Very Rough Lecture Notes for CS2800.
Thu Sep 15 2011
Ver. 422
By Pete Manolios

These lecture notes are on the ACL2 logic.

We just finished studying propositional logic, so let's start by considering the following question:

Why do we need more than propositional logic?

Well, we were able to do a lot with propositional logic. Recall the power of xor and digital logic.

But what we are after is reasoning about programs, and while propositional logic will play an important role, we need more powerful logics.

To see why, let's simplify things for a moment and consider conjectures involving numbers and arithmetic operations.

Consider the conjecture:

1. a+b = ba

What does it mean for such a conjecture to be true? false?

Well, there is a source of ambiguity here. If a, b were constants (like 1, 2, etc) then we could just evaluate the two sides of the equalities and determine if they are true or not.

However, a and b are variables. This is similar to the propositional formulas we saw, eg,

(p/\q) => (p\/q)

Recall that p and q are atoms, and the above formula is valid. What that means is that no matter what value p and q have, (p/\q) => (p\/q) is true.

In conjecture 1, a, b range over a different domain than the booleans, let's say they range over the rationals.

So, what we really mean when we say that conjecture 1 is valid is that for any rational a and any rational b, a+b = ba. Notice the similarity with the Boolean case.

Is conjecture 1 a valid formula?

No, we can come up with a counterexample.

What about the following conjecture?

2. a+b = b+a

Can we come up with a counterexample?

No.

How did we prove that something was valid in the case of propositional logic? We used a truth table, with a row per possible assignment, to show that no counterexample exists. A counterexample is an assignment that evaluates to false.

Can we do something similar here?

Yes, but the number of assignments is unfortunately infinite. That means, we can never completely fill in a table of assignments.

We need a radically new idea here.

We want something that allows us to do a finite amount of work and from that to deduce that there are no counterexamples in the infinite table, were we even able to construct it.

Let’s look at how we might do this, but in the context of programs.

Consider the following conjecture:

```
3. (len (cons x y))
 = (len (cons y x))
```

Remember the definition of len:

```
(defunc len (x)
  :input-contract t
  :output-contract (natp (len x))
  (if (atom x)
      0
    (+ 1 (len (cdr x)))))

(defunc atom (x)
  :input-contract t
  :output-contract (booleanp (atom x))
  (not (consp a)))

(defunc not (a)
  :input-contract (booleanp a)
  :output-contract (booleanp (not a))
  (if a nil t))
```

Is this conjecture true or false?

First, what does it mean for it to be true? That no matter what objects of the ACL2 universe x and y are, the above equality holds.

The conjecture is false, eg, suppose x=nil y=(cons 1 nil)

So, finding a counterexample is "easy".

What about:

```
4. (len (cons x z))
 = (len (cons y z))
```

Here we can’t find a counterexample

How can we go about proving this?

```
   (len (cons x z))
 =   {Definition of len, instantiation}
   (if (atom (cons x z))
       0
     (+ 1 (len (cdr (cons x z)))))
```

```
 =   {Definition of atom}
   (if (not (consp (cons x z)))
       0
     (+ 1 (len (cdr (cons x z)))))
 =   {Definition of not}
   (if (if (consp (cons x z)) nil t)
       0
     (+ 1 (len (cdr (cons x z)))))
 =   {consp axioms}
   (if (if t nil t)
       0
     (+ 1 (len (cdr (cons x z)))))
 =   {if axioms}
   (if nil
       0
     (+ 1 (len (cdr (cons x z)))))
 =   {if axioms}
   (+ 1 (len (cdr (cons x z))))
 =   {car-cdr axioms}
   (+ 1 (len z))
```

What we have shown so far is:

5. (len (cons x z)) = (+ 1 (len z))

which we will free to write as

6. (len (cons x z)) = 1 + (len z)

Because it should be clear how to go from 6 to 5 and because we have been trained to use infix for arithmetic operators since elementary school.

We are not done, but there are at least two reasonable ways to proceed.

First, we might say:

If we simplify the RHS (Right Hand Side), we get

```
   (len (cons y z))
 =   {Definition of len, instantiation}
   ...
 =   {car-cdr axioms}
   1 + (len z)
```

So, the LHS (Left Hand Side) and RHS are equal.

What we realized is that the same steps that we used to simplify the LHS can be used in a symmetric way to simplify the RHS. In this class we will avoid "...". Here’s a better way to make the argument:

First, note that we have that 6 is a theorem. By instantiating 6 with the substitution ((x y)), we get:

7. (len (cons y z)) = 1 + (len z)

Together with 6, we have

4. (len (cons x z)) = (len (cons y z))

So, conjecture 4 is a theorem.

We saw instantiation in propositional logic. It is really important in the ACL2 logic! More below.

This example highlights the new tool we have that will allow us to reason about programs: proof. The game we will be playing is to construct proofs of conjectures involving some of the basic functions we have already defined (e.g., len, app, rev). We will focus on these simple functions because they are simple to understand so we can focus exclusively on how to prove theorems and not on understanding what conjectures mean.

Once we prove that a conjecture is valid, we say that the conjecture is a theorem. We are then free to use that theorem in proving other theorems. This is similar what happens when we program: we define functions and then we use them to define other functions (e.g., we define rev using app).

