CS2800 Fall 2010 - Lecture 27

29 November 2010

Proofs in ACL2 (Example 1)

I will work through an example to demonstrate how to use ACL2 to prove theorems about programs. We will use this implementation of insertion sort:

```
(defun insert (e l)
  (if (endp l)
    (cons e l)
    (if (<= e (car l))
      (cons e l)
      (cons (car l) (insert e (cdr l))))))

(defun isort (l)
  (if (endp l)
    nil
    (insert (car l) (isort (cdr l)))))
```

The function `insert` inserts an element in order into an ordered list:

```
ACL2 >VALUE (insert 2 '(0 1 5 7))
(0 1 2 5 7)
ACL2 >VALUE (insert 2 '())
(2)
ACL2 >VALUE (insert 2 '(3 4))
(2 3 4)
```

And `isort` uses `insert` repeatedly to build a sorted list:

```
ACL2 >VALUE (isort '(9 8 3 7 2))
(2 3 7 8 9)
ACL2 >VALUE (isort '())
NIL
ACL2 >VALUE (isort '(4 5 4 3 3))
(3 3 4 4 5)
```

We can open up ACL2s and create a new Lisp file in Intermediate Mode, add the above definitions, and proceed with attempting the proofs below.

Let us first ask ACL2 to prove that the length of sorting an object is the length of that object:

```
(defthm len--isort
  (equal (len (isort x))
         (len x)))
```

`defthm` asks ACL2 to attempt to prove a formula and, if successful, remember it for use in future proofs. We give it a name, in this case `len--isort`, so that when ACL2 uses it in the future, it can give us the name of what it used.

The proof fails and here is some of the output, with some explanations:

```
<< Starting proof tree logging >>
```

This indicates that ACL2 is sending an outline of the proof attempt to the ACL2s Proof Tree view.

```
Name the formula above *1.
```

This is what ACL2 says when it runs out of things to do without using induction. It will try induction once per proof, so that is what it does next:

```
Perhaps we can prove *1 by induction.  Two induction schemes are suggested
by this conjecture.  These merge into one derived induction scheme.
```

What are the two functions involed in the conjecture? LEN and ISORT. If you look at the induction schemes suggested by those functions, they are the same.

```
We will induct according to a scheme suggested by (LEN X).  This suggestion
was produced using the :induction rules ISORT and LEN.  If we let (:P X)
denote *1 above then the induction scheme we'll use is
(AND (IMPLIES (NOT (CONSP X)) (:P X))
     (IMPLIES (AND (CONSP X) (:P (CDR X)))
              (:P X))).
This induction is justified by the same argument used to admit LEN.
When applied to the goal at hand the above induction scheme produces
two nontautological subgoals.
```

The two subgoals are the base case and the induction step, listed above with an AND.

```
Subgoal *1/2
(IMPLIES (NOT (CONSP X))
         (EQUAL (LEN (ISORT X)) (LEN X))).

But simplification reduces this to T, using the :definitions ISORT
and LEN and the :executable-counterparts of EQUAL and LEN.
```

ACL2 has completed the entire base case proof in what it considers one step. It tells us it used the definitions of ISORT and LEN and executed the functions EQUAL and LEN to determine this is true.

In fact, (ISORT X) is equal to NIL given the assumption and the definition of ISORT, (LEN NIL) can be executed to get 0, (LEN X) is equal to 0 given the assumption and the definition of LEN, and (EQUAL 0 0) can be executed to get T.

On to the next Subgoal, the inductive step:

```
Subgoal *1/1
(IMPLIES (AND (CONSP X)
              (EQUAL (LEN (ISORT (CDR X)))
                     (LEN (CDR X))))
         (EQUAL (LEN (ISORT X)) (LEN X))).

This simplifies, using the :definitions ISORT and LEN, to

Subgoal *1/1'
```

```
(IMPLIES (AND (CONSP X)
              (EQUAL (LEN (ISORT (CDR X)))
                     (LEN (CDR X))))
         (EQUAL (LEN (INSERT (CAR X) (ISORT (CDR X))))
                (+ 1 (LEN (CDR X))))).
```

