Parameterized Congruences in ACL2

David Greve
Rockwell Collins Advanced Technology Center
Cedar Rapids, IA
dagreve@rockwellcollins.com

ABSTRACT

Support for congruence-based rewriting is built into ACL2. This capability allows ACL2 to treat certain predicate relations "just like equality" under appropriate conditions and allows specific theorems involving those equivalence relations to be used as rewrite rules to guide the simplification process. Although this has proven to be an extremely powerful capability, it comes with limitations. One significant limitation is that equivalence relations cannot be parameterized. This precludes one very natural application of congruences: their use in reasoning about arbitrary modulus arithmetic. We have developed a technique for emulating parameterized congruence-based rewriting on a stock ACL2 system using binding hypotheses and `bind-free`. We have also developed a library (nary) that provides a means of expressing and utilizing such congruences with a high degree of automation. We demonstrate the application of this library to problems in modular arithmetic and other similar domains.

1. CONGRUENCE-BASED REWRITING

Support for congruence-based rewriting is built into ACL2. We provide an overview of this capability here, although the curious reader is encouraged to review the ACL2 documentation on this subject under `congruence`[3]. Congruence-based rewriting allows ACL2 to treat certain predicate relations "just like equality" under appropriate conditions, and allows specific theorems involving those equivalence relations to be used as rewrite rules to guide the simplification process. The power of congruence-based rewriting in ACL2 is that it enables the user to construct simple rules that perform context sensitive simplifications and then chain those rules together to simplify expressions of arbitrary depth.

Consider the challenge of writing rules to normalize expressions for a list-based implementation of sets in which order and multiplicity are irrelevant. Assuming cons is a valid set constructor, one useful simplification rule for this library might be:

```
(defthm member-cons-duplicates
  (iff (member a (cons x (cons x y)))
       (member a (cons x y))))
```

Because duplicates are ignored we are able to simplify multiple conses of the same item into a single cons of that item in the context of the second argument of `member`. Looking closely at the `member-cons-duplicates` rule we see that the effect of the rule is to replace the second argument of `member`, `(cons x (cons x y))`, with a new value, `(cons x y)`. While these terms are not equal they behave in the same way in the context of the second argument of member so we are free to replace one with the other. To normalize such expressions, one might conclude that a class of rules that replace any occurrence of `(cons x (cons x y))` with `(cons x y)` within nested `cons` terms appearing in the second argument of member would be useful.

Unfortunately simple rewrite rules are not sufficiently powerful to accomplish this task. The `member-cons-duplicates` rule will not generally simplify a term of the form `(member a (cons w (cons x (cons x y))))`. While another rule specific to this term could be defined, one quickly discovers that no finite set of simple rewrite rules will ever simplify all possible occurrences of `(cons x (cons x y))` appearing in such a context.

More elegant approaches to the immediate problem, employing such syntactic techniques as `:meta` rules or `bind-free`, while more powerful, have limited scalability. They provide only point solutions, specific to a particular set of functions and are difficult to extend to user defined functions in the domain of interest.

Congruence-based rewriting is more powerful than simple rewrite rules and more scalable than other syntactic techniques. Employing congruence-based rewriting in ACL2 requires three basic steps: defining rewriting contexts, proving driver rules that simplify expressions within those contexts, and establishing congruence rules that identify when the rewriter can enter new contexts.

Rather than expressing a context in terms of, say, the second argument of `member`, ACL2 generalizes the notion of rewriting context and requires the formalization of the essential properties of this argument position with an equivalence relation. The equivalence relation is expected to capture the minimal set of properties that must be preserved by an argument in order to preserve the essential properties of that function. ACL2 then associates rewriting contexts with these equivalence relations. New rewriting contexts can be defined by defining and flagging new equivalence relations. An equivalence relation may be any function of two arguments that "acts like equal" in the sense that it satisfies the following property, stated in terms of the equivalence relation `set-equiv`:

```
(and
  (booleanp (set-equiv x y))
  (set-equiv x x)
  (implies (set-equiv x y)
           (set-equiv y x))
  (implies (and (set-equiv x y)
                (set-equiv y z))
           (set-equiv x z)))
```

