CSU290 Lecture Notes Lecture 3 15 Sept 2008

Announcements

* Get the book! (Order A.S.A.P!)

* Download, install, & start using ACL2s

* First homework T.B.A.

ACL2 Values

All data objects in ACL2 can be categorized as follows: (you don’t have to remember all of these)

```
(Description)                       (Predicate/recognizer)
* Atomic data ("atoms")             ATOM or ENDP
  include:
  * ACL2 Numbers                    ACL2-NUMBERP
    are either:
    * Rationals                     RATIONALP
      include:
      * Integers                    INTEGERP
    * Complex rationals             COMPLEX-RATIONALP
  * Symbols                         SYMBOLP
    include:
    * Booleans                      BOOLEANP
    * Keywords                      KEYWORDP
  * Strings                         STRINGP
  * Characters                      CHARACTERP
* Cons pairs (compound data)        CONSP
```

One thing to notice is that ACL2 does not include "inexact" or floating-point numbers natively.

Also, the only way to build compound data is with Cons pairs. We will discuss this in more detail soon.

Here are the categories you should know:

```
(Description)      (Predicate)      (Examples)
* Atoms            ATOM or ENDP
  include:
  * Rationals      RATIONALP         3/4  -20/3
    include:
    * Integers     INTEGERP          -7  0  523
  * Symbols        SYMBOLP           ’green  ’two
    include:
    * Booleans     BOOLEANP          t  nil   (exhaustive)
  * Strings        STRINGP           "hi"   "Who, me?"
  * Characters     CHARACTERP        #\A  #\q
* Cons pairs       CONSP             ’(1 . 2)
```

Booleans

ACL2 and Common Lisp have two special constant symbols that are used, among other things, as boolean values:

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

All of the predicate functions above return booleans. For example,

```
  (booleanp t)   = t      (It is true that t is a boolean)
  (booleanp nil) = t      (It is true that nil is a boolean)
  (booleanp 7)   = nil    (It is false that 7 is a boolean)
```

t and nil are also symbols, which are atoms:

```
  (symbolp t)   = t      (It is true that t is a symbol)
  (symbolp nil) = t      (It is true that nil is a symbol)
  (atom t)      = t      (It is true that t is an atom)
  (consp nil)   = nil    (It is false that nil is a cons pair)
```

Another function that always returns a boolean is EQUAL, which tests data objects for equality:

```
  (equal nil nil)  = t
  (equal nil t)    = nil
  (equal T t)      = t      (Case is ignored when reading symbols)
```

A few functions that are useful on booleans are NOT, AND, and OR:

```
  (not t)              = nil
  (not nil)            = t
  (not (equal nil t))  = t
  (and t t)            = t
  (and nil t)          = nil
  (or nil t)           = t
  (or nil nil)         = nil
```

AND and OR are special in that they can take any number of parameters. AND returns "true" iff all of its parameters are "true", and OR returns "true" iff at least one of its parameters is "true":

```
  (and (equal nil nil)
       (equal t t)
       (not (equal t nil)))  = t
  (and t)                    = t
  (and nil)                  = nil
  (and)                      = t    (All (zero) parameters are "true")
  (or)                       = nil  (No parameter is "true")
  (or t)                     = t
  (or nil)                   = nil
  (or (equal nil t)
      (equal t t)
      (not (equal t t)))     = t
```

Numbers

As mentioned, all ACL2 numbers are exact, and their size is (in theory) unbounded. This makes ACL2 numbers behave as they do in mathematics, but we will stick with just rational numbers--those that can be represented as one integer divided by another.

But first we consider those rationals with a denominator of 1: the
integers. There are many ways to write integers in ACL2--all refering
to the same integers--but the standard way is fine for us:

```
  (integerp -5)    = t
  (rationalp 37)   = t
  (equal 42 042)   = t
```

We can compare them:

```
  (< 4 5)    = t
  (<= 4 -5)  = nil
  (> 0 -5)   = t
  (>= 7 7)   = t
```

We can also do some standard arithmetic:

```
  (+ 4 3)   = 7
  (- 3 5)   = -2
  (* 2 -5)  = -10
  (/ 10 5)  = 2
```

