CSU290 Lecture Notes Lecture 5 18 Sept 2008

Peter Dillinger & Pete Manolios

Announcements

* First homework due 23rd (Tuesday)

* First exam on 24th (Wednesday)

Design Recipe: ACL2 version

```
* Problem analysis and data definition
* Contract and purpose        (no effects in ACL2)
* Examples as tests           (no need to write examples in comments.  just
  (use CHECK=)                 write tests, which serve as examples.)
* 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 where appropriate.
       If contract allows cons data,
         <To be studied/revealed in later lectures>

  2. "Minimize IFs"
     Don't use more IFs than needed.  They are only needed for
     * Solving the problem on contract inputs
     * Eliminting unintended recursion (#1).

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

Notes/Discussion:

Basically, #1, #2 are to make logical reasoning about functions easier when we get to that, and those numbers are their respective priorities. #1 is more important than #2.

If a function does not make any recursive calls--if it never calls itself--there are no parameters that affect how many recursive calls it makes. In this case, just follow #2.

Consider the SIMPLE-INTEREST function,

```
  (defun simple-interest (p r)
    (* p (+ 1 r)))
```

There was nothing to do for #1. #2 tells us that we don't need to add checks and do something special for out-of-contract cases. Just using the arithmetic functions is fine.

Tests outside the contract/intended domain might include:

```
  (check= (simple-interest nil nil) 0)
  (check= (simple-interest 5 "hi") 5)
```

Consider a definition of the COMPOUND-INTEREST function before making it total,

```
(defun compound-interest (p r i)
  (if (= i 0)
    p
    (compound-interest (simple-interest p r) r (- i 1))))
```

In this function, i is the only parameter that determines how many recursive calls are made. We can tell this from the test (zp i) and the replacement for i, (- i 1). Thus, p and r only affect the value that is returned, not the recursion itself.

In accordance with #1, we make sure everything that is outside the intended domain goes to the base case, p. One way to do this would be

```
(defun compound-interest (p r i)
  (if (not (natp i))
    0
    (if (= i 0)
      p
      (compound-interest (simple-interest p r) r (- i 1)))))
```

But that would violate #2 because it uses more IFs than needed. We can use ZP and not increase the number of IFs:

```
(defun compound-interest (p r i)
  (if (zp i)
    p
    (compound-interest (simple-interest p r) r (- i 1))))
```

Cons and lists

Conses are ordered pairs of objects. They are used commonly used objects in ACL2.

Conses are sometimes called ``lists,'' ``cons pairs,'' ``dotted pairs,'' or ``binary trees.''

There are many ways to write a given list. This should not be surprising. Consider the fact that 123, 000123, and 246/2 are all ways to write down the same numeric constant, and stylistic considerations determine which form you use. So too with list constants.

Any two objects may be put together into a cons pair. The cons pair containing the integer 1 and the integer 2 might be writen as <1, 2> or drawn as follows.

```
  /\
 1  2
```

In ACL2 it is written as (1 . 2) and is created with (cons 1 2).

Quoting (a brief digression)

We learned before that if we type something like

```
(1 . 2)
```

into ACL2 it will throw an error, because it expects ``1'' to name a function to call. But if you do type this, you get that result:

```
ACL2 >(cons 1 2)
```

```
  (1 . 2)
  ACL2 >
```

What gives?

The key is evaluation. (1 . 2) describes a cons pair, but if we try to evaluate it, it tries to treat it as code, which fails. (cons 1 2) is valid code that evaluates to (1 . 2).

If we put a quote mark before an object, it will evaluate to that object. So instead of writing (cons 1 2) to use the object (1 . 2) in some code, we can write '(1 . 2).

```
  ACL2 >'(1 . 2)
  (1 . 2)
  ACL2 >(equal (cons 1 2) '(1 . 2))
  T
  ACL2 >
```

We will try to use a quote whenever we are describing an object--so that there's no ambiguity as to whether we are describing an object or code that produces it.

It's OK not to completely understand quoting at this point. Notice how we use it and you'll start to understand it.

Back to ``Cons and lists''

'(1 . 2) is a single cons object in ACL2 with two integer objects as constituents. The left-hand constituent is called the CAR of the pair. The right-hand constituent is called the CDR. These are functions for accessing those constituents:

```
  (car '(1 . 2))    = 1
  (cdr '(3 . 4))    = 4
  (car (cons 5 6))  = 5
```

Recall that CONSP is the predicate for cons pairs. All other data are atoms. Thus, every ACL2 either satisfies CONSP or ATOM, but not both.