Both `equal` and `iff` are examples of equivalence relations that are built-in to the ACL2 system, but any function in ACL2 satisfying the above properties can be flagged as an equivalence relation using `defequiv`:

```
(defequiv set-equiv)
```

This event will associate a new rewriting context with the given relation and will cause ACL2 to treat the equivalence relation just like equality in appropriate contexts. In particular, this means that a theorem of the form:

```
(defthm set-equiv-cons-cons-driver
  (set-equiv (cons a (cons a x)) (cons a x))
```

is treated as a rewrite rule that rewrites `(cons a (cons a x))` into `(cons a x)` in a `set-equiv` context, rather than as a rule that rewrites `(set-equiv (cons a (cons a x)) (cons a x))` into true. This rule is an example of a `set-equiv` driver rule. Such rules greatly enhance the user's ability to automate the production of normalized representations. While there are no specific restrictions on the form of driver rules, they are applied by ACL2 only in appropriate contexts. Those contexts are established by congruence rules. A congruence rule tells ACL2 exactly when it is sound to use certain types of equivalence relations during simplification. An example congruence rule is:

```
(defthm set-equiv-implies-iff-in-2
  (implies
    (set-equiv x y)
    (iff (member a x) (member a y))))
  :rule-classes (:congruence))
```

This theorem tells ACL2 that when it attempts to simplify calls of member in an `iff` (Boolean) context, it is free to simplify the second argument of `member` in a `set-equiv` context. When we say "a `set-equiv` context", we mean that it is sound for ACL2 to apply rewrite rules that employ the `set-equiv` equivalence relation. The form of congruence rules is quite restrictive, applying only to a single function instance (`member`, in this example) and having no hypotheses except a single equivalence relation over one argument of the function. The ACL2 `defcong` macro simplifies the specification of congruence relations. Here we reformulate `set-equiv-implies-iff-in-2 using defcong` and also express a `set-equiv` congruence for `cons` :

```
(defcong set-equiv iff (member a x) 2)
(defcong set-equiv set-equiv (cons a x) 2)
```

ACL2's ability to remember that it is in a particular context while performing simplification allows the user to write concise collections of driver rules. ACL2's ability to chain from one rewriting context to another (via congruence rules) extends the applicability of the driver rules even to arbitrarily nested terms. The `set-equiv` events presented so far are sufficient to program ACL2 to simplify any occurrence of the expression `(cons x (cons x y))` among nests of conses in the second argument of `member`. This demonstrates the power of such rules. These same basic techniques are also scalable. They can be applied locally to each new function in the domain and applied globally, in the context of other functions in the domain.

2. MODULAR ARITHMETIC

Having seen the benefit of congruence-based rewriting in the domain of a set library, let us now consider how one might attempt to extend these techniques to a similar problem in modular arithmetic. In modular arithmetic, two numbers are congruent "mod N" if they have the same *residue*, or remainder, when divided by N. The value of N is called the modulus. The residue of a value x, mod N, can be computed as `(mod x N)`. Consider the following useful simplification rule:

```
(defthm mod-+-mod-1
  (equal (mod (+ (mod x N) y) N)
         (mod (+ x y) N)))
```

Because mod distributes over addition and because mod is idempotent in its first argument, applications of the same modulus nested inside of `+` operations can be removed. Looking closely at the `mod-+-mod-1` rule we see that the effect of the rule is to replace the first argument of `+`, `(mod x N)`, with a new value, `x`. While these two expressions are not generally equal they behave in the same way in the context of the outermost `mod` operator so we are free to replace one with the other. To normalize such expressions, one might conclude that a class of rules that replace any occurrence of `(mod x N)` with `x` within nested `+` terms appearing in the first argument of `(mod * N)` would be useful.

Unfortunately simple rewrite rules are not sufficiently powerful to accomplish this task. The `mod-+-mod-1` rule will not simplify a term of the form `(mod (+ w (mod x N) y) N)`. While another rule specific to this term could be defined, one quickly discovers that no finite set of simple rewrite rules will ever simplify all possible occurrences of `(mod x N)` appearing in such a context.

