# Scalable Normalization of Heap Manipulating Functions

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

## ABSTRACT

Proofs about large systems that manipulate heap-like data structures are challenging. One particularly frustrating aspect of such proofs is the need to articulate, for each combination of heap-manipulating functions, specific non-interference properties. We present a technique that exploits parameterized equivalences and congruence-based rewriting to normalize expressions involving functions that manipulate heaps. The technique is structured, suggesting a systematic approach to proof construction. The technique is automated, being based on parameterized equivalences and the `nary` library. Most importantly, the technique is scalable, allowing function characterizations to be verified locally and applied globally.

## 1. BACKGROUND

In imperative programming, computation is modeled as a sequence of statements that modify program state. Procedures with global side-effects are common in imperative programs. In ACL2 a functional model of an imperative program often employs an additional state parameter that can be accessed and updated during the course of computation and that can be passed to and returned from procedures to model procedural side effects. A common interpretation of such a state object is as a “heap”, in which values or objects are stored at specific address locations. Reading from such a heap is typically done using a read function, which takes an address (or pointer) argument and a heap and returns the value from the heap associated with the address. Updating such a heap is often done using a write function which takes an address, a value, and a heap and returns a modified heap in which the address location provided has been updated with the value provided and no other address location has been changed. For the purposes of this paper, assume that the following property holds of the read and write operations:

```
(defthm read-over-write-non-interference
  (implies
    (not (equal a b))
    (equal (read a (write b v r))
           (read a r))))
```

This property is called *non-interference* because, under the conditions stated, the write operation, although it changed the heap, did not interfere with the value of a subsequent read operation, which otherwise depends upon the heap.

### 1.1 The Challenge of Non-Interference

Complex computing systems are typically constructed from compositions of less complex systems which, in turn, are constructed from even less complex systems and so on until we reach systems that are defined entirely in terms of the primitive operations over the fundamental data structures upon which the system operates. In the case of a system that manipulates a heap, those primitive operations might be the read and write functions described above.

Non-interference for the primitive read and write functions has already been discussed. Now consider the non-interference theorems required for systems defined in terms of these operations. If the new operation updates the heap, a theorem is required that describes when the new operation does not interfere with a read. Conversely, if the new operation accesses data from the heap, a theorem is required that describes when a write does not interfere with the new operation.

Now consider adding a second new operation. Of course the above procedure must be repeated for this new operation as well, but additional steps may be required in order to specify the non-interference properties between this second operation and the first. This is the first indication that scaling is going to become a problem in constructing and reasoning about truly complex systems.

To better illustrate this issue, consider a simple macro that constructs a system consisting of N pairs of (disabled) functions, read_[i] and write_[i], such that read_[i] and write_[i] are defined respectively as read_[i-1] and write_[i-1] and read_[0] and write_[0] are defined respectively as read and write. Now consider the collection of theorems required to ensure our ability to prove the non-interference of read_[j] and write_[k] for arbitrary j and k. To ensure this, for every value of j, a non-interference theorem must be constructed relating read_[j] to write_[k] for every possible value of k. The number of theorems required to do this is quadratic in N.

This contrived example illustrates the challenge a developer faces in constructing large, increasingly complex systems: the verification burden grows non-linearly with the size of the system. Although the non-interference property itself is not a fundamentally difficult property, the most straightforward solution for specifying non-interference, that of proving a collection of theorems about every possible function interaction, does not scale. Similar scalability issues have been addressed in other domains using congruence-based rewriting.

## 1.2 Congruence-Based Rewriting

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. 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, 1].

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.

In congruence-based rewriting, ACL2 generalizes this notion of rewriting context. Rather than expressing a context in terms of, say, the second argument of `member`, ACL2 requires a formalization of the essential properties of this argument position as an equivalence relation. The equivalence relation is expected to capture a minimal set of properties that must be preserved by an argument in order to preserve the essential properties of the function. In the case of the second argument to member, and for set operations in general, the property that must be preserved is the membership of the sets. Location and duplicity are irrelevant.

```
(defun set-equiv (x y)
  (if (consp x)
    (and (member (car x) y)
         (set-equiv (cdr x) (remove (car x) y)))
   (not (consp y))))
```

ACL2 associates rewriting contexts with equivalence relations. New rewriting contexts are specified 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`. Having done this for `set-equiv`, 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.

