Efficient Rewriting of Operations on Finite Structures in ACL2

Matt Kaufmann
matt.kaufmann@amd.com

Rob Sumners
robert.sumners@amd.com

Advanced Micro Devices
5900 E. Ben White Blvd.
Austin, Texas, USA 78741

Abstract

We give a useful set of unconditional rewrite rules for reasoning about *record* structures, which are essentially finite functions. The problem, then, is to define functions for which these rules are true and then prove the rules. We begin with a series of definitions that attempt to satisfy these rules but fall short for various reasons. Then we give two solutions, one of which generalizes to other finite structures. The definitions of our access and update functions are somewhat subtle, complex, and ineffecient, but they return the expected values and the theorems exported are elegant and efficient for automatic, unconditional rewriting.

1 Introduction

It is common programming practice to abstract underlying data structures to the set of operators defined on the structures and the relationships between these operators. Indeed, most C++ programmers use the Standard Template Library without ever looking at the underlying source code defining the implementation of the objects in the library. Since the goal of the implementation is execution efficiency, the implementation may be complex, but the result of the execution should be consistent with the specification.

This programming practice has an analogy in ACL2. It is common practice to define operations on finite structures such as lists, association lists, sets, and trees, and to accompany these definitions with theorems that allow ACL2 to simplify terms involving the operations. In analogy with programming practice, the goal is to provide efficient simplification routines. Users of those routines generally do not need to be concerned with the source code (the definitions).

In this paper, we present techniques for providing definitions of operations on data structures along with associated theorems that enable efficient simplification by ACL2. Since the goal of these definitions is simplification efficiency, the definitions may be complex, but they are consistent with the specification and afford elegant and efficient theorems.

We will focus on the example of so-called "record structures," which can be thought of as stores, memories, or finite functions: they associate non-default values to a finite number of fields. We focus on records in this paper in order to provide a concrete example of the techniques proposed and also because of the simplicity and general applicability of the record operations.

Why not simply use association lists? Imagine for example starting with the empty record (function), then associating the field `'a` with data value `1` and then field `'b` with data value `2`. Presumably the result would be `'((b . 2) (a . 1))`. But if the updates were done in the other order, then the result would presumably be `'((a . 1) (b . 2))`. Sometimes it is convenient to have only one "canonical" representation for a finite function. The record books we present in this paper provide such a representation.

We describe two approaches to providing these utilities, one of which is a generic recipe for generating hypothesis-free equality-based rewrite rules that has also been applied to the definition of a book on flat sets with operations such as intersection, union, membership, etc. In each of the two approaches, we prove the following theorems about functions `s` (*set*, i.e., record update) and `g` (*get*, i.e., record lookup).

```
(defthm g-same-s
  (equal (g a (s a v r))
         v))

(defthm g-diff-s
  (implies (not (equal a b))
           (equal (g a (s b v r))
                  (g a r))))

(defthm s-same-g
  (equal (s a (g a r) r)
         r))

(defthm s-same-s
  (equal (s a y (s a x r))
         (s a y r)))

(defthm s-diff-s
  (implies (not (equal a b))
           (equal (s b y (s a x r))
                  (s a x (s b y r))))
  :rule-classes ((:rewrite :loop-stopper ((b a s)))))
```

The supporting material for this paper includes two records books that provide proofs of the above theorems as well as a book on finite flat sets. In the next section, we will present some history of the development of the definitions for records and the desired theorems provided above. While this history is somewhat tangential to the goal of proving the above theorems, we feel the material is still useful for illustrating the different approaches to defining structures.

2 Defining Records

Most programming languages and logics provide some support for extending the set of built-in types with user-defined structures. In C++ this is carried out by defining *structs* or *classes*, in ML this is provided with *datatypes*, and even in NQTHM this was provided by *shells*. ACL2 is largely an exception in that the only "data" structure in the logic is the `cons` binary node.

Commonly, ACL2 users will "simulate" structures using lists built from `cons` nodes. For example, consider a *person* record consisting of the fields *name*, *address*, and *age*. The following ACL2 code would be a common implementation of a *person* record:

```
(defun person (name address age) (list name address age))

(defun name    (person) (nth 0 person))
(defun address (person) (nth 1 person))
(defun age     (person) (nth 2 person))

(defun update-name    (name person)    (update-nth 0 name person))
(defun update-address (address person) (update-nth 1 address person))
(defun update-age     (age person)     (update-nth 2 age person))
```

Note that the above definitions could easily be (and often are) generated by a macro for defining records. For example, the ACL2 source code has such a macro called `defrec` and there is a distributed book `"books/data-structures/structures.lisp"` that defines the macro `defstructure`.