There are other approaches to the immediate problem, employing such syntactic techniques as `:meta` rules or `bind-free`, which are more powerful than simple rewrite rules. The arithmetic-3 library, for example, has a number of bind-free lemmas for simplifying expressions of exactly this form. Such solutions, however, have limited scalability. They provide point solutions that are specific to a particular set of functions and difficult to extend to user defined functions in the domain of interest.

It was at this point in our discussion of the list-set library that congruence-based rewriting was proposed as a possible solution. The first step in this solution was to establish a rewriting context that captured the essential properties of interest. In this case, the property that we wish to preserve is "mod N". Expressing this property as an equivalence relation, as required by the ACL2 implementation, one might be tempted to define:

```
(defun mod-equiv (x y N)
  (equal (mod x N) (mod y N))
```

ACL2, however, does not support parameterized equivalence relations. It is possible to encapsulate the modulus as a nullary function, `(N)`, and then define `mod-equiv` as:

```
(defun mod-equiv (x y)
  (equal (mod x (N)) (mod y (N)))
```

This version of `mod-equiv` *is* a binary relation and it can be used as an equivalence relation to prove congruence (and possibly even driver) rules. Automating the application of these rules in a specific domain, however, requires that the user functionally instantiate `(N)` for each specific modulus value of interest, potentially making the use of such rules tedious.

We conclude then that, while congruence-based rewriting is an extremely powerful technique, its applicability is limited by the fact that equivalence relations in ACL2 cannot be easily parameterized. This fact inhibits one very natural application of congruences: their use in reasoning about arbitrary modulus arithmetic.

3. PARAMETERIZATION

The nary library was developed specifically to address the need to perform automated reasoning with parameterized congruences in ACL2. With the nary library, like in ACL2 itself, employing parameterized congruence-based rewriting requires three steps: defining parameterized rewriting contexts, proving parameterized driver rules that simplify expressions in those contexts, and establishing parameterized congruence rules that identify when the rewriter can enter new contexts.

3.1 Parameterized Contexts

In the ACL2 implementation rewriting context is determined by the equivalence relations currently active in the genequiv data structure. The genequiv data structure is passed as an argument to the rewriter and it identifies the "active" equivalence relations. Driver rules are allowed to fire if the equivalence relation they preserve is in the genequiv data structure. Congruence rules control modifications to the genequiv data structure, constructing new genequiv data structures as the rewriter dives into the arguments of the various functions it encounters based on the current context and any congruence rules associated with those functions.

One possible solution to the problem of parameterized equivalence relations would involve extending the genequiv data structure to include, along with each equivalence relation, a number of terms reflecting the parameters of that relation. While such an extension may be possible it would require substantial changes to the ACL2 code base.

The nary library enables the emulation of such an extended rewriting context on a stock ACL2 system through the careful application of binding hypotheses and `bind-free`. While traditional rewriting contexts are identified by equivalence relations, parameterized rewriting contexts in the nary library are identified primarily by context (or fixing) functions. In modular arithmetic, for example, the context function is `mod`.

The implicit assumption here is that the desired parametric equivalence relation can be expressed as a simple binary equivalence between two parameterized context functions. For example, a 5-ary equivalence between 2 variables, x and y, "modulo" the three parameters a1, a2, and a3, might be expressed as:

```
(equal (nary-equiv x y a1 a2 a3)
       (equiv (nary-ctx x a1 a2 a3)
              (nary-ctx y a1 a2 a3)))
```

Note that the context functions separate the interaction between each equated term and the parameters from the relative interaction between the two equated terms themselves. It is in this sense that context functions define (or capture) the parameterized rewriting context.

Context functions serve two primary purposes in the nary library and those purposes mirror the purposes of the genequiv data structure in ACL2. First, context functions act as a method for imposing a rewriting context on a specific term. Causing a term to be rewritten in a parametric context is as simple as wrapping that term in its context function and simplifying it. Rewriting the term `x` in a "mod N" context, for example, is simply a matter of simplifying the term `(mod x N)`. By analogy, ACL2 causes a term to be rewritten in an "equiv" context by adding "equiv" to the genequiv data structure and calling the rewriter on that term.