Those are the two things we would do if we were proving this by hand.

```
The destructor terms (CAR X) and (CDR X) can be eliminated by using
CAR-CDR-ELIM to replace X by (CONS X1 X2), (CAR X) by X1 and (CDR X)
by X2.  This produces the following goal.

Subgoal *1/1''
(IMPLIES (AND (CONSP (CONS X1 X2))
              (EQUAL (LEN (ISORT X2)) (LEN X2)))
         (EQUAL (LEN (INSERT X1 (ISORT X2)))
                (+ 1 (LEN X2)))).

This simplifies, using primitive type reasoning, to

Subgoal *1/1'''
(IMPLIES (EQUAL (LEN (ISORT X2)) (LEN X2))
         (EQUAL (LEN (INSERT X1 (ISORT X2)))
                (+ 1 (LEN X2)))).
^^^ Checkpoint Subgoal *1/1''' ^^^
```

Now ACL2 used the fact that X is a cons to make the formula a little simpler by using separate variables for what was the CAR of X and the CDR of X.

"Checkpoint" means that it has exhausted its safest proof techniques and might start doing things that be in a wrong direction. In fact, at the beginning of the proof, it said "Checkpoint" because whenever it tries induction, it may not choose the right induction and may go off in a wrong direction.

```
We now use the hypothesis by substituting (LEN (ISORT X2)) for (LEN X2)
and throwing away the hypothesis.  This produces

Subgoal *1/1'4'
(EQUAL (LEN (INSERT X1 (ISORT X2)))
       (+ 1 (LEN (ISORT X2)))).
```

```
  ^^^ Checkpoint Subgoal *1/1'4' ^^^
```

It used one more tool in its arsenal by using the equality assumption and forgetting about it(throwing it away).

```
  Name the formula above *1.1.
```

Now it has run out of things to do without using induction...

```
  No induction schemes were specified for *1.1, and the depth limit for
  automatic inductions, currently 1, has been reached.  Consequently,
  the proof attempt has failed.
```

... but it's already done induction once, so it won't again.

```
  Summary
  Form:  ( DEFTHM LEN--ISORT ...)
  Rules: ((:DEFINITION ISORT)
          (:DEFINITION LEN)
          (:ELIM CAR-CDR-ELIM)
          (:EXECUTABLE-COUNTERPART EQUAL)
          (:EXECUTABLE-COUNTERPART LEN)
          (:FAKE-RUNE-FOR-TYPE-SET NIL)
          (:INDUCTION ISORT)
          (:INDUCTION LEN))
  Warnings:  None
  Time:  0.03 seconds (prove: 0.01, print: 0.01, proof tree: 0.01, other: 0.00)
```

Here it tells us the rules it used and how much time it took.

```
  ---
  The key checkpoint goals, below, may help you to debug this failure.
  See :DOC failure and see :DOC set-checkpoint-summary-limit.
  ---

  *** Key checkpoint at the top level: ***

  Goal
  (EQUAL (LEN (ISORT X)) (LEN X))
```

```
*** Key checkpoint under a top-level induction: ***

Subgoal *1/1'''
(IMPLIES (EQUAL (LEN (ISORT X2)) (LEN X2))
         (EQUAL (LEN (INSERT X1 (ISORT X2)))
                (+ 1 (LEN X2))))

ACL2 Error in ( DEFTHM LEN--ISORT ...):  See :DOC failure.

******** FAILED ********
```

Very often, we do not have to look at all of a failed proof; we can just look at the "key checkpoints". Basically, ACL2 is telling us that it has reduced the original conjecture to the formula

```
(IMPLIES (EQUAL (LEN (ISORT X2)) (LEN X2))
         (EQUAL (LEN (INSERT X1 (ISORT X2)))
                (+ 1 (LEN X2))))
```

So if you can help ACL2 prove that, it can prove the original conjecture.

In this case, looking at a formula later in the proof is perhaps even more helpful:

```
(EQUAL (LEN (INSERT X1 (ISORT X2)))
       (+ 1 (LEN (ISORT X2)))).
```

This is rather simple, and it should be true, right? How about we ask ACL2 to prove this, so that it can try induction on this? It turns out that ACL2 fails to prove this formula exactly. If we look at it carefully, we see that using induction on exactly this formula does not work out well.

What's the technique we've looked at that allows us to "correct" a formula so that induction works out?

Generalization! How can we make the above proposition more general?

