CSU290 Lecture Notes Lecture 4 17 Sept 2008

Announcements

* First homework due 23rd (Tuesday)

* First exam on 24th (Wednesday)

IF & Generalized Booleans

Recall the ‘‘booleans’’?

```
  t     stands for "true"
  nil   stands for "false" (and has other uses)
```

We briefly introduced IF, which takes three arguments:

```
  (if <test> <true_part> <false_part>)
```

Thus, for example,

```
  (if (= 1 1) "yes" "no")  = "yes"
  (if (= 1 2) "yes" "no")  = "no"
```

("yes" and "no" are strings, of course.)

But what about totality? What if the test of an IF is something other than a boolean?

This brings us to the notion of "generalized booleans," which ACL2 inherits from Common Lisp. In a boolean context, NIL is the only value that stands for ‘‘false’’ and everything else stands for ‘‘true’’. T is just the canonical value for ‘‘true’’.

```
  (if 42    1 2)   = 1
  (if "nil" 1 2)   = 1     ("nil" is a string)
  (if #\a   1 2)   = 1     (#\a is a character)
  (not -3/2)       = nil
  (not 0)          = nil
  (not nil)        = t     (the only value for which NOT returns T)
```

The boolean connectives AND and OR are a bit more complicated because if you pass them values other than just T and NIL, they can return values other than T and NIL. But they behave as expected with regard to the generalized boolean notion of ‘‘true’’ being non-NIL:

```
  (if (and 42 0)   1 2)  = 1
  (if (and nil 0)  1 2)  = 2
  (if (or nil 0)   1 2)  = 1
  (if (or nil nil) 1 2)  = 2
```

That’s all we really need to know about AND and OR. We know when they return NIL and when they return non-NIL.

More functions

```
; SIMPLE-INTEREST:  rational rational -> rational
; Takes a principle amount and a rate of interest and returns
; the amount that must be repaid.
; Amount = Principle * (1 + Rate)
; (see tests for examples)
(defun simple-interest (p r)
  (* p (+ 1 r)))

(check= (simple-interest 100 5/100) 105)
(check= (simple-interest 10 10/100) 11)
(check= (simple-interest 1 7/100) 107/100)
(check= (simple-interest 0 5/100) 0)
(check= (simple-interest 5 0) 5)
(check= (simple-interest nil nil) 0)



; COMPOUND-INTEREST: rational rational nat -> rational
; Takes a principle amount, a rate of interest, and a number of
; times the interest has compounded and returns the amount that
; must be repaid.  This is equivalent to compounding SIMPLE-INTEREST
; the number of times specified.
(defun compound-interest (p r i)
  (if (zp i)
    p
    (compound-interest (simple-interest p r) r (- i 1))))

(check= (compound-interest 100 5/100 0) 100)
(check= (compound-interest 1 7/100 1) 107/100)
(check= (compound-interest 100 10/100 2) 121)
```

What about (compound-interest t "hi" nil)? It returns T. Is that ok? T is not in the intended output range of the function, but we did not give the function something in the intended input domain. Briefly, the parameters p and r do not control when the function terminates; only i does--based on (zp i). Consequently, we won’t put any extra effort into making sure we only compute on p and r if they are in the expected input domain. We’ll look more deeply at this in the future.

Errors

We saw last time that there aren’t any runtime ‘‘type’’ errors; ACL2 functions are untyped and total. But it’s not true that anything we right down gives us an answer. There are many *static* errors, in which ACL2 rejects an expression or definition before it tries to execute it.

If we place something in the function position of an expression which is not a function symbol, then we get an error:

```
  ACL2 > (10 20 30)

  ACL2 Error in TOP-LEVEL:  Function applications in ACL2 must begin
  with a symbol or LAMBDA expression.  (10 20 30) is not of this form.

  ACL2 >
```

Functions cannot be used as values:

```
  ACL2 > (+ endp consp)

  ACL2 Error in TOP-LEVEL:  Global variables, such as CONSP and ENDP,
  are not allowed. ...

  ACL2 >
```

And you must give functions the correct number of arguments. Unlike advanced Scheme, in which this check (and some of the previous) are made a run time, ACL2 makes this check before executing anything:

```
  ACL2 > (if t 42 (booleanp 1 2))

  ACL2 Error in TOP-LEVEL:  BOOLEANP takes 1 argument but in the call
  (BOOLEANP 1 2) it is given 2 arguments.   The formal parameters list
  for BOOLEANP is (X).

  ACL2 >
```

Note that in that expression (booleanp 1 2) would not be evaluated. (By the way! IF is a special function in that it only evalutes either the <true_part> or the <false_part>, never both. That is pretty important to termination of recursive functions.)

