CSU290 Lecture Notes Lecture 13 15 Oct 2008

Peter Dillinger

## Review

A tautology is always true in a given logic. A theorem is always true in a given *theory* under a given logic. We won't ask you to make tough distinctions between these two, but we will soon talk more about logical theories, and that will give you more of an understanding of the essence of the distinction.

First order logic has explicit quantifiers. ACL2 has implicit universal quantification, but we can still state everything we can in first order logic. (We saw how to replace existential quantification with witnesses.)

Implicit universal quantification of free variables means that ACL2 has no distinction between formulas and propositions. The “Generalized Boolean,” in which NIL stands for *false* and anything else stands for true, means that there is no distinction between formulas and expressions. Thus, some simple expressions such as

```
5

'(1 . 2)

(+ 3 4)

"hi"
```

are technically theorems, because they are *true* for all possible assignments to free variables (of which there are none).

## Using the ACL2 logic and theorem prover

We can ask ACL2 to attempt to prove theorems using THM, like so:

```
(thm 5)

(thm "hi")

(thm (or (atom x)
         (consp x)))
```

Now that we are working with ACL2 logically, Programming mode no longer suffices. (Programming mode will not attempt any proofs.) For the rest of CSU290, you should use ACL2s' **Intermediate mode**. We won't be looking at the output of THMs in any detail until the end of the term when we learn to use ACL2's proving capabilities on more complex problems. For now, we will treat THM as something that tells us either, “Yes, ACL2 could prove this automatically, so it is a theorem,” or “ACL2 was unable to prove this automatically, so it may or may not be a theorem.”

Although we aren't discussing exactly how ACL2's THM works, we should discuss how THM doesn't or can't work. For most interesting theorems, THM cannot work by trying all the possibilities and checking that the formula evaluates to *true* (non-NIL) in each case. This is because the possibilities are infinite. Consider trying all the possible ACL2 values on (or (atom x) (consp x)). You would have to try x=0, x=1, x=2, ... Even the natural numbers are infinite!

On the other hand, it only takes one example to show an ACL2 proposition is not a theorem. Consider the proposition that every value is either greater than five or less than five. I could ask ACL2 to prove that as follows:

```
(thm (or (> x 5)
         (< x 5)))
```

It fails, but that does not tell us definitively whether it is a theorem or not. If we can find a **counterexample**, an assignment of values to the free variables, that makes the proposition *false* (NIL), that tells us definitively that the proposition is *not* a theorem.

What if I let x be 5? That is neither greater than 5 nor less than 5, so that should falsify the proposition—make it NIL. Here's how we can check that our counterexample does just that:

(check=

We can also relate the notion of counterexamples back to first order logic. The above ACL2 proposition essentially says

```
∀x (or (> x 5)
       (< x 5))
```

which is the same as

```
¬ ¬ ∀x (or (> x 5)
           (< x 5))
```

We learned last time from identities involving quantifiers that that is the same as

```
¬ ∃x ¬ (or (> x 5)
           (< x 5))
```

We can conclude that is true by providing a witness for x, our counterexample:

```
¬ ¬ (or (> 5 5)
        (< 5 5))
```

which is the same as

```
¬ ¬ nil
```

and in first order logic terms, that is

*false*

## More formalizing

Formalize—write an ACL2 proposition for—this conjecture: Every value is less than or equal to 5 if and only if it is not greater than 5.

```
(thm (iff (<= x 5)
          (not (> x 5))))
```

I wrote this with THM to ask ACL2 to prove it, and it does.

How about, "The length of appending two things is the sum of the lengths of the two things."

```
(thm (equal (len (append x y))
            (+ (len x) (len y))))
```

ACL2 can prove that.

How about, “Appending two true lists results in a true list.”

```
(thm (implies (and (true-listp x)
                   (true-listp y))
              (true-listp (append x y))))
```

And ACL2 can prove that.

## A Fun Problem

A Mersenne prime is a prime number that is equal to $2^n - 1$ for some natural number n. Not all numbers that are $2^n - 1$ are primes. 3 is a Mersenne prime. 7 is also. 15 is not. 31 is.

An unsolved problem in mathematics is whether there are an infinite number of Mersenne primes. Let's write a function that generates them. We can ask ACL2 to include something that already defines what a prime number is with a predicate PRIMEP:

```
(include-book "quadratic-reciprocity/euclid" :dir :system)
```

We can check to make sure PRIMEP is working on some examples:

```
(check= (primep 6) nil)

(check= (primep 7) t)

(check= (primep 6563) t)
```

Now let's write a function that takes the n in $2^n - 1$ and finds the next Mersenne prime for a larger n:

```
(defun next-mprime (n)
  (let ((v (- (expt 2 (+ 1 n)) 1)))
    (if (primep v)
      v
      (next-mprime (+ 1 (nfix n))))))
```

Basically, it checks whether $2^{n+1} - 1$ is prime. If it is, it returns that value. If not, it makes the recursive call to keep checking larger natural numbers.

Programming mode accepts this definition but Intermediate mode does not. For now, we will force Intermediate mode to accept it. Do not concern yourself with the details:

```
(set-termination-method :measure)

(skip-proofs
  (defun next-mprime (n)
    (let ((v (- (expt 2 (+ 1 n)) 1)))
      (if (primep v)
        v
        (next-mprime (+ 1 (nfix n)))))))
```

Do you recall in the last lecture how we formalized that there is no largest integer? We will do something similar here, asking ACL2 to prove that for any integer n, there is a Mersenne prime greater than $2^n - 1$:

```
        (thm (implies (posp n)
                      (primep (next-mprime n))))
```

If I ask ACL2 to prove this, IT PASSES! Wait!? Did I, with the help of ACL2, just solve an unsolved math problem?

The answer is no. ACL2 rejected the function definition because it could not prove that the function terminates. Remember how we require functions to be total and terminate on all inputs? Well, when we leave Programming mode, that requirement is enforced by ACL2 attempting to prove termination of functions before it accepts them.

We have just seen how important termination is. The "skip-proofs" told ACL2 to just believe us that the function terminates, and using that assumption, it was able to prove that there are an infinite number of Mersenne primes. In fact, look at the function. If there are an infinite number of Mersenne primes, it will terminate for all inputs. If there are *not* an infinite number of Mersenne primes, there must be an n beyond which next-mprime does not terminate—because it will search for the next one forever!

Why exactly is termination so important? It establishes a connection between **computational functions** and **mathematical functions**. When we write a function in a programming language, it might not return a value for some inputs because it would run indefinitely on those inputs. Mathematical functions, on the other hand, have no notion of "computing" an output from an input. They are like infinite look-up tables from inputs to outputs.

The ACL2 logic assumes functions can be treated as mathematical total functions. Requiring termination on all inputs guarantees this is the case. Here is an example of a non-terminating function that causes serious problems:

```
        (defun bad (x)  (not (bad x)))
```

We haven't talked much about proving, but we can use boolean logic identities to find equivalent formulas:

*true*

↔ { boolean identity }

`(bad x)` \/ `(not (bad x))`

↔ { by definition of BAD }

`(bad x)` \/ `(bad x)`

↔ { boolean identity }

`(bad x)`

↔ { boolean identity }

`(bad x)` /\ `(bad x)`

↔ { by definition of BAD }

`(bad x)` /\ `(not (bad x))`

↔ { boolean identity }

*false*

I have just used that definition to prove *true* is the same as *false*, a contradiction.