Is this a proposition about `isort`? No. In fact, it does not matter that `isort` is being called. We can replace `(isort X2)` with a new variable to make the formula more general:

```
(EQUAL (LEN (INSERT X1 y))
       (+ 1 (LEN y))).
```

If we put this in a `defthm` with a name, ACL2 is able to prove this. And with that proven, ACL2 can prove `len--isort`, because ACL2 knows it can instantiate `y` with `(isort X2)`.

So after the function definitions, we have

```
(defthm len--isort--lemma
  (equal (len (insert x y))
         (+ 1 (len y))))

(defthm len--isort
  (equal (len (isort x))
         (len x)))
```

For appearance, I made everything lowercase and replaced x1 with just x.

Equality theorems as rewrite rules in ACL2

When you prove a theorem with DEFTHM, ACL2 by default will create a "rewrite rule" for use in future proofs. Let's examine the anatomy of rewrite rules with the above example:

```
(defthm len--insert
  (equal (len (insert x y))
         (+ 1 (len y))))
```

When you prove something of the form

```
(equal lhs
       rhs)
```

Then ACL2 creates a rewrite rule to rewrite instances of *lhs* into *rhs*. *lhs* stands for Left Hand Side and *rhs* for Right Hand Side. Variables in *lhs* can *match* anything, because we can use instantiation to replace each variable with any expression.

So the theorem named `len--insert` above tells ACL2 to look for instances of

```
(len (insert x y))
```

in what it is proving (where x and y can be anything), and rewrite those into

```
(+ 1 (len y))
```

where `y` is replaced with what matched it in the left hand side. Conceptually, this is desirable because the right hand side, `(+ 1 (len y))` is simpler to work with. In fact, we've completely removed `x` from expression; it turns out to be irrelevant to what `(len (insert x y))` is equal to.

The next example is

```
(defthm len--isort
  (equal (len (isort x))
         (len x)))
```

This of course searches for instances of

```
(len (isort x))
```

where x can be anything, and rewrites them into

```
(len x)
```

This is obviously simpler, even though we weren't able to eliminate any variables in the left hand side from the right hand side.

This brings up an interesting point. If we had written

```
(defthm len--isort2
  (equal (len x)
         (len (isort x))))
```

that is an equivalent proposition (because of the symmetry of equality), so it is also a theorem. But when given to ACL2, the rewrite rule created is different from `len--isort` because the right and left hand sides are switched. This new one will look for instances of

```
(len x)
```

and rewrite them to

```
(len (isort x))
```

That is not at all useful. In fact, if ACL2 encounters

```
(len (blah x))
```

it would rewrite that to

```
(len (isort (blah x)))
```

That too is an instance of `(len x)`, so it would rewrite that to

```
(len (isort (isort (blah x))))
```

and that to

```
(len (isort (isort (isort (blah x)))))
```

etc.

The lesson is that the order of the equality matters when it is used to create a rewrite rule. It is best to rewrite more complicated things into simpler

ones but it is not always clear which is simpler.

Proofs in ACL2(Example 2)

We can also ask ACL2 to attempt to prove

```
  (defthm tlp--isort
    (tlp (isort x)))

It fails, and here's the key checkpoint under induction:

  *** Key checkpoint under a top-level induction: ***

  Subgoal *1/2'4'
  (IMPLIES (TLP (ISORT X2))
           (TLP (INSERT X1 (ISORT X2))))
```

And this one calls for the same generalization: the `(isort X2)` could be anything, so we replace that with another variable:

```
  (defthm tlp--insert
    (implies (tlp y)
             (tlp (insert x y))))
Proving the above lemma, helps ACL2 get past the checkpoint
  (defthm tlp--isort
    (tlp (isort x)))
```

Theorems(with hyps) as conditional rewrite rules in ACL2

The next example we will see is of the form:

```
(defthm φ )

for e.g we just saw:
  (defthm true-listp--isort
    (true-listp (isort x)))
```

This is not an equality and not an implication (see below), so basically the whole thing is the left hand side and the right hand side is sort of `T`, but not exactly. Whenever ACl2 sees a `defthm` of the above form, it stores it as the following rewrite rule:

`(iff` ϕ `t)`

Note that `iff` is the same as `equal`, but only in boolean context, so how does ACl2 use the above rule? Simple, it checks wether the formula occurs in the context that is boolean, and if it is indeed the case, then ACL2 rewrites the instance of ϕ to `T`. Here is an example:

```
(if (true-listp (isort y))
    a
   b)
```

is equal to

```
(if T
    a
   b)
```

by the rewrite rule, because the test of an IF is boolean context.

