CSU290 Lecture Notes Lecture 8 29 Sept 2008

Peter Dillinger

Announcements

* Second homework due Oct. 7th (Tuesday)

* Second exam on Oct. 8th (Wednesday)

Design Recipe: ACL2 version (updated)

* Problem analysis and data definition
* Contract and purpose (no effects in ACL2)
* Examples as tests (no need to write examples in comments. just write tests, which serve as examples.) (use CHECK=)
* Function template (this will be inserted before the tests.)
* Function definition
* Make it total (but "keep it simple")

  1. "Eliminate unintended recursion"
     Find all parameters that affect how many recursive calls are made.
     For each one:
       If contract expects only atomic data,
         Make sure all inputs outside of contract go to a base case, using an idiom such as ZP if appropriate.
       If contract allows cons data,
         Make sure all ATOMIC inputs outside of contract go to a base case, using an idiom such as ENDP if appropriate.
         For cons data, recurse as if the cons contains contract values and leave totality concerns for when atomic data is reached.

  2. "Minimize IFs"
     Don't use more IFs than needed. Eliminating unintended recursion (#1) often requires modifying IF tests, but rarely requires introducing more IFs.

* Add tests of the function outside the contract/intended input, to check for totality (use CHECK or CHECK=)

More functions on lists

```
; MY-NTH: nat true-list -> all
; Returns the nth (n is first parameter) element of l (second
; parameter), with zero-based indexing.
(defun my-nth (n l)
  (if (zp n)
    (car l)
    (my-nth (- n 1) (cdr l))))

(check= (my-nth 2 '(1 2 3)) 3)
(check= (my-nth 0 '(1 2 3)) 1)
(check= (my-nth 0 5) nil)


; MY-LEN: true-list -> nat
; Returns the length of the given list
(defun my-len (l)
  (if (endp l)
    0
    (+ 1 (my-len (cdr l)))))
```

```
; an equivalent definition:
(defun my-len (l)
  (if (consp l)
    (+ 1 (my-len (cdr l)))
    0))

(check= (my-len '(4 5 6)) 3)
(check= (my-len '()) 0)
(check= (my-len 5) 0)


; MAX-LIST: (listof nat) -> nat
; Returns the largest element of the given list
(defun max-list (lon)
  (if (endp lon)
    0
    (if (< (max-list (cdr lon)) (car lon))
      (car lon)
      (max-list (cdr lon)))))

(check= (max-list '(1 2 3 2 1)) 3)
(check= (max-list '(5)) 5)
(check= (max-list nil) 0)
(check= (max-list 5) 0)
```

Now let us consider some more interesting examples:

```
  (max-list '(-3 -4))
```

What should it return? If we think of the function as returning the largest *number* in the list, the return value seems wrong:

```
  ACL2 p>VALUE (max-list '(-3 -4))
  0
  ACL2 p>
```

The contract calls for a list of natural numbers and we used that assumption when we decided to have the max-list of the empty list be 0. What this example returns is the maximum of -3, -4, and 0, which is 0.

Let's consider a somewhat large example:

```
  (check= (max-list '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5)) 5)
```

You might end up waiting a very long time for that to finish. How do we know it will finish? Well, for now, you have to figure that out for yourself. Do all your recursions decrease toward a base case? This one does. So why is it taking so long?

First of all, we aren't really concerned with efficiency in this class, so if you write something like that on a test, you will get full credit. Aside from being inefficient, it is logically a simple definition.

The reason for the inefficiency is that each call can make two recusive calls, the two instances of (max-list (cdr lon)). Thus, (max-list '(0 5)) will call (max-list '(5)) twice. (max-list '(0 0 5)) will call (max-list '(0 5)) twice. Double the execution time enough times, and it becomes intractable.

What can we do about this? Well, in this case, the two recursive calls are computing the same value! Can't we reuse this value without asking to recompute it? That is what LET is for:

```
(defun max-list (lon)
  (if (endp lon)
    0
    (let ((m (max-list (cdr lon))))
      (if (< m (car lon))
        (car lon)
        m))))
```

The form of a LET is

```
  (let ((<var1> <expr1>)
        (<var2> <expr2>)
        ...)
    <body>)
```

and the meaning is that it executes <expr1>, <expr2>, ... and binds the values returned to <var1>, <var2>, ... respectively in evaluating <body>.