We will encounter some other kinds of static errors, and we will consider those in more detail as they become more important.

More Functions

We saw how ZP is useful for recursions that count down to zero. Let’s consider some variations on that and how that changes the job of writing a function that terminates on all inputs.

Let’s write a function LOG2 that computes the base-2 logarithm of a number--not the exact logarithm but the next highest integer. Recall that the base-2 logarithm of a number is the power to which 2 must be raised to get that number. With our rounding up, we are essentially counting how many powers of 2 are less than a number. The powers of two are

```
  2^0 = 1
  2^1 = 2
  2^2 = 4
  2^3 = 8
  ...
```

So we want

```
  (log2 1)  = 0
  (log2 2)  = 1
  (log2 3)  = 2
  (log2 4)  = 2
  (log2 5)  = 3
  (log2 8)  = 3
  (log2 16777216) = 24
  (log2 16777217) = 25
```

The key observation in solving this problem recursively is the mathematical equation

```
  log2 (2 * x) = 1 + log2 x
```

or

```
  log2 x  =  1 + log2 (x/2)
```

And we have the base case that

```
  log2 1 = 0     (because 2^0 = 1)
```

Now let us begin to write this function

```
  ; LOG2: pos -> nat
  ; computes the log base 2 of the positive natural parameter, rounded
  ; up to the nearest natural.
  ; (see tests as examples)

  (defun log2 (x)
     <body>)

  (check= (log2 1) 0)
  (check= (log2 2) 1)
  (check= (log2 3) 2)
  (check= (log2 4) 2)
  (check= (log2 5) 3)
  (check= (log2 8) 3)
  (check= (log2 16777216) 24)
  (check= (log2 16777217) 25)
```

Based on the recursive decomposition of logarithms (math formula above),
it will probably look like this:

```
  (defun log2 (x)
    (if <test>
      0
      (+ 1 (log2 (/ x 2)))))
```

So what should <test> be? If all we check is (= x 1), then our
function will not terminate for rationals less than one, negative
rationals, and many other inputs. (Feel free to test it!)

Instead, we should make our job easier by only worrying about our
recursion terminating on the intended inputs. Thus, we’re going to
treat everything less than 1 or not a natural number as 1 and return 0.
(If you foresee trouble, you might be correct.) There are many ways we
can express this base case:

```
  (or (not (posp x))
      (= x 1))
  (or (zp x)
      (= x 1))
  (or (zp x)
      (<= x 1))
  (or (not (integerp x))
      (<= x 1))
```

So let’s go with

```
  (defun log2 (x)
    (if (or (not (integerp x))
            (<= x 1))
      0
      (+ 1 (log2 (/ x 2)))))
```

Does that pass all our tests? No!?!

```
  ACL2 > (check= (log2 3) 2)

  ACL2 Error in CHECK=:  Check failed (values not equal).
  First value:  1
  Second value: 2
  ACL2 >
```

What happened?

```
  (log2 3)
  =  { 3 is an integer and > 1 }
  (+ 1 (log2 (/ 3 2)))
  =
  (+ 1 (log2 3/2))
  =  { 3/2 is NOT an integer! }
  (+ 1 0)
  =
  1
```

Oops! We forgot to make sure that our recursive calls adhere to the input specification (type ‘‘pos’’). It is possible to modify our recursive calls to fix this, but we haven’t learned the functions to do that. What we can do, is make this function operate on rational numbers:

```
  ; LOG2: rational -> nat
  ; computes the log base 2 of the given rational, rounded up to the
  ; nearest natural.
  ; <examples from above would go here>

  (defun log2 (x)
    (if (or (not (rationalp x))
            (<= x 1))
      0
      (+ 1 (log2 (/ x 2)))))
```

NOTE: This LOG2 function is in some ways more complicated than what you will be expected to write. The point was to encounter some problems by trying to solve an atypical problem without thinking about it very deeply.

In fact, the termination proof for this function is complicated enough that ACL2 needs a little help to admit it for logical reasoning--in modes outside of of Programming Mode. This function uses rational numbers in figuring out when it’s in a base case, and you’re not likely to see that again. But you might very well encounter similar problems if you blindly try to apply simple solutions. Think!