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

January, 2023

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.
- Example terms.
- Recognizing a term.
- Evaluation examples.
- An evaluation function.
- Tracing evaluation.
- The rest of the semester.

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.

MC-FLATTEN – McCarthy's FLATTEN Function

On HW #1, 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.

Our Variable Lookup Function

Imagine we wished to evaluate arithmetic expressions. We will need a way to look up the values of variables

```
(defun integer-val-alistp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (null x)
    (if (atom (car x))
        NIL
      (and (integerp (cdar x))
           (integer-val-alistp (cdr x))))))


(defun assc (k al)
  (declare (xargs :guard (alistp al)))
  (if (atom al)
      NIL
    (let* ((pair (car al))
           (key  (car pair)))
      (if (equal k key)
          pair
        (assc k (cdr al))))))
```

Can we have duplicate keys?

Expression Recognizer

```
(defun exprp (x)
  (if (atom x)
      (legal-varp x)
    (let ((fn (car x))
          (args (cdr x)))
      (cond ((eq fn 'quote)
             (and (consp args)
                  (null (cdr args))
                  (integerp (car args))))
            ((symbolp fn)
             (case fn
               (- (and (consp args) (null (cdr args))
                       (exprp (car args))))
               (+ (and (consp args) (consp (cdr args))
                       (null (cddr args))
                       (exprp (car args)) (exprp (cadr args))))
               (* (and (consp args) (consp (cdr args))
                       (null (cddr args))
                       (exprp (car args)) (exprp (cadr args))))
                  (otherwise NIL)))
            (t nil)))))
```

Expression Evaluator

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

```
(defun evx (x a)
  (if (atom x)
      ;; Lookup variable value
      (let ((pair (assc x a)))
        ;; If not found, return 0
        (if pair (cdr pair) 0))
    (let ((fn (car x))
          (args (cdr x)))
      (cond ((eq fn 'quote)  ;; Constant?
             (car args))
            ((symbolp fn)    ;; Function?
             (case fn
               (- (- (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 (and (exprp x)
                (integer-val-alistp a))
           (integerp (evx x a))))
```

Can you define a constant-folding expression optimizer?

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

For example:

```
(optimize '(+ x (* '3 '2)))
==>
'(+ x '6)
```

Can you prove your constant-folding expression optimizer correct?