Also,

```
(implies something
         (true-listp (isort z)))
```

is equivalent to

```
(implies something
         T)
```

which, by propositional reasoning, is equivalent to `T`.

However, in `(integerp (true-listp (isort x)))`

the rewrite rule does not apply because the argument to integerp is not a boolean context. It usually works to think of ACL2 as rewriting formulas like `(true-listp (isort x))` to `T` only if it makes sense for them to be only *True* or *False*.

Lets look at an theorem of the form (where the formula is an implication, i.e has hypotheses):

```
  (implies hyp
           concl
or
  (implies (and hyp_1
                hyp_2
                ...
                hyp_n )
           concl)
```

Where *concl* can be just a *lhs* which rewrites to `T` or like before it can just be (equal *lhs rhs*).

An example is

```
(defthm true-listp--insert
  (implies (true-listp y)
           (true-listp (insert x y))))
```

As you might guess, such theorem give rise to **conditional** rewrite rules, where the *hypotheses* identify the conditions under which a rewrite rule can be used. When a conditional rewrite rule **matches the formula**[1], ACL2 quietly attempts a miniature proof[2] of the hypotheses of the rewrite rule. If it succeeds, the rule is applied(we also say the rule **fires**), but if it fails, ACL2 proceeds with the original proof without applying the rule(i.e the rule does not fire).

```
(implies (true-listp y)
         (true-listp (insert x y)))
```

[1]See lecture 26 rewriting notes. It nothing new, basically, a rule/lemma "matches" a formula, is the same as saying, there is an instantiation of this rule/lemma that makes it equal to the formula

[2]That short proof attempt is called **Backchaining**

means that ACL2 will search for instances of

```
(true-listp (insert x y))
```

where `x` and `y` can be anything. When it finds that, it will attempt to prove `(true-listp y)`, where `y` is what was matched for `y` in `(true-listp (insert x y))`.

For example, if I were to ask ACL2 to prove

```
(true-listp (insert e (isort x)))
```

It will see that the rule `true-listp--insert` matches with the substitution `((x e) (y (isort x)))`, but there is a hypothesis to prove, which will call for proof of

```
(true-listp (isort x))
```

This rewrites to `T` by the rewrite rule `true-listp--isort`, so the rule `true-listp--insert` can be applied, which rewrites the conjecture to `T`.

Heres an example of a conditional rewrite rule not firing, even though it matches:

```
(defthm app-nil
  (implies (tlp x)
           (equal (app x nil) x)))

(defthm tlp-app
  (equal (tlp (app x y))
         (tlp y)))

(defthm app-assoc
  (equal (app (app x y) z)
         (app x (app y z))))

(defthm rev-app
  (equal (rev (app A B))
         (app (rev B) (rev A))))
```

ACL2 proof of the rev-app lemma fails at this checkpoint:

```
Subgoal *1/1''
(IMPLIES (NOT (CONSP A))
         (EQUAL (REV B)
                (APP (REV B) NIL)))
```

If ACL2 tries to rewrite the above formula, it will *match* `(APP (REV B) NIL)` with the `app-nil` rule. Why does it match?
Simply, because `app-nil concl`$|_{\sigma=((x\ (REV\ B)))}$ is the same sequence of symbols as `(app (REV B) nil)`[3]. But clearly ACL2 is not using this lemma/rule, the reason why `app-nil` lemma is not firing is because ACL2 cant prove the condition `(tlp (REV B))` under which the rewrite rule can be applied. So we help ACL2, by giving it the `rev-tlp` lemma:

```
(defthm tlp-rev
 (tlp (rev x)))
```

This helps ACL2 relieve/satisfy the hypothesis(condition) of `app-nil` rule, so that it fires and ACL2 proof goes through.

