CS 2800: Logic and Computation Fall 2010
(Lecture 14)

14 October 2010

1 Intro to ACL2 Logic

I finished the proof for "There is no largest integer", you can find that in the last lecture notes.

After that I said, ACL2 Logic is similar to FOL in many ways, but there are a few differences.

1. *ACL2 Logic does not have any (explicit) quantifiers*
2. *All (free) variables in ACL2 formulas are (implicitly) universally quantified*
   e.g. `(or (atom x) (consp x))`
   means that for all values of x, i.e. ranging over all the objects of the ACL2 Universe, x satisfies the predicate functions `atom` and `consp`. In the logic and theory of ACL2, this is a theorem, because there are no values of x that make this *false*(nil).
3. *Existential quantification is emulated using witnesses*
   e.g. we saw how we got rid of the existential quantifiers in the FOL formula for "there is no largest integer". We can formalize the final formula in ACL2:

```
(implies (integerp x)
           (and (integerp (+ x 1))
                (> (+ x 1) x)))
```

4. *There are no relation symbols in ACL2 Logic, they are emulated using predicate functions(i.e. return* `t` *or* `nil` *as output)*

5. *The distinction between terms(expressions) and formulas in ACL2 is blurred due to the concept of generalized booleans(nil and non-nil) and Note 1 mentioned below.*
   e.g. The following expressions are not only formulas in ACL2, but also theorems, since they evaluate to a non-nil object:
   ```
   5
   '(1 2 3)
   (+ 3 4)
   ``hi''
   ```

Recall from Lecture 5, an ACL2 *expression* is one of:

- a variable symbol
- a constant symbol
- a constant expression, or
- application of n-arity function symbol to n expressions $a_1, \ldots, a_n$,written as $(f \quad a_1 \ldots a_n)$

Assuming ***expr1*** ,***expr2*** are ACL2 expressions, we can precisely define ACL2 *formulas*:

- `(equal` ***expr1 expr2***`)` is a (atomic) formula
- If ***form*** is an ACL2 formula, so is `(not` ***form***`)`
- If ***form1*** and ***form2*** are ACL2 formulas, then so is (□ ***form1 form2***), where □ is one of `and, or, implies, iff`

Notes:

1. To be pedantic, any ACL2 expression `expr` used in the context of a formula, can be thought of as an abbreviation for `(not (equal expr nil))`, i.e. we can think of it as (= *expr* non-nil). This blurs the distinction between ACL2 formulas and expressions.

2. Forcing the user to use witnesses as opposed to existential quantifiers (immensely) helps ACL2 to prove theorems automatically. AFAIK, no Interactive Theorem Prover in the world has automation comparable to ACL2.

3. There is another problem with writing quantifiers: they are not programming language constructs; they are not computable. How would one compute:

   ```
   (exists x (and (perfect-numberp x)
                  (oddp x)))
   ```

   Would you try all possible values of x? Even if you only tried all the natural numbers, they are infinite! (In this case, I'm looking to solve an unsolved problem in mathematics: the existence of an odd perfect number(See wikipedia for more information).
   So a nice thing about ACL2 formulas is that they are executable – if you assign a value to all the variables. For example, we could test our conjecture on there being no largest integer on the value 1000 like so:

   ```
   (let ((x 1000))
     (implies (integerp x)
              (and (integerp (+ x 1))
                   (> (+ x 1) x))))
   ```

   And that evaluates to `T`, meaning it is *true*.

4. Since ACL2 is quantifier-free and free variables are implicitly universally quantified, all ACL2 formulas are propositions. Thus, all ACL2 expressions are propositions. Here are some ACL2 theorems. They are theorems because there are no assignments to variables under which they evaluate to nil:

   - `(< 1 2)`
   - `t`
   - `(integerp 5)`
   - `(implies (natp x) (integerp x))`
   - `5`
   - `"false"`

