding a free variable, its binding can't be local to a lambda, but must be set in the substitution σ of the current rewrite rule application. Therefore the FGL rewriter actually tracks two sets of bindings: σ_u, the unifying substitution plus any free variables that have been bound so far, and σ_λ, the combined bindings from the current nesting of lambdas. A variable's binding is found by looking it up first in σ_λ, then in σ_u if not found.

4.2 Inductive Correctness with Free Variable Bindings

Recall that we stated the correctness contract of a rewriter, in Section 2:

$$\mathrm{Ev}\,(in \setminus \sigma, env) \equiv \mathrm{Ev}\,(out, env)$$

In FGL, the substitution σ is both an input to and output from the rewriter. For the current discussion we will ignore the fact mentioned previously that the full substitution consists of two parts, σ_u and σ_λ; here, σ represents both. In each call of the rewriter, the current substitution σ_i is passed in, and a modified substitution σ_o, potentially with some new free variables bound, is returned. The correctness statement we want has to do with the resulting substitution σ_o:

$$\mathrm{Ev}\,(in \setminus \sigma_o, env) \equiv \mathrm{Ev}\,(out, env)$$

However, as stated this property is not inductive. Consider rewriting a function call of two arguments $f(a,b)$; we'll try and show that the rewriter's treatment of this term is correct assuming inductively that it correctly rewrites a and b. Suppose we start with substitution σ_i. First we rewrite a with this substitution, producing output a' and substitution σ_a. Then then rewrite b with substitution σ_a, producing output b' and subsitution σ_b. Then for simplicity suppose we have no rewrite rules about f, so we just return $f(a',b')$ and substitution σ_b. The facts we may assume inductively are:

$$\mathrm{Ev}\,(a \setminus \sigma_a, env) \equiv \mathrm{Ev}\,(a', env)$$

$$\mathrm{Ev}\,(b \setminus \sigma_b, env) \equiv \mathrm{Ev}\,(b', env)$$

We want to prove:

$$\mathrm{Ev}\,(f(a,b) \setminus \sigma_b, env) \equiv \mathrm{Ev}\,(f(a',b'), env)$$

Basic facts about evaluation and substitution reduce this to:

$$f\left(\mathrm{Ev}\,(a \setminus \sigma_b, env), \mathrm{Ev}\,(b \setminus \sigma_b, env)\right) \equiv f\left(\mathrm{Ev}\,\left(a', env\right), \mathrm{Ev}\,\left(b', env\right)\right)$$

Notice we have an assumption about $\mathrm{Ev}\,(a \setminus \sigma_a, env)$ where we need a fact about $\mathrm{Ev}\,(a \setminus \sigma_b, env)$. With only these induction hypotheses, we are stuck. However, a basic property of the rewriter is that σ_o is an extension of σ_i—for any variable bound in σ_i, it must be bound to the same value in σ_o. We'll denote this using set notation as $\sigma_i \subseteq \sigma_o$. Intuitively, σ_a should bind all the variables that are needed to evaluate a, so that the new variables bound in σ_b don't affect the result. Therefore, the inductive assumption we need is that the evaluation equivalence holds for any extension to the resulting substitution:

$$\forall \sigma_+ : \sigma_o \subseteq \sigma_+ \Rightarrow \mathrm{Ev}\,(in \setminus \sigma_+, env) \equiv \mathrm{Ev}\,(out, env).$$

Trying our proof again, we have inductive assumptions:

$$\forall \sigma_+ : \sigma_a \subseteq \sigma_+ \Rightarrow \mathrm{Ev}\,(a \setminus \sigma_+, env) \equiv \mathrm{Ev}\,(a', env)$$

$$\forall \sigma_+ : \sigma_b \subseteq \sigma_+ \Rightarrow \mathrm{Ev}\,(b \setminus \sigma_+, env) \equiv \mathrm{Ev}\,(b', env)$$

and we want to prove

$$\forall \sigma_+ : \sigma_b \subseteq \sigma_+ \Rightarrow \mathrm{Ev}\,(f(a,b) \setminus \sigma_+, env) \equiv \mathrm{Ev}\,(f(a',b'), env)$$

or equivalently

$$\forall \sigma_+ : \sigma_b \subseteq \sigma_+ \Rightarrow f\left(\mathrm{Ev}\left(a \backslash \sigma_+, env\right), \mathrm{Ev}\left(b \backslash \sigma_+, env\right)\right) \equiv f\left(\mathrm{Ev}\left(a', env\right), \mathrm{Ev}\left(b', env\right)\right).$$

Fortunately, we can apply the transitivity of substitution extension to conclude $\sigma_a \subseteq \sigma_+$ so that this time we may apply the induction hypothesis about a as well as the one about b. Of course, the full proof that the FGL rewriter is correct is beyond the scope of this paper, but may be examined in the ACL2 community book "centaur/fgl/interp.lisp."