In this case, (max-list (cdr lon)) is evaluted and bound to the variable m in the expression (if (< m (car lon)) (car lon) m). I could have chosen most any variable name instead of m. I just chose m to be short for "max".

But, there is a cleaner solution to this problem, and that is to write a function that takes the maximum of two numbers and use that to implement max-list:

```
; Acutally, this is already defined in ACL2:
(defun max (x y)
  (if (> x y)
    x
    y))

; MAX-LIST: (listof nat) -> nat
; Returns the largest element of the given list
(defun max-list (lon)
  (if (endp lon)
    0
    (max (car lon)
         (max-list (cdr lon)))))
```

Using the auxilliary function has a similar effect as using the LET, because the result of the recursive call (max-list (cdr lon)) gets bound to the parameter y in MAX and is potentially used twice: once in comparing and possibly once in returning a value. But because it was bound to a variable or parameter, the recursive call is only made once.

More on quoting

When we write

```
  '(1 2 3)
```

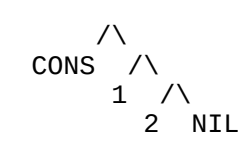
that is shorthand for

```
  (quote (1 2 3))
```

and QUOTE is special in ACL2. Clearly it must be so because (1 2 3) is not a valid function call. In fact,

```
   (quote (cons 1 2))
or '(cons 1 2)
```

evaluates to the list



where CONS is a symbol. But CONS is not a valid expression:

```
ACL2 >cons

ACL2 Error in TOP-LEVEL:  Global variables, such as CONS, are not allowed.
...
```

If we want an expression that evaluates to the symbol CONS, we need to quote it to indicate we aren't refering to a variable, parameter, or function name:

```
ACL2 >'cons
CONS
```

For example, if we define

```
(defun foo (x) (cons x 'x))
```

The unquoted x will evaluate to the value passed in, while the quoted x will evaluate to the symbol x:

```
ACL2 >(foo 3)
(3 . X)
```

The symbols T and NIL are special, though, because they evaluate to themselves. You can optionally quote them also:

```
ACL2 >t
T
ACL2 >'t
T
ACL2 >nil
NIL
ACL2 >'nil
NIL
ACL2 >()
NIL
ACL2 >'()
NIL
```

We've talked about writing down literal lists using quoting, as in '(1 2 3). What about a list of lists? It might stand to reason that

```
'('(1 2) '(3) '(4 5 6))
```

is a list of lists of numbers, but actually it is not. The CAR of the CAR of that should be 1, but in fact,

```
ACL2 >(car (car '('(1 2) '(3) '(4 5 6))))
QUOTE
```

In short, you will never need to use quote inside of something that is already quoted, because the quote affects the entire expression. Nothing in a quoted expression is actually evaluted; instead, it is all returned as a literal value:

```
ACL2 >(car (car '((1 2) (3) (4 5 6))))
1
```

More Boolean Logic

We will learn more boolean logic identities that will help in manipulating expressions quickly. Here are some important ones:

```
  ~(p /\ q)   =   ~p \/ ~q     (De Morgan's law 1)

  ~(p \/ q)   =   ~p /\ ~q     (De Morgan's law 2)

      ~~p     =      p         (Double negation)
```

For confirmation, you can construct a truth table for the formulas on both sides of each equality and check that they have the same truth value for all possible truth values of p and q.

Suppose we have to determine whether ~(p -> q) is equivalent to ~p /\ ~q. We might write

```
  ~(p -> q)  ?=  ~p /\ ~q
```

Recall from last time that p -> q is equivalent to ~p \/ q. We can make that replacement:

```
  ~(~p \/ q) ?=  ~p /\ ~q
```

We can use De Morgan's law 2 where p is replaced with ~p:

```
  ~~p /\ ~q  ?=  ~p /\ ~q
```

Now we can replace double negation:

```
   p /\ ~q  ?=  ~p /\ ~q
```

We know those aren't equivalent. If, for example, p is True and q is False, the left side is True and the right is False.

Thus, ~(p -> q) is not equivalent to ~p /\ ~q.