Some Basic ACL2-Lisp Functions

ACL2 Lecture 3

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

REV, Accumulators, Complexity, and Tracing

In this lecture, we investigate some basic list and tree processing functions.

- Discussion of REV
- `TRACE$` of REV
- Using accumulators; tail recursion
- We define a way to sum the leaves of a tree with integer leaves.
- We define an ACL2 predicate that recognizes trees with integer leaves.
- We introduce `FLATTEN`, a function to flatten a tree into a list.
- We postulate flattening a tree without using `APP`.
- We will use `TRACE$` to animate our definitions.

At the end, we will look at HW 1.

REMINDER: Using examples, we are introducing functional programming.

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

List REVerse

We can use APP to create a REVerse function.

Consider:

```
(defun app (x y)
  (if (atom x)
      y
    (cons (car x)
          (app (cdr x) y))))


(defun rev (x)
  (if (atom x)
      nil
    (app (rev (cdr x))
         (list (car x)))))
```

Does REV return a TRUE-LISTP?

Remark: in running text, we often write ACL2 terms in upper case because Lisp *up-cases* everything.

TRACE of REVerse

Let's *animate* the `REV` function.

```
(trace$ REV)
```

It appears linear in its performance, but what about `APP`?

```
(trace$ APP)
```

Does `REV` return a `TRUE-LISTP`?

Does reversing a list twice produce return the original input?

```
(equal (rev (rev x)) x)
```

Deeper Question: Should we trace `CONS`?

Using Accumulators; Tail Recursion

We see that REV may be expensive to evaluate.

What is the complexity of REV?

Let's consider the use of an accumulator.

```
(defun rev2-help (x acc)
  (if (atom x)
      acc
    (rev2-help
      ;; Trim off first element
      (cdr x)
      ;; Extend the accumulator
      (cons (car x) acc))))

(defun rev2 (x)
  (rev2-help x nil))
```

Let's trace things to see what happens.

Tree Copy

So far, our definitions have concerned only right-associated trees (lists).

Consider:

```
(defun tree-copy (x)
  ;; If no pair
  (if (atom x)
      ;; return the atom
      x
    ;; otherwise, pair a copy
    (cons
      ;; the left subtree, and
      (tree-copy (car x))
      ;; the right subtree
      (tree-copy (cdr x)))))
```

Note that we *move* both left and right. Is this OK?

A Tree with Integer Leaves

Can we define a function that recognizes a tree containing just integers?

Will we allow *empty* trees?

```
(defun tree-integerp (x)
  ;; If pair recognized
  (if (atom x)
      ;; do we have an integer?
      (integerp x)
    ;; otherwise, check
    (and
      ;; the left subtree, and
      (tree-integerp (car x))
      ;; the right subtree
      (tree-integerp (cdr x)))))
```

Are we under utilizing `CONS`? How many `CONS`es are needed to *hold n* atoms?

FLATTEN a Tree

Can we define a function that, from left-to-right, takes each tip of a tree and creates a list?

```
(flatten (cons (cons 1 2) (cons 3 4)))
==>
(1 2 3 4)
```

We want a list with all tree tips. So, let's append the left and right subtrees together.

What do we need to do?

Let's have a look at HW 1.