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.

Congruence based rewriting is an extremely powerful tool in the battle for scalability. It allows simplification rules to be expressed in terms of equivalence relations (such as `set-equiv`), rather than in terms of specific function symbols (such as `member`). Congruence rules characterize functions once, locally, and then can be applied globally in the context of other functions in the domain. Finally, the characterization of a functions can leverage the characterization of its constituent operations, enabling a compositional proof architecture.

## 1.3 Parameterized Congruence

A natural extension of congruence based rewriting allows for parameterized congruences. This is perhaps most obvious when one considers 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 first argument of the outermost `mod` operator so we are free to replace one with the other.

Previously, in our discussion of simplifications in the context of the second argument of `member`, a more general solution was presented in which the equivalence relation `set-equiv` was defined to capture the essence of what being in that context meant. In this example, the property that needs to be preserved is "mod N" and a reasonable equivalence relation might be `mod-equiv`:

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

Note, however, that `mod-equiv` is a function of three arguments. It is a parameterized equivalence. A parameterized equivalence may be any function of two or more arguments that "acts like an equivalence" between two of its arguments. Assuming that the equated arguments are the first two arguments and the function takes some number of parameters, `[n .. m]`, as additional arguments, a function "acts like a [parameterized] equivalence" if it satisfies the following properties:

```
(and
  (booleanp (nary-equiv x y [n .. m])
  (nary-equiv x x [n .. m])
  (implies (nary-equiv x y [n .. m])
           (nary-equiv y x [n .. m]))
  (implies (and (nary-equiv x y [n .. m])
                (nary-equiv y z [n .. m]))
           (nary-equiv x z [n .. m])))
```

Out of the box, ACL2's congruence-based rewrite capabilities do not support parameterized equivalence relations. We have, however, developed a library, the `nary` library, that supports congruence based rewriting with parameterized equivalence relations. This library was discussed in some detail previously[2]. Nonetheless, because the use of this library would be unfamiliar to most users of ACL2, we present parameterized congruence rules as they might appear if they were supported natively within ACL2 using the ficticious rule class `:nary-congruence`.

## 2. USE EQUIVALENCE

In order to apply congruence-based rewriting to the problem of non-interference it is first necessary to identify and formulate a more general equivalence relation of which non-interference is a specific instance. One such equivalence is use equivalence. Two heaps are `use-equiv` modulo a set of address locations if the values stored in the heaps agree at every one of the locations contained in the set. If two heaps are `use-equiv` modulo a set, we know that values read from each of the heaps at an address location contained in the set will be equal.

```
(defun use-equiv (r1 r2 uset)
  (if (consp uset)
    (and (equal (read (car uset) r1)
                (read (car uset) r2))
         (use-equiv r1 r2 (cdr uset)))
  t))
```

It is possible to prove that `use-equiv` satisfies the properties of a parameterized equivalence relation.

### 2.1 Characterizing the Primitive Functions

Use equivalence can be employed to characterize functions that manipulate the heap. The following theorem shows a characterization of the read function.

```
(defthm read-use-cong
  (implies
   (use-equiv x y (list a))
   (equal (read a x)
          (read a y))
:rule-classes (:nary-congruence))
```

The effect of this rule is to cause ACL2 to simplify the second (heap) argument of `read` in a `use-equiv` context in which the address (a) is the only element in the use set. Note that this rule chains the rewriter from an `equal` context to a `use-equiv` context.

