Using ACL2 for Set Operations

ACL2 Lecture 4

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

Computer Science Department
University of Texas
2317 Speedway, M/S D9500
Austin, TX 78712-0233

January, 2023

Sets and Set Operations

In this lecture, we investigate using ACL2 to define sets and operations on sets.

- Set objects.
- Recognizing an acceptable set.
- Removing duplicate elements.
- Set union.
- Lookup by index.
- Update list at position.
- Lookup at write location.
- Access by name.
- Update by name.

Repeated REMINDER: We are introducing functional programming.

The lack of side effects provides opportunities for analysis. Much of this course concerns the pursuit of such opportunities.

Set Objects

In this lecture, we will define set operations.

We first define what elements our sets may contain.

```
(defun eqlablep (x)
  ;; Set element recognizer
  (or (acl2-numberp x)
      (symbolp x)
      (characterp x)))

(defun eqlable-listp (l)
  ;; Set recognizer
  (if (consp l)
      (and (eqlablep (car l))
           (eqlable-listp (cdr l)))
    (equal l nil)))
```

Using `EQLABLE-LISTP` as our set recognizer restricts set members to be characters, numbers, and symbols.

Is this an adequate definition?

Sets or Bags

Is our use of EQLABLE-LISTP as our set recognizer good enough? Consider:

```
(eqlable-listp '(a b c b))  ==>  T
```

Should our set recognizer allows duplicate members?

We can further restrict our set recognizer by requiring that there are no duplicates.

```
(defun no-dups (x)
  (if (atom x)
      t
    (let ((e (car x))
          (rst (cdr x)))
      (and
        ;; Check that E doesn't later appear
        (not (mem e rst))
        ;; Check the rest of the elements
        (no-dups rst)))))
```

NO-DUPS returns T when no duplicates are found.

We combine EQLABLE-LISTP with NO-DUPS to recognize a set, but not a bag.

What About Removing Duplicates?

To *clean up* a bag, we can write a function to remove duplicates.

```
(defun rm-dups (x)
  ;; Remove duplicates if they exist
  (declare (xargs :guard (eqlable-listp x)))
  (if (atom x)
      NIL
    (let ((e (car x))
          (rst (cdr x)))
      (if (mem e rst)
          (rm-dups rst)
        (cons e (rm-dups rst))))))
```

Let's trace things to see what happens.

```
  (trace$ rm-dups)
  (rm-dups '(1 2 3 2 4 2 3 2))
```

Confirm the Operation of RM-DUPS

Are we sure RM-DUPS works properly? Can we state (and prove) a property that would increase our confidence?

Consider:

```
(defthm not-mem-rm-dups
  ;; If no E in X, then no E in (RM-DUPS E X).
  (implies (not (mem e x))
           (not (mem e (rm-dups x)))))


(defthm no-mem-rm-all
  ;; There are no duplicates after removing duplicates.
  (no-dups (rm-dups x)))
```

It is important that we can explore our definitions.

We often perform such explorations by proof.

Set Union

Given two sets, can we create their union?

```
(defun set-union (x y)
  (if (atom x)
      ;; If X empty, return Y
      y
    (let ((e (car x))
          (rst (cdr x)))
      (if (mem e y)
          ;; If first element (E) of X appears in Y, then skip
          (set-union rst y)
        ;; Otherwise, include E, and continue...
        (cons e (set-union rst y))))))
```

Is this what we want? Let's check SET-UNION by proof.

```
(defthm eqlable-listp-set-union
  ;; Set union returns objects of the same type.
  (implies (and (eqlable-listp x)
                (eqlable-listp y))
           (eqlable-listp (set-union x y))))
```

Properties of SET-UNION

To increase our confidence, we state several desired properties.

```
(defthm not-mem-set-union
  ;; If E not member of X nor Y, then not in their SET-UNION.
  (implies (and (not (mem e x))
                (not (mem e y)))
           (not (mem e (set-union x y)))))

(defthm no-dups-set-union
  ;; No duplicates in X and Y, then no duplicates in SET-UNION.
  (implies (and (no-dups x)
                (no-dups y))
           (no-dups (set-union x y))))

(defthm mem-set-union
  ;; If E is in X or Y, then E is in their SET-UNION.
  (implies (or (mem e x)
               (mem e y))
           (mem e (set-union x y))))
```

We can check these properties by proof – this is something everyone will learn to do.

Lookup and Update by Position

We can use lists as a memory.

```
(defun ith (n l)
  ;; If at the end of memory  L?
  (if (endp l)
      ;; then, return default value
      nil
    ;; If at address, access item
    (if (zp n)
        (car l)
      ;; otherwise, keep looking...
      (ith (- n 1) (cdr l)))))


(defun !ith (key val l)
  (if (zp key)
  ;; If at the end position, add element
      (cons val (cdr l))
    ;; otherwise, copy element, and continue...
    (cons (car l)
          (!ith (1- key) val (cdr l)))))
```

One should consider what happens when `(< (LEN L) N))`.

Lookup and Update Properties

Have we defined a useful memory? Consider:

```
(defthm ith-!ith
  ;; We read what we wrote
  (equal (ith n (!ith n v l)) v))

(defthm ith-!ith-different-addresses
  (implies (and (natp i)
                (natp j)
                (not (equal i j)))
           ;; Writes at other locations
           (equal (ith i (!ith j v l))
                  ;;don't change what is at position  I
                  (ith i l)))))
```

Lemma ITH-!ITH confirms that we can read back what was written.

Lemma ITH-!ITH-DIFFERENT-ADDRESSES says a write other than at I doesn't change the value at position I.

Is this enough?

Associative Memory

Instead of a lookup by index, often we prefer to lookup by name (key). `ASSCP` recognizes a list of pairs where each pair is: `(CONS` *key value*`)`.

```
(defun asscp (x)
  (if (atom x)
      (null x)
    (and (consp (car x))
         (asscp (cdr x)))))

(defun assc (k al)
   ;; Indicate the structure of  AL.
  (declare (xargs :guard (asscp al)))
  (if (atom al)
      NIL
    (let* ((pair (car al))
           (key  (car pair)))
      (if (equal k key)
          ;; If found, return pair.
          pair
        (assc k (cdr al))))))
```

Why does `ASSC` return a pair instead of just the value?

Update Associative Memory

Our update function is simple, we just add a key-value pair to the *front* of our memory.

```
(defun update (k v al)
  (declare (xargs :guard t))
  (cons (cons k v)
        al))
```

We can observe various properties of this approach? For instance,

```
(defthm assc-update
  (equal (assc k (update k v a))
         (cons k v)))
```

But, is this enough? What about blocked (unreachable) entries?