Context functions also act as triggers for driver rules. Which is to say that driver rules for the nary library are simply rewrite rules whose left-hand side's leading function symbol is a context function. The nary library effectively "activates" such rules when it wraps terms as described above. For example, when rewriting the term `x` in a "mod N" context, the driver rules include any those active rewrite rules whose left-hand side is an instance of `(mod x N)`. This behavior is comparable to how ACL2 activates certain rewrite rules based on equivalence relations found in the current genequiv data structure.

3.2 Parameterized Driver Rules

As we saw in the list-set example above it is driver rules that perform the actual work of term simplification in congruence-based rewriting. Parameterized driver rules are nothing more than rewrite rules whose leading function symbol is a parameterized context function. For example, the simple rule:

```
(defthm mod-N-N
   (implies
     (and (integerp N)
          (not (equal N 0)))
     (equal (mod N N) 0)))
```

is a perfectly good driver rule for modular arithmetic. Note that it is possible to combine standard congruence-based rewriting with parameterized congruence-based rewriting. This means it is possible to use traditional equivalence relations to construct driver rules for parameterized context functions.

3.3 Parameterized Congruences

The primary purpose of a traditional congruence rule is to cause ACL2 to simplify a selected argument of a targeted function in a new rewriting context whenever that function symbol is encountered in an appropriate rewriting context. While the form of a simple congruence rule resembles that of a rewrite rule, it is not treated as such by ACL2. Consider for example:

```
(defthm set-equiv-cons
  (implies
    (set-equiv a x)
    (set-equiv (cons v a)
               (cons v x)))
  :rule-classes (:congruence))
```

The conclusion of the congruence rule dictates the context in which the rule will fire and the hypothesis suggests the context in which the simplification of the argument will take place.

The result of applying nary congruence rules is very similar in effect to that of traditional congruence rules. Unlike traditional congruence rules, however, nary congruence rules *are* treated by ACL2 as simple rewrite rules. Like traditional congruence rules, the conclusion of the rule dictates the context in which the rule will fire. However, because nary congruence rules are just rewrite rules, this context is influenced by pattern matching. Also, as with traditional congruence rules, the hypothesis of nary congruence rules dictates the context in which the simplification of the function argument(s) will take place. The manner in which this simplification is performed, however, is quite different. In fact it is the hypotheses of nary congruence rules that house nearly all of the complexity of the nary library. Following is an example of a parameterized congruence rule for modular arithmetic[1]:

```
(defthm mod-+-cong-1
 (implies
  (and
    ;; -- Guard Hypotheses --
    (integerp N)
    (not (equal N 0))
    (integerp a)
    (integerp b)
    ;; -- Congruence Machinery --
    (equal wrap (mod a N))
    (bind-free
      (mod_unfix wrap N 'wrap? 'x)
      (wrap? x))
    (if wrap? (equal wrap (mod x N))
      (equal wrap x))
    (syntaxp (not (equal a x)))
    ;; -- Check Hypotheses --
    (integerp x))
 (equal (mod (+ a b) N)
        (mod (+ x b) N))))
```

The primary purpose of congruence rules is to induce ACL2 to simplify function arguments in a specified context. The rule above induces ACL2 to simplify the first argument of `+` in a "mod N" context when `+` itself is encountered in a "mod N" context. In order to see this, consider what is going on in the hypotheses of this rule. The first four hypotheses in this rule are simply guard hypotheses that dictate when this congruence rule is valid.

```
(integerp N)
(not (equal N 0))
(integerp a)
(integerp b)
```

The ability to guard congruence rules in this way illustrates one significant difference between traditional congruence based rewriting in ACL2 and the capabilities provided by the nary library.

The next four hypotheses are referred to as the *congruence machinery* hypotheses since they perform the heavy lifting of parameterized congruence based rewriting. An expression of the form `(equal x term)` appearing in the hypothesis of a rewrite rule, where `x` is a free variable and all variables appearing in the right-hand side term are bound, is called a binding expression[2]. The effect of a binding expression is to bind the free variable to the result of simplifying the term on the right-hand side. Thus, the effect of the fifth hypothesis of our rule is to bind to the variable `wrap` the

[1]The exact form of this rule has been simplified for the purpose of illustration.

