ACL2 Terms, Evaluation, Mutual Recursion

ACL2 Lecture 5

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

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

March, 2025

ACL2 Terms, Evaluation, Mutual Recursion

In this lecture, we investigate using ACL2 to define terms and their evaluation.

- Review item: a `FLATTEN` function with an accumulator.
- Lookup and Update by position
- Lookup and Update Properties
- Measure functions
- Mutual Recursion – Shuffling
- Example terms
- Recognizing a term
- Evaluation examples
- An evaluation function
- Tracing evaluation
- Class Projects

MC-FLATTEN – McCarthy's FLATTEN Function

Earlier this semester, we asked that you define a FLATTEN function using CONS instead of APPend.

Why? Because, repeated use of APP is expensive in the number of CONS cells created.

So, can we use an accumulator to assist our collection effort?

```
(defun mc-flatten (x acc)
  (if (atom x)
      (cons x acc)
    ;; Accumulate the left elements second
    (mc-flatten (car x)
                ;; Accumulate the right elements first
                (mc-flatten (cdr x) acc))))
```

Suggestion: using TRACE$, compare FLATTEN with MC-FLATTEN.

Can you prove?

```
(equal (flatten x)
       (mc-flatten x nil))
```

Lookup and Update by Position

We can use lists as a memory.

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


(defun !ith (n val l)
  (if (zp n)
  ;; If at the specified position, place element
      (cons val (cdr l))
    ;; otherwise, copy current element, and continue...
    (cons (car l)
          (!ith (1- n) 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?

NTH and !NTH Properties

Here we defined an abbreviation for UPDATE-NTH.

```
(defmacro !nth (key val l)
 `(update-nth ,key ,val ,l))

(add-macro-fn !nth update-nth) ;; UPDATE-NTH shown as !NTH
```

These five rules relate NTH and !NTH – the first rule is built in.

```
(defthm nth-!nth
  ;; Read-over-Write; redundant with lemma NTH-UPDATE-NTH
  (equal (nth a1 (!nth a2 v l))
         (if (equal (nfix a1) (nfix a2))
             v
           (nth a1 l))))

(defthm !nth-nth
  ;; Write-over-Read
  ;; A read "off the end" is NIL, but a write extends L
  (implies (and (equal a1 a2)
                (< (nfix a1) (len l)))
           (equal (!nth a1 (nth a2 l) l)
                  l)))
```

NTH and !NTH Properties, continued

```
(defthm !nth-!nth-same-address
  ;; Write-over-write; memory reflects last value written
  (implies (equal a1 a2)
           (equal (!nth a1 v (!nth a2 w st))
                  (!nth a1 v st))))


(defthm !nth-!nth-different-addresses
  ;; Order of writes to different memory locations is irrelevant
  (implies (not (equal (nfix a1) (nfix a2)))
           (equal (!nth a1 v1 (!nth a2 v2 st))
                  (!nth a2 v2 (!nth a1 v1 st)))))


(defthm nth-from-atom-or-short-list
  ;; A read ``off the end''
  (implies (or (atom l)
               (and (integerp i)
                    (< (len l) i)))
           (not (nth i l))))
```

Measure Function Example

ACL2 provides basic idioms for proving that recursive definitions will terminate.

Through our use of the *Principle of Structural Recursion* we have investigated the use of the `CAR–CDR` idiom.

But, imagine we want to recur by removing an *item* first from one pile and then from a second pile, and then repeating.

This is like shuffling (mixing) two stacks of playing cards. Could we define:

```
(defun mix (s x y)
  (if s
      (if (atom x)
          y
        (cons (car x)
              (mix nil (cdr x) y)))
    (if (atom y)
        x
      (cons (car y)
            (mix t x (cdr y))))))
```

where `X` and `Y` contain our cards?

ACL2 fails to accept this function. Why?

Measure Function Example, continued

For a measure, one may specify a measure function that involves relationships among a function's arguments.