The only functions that should appear inside of a `use-equiv` context are functions that modify the heap, and the simplest function that modifies the heap is the write operation. If the address being updated by the write operation is outside of the set of addresses of interest, we can ignore the write operation. This fact can be expressed as follows:

```
(defthm write-use-elim
  (implies
   (not (member a uset))
   (use-equiv (write a v x)
              x
              uset)))
```

The effect of this rule is to rewrite `(write a v x)` into `x` when it appears in a `use-equiv` context in which `a` is not a member of `uset`. The simple combination of `read-use-cong` and `write-use-elim` permits the following proof:

```
(defthm read-over-write-normalization-1
  (implies
   (not (member a (list b c d)))
   (equal (read a (write b v1
                   (write c v2
                    (write a v3

                     (write d v4
                      (write a v5 x))))
          (read a (write a v3
                   (write d v4
                     (write a v5 x))))
```

Whether or not the address is a member of the use set, the following congruence rule is true:

```
(defthm write-use-cong
  (implies
   (use-equiv x y uset)
   (use-equiv (write a v x)
              (write a v y)
              uset))
  :rule-classes (:nary-congruence))
```

The effect of this congruence rule is to simplify the third (heap) argument of `write` in a `use-equiv` context whenever `write` is encountered in a `use-equiv` context. With the addition of this rule our example reduces even further:

```
(defthm read-over-write-normalization-2
  (implies
   (not (member a (list b c d)))
   (equal (read a (write b v1
                   (write c v2
                    (write a v3
                     (write d v4
                      (write a v5 x))))
          (read a (write a v3
                   (write a v5 x)))))
```

An even stronger congruence rule exists for the write operation. The act of writing a value to a location "shadows" that location in the heap from the use set. Because of this, we can actually remove the address being written from the use set and prove the following congruence:

```
(defthm write-use-cong-stronger
  (implies
   (use-equiv x y (remove a uset))
   (use-equiv (write a v x)
              (write a v y)
              uset))
  :rule-classes (:nary-congruence))
```

The effect of this congruence rule is to simplify the third (heap) argument of `write` in a `use-equiv` context less the address (`a`) whenever `write` is encountered in a `use-equiv` context. Using nothing but congruence relations, we are now able to perform the following heap normalization:

```
(defthm read-over-write-normalization-3
  (implies
   (not (member a (list b c d)))
   (equal (read a (write b v1
                   (write c v2
                    (write a v3
                     (write d v4
                      (write a v5 x))))
          (read a (write a v3 x)))))
```

In certain applications the congruence relations discussed so far can actually replace (and sometimes produce better normalization results than) the following three rewrite rules:

```
(defthm read-over-write-noninterference
  (implies
   (not (equal a b))
   (equal (read a (write b v1 x))
          (read a x))))

(defthm write-over-write-noninterference
  (implies
   (not (equal a b))
   (equal (write a v1 (write b v2 x))
          (write b v2 (write a v1 x)))))

(defthm write-of-write
 (equal (write a v1 (write a v2 x))
        (write a v1 x)))
```

## 2.2 Applying the Technique

Consider again the simple macro that constructs a system consisting of N pairs of (disabled) functions, but now, along with each read_[i] and write_[i], assume that the macro also generates the following theorems:

```
(defthm read_[i]-use-cong
  (implies
   (use-equiv x y (list a))
   (equal (read_[i] a x)
          (read_[i] a y))
  :rule-classes (:nary-congruence))

(defthm write_[i]-use-cong
  (implies
   (use-equiv x y (remove a uset))
   (use-equiv (write_[i] a v x)
              (write_[i] a v y)
              uset))
  :rule-classes (:nary-congruence))

(defthm write_[i]-use-elim
  (implies
   (not (member a uset))
   (use-equiv (write_[i] a v x)
              x
              uset)))
```

These theorems, each verified locally for each read_[i] and write_[i], are sufficient to perform the following (global) simplification:

```
(defthm read_[i]-write_[k]-normalization
  (implies
   (not (member a (list b c d)))
   (equal (read_[i] a (write_[j] b v1
                       (write_[k] c v2
                        (write_[x] a v3
                         (write_[y] d v4
                          (write_[z] a v5 x))))
          (read_[i] a (write_[x] a v3 x)))))
```

Note that the number of theorems required to completely characterize the non-interference properties of the system is linear in the size of the system. Note also that the technique suggests a systematic approach to developing and characterizing functions that manipulate heap data structures.