5 Porting to the ACL2 Rewriter

If these features of the FGL rewriter are useful, it raises the question of whether they could be ported to the ACL2 rewriter. In this section we try to assess how difficult it would be to add these features, whether they conflict with any other features, etc. These assessments are based on examination of the relevant ACL2 code, and not on any attempt to implement them; therefore, there might be impediments that we didn't foresee.

5.1 `Unequiv` context

Support for `unequiv` would the easiest of these features to add to the ACL2 rewriter. We believe the following changes would be the only ones necessary:

- Set `unequiv` as the equivalence context for all arguments whenever a function or lambda call occurs in `unequiv` context. This could be done by adding a special case to the function `geneqv-lst`, which computes the equivalence contexts for rewriting a function's arguments given the equivalence context in which the function is being rewritten.
- Recognize that any other equivalence relation is a refinement of `unequiv`. This could be done by adding a special case to the function `geneqv-refinementp`, which determines whether an equivalence is a refinement of the current equivalence context.

These changes would suffice for allowing the rewriter to use `unequiv`-based rewrite rules in logically irrelevant contexts. Additional features that are allowed only under `unequiv` context, such as `syntax-interp`, could be considered separately.

5.2 `abort-rewrite`

In order to implement `abort-rewrite`, the call of `rewrite` in the function `rewrite-with-lemma` would need to return a flag saying that the rule application attempt failed. This would affect most of the mutually recursive clique implementing the rewriter, though not in a very deep way. Rewriter functions `rewrite`, `rewrite-if`, `rewrite-args`, `rewrite-with-lemmas`, and `rewrite-fncall` would all need to return the abort flag, and callers of those functions in `rewrite-equal`, `relieve-hyp`, `rewrite-with-lemma`, `rewrite-linear-term`, and `multiply-alists2` would need to deal with the possibility of an abort. Rather than inserting tests in order to exit early after calls that produce an abort, it might be cleaner to pass the abort flag along through the rewriter, checking for it at a convenient point such as the entry to `rewrite`.

As an additional consideration, it would be unfortunate to abort a proof completely due to the presence of `abort-rewrite` in the statement of the conjecture to be proved. This could be dealt with by

adding an input flag to the rewriter (perhaps part of the `rcnst` structure) that says whether `abort-rewrite` calls are to be respected, or by simply stripping out `abort-rewrite` calls from the conjecture before attempting its proof.

5.3 Free Variable Binding

At first glance adding the ability to bind free variables anywhere would seem not to touch any more code than `abort-rewrite`. Essentially it would require passing the unifying substitution out of all the same functions that would need to return the abort flag. One could even get clever and combine the unifying substitution with the abort flag, since the substitution is irrelevant if the rule application is to be aborted.

However, there is another logical issue to work out. When relieving hypotheses, the ACL2 rewriter is careful to ensure that variables aren't used before they're bound in the unifying substitution. But otherwise, it assumes that a variable not bound in the substitution is implicitly bound to itself. It uses the following form to look up variables in the substitution, where `term` is the variable:

```
(let ((temp (assoc-eq term alist)))
  (cond (temp (cdr temp))
        (t term)))
```

In fact, in several places the ACL2 rewriter uses alist `nil` to signify that all variables in the term to be rewritten are bound to themselves. If we allowed free variables to be bound arbitrarily within terms being rewritten, we'd need to know that they hadn't been previously assumed to be bound to themselves. Otherwise, we could prove `nil` by applying something like the following rule:

```
(equal (always-true)
       (equal a (bind-var a (syntax-interp '(not a)))))
```

Additionally, in order to be able to bind variables within `let` or `lambda` expressions, the ACL2 rewriter would need to adopt a similar style of variable binding as we discussed in Subsection 4.1, splitting the variable assignment into the unifying substitution and local lambda bindings and filtering out self-bindings from lambdas.

If it isn't feasible to add the capability of binding free variables within a term, something akin to binder rules could still be pursued for use on top-level hypotheses. Instead of binder functions that fix a free variable so that it complies with some property, we could use binder hypothesis functions that simply describe the properties satisfied by a free variable binding, and binder hypothesis rules that say how to bind those free variables in such a way that the binder hypothesis function is true. An example based on the `split-list-by-membership` binder described in Subection 3.3 is shown in Listing 7.