Often, we can use a measure that produces a natural number; this is the simplest kind of ordinal. For now, that is what we will do.

```
(defun m (x y)
  (+ (len x) (len y)))
```

Given the measure function above, we can now define the MIX function.

```
(defun mix (s x y)
  (declare (xargs :measure (m x y)))  ;; A specific measure
  (if s
      (if (atom x)
          y
        (cons (car x)
              (mix (not s) (cdr x) y)))
    (if (atom y)
        x
      (cons (car y)
            (mix (not s) x (cdr y))))))
```

Mutual Recursion

Instead of a single function with a *flag* argument, can we define two, mutually-recursive functions?

```
(mutual-recursion

 (defun l (x y)
   (declare (xargs :measure (m x y)))
   (if (atom x)
       y
     (cons (car x)
           (r (cdr x) y))))

 (defun r (x y)
   (declare (xargs :measure (m x y)))
   (if (atom y)
       x
     (cons (car y)
           (l x (cdr y))))))
```

Is MIX equivalent to L and R?

Can we establish a relationship between MIX and L and R?

```
(defthm mix-is-l-r
  (and

   (implies (double-rewrite s)
            (equal (mix s x y)
                   (l x y)))

   (implies (not (double-rewrite s))
            (equal (mix s x y)
                   (r x y))))
)
```

Advice: At first, implement mutual recursion with *flag* functions.

An Expression Evaluator

Can we define the syntax and semantics of a simple calculator? Let's try...

Our variable symbols will be ACL2 symbols. Our values are integers.

```
(defun symbol-integer-alistp (al)
  (declare (xargs :guard t))
  (if (atom al)
      (null al)
    (let* ((pair (car al)))
      (and (consp pair)
           (symbolp (car pair))
           (integerp (cdr pair))
           (symbol-integer-alistp (cdr al))))))
```

Our language will allow embedded integer-valued constants.

Our calculator will implement negation, addition, and multiplication.

Our Variable Lookup Function

Imagine we wished to evaluate arithmetic expressions that include variables.

We will need a way to look up the values of variables

```
(defun assc (k al)
  (declare (xargs :guard (symbol-integer-alistp al)))
  (if (atom al)
      0
    (let* ((pair (car al))
           (key  (car pair)))
      (if (equal k key)
          (mbe :logic (ifix (cdr pair))
               :exec  (cdr pair))
        (assc k (cdr al))))))
```

Can we have duplicate keys?

Expression Recognizer

```
(defun exprp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (symbolp x)
    (let ((fn (car x))
          (args (cdr x)))
      (and (symbolp fn)
           (consp args)
           (case fn
             (quote (and (null (cdr args))
                         (integerp (car args))))
             (-     (and (null (cdr args))
                         (exprp (car args))))
             (+     (and (consp (cdr args))
                         (null (cddr args))
                         (exprp (car args))
                         (exprp (cadr args))))
             (*     (and (consp (cdr args))
                         (null (cddr args))
                         (exprp (car args))
                         (exprp (cadr args))))
             (otherwise NIL))))))
```

Expression Evaluator

Given a term and an association list, we can define an evaluator.

```
(defun evx (x a)
  (declare (xargs :guard (and (exprp x)
                              (symbol-integer-alistp a))
                  :verify-guards nil))
  (if (atom x)
      ;; Lookup variable value
      (assc x a)
    (let ((fn (car x))
          (args (cdr x)))
      (case fn
        (quote (car args))  ;; Constant
        (- (- (evx (car args) a)))
        (+ (+ (evx (car args) a)
              (evx (cadr args) a)))
        (* (* (evx (car args) a)
              (evx (cadr args) a)))))))
```

Question: does this function always return an integer?

Expression Return Type, Optimizer

What is the type of our evaluator?

```
(defthm integerp-evx
  (implies (exprp x)
           (integerp (evx x a)))
  :rule-classes :type-prescription)
```

Can you define a `FOLD-CONSTANT` expression optimizer?

```
(defthm integerp-evx
  (implies (and (exprp x)
                (integer-val-alistp a))
           (equal (evx (fold-constants x) a)
                  (evx x a))))
```

For example:

```
(fold-constants '(+ x (* '3 '2)))
==>
'(+ x '6)
```

Can you prove your constant-folding expression optimizer correct?