What’s new here?

Well, we are beyond the realm of the propositional.
We have functions, and equality, and variables ranging over the ACL2 universe.

Let’s look at equality. How can we reason about it?

```
Reflexivity:              x = x
Symmetry of Equality:     x = y   =>  y = x
Transitivity of Equality: x = y /\  y = z  =>  y = z
```

This is what allows us to chain together the sequence of equalities in the proof of 5 above.

Equality Axiom Schema for Functions:
For every function symbol f of arity n we have the axiom

```
x1 = y1 /\ ... /\ xn = yn  =>  (f x1 ... xn) = (f y1 ... yn)
```

One more thing. In ACL2, we would write conjecture 4 as:

```
(equal (len (cons x z))
       (len (cons y z)))
```

We will feel free to go back and forth.

If we want to be pedantic, here is what we know about equal.

```
x  = y  =>  (equal x y) = t
x != y  =>  (equal x y) = nil
```

We are also reasoning about builtin functions, such as cons, car, and cdr. We have axioms for that.

```
(equal (car (cons x y)) x) != nil
```

More ACL2 axioms:

```
(cdr (cons x y)) = y
(consp (cons x y)) = t
```

If is also builtin. Recall that the axioms for if are:

```
- x = nil =>  (if x y z) = z
- x != nil => (if x y z) = y
```

```
What about instantiation? It is a rule of inference:

Derive phi|sigma from phi. That is, if phi is a theorem, so is
phi|sigma.

Example: From (equal (car (cons x y)) x)
I can derive (equal (car (cons (foo x) (bar z))) (foo x))

More carefully, a substitution is just a list of the form:
( (var1 term1) ... (varn termn) ), where the vars are "target
variables" and the terms are their images.  The application of
this substitution to a formula uniformly replaces every free
occurrence of a target variable by its image.

What does it mean to prove the theorem

4. (len (cons x z)) = (len (cons y z))

That no matter what you replace x, y, and z with from the ACL2
universe, if you evaluate the lhs and rhs, you get the same
answer.

By the way, we are now considering equational reasoning in
ACL2. Sectiond 6.1 and 6.2 of the ACL2 book.

Let’s try to prove another theorem

8. (app (cons x y) z)
 = (cons x (app y z))


  (app (cons x y) z)
=   {Definition of app, instantiation}
  (if (endp (cons x y))
      z
    (cons (car (cons x y)) (app (cdr (cons x y)) z)))
=   {Definition of endp, axioms consp}
  (if nil
      z
    (cons (car (cons x y)) (app (cdr (cons x y)) z)))
=   {Axioms for if, car, cdr}
  (cons x (app y z))

Unfortunately, the above "proof" has a problem. Unlike len, which
is defined for the whole ACL2 universe, app is only defined for
true-lists.

Recall the definitions:

(defunc listp (x)
  :input-contract t
  :output-contract (booleanp (listp x))
  (or (equal x nil)
      (consp x)))

(defunc endp (a)
  :input-contract (listp a)
  :output-contract (booleanp (endp a))
  (not (consp a)))

(defunc true-listp (a)
  :input-contract t
  :output-contract (booleanp (true-listp a))
```

```
  (if (consp a)
      (true-listp (rest a))
    (equal a nil)))

(defunc app (a b)
  :input-contract (and (true-listp a) (true-listp b))
  :output-contract (and (true-listp (app a b))
                        (equal (len (app a b))
                               (+ (len a) (len b))))
  (if (endp a)
      b
    (cons (first a) (app (rest a) b))))
```

The definition of functions such as app give rise to "definitional axioms", e.g.:

```
(true-listp a) /\ (true-listp b)
=>
(app a b)
=
(if (endp a)
    b
  (cons (car a) (app (cdr a) b)))
```

In general, every time we successfully define a function, we get an axiom of the form

```
ic => [(f x1 ... xn) = body]
```

So, I can’t expand the definition of app in the proof of conjecture 8, unless I know:

```
(true-listp (cons x y)) /\ (true-listp z)
```

which is equivalent to:

```
(true-listp y) /\ (true-listp z)
```

So, what we really proved was:

```
9.
   (true-listp y) /\ (true-listp z)
=>
   (app (cons x y) z)
 = (cons x (app y z))
```

When we write out proofs, I will not require you to explicitly mention input contracts when using a function definition because the understanding is that every time we use a definitional axiom to expand a function, we have to check that we satisfy the input contract, so we don’t need to remind the reader of our proof that we did something we all understand always needs to be done.