When proving theorems about these record functions, we would like some basic relationships to hold. In particular, we would like the access of a particular field to return the value from the most recent update of that field. For example, the following theorems would define the rules for reducing an access of the *address* field:

```
(defthm address-of-update-name
  (equal (address (update-name n p))
         (address p)))

(defthm address-of-update-address
  (equal (address (update-address a p))
         a))

(defthm address-of-update-age
  (equal (address (update-age a p))
         (address p)))
```

There are several problems with this approach. First, the number of theorems required for a given record is quadratic in the number of fields in the record. We can address this problem by enabling the field access and update function definitions and using the following rule (amongst others) for reducing terms with `nth` and `update-nth`:

```
(defthm nth-diff-update-nth
  (implies (and (integerp a) (>= a 0)
                (integerp b) (>= b 0)
                (not (equal a b)))
           (equal (nth a (update-nth b v r))
                  (nth a r))))
```

Unfortunately, in the proof output from ACL2, all of the accesses and updates will be reduced to `nth` and `update-nth` operations. For records with a large number of fields, it becomes a very cumbersome task to match up the indices in the nth and update-nth terms with the corresponding fields in the record. We can rectify this problem (somewhat) and allow a general set of field names by using association lists instead. For example, we could define the record accessors using assoc and record updating using acons as follows:

```
(defun rcd-access (field rcd) (cdr (assoc field rcd)))
(defun rcd-update (field value rcd) (acons field value rcd))

(defun name    (person) (rcd-access 'name person))
(defun address (person) (rcd-access 'address person))
(defun age     (person) (rcd-access 'age person))

(defun update-name    (name person)    (rcd-update 'name name person))
(defun update-address (address person) (rcd-update 'address address person))
(defun update-age     (age person)     (rcd-update 'age age person))
```

So, the proof output from ACL2 will now involve terms with rcd-update and rcd-access operators, but now these operators will be tagged with quoted symbols in the field parameters instead of natural numbers. Additionally, since the access is "positionless", the same access function can be used on records of different "types" but with fields that share names. We can now prove the following theorem, which corresponds to the theorem nth-of-update-nth stated before:

```
(defthm rcd-access-diff-rcd-update
  (implies (not (equal a b))
           (equal (rcd-access a (rcd-update b v r))
                  (rcd-access a r))))
```

Note that the hypotheses requiring n and m to be non-negative integers have been removed. Unfortunately, the best we can do for a theorem that reduces nested updates on the same key is the following, for a suitable alist-equality relation, alist=; for example, the alists '((a . 1) (b . 2)) and '((b . 2) (a . 1)) are not equal, but they are alist=.

```
(defthm rcd-update-same-rcd-update
  (alist= (rcd-update a y (rcd-update a x r))
          (rcd-update a y r)))
```

Alist= is far more complex than equal, and its effective use may require numerous congruence rules showing the preservation of alist=. Thus, we would like to define record access and update so that the rule above used equal instead of alist=.

We can achieve this goal if we normalize the association lists by (a) ordering the entries by the fields or keys, and (b) removing any entries where the value stored is nil. In ordering the keys in the association list, we will make use of the total order << [3] distributed with ACL2 Version 2.6 in "books/misc/total-order.lisp". This use of a total order on all of the ACL2 objects avoids having to introduce hypotheses for the keys in the records. (For example, an early version of this work restricted keys to be symbols so that we could use the total order symbol-< on symbols.) This normalization of the record structures leads to the definitions and theorems in Figure 1.

```
(include-book "total-order")

(defun rcd-access (field rcd)
  (cond ((or (endp rcd)
             (<< field (caar rcd)))
         nil)
        ((equal field (caar rcd))
         (cdar rcd))
        (t (rcd-access field (cdr rcd)))))

(defun acons-if (field value rcd)
  (if value (acons field value rcd) rcd))

(defun rcd-update (field value rcd)
  (cond ((or (endp rcd)
             (<< field (caar rcd)))
         (acons-if field value rcd))
        ((equal field (caar rcd))
         (acons-if field value (cdr rcd)))
        (t (cons (car rcd)
                 (rcd-update field value (cdr rcd))))))

(defun rcdp (x)
  (or (null x)
      (and (consp x)
           (consp (car x))
           (cdar x)
           (rcdp (cdr x))
           (or (endp (cdr x))
               (<< (caar x) (caadr x))))))

(defthm rcd-access-diff-rcd-update
  (implies (and (rcdp r)
                (not (equal a b)))
           (equal (rcd-access a (rcd-update b v r))
                  (rcd-access a r))))

(defthm rcd-update-same-rcd-update
  (implies (rcdp r)
           (equal (rcd-update a y (rcd-update a x r))
                  (rcd-update a y r))))
```