2 Using ACL2 Logic and the theorem prover

We can ask ACL2 to attempt to prove theorems using `thm`, like:

```
(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 CS2800, you should use the ACL2s session mode **Intermediate mode**. We won't be looking at the output of `thm` 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, ACL2s could prove this automatically, so it is definitely a theorem," or "ACL2s was able to find counterexamples, in which case its definitely not a theorem" or "ACL2s was unable to prove this automatically, and unable to find counterexamples 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(and no practical machine can check infinite possibilities). Consider trying all the possible ACL2 values on `(or (atom x) (consp x))`. You would have to try $x = 0, x = 1, x = 2, \ldots$. Even the natural numbers are infinite, which is just the subset of the whole ACL2 Universe! If you ask ACL2 to prove the above using `thm` it will succeed("Yes! its a theorem") and give a small, finite proof(whats a proof will be explained in a future class). This is pretty awesome because a proof gives us a finite way of running an infinite number of examples. Thats the power of logic and mathematics.

On the other hand, it only takes just one example(a counterexample) 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(unless it explicitly gives a counterexample). If we can find a counterexample(by ourselves), 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= ... nil)
```

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

$$\forall x.(\text{or } (> x\ 5)\ (< x\ 5))$$

which is the same as

$$\neg\neg\forall x.(\text{or } (> x\ 5)\ (< x\ 5))$$

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

$$\neg\exists x.\neg(\text{or } (> \text{x}\ 5)\ (< x\ 5))$$

We can eliminate the existential quantifier providing a witness for x, which will serve as our counterexample:

$$\neg\neg(\text{or } (> 5\ 5)\ (< 5\ 5))$$

which is the same as

$$\neg\neg\text{nil}$$

and in FOL terminology, that is

$$\mathit{false}$$

In these 2 cases, it was easy to decide if a conjecture was true (`thm` gives a proof) or false(with a good amount of testing, we would have identified the false conjectures).

Is this always the case?

No, it is not!

Anyone heard of Fermats last theorem?
For all positive integers x, y, z, and n, where $n > 2$,

$$x^n + y^n \neq z^n$$

In 1637, Fermat wrote about the above: "I have a truly marvelous proof of this proposition which this margin is too narrow to contain."
This is called Fermats Last Theorem. It took 357 years for a correct proof to be found (by Andrew Wiles in 1995). Can someone use the above to construct a conjecture that would be hard to prove in ACL2?

```
(defun f (x y z n)
  (if (and (posp x)
           (posp y)
           (posp z)
           (natp n)
           (> n 2)
           (= (+ (expt x n) (expt y n))
              (expt z n)))
      1
    0))

(thm (= (f x y z n) 0))
```

So, proving theorems may be hard. But, if they arent theorems, we should be able to find counterexamples quickly, right?
No!
A conjecture may be false, but it may be very hard to find a counterexample.

Let $N^{\geq 2}$ be the set of natural numbers >= 2. Remember that we can factor any number in $N^{\geq 2}$ into a product of primes. This product is uniquely defined. If the number of primes is even, we say that the number is of "even type"; otherwise it is of "odd type". Examples: $2 = 2$ is of odd type and $4 = 2 * 2$ is of even type.

Let
$E(n)$= the number of naturals in $N^{\geq 2}$ that are of even type and are <= n.
$O(n)$= the number of naturals in $N^{\geq 2}$ that are of odd type and are <= n.

Polyas conjecture: $O(n) >= E(n)$ for all n in $N^{\geq 2}$. This conjecture was made in 1919. Check Wikipedia. It was checked for many values of n, and

it was widely believed to be true.
However, in 1962 a counterexample was found (by Lehman) for n=906,180,359.
Phew!

This is related to Computer Science and especially to software. Perhaps the most well-known statement explaining this:
"Program testing can be used to show the presence of bugs, but never to show their absence" – Edsger W Djisktra (EWD249, 1970)

3 Substitution

A **substitution**(σ) is a mapping(a one-to-one function) from the variable symbols to terms(expressions). In CS2800 we will represent it as a list of 2-element lists:

```
Substitution := (listof (list varSymbol expression))
```

If a substitution σ maps variable symbol x to expression e, we say x is a **target variable** of σ whose **image** is e.

The **application of a substitution** σ to a ACL2 formula ϕ , denoted by $\phi|_\sigma$ uniformly replaces, in ϕ, every occurence of the target variable by its image.

Note1: By uniformly we mean, σ replaces **every** occurence of a variable symbol with its unique image.

Note2: The substitution process happens simultaneously. e.g.
ϕ: `(app x y)`, applying the following
σ: `((x (cons y w)) (y (+ a b)) (w 42))` gives:
$\phi|_\sigma$: `(app (cons y w) (+ a b))`
If you think the answer is something else, then you are not applying the substitution simultaneously i.e. in one go.

Do the Lab assignment, to make sure you completely understand substitution since it is an important concept and will be used extensively when we are doing theorem proving.