It is often the case that when we think about conjectures that we expect to be valid, we often forget to carefully specify the hypotheses under which they are valid. These hypotheses depend on the input contracts of the functions mentioned in the conjectures, so get into the habit of looking at conjectures and making sure that they have the hypotheses needed in order for the conjectures to be true. This is similar to what you do when you write functions: you check the body contracts of the functions you define. In the case of function definitions, as we have seen,

```
chances are that if the function definition is wrong, there is
also a contract violation. Similarly, if a conjecture is not
true, chances are that there is a contract violation.

Let's look at another example:

Conjecture:

10.
(endp x) =>   (app (app x y) z)
            = (app x (app y z))

Can I prove this? Check the contracts?

I need to take the contracts into account. That gives rise to
(below I write (tlp x) as a shorthand for (true-listp x)):

11.
(endp x) /\ (tlp x) /\ (tlp y) /\ (tlp z)
 =>   (app (app x y) z) = (app x (app y z))

By the way, notice all of the hypotheses. Notice the Boolean
structure. This is why we studied Boolean logic first! Also,
almost everything we will prove will include an implication.

Notice that in ACL2, we would technically write:

  (implies (and (endp x)
                (and (true-listp x)
                     (and (true-listp y) (true-listp z))))
           (equal (app (app x y) z)
                  (app x (app y z))))


The first thing to do is to when proving theorems is to take the
Boolean structure into account and to try and write the
conjecture into the form: hyp1 /\ hyp2 /\ ... /\ hypn => conc
where we have as many hyps as possible. We will call the set of
top-level hypotheses (hyp1, ..., hypn) our "context".

Our context for 11, is:

C1. (endp x)
C2. (tlp x)
C3. (tlp y)
C4. (tlp z)

We then look at our context and see what obvious things our
context implies. The obvious thing here is that C1 and C2 imply
that x must be nil, so we add to our context the following:

C5. x = nil { 1, 2}

Notice that any new facts we add must come with a justification.
We will use the convention that all elements of our context will
be given a label of the form Ci, where i is a positive integer.

The next thing we do is to start with the LHS of the conclusion
and to try and reduce it to the RHS, using our proof format. If
we need to refer to the context in one of proof step
justifications, say context #5, we write C5.

   (app (app x y) z)
 =   {Def of app, C5, Def of endp, if axioms}
```

```
   (app y z)
 =   {Def of app, C5, Def of endp, if axioms}
   (app x (app y z))
```

Notice that we took bigger steps than before. Before we might
have written:

```
   (app (app x y) z)
 =   {Def of app}
   (app (if (endp x) y (cons (car x) (app (cdr x) y))) z)
 =   {C5}
   (app (if (endp nil) y (cons (car nil) (app (cdr nil) y))) z)
 =   {Def of endp}
   (app (if t y (cons (car nil) (app (cdr nil) y))) z)
 =   {If axioms}
   (app y z)
 ...
```

So, the above four steps were compressed into 1 step. Why?
Because many of the steps we take involve expanding the
definition of a function. Function definitions tend to have a
top-level if or cond and as a general rule we will not expand the
definition of such a function unless we can determine which case
of the top-level if-structure will be true. If we just blindly
expand function definitions, we’ll wind up with a sequence of
increasingly complicated terms that don’t get us anywhere. So, if
we know which case of the top-level if will be true, then we go
to the trouble of writing out the whole body of the function? Why
not just write out that one case? Well, that’s why we allow
ourselves to expand definitions as in the first proof of
Conjecture 11.

One other comment about the first proof of conjecture 8. Students
often have no difficulty with the first step, but have difficulty
with the second step. The second step requires one to see that
the simple term:

```
   (app y z)
```

can be transformed into the RHS

```
   (app x (app y z))
```

This may seem like a strange thing to do because students are used
to thinking about computation as unfolding over time. So, if x is
nil then of course the following holds.

```
   (app (app x y) z)
 =   {Def app, ...}
   (app y z)
```

Because when we compute (app x y) we get y.

What students initially have difficulty with is seeing that you
can reverse the flow of time and everything still works. For
example the following is true,

```
   (app y z)
 =   {Def app, ...}
   (app x (app y z))
```

Because starting with (app y z) we can run time in reverse to get
(app x (app y z)) (recall x is nil). In fact, this is "obvious"
from the equality (=) axioms that tell us that equality is an

equivalence relation (reflexive, symmetric, and transitive). The symmetry axiom tells us that view computation as moving forward in time or backward. It just doesn’t make a difference.

As an aside, it turns out that in physics, that we can’t reverse time and so this symmetry we have with computation is not a symmetry we have in out universe. One reason why we can’t reverse time in physics is that the second law of thermodynamics precludes it. The second law of thermodynamics implies that entropy increases over time. There is an even more fundamental reason why time is not reversible. This second reason has to do with the fundamental laws of physics at the quantum level, whereas the second law of thermodynamics is thought to be a result of the initial conditions of our universe. The second reason is that in our current understanding of the universe, there are very small violations of time reversibility exhibited by subatomic particles. The extent of the violations is not fully understood and probably has something to do with the imbalance of matter and antimatter in the visible universe. There is almost no antimatter in the visible universe and one of the big open problems in physics is trying to understand why that is the case.

Recall that since 11 is a theorem, whatever we replace the free variables with, it will evaluate to t. A convenient way of checking 11 using ACL2 is to use let, as follows:

```
(let ((x nil)
      (y nil)
      (z nil))
  (implies (and (endp x)
                (and (true-listp x)
                     (and (true-listp y) (true-listp z))))
           (equal (app (app x y) z)
                  (app x (app y z)))))
```

=======================================
Begin: An aside on let.
=======================================

Let:

A let expression:

```
(let ((v1 x1)
      ...
      (vn xn))
  body)
```

binds its local variables, the vi, in parallel, to the values of the xi, and evaluates it’s body.