[2]The term `(equiv x (double-rewrite y))` is also a binding expression for `x` if equiv is a recognized equivalence relation and all the variables appearing in `y` are bound. This fact allows `nary` congruence rules to leverage standard congruence-based rewriting as a part of the simplification process.

result of simplifying the first argument of the + function, a, in a "mod N" context:

```
  (equal wrap (mod a N))
```

Note that the simplification of (mod a N) may involve the application of some number of nary driver rules (such as mod-N-N, above) that trigger off of the nary context function (mod, in this case).

The sixth hypothesis in our theorem is a bind-free term that calls the function mod_unfix:

```
(bind-free
  (mod_unfix wrap N 'wrap? 'x)
  (wrap? x))
```

The mod_unfix function is defined as follows:

```
  (defun mod_unfix (wrap N wrap? x)
    (if (and (consp wrap)
             (equal (car wrap) 'mod)
             (equal (caddr wrap) N))
        (list (cons wrap? '(quote t))
              (cons x (cadr wrap)))
      (list (cons wrap? '(quote nil))
            (cons x wrap))
```

This function binds two variables, wrap? and x. The variable wrap? is bound to true when the term bound to wrap is of the form (mod P N). If wrap? is true x is bound to P (we call this the "unwrapped" value of wrap) otherwise it is bound to wrap.

There is no logical mechanism that we can use to verify a priori the correctness of the syntactic transformation performed by mod_unfix. We are thus obligated to prove that it unwrapped wrap correctly. This check takes place in the seventh hypothesis:

```
  (if wrap?
     (equal wrap (mod x N))
    (equal wrap x))
```

This check ensures that the value of x has the property that it is the result of simplifying a in a "mod N" context (ie: (mod a N)). If the result of that simplification was of the form (mod P N), x is equal to P and, if not, x is just equal to the simplification of (mod a N). Note that in computing x, our replacement for a, we are taking some care not to produce a term of the form (mod P N). This feature of the rule helps to ensure that we do not litter our expressions with (mod * N) terms. Our eighth hypothesis mitigates looping and ensures and that we have, in fact, accomplished something in this process:

```
  (syntaxp (not (equal a x))))
```

The final hypothesis is a logical check on the type of our replacement for a, the value x. Such checks may be required because, again, we have no assurance that mod_unfix did the right thing.

```
  (integerp x)
```

It is worth observing that, because nary congruence rules are just rewrite rules, they are strictly more expressive (flexible) than ACL2 congruence rules. ACL2 congruence rules work only with simple, non-parameterized equivalence relations, they apply only to individual functions, and they cannot have guarding hypotheses. Congruence rules defined using nary enable parameterization, can be made to pattern match on arbitrary terms and can be guarded by arbitrary hypotheses.

4. PARAMETRIC SUPPORT IN NARY

While the theorems and functions presented so far may implement parameterized congruence based reasoning, they are simply too complex to write by hand. To address this issue, a number of macros are provided by the nary library to simplify the process of specifying parameterized congruences. Here we provide an overview of those macros and some examples of their use.

4.1 defcontext

The defcontext macro is analogous to ACL2's defequiv macro in that it is used to establish new parameterized rewriting contexts and paves the way for parametric congruence lemmas to be proven later on. The form of a defcontext event is:

```
(defcontext (mod x n) 1)
```

The macro accepts a single function expression and a numeric designator. The function provided can henceforth be used to designate a rewriting context. The numeric designator identifies which argument to the function is to be treated as the context argument. The effect of this macro is to define the set of auxiliary functions used in expressing congruence rules. The mod_unfix function encountered in the example above was generated by this macro.