These functions are special, however, because they can different numbers
of parameters. For example, + and * can take any number of parameters:

```
  (+ 5)      = 5
  (+ 3 2 1)  = 6
  (* -2)     = -2
  (* 1 2 3)  = 6
  (+)        = 0   (Additive identity, the sum of zero numbers)
  (*)        = 1   (Multiplicative identity, the product of zero numbers)
```

- and / can take one or two parameters. Given one argument, they perform
negation and multiplicative inversion respectively:

```
  (- 5)    = -5
  (- -10)  = 10
  (- 0)    = 0
  (/ 1)    = 1
  (/ 5)    = 1/5
  (/ -2/3) = -3/2
```

And that brings us to non-integer rationals. Equality among rationals
is mathematical equality:

```
  (equal 7/9 14/18)  = t
  (equal 8/2 4)      = t
```

In fact, ACL2 automatically puts rationals in lowest terms, as shown
when printing them out or accessing the numerator or denominator:

```
  -4/6          = -2/3
  20/4          = 5
  (/ -6 -4)     = 3/2
  (/ 15/6 1/2)  = 5
  (numerator -4/6)   = -2
  (numerator 22/8)   = 11
  (numerator -5)     = -5
  (denominator -4/6) = 3
  (denominator 22/8) = 4
  (denominator -5)   = 1
  (denominator 0)    = 1
```

Arithmetic on rationals works as expected:

```
  (+ 2/3 3/5)   = 19/15
  (+ 3/4 5/6)   = 19/12
  (* 2/3 3/5)   = 6/15
  (* 3/4 5/6)   = 5/8
```

Which raises an interesting question: what is (/ 1 0) ? We know in mathematics that anything divided by zero is undefined. So does ACL2 throw an error or what?

Introduction to Totality

ACL2 functions are total, which means every function call returns a value. This is one of the fundamental design choices in the Boyer-Moore theorem prover, ACL2. For now, you can consider this choice to keep some things simple by eliminating exceptional cases. For example, ACL2 defines anything divided by 0 to be 0. Consequently, division in ACL2 always returns a number, and we don’t need to figure out anything about the input to come to that conclusion. (More on this stuff comes later in the course.)

```
  (/ 1 0)   = 0
  (/ 0 0)   = 0
```

Also for totality, arithmetic functions treat non-numbers as 0. Here are some examples:

```
  (+ 5 nil)    = 5
  (* t 12)     = 0
  (- nil)      = 0
  (/ 5 nil)    = 0
  (< nil 5)    = t
  (> nil 5)    = nil
  (< -5 nil)   = t
```

Totality also means there are no "type errors" in which a function was expecting one type of input and got another. The only thing we have to get right is the number of parameters, and that’s an easy check for ACL2:

```
 ACL2 >(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 >
```

Back to Numbers, with Function Definition

Here are some extra functions pertaining to numbers:

```
  NATP - predicate for natural numbers  (0, 1, 2, ...)
  POSP - predicate for positive naturals (1, 2, 3, ...)
  ZP   - (not (posp x))
```

We will see how useful ZP is in our first function definition. Let us define the factorial function, call it FACT. Here are examples of how factorial is written and defined in mathematics:

```
  5! = 5 * 4 * 3 * 2 * 1
  2! = 2 * 1
  1! = 1
  0! = 1
```

So for any integer greater than 0, the factorial is that integer times the factorial of one less than the integer. (Recursion!) Here’s a first try at a definition:

```
  ; fact: nat -> nat
  ; Computes the factorial function
  ; (examples above)

  (defun fact (x)
    (if (= x 0)
      1
      (* x (fact (- x 1)))))
```

An IF looks like

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

If the test is true, then it evaluates to the true_part, otherwise the false part.

Our definition works on natural numbers, but is it total? Does it return a value on all inputs?

The answer is No. If we give it a negative number (or indeed, a non-number) it does not terminate:

```
  (fact -1)
  =
  (* -1 (* -2 (* -3 ...
```

A solution is to treat everything outside the intended domain, the naturals, as a base case, 0. So the new base case is 0 or anything not a natural number. If only we had a function that captured all of that...

```
  (defun fact (x)
    (if (zp x)
      1
      (* x (fact (- x 1)))))
```