Example:

```
(let ((x ’(1 2 3))
      (y ’(3 4)))
  (append (append x y) (append x y)))
```

This saves us having to type ’(1 2 3) and ’(3 4) multiple times.

Maybe we can avoid having to type (append x y) multiple times. What about?

```
(let ((x ’(1 2 3))
      (y ’(3 4))
```

```
      (z (append x y)))
  (append (append x y) z))

It doesn’t work.  What does ’(1 2 3) evaluate to at the top
level? ’(3 4)? (append x y)?.  Let binds in parallel, so x and y
in z binding are not yet bound.

This brings us to Let*:

(let* ((v1 x1)
       ...
       (vn xn))
  body)

binds its local variables, the vi, sequentially, to the values of
the xi, and evaluates it’s body.

(let* ((x ’(1 2 3))
       (y ’(3 4))
       (z (append x y)))
  (append (append x y) z))

Let’s simplify further:

(let* ((x ’(1 2 3))
       (y ’(3 4))
       (z (append x y)))
  (append z z))

So, let, let* give us abbreviation power.

=======================================
End: An aside on let.
=======================================

So, getting back to our theorem: no matter what we bind x, y, and
z with, we will evaluate to t.

Let’s continue with more examples.

12.
(consp x)
=>
  [
    [(tlp (cdr x)) /\ (tlp y) /\ (tlp z)
 =>  (app (app (cdr x) y) z) = (app (cdr x) (app y z))]

    [(tlp x) /\ (tlp y) /\ (tlp z)
 =>  (app (app x y) z) = (app x (app y z)) ]]

The above conjecture has the form

A => [B => C]

where

A is (consp x)
B is     [(tlp (cdr x)) /\ (tlp y) /\ (tlp z)
      =>  (app (app (cdr x) y) z) = (app (cdr x) (app y z))]

C is     [(tlp x) /\ (tlp y) /\ (tlp z)
      =>  (app (app x y) z) = (app x (app y z)) ]

What we are doing here is identifying some of the propositional
```

```
structure of conjecture 12. Here's why. It turns out that

(*) A => [B => C]  =   [A /\ B] => C

This propositional equality is one we will use over and over.
We will use (*) to rewrite conjecture 12 so that the context is as
big as possible. After applying (*) to 12, we get:

13.
[ (consp x) /\
  [(tlp (cdr x)) /\ (tlp y) /\ (tlp z)
   =>  (app (app (cdr x) y) z) = (app (cdr x) (app y z))] ]
=>
    [(tlp x) /\ (tlp y) /\ (tlp z)
 =>  (app (app x y) z) = (app x (app y z)) ]

Applying (*) again (and rearranging conjuncts) gives us:

14.
[ (consp x) /\ (tlp x) /\ (tlp y) /\ (tlp z) /\
  [(tlp (cdr x)) /\ (tlp y) /\ (tlp z)
   =>  (app (app (cdr x) y) z) = (app (cdr x) (app y z))]]
 =>  (app (app x y) z) = (app x (app y z)) ]

Now, we can extract the context. Doing so gives us:

C1. (consp x)
C2. (tlp x)
C3. (tlp y)
C4. (tlp z)
C5. [(tlp (cdr x)) /\ (tlp y) /\ (tlp z)]
   =>  [(app (app (cdr x) y) z) = (app (cdr x) (app y z))]

Notice that we cannot use (*) on C5 to add the hypotheses of C5 to
our context. Why?

We will be confronted with implications in our context (like C5)
over and over. Usually what we will need is the consequent of the
implication, but we can only use the consequent if we can also
establish the antecedent, so we will try to do that. Here's how:

C6. (tlp (cdr x)) {1, 2, Def tlp}
C7. (app (app (cdr x) y) z) = (app (cdr x) (app y z)) {6, 3, 4, 5, MP}

So, notice what we did. First we added C6 to our context. How did
we get C6? Well, we know (tlp x) (C2) and (consp x) (C1) so if we
use the definitional axiom of consp, we get C6: (tlp (cdr x)).

Now, we have extended our context to include the antecedent of
C5, so by propositional logic (Modus Ponens, abbreviated MP), we
get that the conclusion also holds, i.e., C7.

Recall that Modus Ponens tells us that if the following two
formulas hold

A => B
A

Then so does the formula

B

We are now ready to prove the theorem. We start with the LHS of
the equality in the conclusion of 14
```

```
  (app (app x y) z)
=  { Def app, C1, C2, C3 }
  (app (cons (car x) (app (cdr x) y)) z)
=  { Theorem 8 }
  (cons (car x) (app (app (cdr x) y) z))
=  { C7 }
  (cons (car x) (app (cdr x) (app y z)))
=  { Def app, C1, C2, C3, C4 }
  (app x (app y z))
```

The difference between theorems and context

It is very important to understand the difference between a formula that is a theorem and one that appears in a context. A formula that appears in a context cannot be instantiated. It can only be used as is, in the proof attempt for the conjecture from which it was extracted. This is a major difference. Our contexts will never include theorems we already know. Theorems we already know are independent of any conjecture we are trying to prove and therefore do not belong in a context. A context is always formula specific.