Figure 1: Normalized Record Structure Definitions and Theorems

Using the normalized record structures, we have defined theorems and rewrite rules which have resulted in `equal`-ity based theorems. Unfortunately, in the process we have introduced `(rcdp l)` hypotheses. As is common knowledge among ACL2 users, it is generally desirable to eliminate hypotheses of rewrite rules when possible: even when it seems easy to relieve such hypotheses, situations invariably arise where unforeseen work is required in the form of additional lemmas in order to relieve hypotheses. We briefly report on two approaches for removing the `rcdp` hypothesis in the next two sections while still maintaining the desired properties on record access and update.[1]

In the following two sections we provide two implementations of records, with associated rules. The first of these came first historically, and may have more intuitive computed results. The second approach is more general and has application to other data structures. In both cases we use the very short names `g` (record "get") and `s` (record "set") for readability of the ACL2 proof output, since the record operations tend to appear frequently.

3 Dealing with Non-Record objects

This section describes the book `"records0.lisp"` in the supporting materials, which is also (essentially) `"books/misc/records0.lisp"` distributed with ACL2 Version 2.6. This book contains our first approach to eliminating the `rcdp` hypothesis. Its exported rules originally required the keys to be symbols, but that restriction was removed when a total order was added for ACL2 [3].

As before, a record is an ordered alist whose value components are non-nil.

A lookup structure is the cons of a record onto any object, such that this object is (recursively) not a lookup structure.

```
(defun lsp (x)
  (or (rcdp x)
      (and (consp x)
           (not (lsp (cdr x)))
           (rcdp (car x))
           (car x))))
```

So, the `cdr` of a lookup structure is "junk," while the `car` is the record that holds the values. However, if we have other than a lookup structure then we view its entirety as "junk." Thus every ACL2 object represents a record structure: a record represents itself, a lookup structure represents the record in its `car`, and everything else represents the empty record.

`G` ("get") returns `nil` on a non-lookup structure, else looks up in the record part (the `car`).

```
(defun g (a x)
  (cond ((rcdp x)
         (cdr (assoc-equal a x)))
        ((lsp x)
         (cdr (assoc-equal a (car x))))
        (t nil)))
```

[1] We posted this problem to the ACL2 mailing list after our initial solution. A reply from Dave Greve and Matt Wilding gave another solution, quite similar to the one presented in Section 3.

The function s-aux updates a record by setting the value for a given key. We also need a function delete-key that removes a key-value pair (if any) for a given key. Using these notions we can define our record update function.

```
(defun s (a v x)
  (cond ((rcdp x)
         (if (null v)
             (delete-key a x)
           (s-aux a v x)))
        ((not (lsp x))
         (if v
             ;; then make x into the cdr of the returned lookup structure
             (cons (list (cons a v)) x)
           ;; else x already is an appropriate result
           x))
        ;; else we have (cons record junk)
        ((null v)
         (if (and (null (cdr (car x)))
                  (equal (caar (car x)) a))
             ;; then return the junk; there is nothing left of the record
             (cdr x)
           ;; else delete the key, a, from the record portion
           (cons (delete-key a (car x))
                 (cdr x))))
        (t
         ;; since (lsp x) and v is non-nil, we simply update the record portion
         (cons (s-aux a v (car x))
               (cdr x)))))
```

The theorems from the introduction follow from these definitions. The supporting materials contain lemmas used by ACL2 to prove those results.

4 Mapping ACL2 objects into Records

This section describes the book "records.lisp" in the supporting materials, which is also (essentially) "books/misc/records.lisp" distributed with ACL2 Version 2.6.

While the approach presented in the previous section achieves the desired results with respect to properties on records, we found it difficult to apply to a different setting, namely flat sets, where a larger number of functions and properties are required. In this section we define a strategy for achieving unconditional rewrite rules on structures which we believe is more easily applied to other contexts.

Before addressing the problem of removing the rcdp hypothesis, we first review and generalize the points in Section 2. In short, given a set of operations on some common data structure, (1) determine a normal form for the data structure such that "equivalent" structures are equal – the ACL2 total order is useful for this, (2) define the desired operations assuming well-formed data structures, and (3) prove the desired theorems about the operations assuming well-formed data structures. This leads to equal-based theorems with some hypothesis about normalized or well-formed objects. In order to remove these well-formed object assumptions, we (4) define an invertible mapping from all ACL2 objects into well-formed objects, (5) define "translated" versions of the "nice" operations from (2) using the mapping and inverse to translate ACL2 objects into well-formed objects and back, and (6) prove the desired properties about the "translated" operations removing the well-formed hypothesis.