Generalisation

Since nobody asked me to elaborate on the email I sent you, I assume, you did understand my email. So instead of repeating myself in greater detail, I will solve a few questions from the homework:

Let me rephrase the question slightly. But first let me repeat what I said in my email, but I will be more precise and closer to whats expected in the HW:
Lets define the following:
A is a **generalization** of B :iff $A|_\sigma \Rightarrow B$

Lets motivate this definition in the context of rewriting: why do we care wether A is more general than B? Simply because the more general formula makes a better rewrite rule, for example, in one of the quizzes I asked you to find a general lemma for the following checkpoint:

[3] ACL2 is case-insensitive

```
Subgoal *1/3'4'
   (EQUAL (REV (APP (REV X2) (LIST X1)))
          (CONS X1 (REV X2)))
```

This is a slight variation of a checkpoint that ACL2 gets stuck in while proving that the reverse of the reverse of the list gives back the original list. Now here we need to come up with a general lemma which would help ACL2 get past this checkpoint, suppose we have a choice, we come up with 3 lemmas:

```
Lemma A:
   (EQUAL (REV (APP (REV X2) (LIST X1)))
          (CONS X1 (REV (REV X2))))
Lemma B:
   (EQUAL (REV (APP A (LIST X1)))
          (CONS X1 (REV A)))
Lemma C:
   (EQUAL (REV (APP A B))
          (APP (REV B) (REV A)))
```

When we have a choice, we want to know which is the best, so we want to choose a rewrite rule, which will help ACL2 the most, clearly all of them help ACL2 prove the above checkpoint, but since this lemma will go into the database of rewrite rules, we want a nice rule, which can be applied in as many proofs as possible. So the question is what lemma should I choose, well choose the one which is most general, that is which can be applied in most contexts, that is which can be instantiated with least restrictions. Look at lhs of B and lhs of A, clearly lhs of B has a free variable `A` in place of `(rev X2)` and so can match any term, but `(rev X2)` can only match terms starting with `(rev ...)`. So Lemma B is a better rewrite rule. Similarily, look at lhs of C and lhs of B, clearly, the lhs of C has a free variable `B` instead of `(list X1)` and will match many more terms. Therefore Lemma C is a much nicer rewrite rule than Lemma B.

Finally another way of answering which lemma is a nicer rewrite rule, one can just ask the question: does Instantiation of lemma C $\Rightarrow$ lemma B. The answer ofcourse is yes and in fact, B is a generalization of A, and C is a generalization of C, and in more precise terms: $C|_\sigma \Rightarrow B$ and $B|_\sigma \Rightarrow A$

But the reasons of the above implications should not be very long. To be more precise they should be something like:

C'
$\Rightarrow$ {`Instantiation of C, plus few obvious reasons that relieve hyps`}
B

If the above is true, then we say C is a generalization of B.

Lets look at some worked out solutions: Part 3 of Problem 4 in HWK:

A`(Original):`
```
(implies (and (integer-listp x)
              (integer-listp y))
         (true-listp (app y x)))
```

B`(Generalization):`
```
(implies (and (true-listp x)
              (true-listp y))
         (true-listp (app x y)))
```

Solution: **True**. To show that is true we will prove(with obvious lemmas) that $B_\sigma \Rightarrow A$:
Using all the simplifications[4] we normally do, we have the following simplified proof with context:
Proof:

[4]See Lec 23

```
(true-listp (app y x))
⇐ {D3 }
t
```

Context

```
Instantiate with σ = ((x y) (y x))
A1:
(implies (and (true-listp y)
              (true-listp x))
         (true-listp (app y x)))
A2: (integer-listp x)
A3: (integer-listp y)
Using obvious lemma L1
(implies (integer-listp A)
         (true-listp A))
we have the following derived
context:
D1: (true-listp x) {A2,L1}
D2: (true-listp y) {A3,L1}
Using {D1,D2,A1,Modus Ponens}
D3: (true-listp (app y x))
```

Part 4 of Problem 4:

```
φ(Original):
(implies (and (integer-listp x)
              (integer-listp y))
         (integer-listp (app x y)))

ψ(Generalization):
(implies (and (integer-listp x)
              (integer-listp y))
         (true-listp (app x y)))
```

Sol: **False**, Clearly (true-listp A) does not imply (integer-listp A). Heres a simple counterexample: A = `'(a 1/3 ``ok'')` which is a true-list, but not an integer-list.