Here is an example that shows why instantiation of context formulas leads to unsoundness. Here is a "proof" of

15. x=1 => 0=1

Context:
C1. x=1

```
Proof
   0
= { Instantiate C1 with ((x 0)) }
   1
```

So, now we have a "proof" of 15, but using 15, we can get:

16. false

How? (Instantiate 15 with ((x 1)), use Propositional logic, Arithmetic)

Now we have a proof for any conjecture we want, e.g.,

17. phi (any conjecture)

How?

Well, false implies anything, so this is a theorem

false => phi

Now, phi follows using 16 and Modus ponens.

The point is that a context is *completely* different from a theorem. The context of 15 does not tell us that for all x, x=1. It just tells us that x=1 in the context of conjecture 15. Contexts are just a mechanism for extracting propositional structure from a conjecture, which allows us to focus on the important part of a proof and to minimize the writing we have to do.

```
----------------------------------------------------------
How to prove theorems, part 1
----------------------------------------------------------

When presented with a conjecture, make sure that you check
contracts, as shown above.

If the contracts checking succeeds, make sure you understand what
the conjecture is saying.

Once you do, see if you can find a counterexample.

If you can’t think about how to prove that the conjecture is a
theorem.

One often iterates over the last two steps.

During the proof process, you have available to you all the
theorems we have proved so far. This includes all of the axioms
(car-cdr axioms, if axioms, ...), all the definitional axioms
(def of app, len, ...), all the definitional contracts (contacts
of app, len, ...). These theorems can be used at any time in any
proof and can be instantiated using any substitution. They are a
great weapon that will help you prove theorems, so make sure you
understand the set of already proven theorems.

There are also local facts extracted from the conjecture under
consideration. Recall that the first step is to try and rewrite
the conjecture into the form:

 [C1 /\ C2 /\ ... /\ Cn] => RHS

where we try to make RHS as simple as possible.  C1, ..., Cn are
going to be the first n components of our context. Formulas in
the context are specific to the conjecture under
consideration. They are completely different from theorems (as
per the above discussion). A good amount of manipulation of the
conjecture may be required to extract the maximal context, but it
is well worth it.

The next step is to see what other facts the C1, ..., Cn
imply. For example, if the current context is:

C1. (endp x)
C2. (tlp x)

then we would add

C3. x=nil { C1, C2 }

This will happen a lot. Another case that will happen a lot is:

C1. (consp x)
C2. (tlp x)
C3. (tlp (cdr x)) => phi

then we would add

C4. (tlp (cdr x)) { C1, C2, Def tlp }
C5. phi           { C4, C3, MP }

where MP is modus ponens.

***************************
```

Proving theorems, general comments
****

We can also have "word problems". For example, consider:

Conjecture: x <= xy if y>=1

Discussion: What does the above conjecture mean, anyway?

It means that for any values of x, and y, ...

Really? Any values? What if x and y are functions or strings or ...?
Usually the domain is implicit, i.e., "clear from context".

We will be using ACL2, and we can't appeal to "context". This is a good thing!

Notice also that we can use ACL2, a programming language, to make mathematical statements. Duh! Programming languages are mathematical structures and you reason about programs the way you reason about the natural numbers, the reals, sets, etc.: you prove theorems.

In ACL2, we have to be precise about the conditions under which we expect the conjecture to hold. The conjecture can be formalized in ACL2 as follows:

```
(...
 (implies (and (rationalp x)
               (rationalp y)
               (>= y 1))
          (<= x (* x y))))
```

In standard mathematical notation it is:

<forall x, y: x,y \in Q and y>=1 : x <= xy>

Is the above conjecture true?

Well, when given a conjecture, we can try one of two things:

1. Try to falsify it.
2. Try to prove it is correct.

How do we falsify a conjecture?

Simple exhibit a counterexample.

Remember that in the design recipe, we construct examples and tests. You should do the same thing with conjectures. That is, we can test that the conjecture is true on examples. Here are some:

x=0,y=0
x=12,y=1/3
x=9,y=3/2

Any others?

How do we test this in ACL2s? Put the conjecture in the body of a let.

```
(let ((x 0)
      (y 0))
  (implies (and (rationalp x)
```

```
                (rationalp y)
                (>= y 1))
           (<= x (* x y))))
```

We are using a programming language, so we can do better. We can write a program to test the conjecture on a large number of cases. How many cases are there? We can use a random number generator to "randomly" sample from the domain.

We'll see how to do that in ACL2s.

If all of the tests pass, then we can try to prove that the conjecture is a theorem.

What would a "proof" of the above conjecture look like?

Most proofs are informal and it takes a long time for students to understand what constitutes an informal proof. This happens by osmosis over time.

In our case, we have a simple rule: it's a proof if ACL2 says it is.

```
(thm
 (implies (and (rationalp x)
               (rationalp y)
               (>= y 1))
          (<= x (* x y))))
```

Of course, this isn't a theorem.

Let's consider another example:

If I have time:

Conjecture: x(y+z) = xy + xz

How do we write this in ACL2s?

```
(thm (implies (and (rationalp x)
                   (rationalp y)
                   (rationalp z))
              (equal (* x (+ y z))
                     (+ (* x y) (* x z)))))
```

Is the above conjecture true?

