Efficient Rewriting of Operations on Finite Structures in ACL2

ACL2 Workshop

Grenoble, France

April 9, 2002

Matt Kaufmann

matt.kaufmann@amd.com

Rob Sumners

robert.sumners@amd.com

⌊ “efficient rewriting”? ⌋

- Remove constraints on the contexts in which the rules can be applied
  - Eliminate hypothesis or conditions for applying the rewrite rule
  - Define the rewrite rules based on `equal` instead of a weaker equivalence
- Define enough rules to effectively reduce (normalize) the terms encountered

```
(implies (true-listp x)
         (equal (append x ()) x))
```

⌊ “operations on finite structures”? ⌋

- Most programming languages provide support for defining data structures
  - A data structure is a collection of operations and an underlying implementation
  - Execution efficiency considerations may affect the choice of implementation
    - but, the properties of the operations should remain the same

- In ACL2, these properties are codified by a set of rewrite rules referring to the operations
  - Simplification efficiency considerations (which properties are provable) may affect the choice of definition

⌊ Example: records ⌋

- Records associate some finite number of keys ("fields") to (non-default) values
  - Two operations on records: access (`g`, "get") and update (`s`, "set")
  - `nil` is an empty record (i.e. no fields are associated with non-default value)
- ACL2 has support for records using `defrec` and `defstructure`
  - Fixed set of fields, quadratic number of rewrite rules
- How about using `nth` and `update-nth`? or `assoc` and `acons`? to define our records?
- What properties do we want? What definitions are required?

⌊ What properties do we want? ⌋

- Desired properties (a fixed set of rewrite rules):

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

⌊ Structure normalization ⌋

- Normalize structures such that equivalent structures are `equal`

  – affords `equal` based rewrite rules

- Normalized records are alists where the keys are ordered via `<<`

  – `<<` is a strict (no duplicate keys) total order on ACL2 objects derived from `lexorder`

  – The alists cannot bind a key to the default value of `nil`

⌊ Initial definitions ⌋

- Definitions of `s-rcd`, `g-rcd`, and `rcdp`:

```
(defun g-rcd (a r)
  (cond ((or (endp r) (<< a (caar r)))
         nil)
        ((equal a (caar r))
         (cdar r))
        (t (g-rcd a (cdr r)))))

(defun acons-if (a v r)
  (if v (acons a v r) r))

(defun s-rcd (a v r)
  (cond ((or (endp r) (<< a (caar r)))
         (acons-if a v r))
        ((equal a (caar r))
         (acons-if a v (cdr r)))
        (t (cons (car r) (s-rcd a v (cdr r))))))

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

- We can prove the desired properties, but we have to add `rcdp` hypothesis

⌊ **Removing `rcdp` hypothesis #1** ⌋

- Basic idea: interpret ACL2 objects as suitable records

- Details: every ACL2 object is either a record (i.e. **`rcdp`**), the **`cons`** of a record with *junk* (i.e. **`lsp`**), or just *junk*

  – Notice that the definition of *junk* is recursive

  – We interpret *junk* as an empty record

```
(defun g (a x)
  (cond ((rcdp x)
         (g-rcd a x))
        ((lsp x)   ;; (<record> . <junk>)
         (g-rcd a (car x)))
        (t nil)))
```

⌊ Definition of s ⌋

- We now define the update function:

```
(defun s (a v x)
  (cond ((rcdp x)
         (s-rcd a v x))
        ((lsp x)
         (let ((r (s-rcd a v (car x))))
           (if r (cons r (cdr x)) (cdr x))))
        (t  ;; otherwise we have junk
         (let ((r (s-rcd a v nil)))
           (if r (cons r x) x)))))
```

- The proofs of the record properties go through with a few lemmas

- We found this approach difficult to transfer to other structures (e.g. flat sets)

  – We may need to continually modify the interpretation of *junk* based on the theorems we want to prove

⌊ Removing `rcdp` hypothesis #2 ⌋

- Basic idea: *translate* operations on records to operations on ACL2 objects using an invertible mapping of ACL2 objects to records

- Define a mapping **`acl2->rcd`** of ACL2 objects to records and an inverse mapping **`rcd->acl2`**

  – We must be careful to leave enough **`room`** in order to map ACL2 objects into a subset of the ACL2 objects

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

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

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

⌊ Definitions continued... ⌋

- A few theorems about the translation:

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

- We now have to translate `s-rcd` and `g-rcd` to ACL2 objects:

```
(defun g (a x)
  (g-rcd a (acl2->rcd x)))

(defun s (a v x)
  (rcd->acl2 (s-rcd a v (acl2->rcd x))))
```

- Potential downside: executable-counterpart does not map records to records

⌊ Conclusion ⌋

- We presented a few approaches for defining ACL2 functions on finite structures which afford efficient rewrite rules

  – We focused on the application of records, but a book on flat sets using the second approach is included in the supporting materials

- We would like to develop a library of books on finite structures with optimized rewrite rules

  – partitions, relations, etc.

- We should note that in a higher-order logic, one could define records by functions without having to construct a normal structure

  – Well, some normalization would be needed at the term level in order for syntactic equality between terms defining functions (records) to coincide with `equal`