4.2 defcong+

The defcong+ macro is analogous to ACL2 defcong macro. The following is an example of using the defcong+ macro in its most general form. It also illustrates how one might program ACL2 to apply the distributivity of mod over plus.

```
  (defcong+ mod-+-cong
    (mod (+ a b) N)
    :hyps (and (rationalp N)
               (rationalp a)
               (rationalp b)
               (not (equal N 0)))
    :cong ((a (equal x (mod a N)))
           (b (equal y (mod b N))))
    :check (and (rationalp x)
                (rationalp y))
    :equiv equal
    :hints (("goal" :in-theory
             (enable mod-+-fixpoint-helper))))
```

This expanded form of this macro is very similar to `mod-+-cong-1`, presented above, except that this theorem will cause the simplification of both arguments of `+` rather than just the first. This theorem says that we can rewrite both arguments to `+` in a "mod N" context when `+` itself is in a "mod N" context and `a`, `b` and `N` are rational and `N` is not zero.

The first argument to the macro is the name of the congruence theorem. The second argument defines the target (left hand side) of the rule and can be an arbitrary term expression. In this example, we will be simplifying expressions of the form `(mod (+ a b) N)`. The `:hyps` argument corresponds to a guarding hypothesis and defines the conditions under which the theorem should be applied. The `:cong` term accepts a list of substitutions to be performed on the target expression. Each substitution must be of the form `(var1 (equiv var2 expr))`, where `var1` is a variable appearing in the target, `equiv` is a valid equivalence relation, `var2` is a new, unique variable name, and `expr` is a term whose leading function symbol has been defined (using `defcontext`) as an nary context[3]. The `:cong` term in this example directs ACL2 to rewrite `(mod (+ a b) N)` into `(mod (+ x y) N)` by replacing the "a" term of `(mod (+ a b) N)` with `x`, the result of simplifying `a` in the context `(mod a N)`, and replacing the "b" term with `y`, the result of simplifying `b` in the context `(mod b N)`. Because `x` and `y` are, from a logical perspective, plucked out of thin air, the `:check` term may be required to ensure that they are of an appropriate type. The `:equiv` keyword is used to specify the equivalence relation used to state the conclusion of the theorem. The default value is `equal`. Finally, `:hints` can be provided to guide the proof of the congruence theorem. All of the macro keywords are optional except the `:cong` keyword, which must specify at least one substitution.

4.3 `bind-context`

There are circumstances in which it is convenient to have access to the congruence machinery hypotheses without all the trappings of the `defcong+` macro. To support this, the nary library provides the `bind-context` macro. The `bind-context` macro accepts a list of substitutions and generates the associated congruence machinery hypotheses. Using `bind-context`, the `defcong+` example from above could be reformulated as:

```
  (defthm mod-+-cong
    (implies
      (and (rationalp N)
           (rationalp a)
           (rationalp b)
           (not (equal N 0)))
           (bind-context ((a (equal x (mod a N))))
                         (b (equal y (mod a N)))))
           (rationalp x)
           (rationalp y))
      (equal (mod (+ a b) N)
             (mod (+ x y) N)))
```

[3]These substitutions are converted into binding expressions like we saw earlier in the congruence machinery hypotheses. If the equivalence relation specified in the substitution is not `equal`, it will generate a binding expression of the form `(equiv var2 (double-rewrite expr))`.

4.4 Parametric Examples

Consider functions `foo1` and `foo2` with the following properties:

```
 (defthm foo1-prop
   (equal (mod (foo1 x n) n) (mod x n)))

  (defcong+ foo2-cong
    (mod (foo2 x) n)
    :cong ((x (equal a (mod x n)))))
```

Having defined `mod` as a parametric rewriting context (using `defcontext`) and having proven the `mod-+-cong` theorem presented above, we can now, in conjunction with the standard `arithmetic-3/floor-mod/floor-mod` book and properties such as those above, automate simplifications such as:

```
  (defthm mod-+-normalization-example
    (implies
      (and
        (integerp n)
        (not (equal n 0))
        (rationalp-guard a b c d e))
      (equal (mod (+ a
                     (mod b n)
                     (foo1 c n)
                     (foo2 (+ (mod d n)
                              (mod e n))))
                  n)
             (mod (+ a b c (foo2 (+ d e))) n))))
```