Well, we can try to falsify it.

```
(let ((x 0)
      (y 0)
      (z 0))
  (= (* x (+ y z))
     (+ (* x y) (* x z))))
```

We can try many examples. We can automatically generate random examples.

When do we give up falsifying this?

Can we just try all the possibilities? If we had infinite time. Do we? Maybe (ask a physicist), but, as a practical matter, we currently don't.

```
Maybe we should consider a proof. Can we prove the above?

One answer might be: "of course, multiplication distributes over
addition".

In ACL2, the conjecture turns out to be true

(thm (implies (and (rationalp x)
                   (rationalp y)
                   (rationalp z))
              (equal (* x (+ y z))
                     (+ (* x y) (* x z)))))

This is pretty amazing because a proof gives us a finite way of
running an infinite number of examples.  That's the power of
logic and mathematics.

When ACL2 proves this theorem, is it thinking?

The question of whether Machines Can Think ... is about as
relevant as the question of whether Submarines Can Swim.
EWD898, 1984

See the EWD archives at the University of Texas at Austin.

----------------------------------------------------------
Here is another example
----------------------------------------------------------

(defunc app (a b)
  :input-contract (and (true-listp a) (true-listp b))
  :output-contract (and (true-listp (app a b))
                        (equal (len (app a b))
                               (+ (len a) (len b))))
  (if (endp a)
      b
    (cons (first a) (app (rest a) b))))

(defunc rev (x)
  :input-contract (true-listp x)
  :output-contract (and (true-listp (rev x))
                        (equal (len (rev x))
                               (len x)))
  (if (endp x)
      nil
    (app (rev (cdr x)) (list (car x)))))


(defunc in (a X)
  :input-contract (true-listp x)
  :output-contract (booleanp (in a X))
  (if (endp x)
      nil
    (or (equal a (car X))
        (in a (cdr X)))))


(defunc del (a X)
  :input-contract (true-listp x)
  :output-contract (true-listp (del a X))
  (cond ((endp x) nil)
        ((equal a (car x)) (cdr x))
        (t (cons (car x) (del a (cdr x))))))
```

```
Conjecture.
(tlp x)
=>
(in a x) => (not (in a (del a x)))

Using induction (something we will describe later), the above
conjecture leads to the following proof obligation:

18.
(and (implies (endp x)
              (implies (true-listp x)
                       (implies (in a x)
                                (not (in a (del a x))))))
     (implies (and (consp x)
                   (equal a (car x)))
              (implies (true-listp x)
                       (implies (in a x)
                                (not (in a (del a x))))))
     (implies (and (consp x)
                   (not (equal a (car x)))
                   (implies (true-listp (cdr x))
                            (implies (in a (cdr x))
                                     (not (in a (del a (cdr x)))))))
              (implies (true-listp x)
                       (implies (in a x)
                                (not (in a (del a x)))))))

Is this true? If so, give a proof. Is it false? Is so, exhibit a
counterexample.

Try this before reading further.

18 is false, e.g., consider

(let ((x '(1 1))
     (a 1))
     18)

where 18 is the conjecture 18 in the above let.

What about the following conjecture?

(tlp x)
=>
(in a x) => (in a (app x y))

Which by induction leads to the following proof obligation:

19.
(and (implies (endp x)
              (implies (true-listp x)
                       (implies (in a x)
                                (in a (app x y)))))
     (implies (and (consp x)
                   (equal a (car x)))
              (implies (true-listp x)
                       (implies (in a x)
                                (in a (app x y)))))
     (implies (and (consp x)
                   (not (equal a (car x)))
                   (implies (true-listp (cdr x))
                            (implies (in a (cdr x))
```

```
                                     (in a (app (cdr x) y)))))
              (implies (true-listp x)
                       (implies (in a x)
                                (in a (app x y))))))
```

Is this true? If so, give a proof. Is it false? Is so, exhibit a
counterexample.

Try this before reading further.

This is true and you should be able to prove it by breaking 19
into three parts and proving each in turn.

In these cases we have seen so far, it was easy to decide if a
conjecture was true or false, and with a good amount of testing,
we would have identified the false conjectures.

Is this always the case?

No.

Anyone heard of Fermat's last theorem?

For all positive integers x, y, z, and n, where n>2,
x^n + y^n != 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 Fermat's 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 aren't theorems, we should be able to find
counterexamples quickly, right?

That is not true either.