As an example, we return to the problem of removing the `rcdp` hypothesis. Our goal is to define an invertible mapping of ACL2 objects into records. The simplest approach is to map any non-`rcdp` ACL2 object `X` into a record with a single association of some default key (say `nil`) to this object `X`. Unfortunately, since we want the mapping to be invertible, we cannot map singleton records with `nil` keys to themselves, and so we stuff these objects into other records. This process leads to the following definitions of our mappings between ACL2 objects and records (along with the desired properties of the mappings):

```
(defun ifrp (x) ;; ill-formed rcdp
  (or (not (rcdp x))
      (and (consp x)
           (null (cdr x))
           (consp (car x))
           (null (caar x))
           (ifrp (cdar x)))))

(defun acl2->rcd (x)
  (if (ifrp x) (list (cons nil x)) x))

(defun rcd->acl2 (x)
  (if (ifrp x) (cdar x) x))

(defthm acl2->rcd-returns-rcdp
  (rcdp (acl2->rcd x)))

(defthm acl2->rcd-rcd->acl2-of-rcdp
  (implies (rcdp x)
           (equal (acl2->rcd (rcd->acl2 x))
                  x)))

(defthm rcd->acl2-acl2->rcd-inverse
  (equal (rcd->acl2 (acl2->rcd x)) x))
```

We can now define the "translated" versions of our record access and update operations and prove the desired properties without `rcdp` hypotheses.

```
(defun g (field obj)
  (rcd-access field (acl2->rcd obj)))

(defun s (field value obj)
  (rcd->acl2 (rcd-update field value (acl2->rcd obj))))
```

Once a few simple theorems have been proven about the inverse mappings and the "nice" operations, the theorems from the introduction are automatically proven by ACL2.

A downside to this approach of using an invertible mapping to transform the "nice" functions to handle all ACL2 objects is that the resulting definitions may have unintuitive executable-counterparts. For example:

```
(rcd-get nil 2) = 2
(rcd-get nil '((nil . 2))) = '((nil . 2))
(rcd-set nil 2 ()) = 2
```

Evaluations such as these may occur during ACL2 simplification and may lead to confusing proof output. This problem can be mitigated substantially by "tagging" the mapping of the ill-formed ACL2 objects so that they do not interfere with the normal evaluations of the operations. In the case of records, we could simply change the definitions of `ifrp` and `acl2->rcd` as follows (and indeed, we have certified the resulting book).

```
(defun ifrp-tag () 'very-unlikely-to-occur-in-any-evaluation-of-ifrp)

(defun ifrp (x) ;; ill-formed rcdp
  (or (not (rcdp x))
      (and (consp x)
           (null (cdr x))
           (consp (car x))
           (equal (caar x) (ifrp-tag))  ;; instead of (null (caar x))
           (ifrp (cdar x)))))

(defun acl2->rcd (x)
  (if (ifrp x) (list (cons (ifrp-tag) x)) x))
```

5 Conclusions

We considered in this paper the problem of optimizing theorems about operations on data structures. We addressed this problem for the simple but generally useful case of record structures where the operations are field access and update. We presented two approaches to addressing this problem where we feel the latter approach has more general application with the potential downside of an unintuitive executable-counterpart. The records books have been used in several applications including the proof of a concurrent program [4] which used the records in various nestings and with different types of keys. We have also applied the latter approach to the problem of defining an elegant set of reduction rules for operations on flat sets. Although we did not discuss this application, we nevertheless provide a commented, certified book in the supporting materials for this paper.

Some possible areas of future work are worth mentioning. It would be useful to define and maintain a library of data structures with optimized (hypothesis-free and `equal`-based) reduction rules. Finite partitions and relations are clear candidates for future work, but numerous other candidates may exist. In addition, it would be useful to define computationally efficient versions of the logically efficient functions and show that these computationally efficient versions (in the appropriate contexts) coincide with the logically efficient versions. There have been discussions of a "`defexec`" proposal for ACL2, where the user can substitute a computationally efficient definition for execution while the logically efficient definition is used for proofs, where an appropriate proof obligation is generated by ACL2 to ensure the appropriateness of the substitution for execution.

Finally, it may be interesting to compare our handling of records to the handling of finite functions in a higher-order theorem prover. Functions are first-class objects in a higher-order logic, so, unlike the case with ACL2, there is no need to pick out a unique object to represent a functional relationship. We still suspect, however, that normalization rules are necessary in order to simplify terms efficiently that involve function updates.

References

[1] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June, 2000.

[2] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, June, 2000.

[3] P. Manolios and M. Kaufmann. "Adding a Total Order to ACL2," ACL2 Workshop 2002, http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/.

[4] R. Sumners. "An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2," ACL2 Workshop 2000, http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/.