Adding this capability would require many changes as well. Presumably, binder hypothesis rules would need to have their own rule class, with support for processing appropriate forms of `defthm` as well as support for applying the rules in the rewriter within `relieve-hyp`. Binder hypothesis meta rules would likely be an incremental addition on top of this.

6 Conclusion

FGL's rewriter adds new features that allow a style of metaprogramming via rewrite rules that cannot practically be done with the ACL2 rewriter's current feature set. The new free variable binding capabilities allow syntactic information to be used in directing the rewriter without losing (or needing to re-verify) the semantic information that the syntactic properties imply. Rewrite rules using these capabilities provide an alternative to complicated metafunctions that must be proved correct relative to an

Listing 7: Binder hypothesis rule concept

```
(defun-nx bind-split-list-by-membership (free-var x y)
  (mv-let (part1 part2) free-var
    (and (set-equiv (append part1 part2) x)
         (subsetp-equal part1 y))))

(defthm bind-split-list-by-membership-default
   (implies (equal free-var (mv nil x))
            (bind-split-list-by-membership free-var x y))
   :rule-classes :binder-hyp)

(defthm bind-split-list-by-membership-nonmember
  (implies (and (consp x)
                (bind-split-list-by-membership rest-split (cdr x) y)
                (equal free-var (mv-let (rest1 rest2) rest-split
                                  (mv rest1 (cons (car x) rest2)))))
           (bind-split-list-by-membership free-var x y))
   :rule-classes :binder-hyp)

(defthm bind-split-list-by-membership-member
  (implies (and (consp x)
                (member (car x) y)
                (bind-split-list-by-membership rest-split (cdr x) y)
                (equal free-var (mv-let (rest1 rest2) rest-split
                                  (mv (cons (car x) rest1) rest2))))
           (bind-split-list-by-membership free-var x y))
   :rule-classes :binder-hyp)
```

evaluator. We hope to show in future work that it also provides a platform on which significant proof and analysis routines can be built quickly and easily, using powerful automatic tools such as incremental SAT and Boolean circuit simplifiers.

Acknowledgements

I would like to thank Matt Kaufmann for maintaining ACL2, and more specifically for working out the removal of a restriction involving the use of attachments in metafunctions which has been crucial to the development of this work. I'd also like to thank my colleagues at Centaur Technology, Shilpi Goel, Anna Slobodova, and Rob Sumners, for their help and encouragement in developing FGL.

References

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

[2] Warren A. Hunt Jr., Matt Kaufmann, Robert Bellarmine Krug, J. Strother Moore & Eric Whitman Smith (2005): *Meta Reasoning in ACL2*. In Joe Hurd & Tom Melham, editors: *Theorem Proving in Higher Order Logics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 163–178, doi:10.1007/11541868_11.

[3] Matt Kaufmann & J. Strother Moore (2014): *Rough Diamond: An Extension of Equivalence-Based Rewriting*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving*, Springer International Publishing, Cham, pp. 537–542, doi:10.1007/978-3-319-08970-6_35.

[4] Matt Kaufmann, J Strother Moore, Sandip Ray & Erik Reeber (2009): *Integrating external deduction tools with ACL2*. *Journal of Applied Logic* 7(1), pp. 3 – 25, doi:10.1016/j.jal.2007.07.002. Special Issue: Empirically Successful Computerized Reasoning.

[5] David M. Russinoff (2019): *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, doi:10.1007/978-3-319-95513-1.

[6] Sol Swords (2017): *Term-Level Reasoning in Support of Bit-blasting*. In Anna Slobodova & Warren A. Hunt, Jr., editors: Proceedings 14th International Workshop on the *ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, May 22-23, 2017, *Electronic Proceedings in Theoretical Computer Science* 249, Open Publishing Association, pp. 95–111, doi:10.4204/EPTCS.249.7.

[7] Sol Swords (Accessed: 2020): *FGL source distribution*. Available at `https://github.com/acl2/acl2/tree/master/books/centaur/fgl`.

[8] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In David Hardin & Julien Schmaltz, editors: Proceedings 10th International Workshop *on the ACL2 Theorem Prover and its Applications*, Austin, Texas, USA, November 3-4, 2011, *Electronic Proceedings in Theoretical Computer Science* 70, Open Publishing Association, pp. 84–102, doi:10.4204/EPTCS.70.7.