Arithmetic

We can also reason about arithmetic functions. For example:

```
Prove that \Sigma_{i=0}^{n} i = n(n+1)/2

That is, summing up 0, 1, ..., n gives n(n+1)/2

We can prove this using mathematical induction.

Here is how we do it in ACL2s. First, we have to define \Sigma.

(defunc sum (n)
  :input-contract (natp n)
  :output-contract (natp (sum n))
  (if (equal n 0)
      0
    (+ n (sum (- n 1)))))

We can prove that (sum n) = n(n+1)/2, which is formalized as:

(implies (natp n)
         (equal (sum n)
                (/ (* n (+ n 1)) 2)))

by induction. How?

Base case: (equal n 0)

induction step (and (natp n) (not (equal n 0)),
i.e., n is a natural number >0 and above holds for n-1.

20.     (equal n 0) /\ (natp n)
    =>  (sum n) = (/ (* n n+1) 2)

21.     n>0 /\ (natp n) /\ (sum n-1) = (/ (* n-1 n) 2)
    =>  (sum n) = (/ (* n n+1) 2)



So, here is the proof.

20.
Context.

C1. (natp n)
C2. (equal n 0)

Proof.

  (sum n)
=  { C2, Def sum }
  0
=  { Arithmetic, C2 }
  (/ (* n n+1) 2)

21.
Context.

C1. (natp n)
C2. n != 0
C3. (natp n-1) => (sum n-1) = (/ (* n-1 n) 2)
C4. (natp n-1) { C1, C2 }
C5. (sum n-1) = (/ (* n-1 n) 2) { C3, C4, MP }

Proof.
```

```
  (sum n)
=  { C2, Def sum }
  n + (sum (- n 1))
=  { C5 }
  n + (/ (* n-1 n) 2)
=   { Arithmetic }
  (2n + n(n-1))/2
=   { Arithmetic }
  (2n + n^2 -n)/2
=   { Arithmetic }
  (n^2 +n)/2
=   { Arithmetic }
  n(n+1)/2
```

****
The Definitional Principle
****

We've already seen that when you define a function, say

```
(defunc f(x)
  :input-contract ic
  :output-contract oc
  body)
```

then ACL2 adds the definitional axiom

ic => (f x) = body

and the contract

ic => oc

Today, we'll more carefully examine what happens when you define functions.

First, let's see why we have to examine anything at all.

In fundies 1, you were allowed to write functions such as the following:

```
(defunc f(x)
  :input-contract (natp x)
  :output-contract (natp (f x))
  (+ 1 (f x)))
```

This is a nonterminating recursion.

There has been no reason for you to write nonterminating functions in 211 or in this class, but you had the ability to do it.

Presumably, a reasonable language would have prevented you from doing so.

As a second best option is to use a design recipe that does not lead to non-termination.

In fact, ACL2s does not allow you to write such functions.

Suppose we add the axioms

A1. (natp x) => (f x) = (+ 1 (f x))
A2. (natp x) => (natp (f x))

```
Then what?

We get a contradiction, since in ACL2s, we can prove

1. natp x => x != x+1

(thm (implies (natp x) (not (equal x (1+ x)))))

That is, in ACL2s, we can prove nil.  How?

Notice
2. (natp x) => (f x) != (+ 1 (f x))

Proof:

   (natp 1)
=>  { A2 }
   (natp (f 1))
=>  { 1 under ((x (f 1))), A1 }
   [(f 1) != 1+(f 1)] /\ [(f 1) = 1+(f 1)]
=>  { Propositional Logic }
   nil

So, we proved t => nil, which is nil (because (natp 1) = t).

3. nil

What's so bad about that?

Consider any formula f (say 3=4).

Here's a proof of it:

  f
=   { 3 (nil is a theorem), propositional logic }
  nil => f
=   { propositional logic }
  t

So, we proved f = t, that is f is valid, a theorem! We also only
used propositional logic.

So, some nonterminating recursive equations introduce
unsoundness. Therefore, ACL2s does not allow you to define
nonterminating functions.

Question: does every non-terminating recursive equation introduce
unsoundness?

No. Consider (f x) = (f x).  You can prove such things in ACL2,
since every function satisfies this (it's the Leibniz axiom);
you just can't define a function this way.

But, can some terminating recursive equations introduce
unsoundness?

Well, yes.

Consider:

(defun f (x) y)

   4
```

```
=    { Instantiation f axiom ( (x 1) (y 4) ) }
   (f 1)
=    { Instantiation f axiom ( (x 1) (y 3) ) }
   3
```

But this happened because we allowed a "global" variable. It will turn out that we can rule out bad terminating equations with some simple checks.

So, modulo some checks we are going to get to soon, terminating recursive equations do not introduce unsoundness, because we can prove that if a recursive equation can be shown to terminate then there exists a function satisfying the equation.

The above discussion should convince you that we need a mechanism for making sure that when users add axioms to ACL2 by defining functions, then the logic stays sound.

That’s what the *definitional principle* does.

Definitional Principle:

The definition

```
(defunc f (x1 ... xn)
  :input-contract ic
  :output-contract oc
  body)
```

is *admissible* provided:

1. f is a new function symbol, i.e., there are no other axioms about it; (note that this happens in the context of a history)

(BTW, why do we need this? Well, what if we already defined app? Then we would have two definitions. What about redefining? We may already have theorems proven about app. We’ve have to throw them out. ACL2s allows you to undo, but not redefine.)

2. The xi are distinct variable symbols;

(Why? If the variables are the same, say (defun f (x x) body) then what is (f 1 2)? ACL2 is total.)

3. body is a term, possibly using f recursively as a function symbol, mentioning no variables freely other than the xi;

(Why? Well, we already saw that (defun f(x) y) can lead to unsoundness. body is a term means that it is a legal expression in the current history)

4. the function is terminating;
(Why? We saw that nontermination can lead to unsoundness.)

There are also two other conditions that I state separately.

5. ic => oc is a theorem;

6. the body contracts hold under the assumption that ic holds.