```
  (consp '(1 . 2))   = t
  (consp nil)        = nil
  (consp 5/4)        = nil
```

Let's look at nested conses. <1, <2, 3>> may be drawn as:

```
      /\
     1 /\
      2  3
```

and can be written in ACL2 as '(1 . (2 . 3))

What about '(1 . (2 . (3 . nil)))?

What tree corresponds to this?

Does every binary tree correspond to a cons? Yes.

The notation we are using to write list constants is called *dot notation*. It is a straightforward translation of the familiar Cartesian coordinate notation in which parentheses replace the

brackets and a dot (which must be surrounded by whitespace) replaces the comma.

ACL2 has two syntactic rules that allow us to write cons pairs in a variety of ways. The first rule provides a special way to write trees like (1 . nil). This tree may be written as (1). That is, if a cons pair has the symbol nil as its cdr, you can drop the ``dot'' and the nil when you write it.

Examples:

```
(equal '(1 . ((2 . nil) . (3 . nil)))
       '(1 . ((2) . (3))))
=
t
```

The second rule provides a special way to write a cons pair that contains another cons pair in the cdr: you may drop the dot and the balanced pair of parentheses following it. Thus (x . (...)) may be written as (x ...). For example, '(1 . ((2) . (3))) simplifies to '(1 (2) 3).

Another example: '(1 . (2 . (3 . nil))) may be written as:

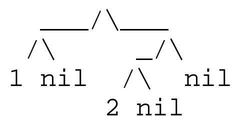
 - '(1 . (2 . (3)))
 - '(1 . (2 3))
 - '(1 2 3)
 - '(1 2 . (3 . nil))
 - etc.

Binary trees that terminate in nil on the right-most branch, such as the tree above are called *true lists*. By extension, the symbol nil is also called a (true) list or, in particular, the *empty list*, and is sometimes written (). Note that the empty list is equal to nil.

```
  (equal nil ())  = t
```

The *elements* of a list are the successive cars along the ``cdr chain.'' That is, the elements are the car, the car of the cdr, the car of the cdr of the cdr, etc. The elements of the list (1 2 3) = (1 . (2 . (3 . nil))) are 1, 2, and 3, in that order. There are no elements in the empty list.

Of course, the elements of a list may be conses. Consider the tree:



This is equivalent to

```
  '((1 . nil) . ((2 . nil) . nil))
= '((1)       . ((2)           ))      (got rid of three ``. nil''s)
= '((1)          (2))                  (got rid of ``. (   )'')
```

This happens to be a true list containing two elements. The first element is a (true) list with one element, the number 1. The second element is also a single-element list, containing the number 2.

How about

```
     _/\
    /\  nil
 nil  nil
```

This is

```
  '((nil . nil) . nil)
= '((nil))
```

Beware:

```
  (equal nil '((nil)))    = nil
  (consp '((nil)))        = t
  (equal '(nil) '((nil))) = nil
```

Cons structures with nils are not the same as nil! And adding parentheses around something changes its meaning. '(nil) is '(nil . nil), which is

```
    /\
 nil  nil
```

So the nil list is different from the list containing nil, which is different from the list containing the list containing nil.

Finally, lets return to

```
  /\
 1 /\
  2  3
```

Why not use this to represent the list containing 1, 2, and 3, rather than

```
  /\
 1 /\
  2 /\
   3  nil
```

After all, the first uses only two Cons pairs, while the second uses three!

Consider a two-element list using the first method:

```
  /\
 1  2
```

What would a 1-element list be? The element itself? What would an empty list be? nil? How would you represent the list with nil as its only element?

In addition to providing an unabiguous way of representing any list, the second method, the standard method, has each element of the list as the CAR of a CONS--the first element of a sublist. This gives it a nice recursive structure.

Finally finally, note that CAR and CDR are ACL2 functions and, thus, must return a value even if their parameter is not a cons. In that case, they return nil:

```
  (car 54)    = nil
  (cdr "hi")  = nil
  (car t)     = nil
  (cdr (cdr '(1 . 2))) = nil
```

But CAR or CDR returning nil does not necessarily imply the parameter is

not a cons:

```
  (cdr '(1))        = nil
  (car '(nil . 3)  = nil
```