Modular arithmetic, however, is not the only domain that benefits from parameterized congruence-based reasoning. Another potential domain includes functions operating on data structures such as those defined by `defstobj`. Of concern in such examples is the use set of the functions in the domain. We define a "use set" context as:

```
  (defun copy-nth* (list st1 st2)
    (if (null list) st2
      (update-nth (car list)
        (nth (car list) st1)
        (copy-nth* (cdr list) st1 st2))))

  (defun use (list st)
    (copy-nth* list st nil))

  (defthm use-over-update-nth
    (implies
      (not (member (nfix b) list))
      (equal (use list (update-nth b v st))
             (use list st))))

  (defcontext (use list st) 2)
```

With `use` we are able to prove the following congruence rule for `update-nth`:

```
(defcong+ use-update-nth-cong
  (use list (update-nth a v x))
  :cong ((x (equal z (use list x)))))
```

Now consider a function, `foo`, that updates selected fields in a stobj with values from selected other fields of that stobj. The list of fields updated by `foo` are exactly those in `(foo-defs)` and the list of fields used by `foo` in that process are exactly `(foo-use)`. We can then define a congruence rule for `foo`:

```
(defcong+ nth-foo-use
  (nth a (foo st))
  :cong ((st (equal z (use (foo-use) st))))
  :hyps (member (nfix a) (foo-def)))
```

This theorem tells us that, if we are accessing field `a` of the stobj computed by `foo` from `st` and field `a` is a member of the fields updated by `foo`, then we need only consider those fields of `st` listed in `(foo-use)`. This rule is useful because it, in conjunction with `use-update-nth-cong`, allows us to use the rule `use-over-update-nth` to normalize state expressions appearing as arguments to `foo` by removing superfluous state updates from within an arbitrary nest of state updates. For example, if field 0 is in `(foo-def)` and field 3 is not in `(foo-use)`, our congruence rules cause the following simplification to take place:

```
(defthm test-nth-foo
  (equal (nth 0 (foo (update-nth a w
                       (update-nth 3 v st))))
         (nth 0 (foo (update-nth a w st)))))
```

While we have not yet applied such “use set” rules in practice we see this as one very appealing use of this technology. The nary library has been used successfully to prove parameterized congruences involving the ihs library `loghead` function and those congruences have been used to greatly simplify proofs about a variety of software programs running on a model of the Rockwell Collins AAMP7 microprocessor[1].

5. ISSUES AND FUTURE DIRECTIONS

The primary issue currently facing the nary library is efficiency. The conclusions of nary congruence rules are often of the form `(equal (fix a) (fix x))`. Note that the right hand side of this rule matches the left hand side, so as soon as ACL2 applies this rule it immediately attempts to apply it again. This leads to inefficiencies which can be measured using accumulated persistence. While nary congruence rules are designed to minimize the performance impact of this double application, it would be ideal if this second application could somehow be prevented altogether.

An interesting connection exists between congruence based rewriting and the nu-rewriter[2] in ACL2. Both employ information about rewriting context in order to perform their tasks and both profit from outside-in rewriting. The nu-rewriter embodies an under-the-hood implementation of simple parameterization, carrying with it the index of the `nth` function as it dives into terms. This connection suggests the possibility of generalizing the nu-rewriter capabilities to apply to a broader range of function symbols and domains such as those found in this library.

6. ACKNOWLEDGMENTS

I would like to thank Matt Kaufmann for providing substantial input on the form and content of this report. I would also like to thank the ACL2 workshop referees for their helpful reviews of this work.

7. CONCLUSION

We have developed an ACL2 library (nary) that supports parameterized congruence-based rewriting on a stock ACL2 system. This library provides a means of expressing and utilizing parameterized congruences with a high degree of automation and we have demonstrated the application of this library to problems in modular arithmetic and other similar domains.

8. REFERENCES

[1] David Greve, Raymond Richards, and Matthew Wilding. A Summary of Intrinsic Partitioning Verification. In *ACL2 Workshop 2004*, November 2004.

[2] J Moore. Rewriting for Symbolic Execution of State Machine Models. In *Computer Aided Verification*, 2001.

[3] J Moore and Matt Kaufmann. ACL2 Documentation. http://www.cs.utexas.edu/users/moore/acl2.