If admissible, the logical effect of the definition is to add two new axioms:

Definitional Axiom for f: ic => [(f x1 ... xn) = body].

Contract Axiom for f: ic => oc.

But, how do we prove termination?

A very simple first idea is to use what are called measure functions. There are functions from the parameters of the function at hand into the natural numbers, so that we can prove that on every recursive call the function terminates. Let’s try this with app. What is a measure fuction for app?

How about the length of x? So, the measure function is (len x).

In more detail we have a measure function m that is defined over the parameters of app, which has the same input contract as app, which has an output contract that it always returns a natural number, and for which we can prove that on every recursive call, m applied to the arguments to that recursive call decreases, under the conditions that led to the recursive call.

Here then is m:

```
(defunc m (x y)
  :input-contract (and (true-listp x) (true-listp y))
  :output-contract (natp (m x y))
  (len x))
```

This is a non-recursive function, so it is easy to admit. Notice that we do not use the second parameter (but that is OK; it just tells us that the second parameter is not needed for the termination argument).

Next, we have to prove that m decreases on all recursive calls of app, under the conditions that led to the recursive call. Since there is one recursive call, we have to show:

```
(implies (and (true-listp x)
              (true-listp y)
              (not (endp x)))
         (< (m (cdr x) y) (m x y)))
```

which is equivalent to:

```
(implies (and (true-listp x)
              (true-listp y)
              (not (endp x)))
         (< (len (cdr x)) (len x)))
```

which is a true statement.

More examples:

```
(defun rev (x)
  (if (endp x)
      nil
    (app (rev (cdr x)) (list (car x)))))
```

Is this admissable? It depends if we defined app already. Suppose app is above. What is a measure function?

len.

What about:

```
(defun drop-last (x)
```

```
  (if (= (len x) 1)
      nil
    (cons (first x) (drop-last (rest x)))))
```

No. It is non-terminating, e.g., when x is nil. How can we define it (using the design recipe)?

```
(defun drop-last (x)
  (cond ((endp x) nil)
        (t (cond ((endp (cdr x)) nil)
                 (t (cons (first x) (drop-last (rest x))))))))
```

What is a measure function?

What about the following:

```
(defun prefixes (l)
  (cond ((endp l) '( () ))
        (t (cons l (prefixes (drop-last l))))))
```

What is a measure function?

Yes. It satisfies the conditions of the definitional principle; in particular, it terminates because we are removing the last element from l.

Caveat: Checking for termination is undecidable; Turing showed that. So, you can define functions that terminate, but that ACL2s can’t prove terminating automatically. How would you write a program that check if other programs terminate? However, we expect that for the programs you write, ACL2s will be able to prove termination automatically. If not, send email to us and we can help you.

By the way, remember big-Oh notation? It is connected to termination. How?

Well if the running time for a function is O(n^2), say, then that means that:

1. the function terminates
2. there is a constant c s.t. the function terminates within cn^2
   steps, where n is the "size" of the input

so, big-Oh analysis is just a refinement of termination, where we are not interested in only whether a function terminates, but also we want to know how long it will take.

Terminating functions give rise to induction schemes.

Consider the following function:

```
(defunc nind (n)
  :input-contract (natp n)
  :output-contract t
  (if (equal n 0)
      0
    (nind (- n 1))))
```

Suppose you want to prove phi using the induction scheme you get from (nind n). What are your proof obligations?

1. (not (natp n)) => phi

```
2. (natp n) /\ (equal n 0) => phi
3. (natp n) /\ (not (equal n 0))  /\ phi| n<-n-1 => phi

More generally we have the following.

1. Given a function definition of the form:

(defunc foo (x1 ... xn)
  :input-contract ic
  :output-contract oc
  (cond (t1 c1)
        (t2 c2)
        ...
        (tm cm)
        (t cm+1)))

where none of the ci’s have any ifs in them.

Notice that any function definition can be written in this form.

If ci contains a call to foo, we say it is a RECURSIVE case;
otherwise it is a BASE case. If ci is a RECURSIVE case, the it
includes at least one call to foo. Say there are Ri calls to foo
and they are foo_i^j where 1<=j<=Ri. Let s_i^j be the
substitution such that (foo x1 ... xn)|s_i^j = foo_i^j.

Let tm+1 be t.

Let Casei be ti /\ ~tj for all j<i,

eg, Case0 is t1
    Case1 is t2 /\ ~t1
    Case2 is t3 /\ ~t1 /\ ~t2
    Casem+1 is t /\  ~t1 /\ ~t2 /\ ... /\ ~tm

2. foo gives rise to the following induction scheme:

To prove phi, you can instead prove

1. ~ic  =>  phi
2. [ic /\ Casei]  =>  phi (for all ci that are BASE cases)
3. [ic /\ Casei /\_{1<=j<=Ri} phi|s_i^j]  =>  phi (for all ci that are RECURSIVE cases)

We can play this game in reverse. For example, if I were to ask
you to write a function that gives rise to the following
induction scheme:

1. (not (natp n)) => phi
2. (natp n) /\ (equal n 0) => phi
3. (natp n) /\ (not (equal n 0))  /\ phi| n<-n-1 => phi

You would give me nind.
```