## 2.3 Read Modify Write

Only simple read and write functions have been considered so far. Interesting computing systems will include functions that both access and modify the heap. Consider the `move` function defined below:

```
(defun move (rptr wptr r)
 (write wptr (read rptr r) r))
```

The characterization of this function requires that the use set be updated to include the "read pointer." However, the "write pointer" may be removed, since it is shadowed by the write operation.

```
(defthm move-use-cong
  (implies
   (use-equiv x y (cons rptr (remove wptr uset)))
   (use-equiv (move rptr wptr x)
              (move rptr wptr y)
              uset))
  :rule-classes (:nary-congruence))
```

A complementary `use-equiv` driver rule enables us to ignore the `move` operation if the write pointer is not in the use set.

```
(defthm move-use-elim
  (implies
   (not (member wptr uset))
   (use-equiv (move rptr wptr x)
              x
              uset)))
```

## 2.4 Data Structures and Crawlers

Complex computing systems typically manipulate well defined data structures. Data structures that inhabit a heap often include pointers. Consider a simple heap-based two cell cons structure whose base address is provided as an argument to the function. The base address points to the value portion of the cons cell and adding one to the base address produces a pointer to the cdr cell. Now consider the function `get-cadr` that operate on this list-like data structure:

```
(defun get-cadr (ptr r)
  (read (read (+ ptr 1) r) r))
```

The use set of `get-cadr` is somewhat complex, but we can define a function that computes it.

```
(defun get-cadr-uset (ptr r)
  (list (+ ptr 1) (read (+ ptr 1) r)))
```

The following theorem now characterizes `get-cadr`:

```
(defthm get-cadr-use-cong
  (implies
   (use-equiv x y (append (get-cadr-uset ptr x) uset)
   (equal (get-cadr ptr x)
          (get-cadr ptr y)
         uset))
:rule-classes (:nary-congruence))
```

Note that the use set of `get-cadr` is a function of the heap. Functions that traverse the heap to compute the use set of another function are called *data structure crawlers* or simply *crawlers*. Crawler are, therefore, nothing more than generalized read functions. As such, they too can be characterized.

```
(defthm get-cadr-uset-use-cong
  (implies
   (use-equiv x y (cons (+ 1 ptr) uset)
   (equal (get-cadr-uset ptr x)
          (get-cadr-uset ptr y)
          uset))
:rule-classes (:nary-congruence))
```

The full power of congruence-based normalization can therefore be used to normalize the very functions used to characterize the congruences in the first place.

## 2.5 Self-Characterizing Functions

Crawlers can be used to characterize functions, and crawlers themselves are functions that can be characterized. In this strange loop it turns out that most useful crawler functions also characterize themselves. This follows from the fact that it is always conservative to add additional elements to the use set of a congruence relation. The problem with using a function as its own characterization is that, in order to do so, the function must appear in the hypothesis of its own congruence relation. This may lead to simplification loops in the rewriter. Characterizing crawlers with other, more restricted crawlers is one safe solution to this problem, but there are cases when this is not generally possible. The problem arises when we attempt to model recursive data structures.

Consider the function that computes the size of the heap list data structure defined above.

```
(defun list-len (ptr r)
  (if (null ptr) 0
    (+ 1 (list-len (read (+ ptr 1) r) r))))
```

A crawler that would characterizes this function is as follows:

```
(defun list-len-uset (ptr r)
  (if (null ptr) nil
    (cons (+ ptr 1)
      (list-len-uset (read (+ ptr 1) r) r))))
```

There is, however, no simpler function that characterizes this crawler. How best to deal with such self-referential systems remains an open issue.

## 3. CONCLUSION

We have presented a technique that exploits parameterized equivalences and congruence-based rewriting to normalize expressions involving functions that manipulate heaps. The technique is structured, suggesting a systematic approach to proof construction. The technique is automated, being based on parameterized equivalences and the `nary` library. Most importantly, the technique is scalable, allowing function characterizations to be verified locally and applied globally. We conclude with an open issue concerning how best to deal with self-characterizing recursive functions.

## 4. REFERENCES

[1] B. Brock, M. Kaufmann, and J S. Moore. Rewriting with equivalence relations in ACL2. in preparation.

[2] David Greve. Parameterized Congruences in ACL2. In *ACL2 2006*, August 2006.

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