Using the ACL2 Theorem Prover

We already saw that we could use the ACL2 theorem prover to prove validity of formulas of propositional logic.

```
ACL2 > (thm (implies (and a b) a))

But we reduce the conjecture to T, by case analysis.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: NIL
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)

Proof succeeded.
```

The `(thm F)` form is used to query the theorem prover, asking it to prove formula `F`. This can either succeed or fail. The above generalizes to any formula that can be proved using propositional reasoning alone.

```
ACL2 > (thm (implies (and (true-listp a)
                          (true-listp b))
                     (true-listp a)))

But we reduce the conjecture to T, by case analysis.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: NIL
Warnings:  None
```

```
Time:  0.00 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)

Proof succeeded.
```

What about validity of formulas in ACL2 logic? Let's work through some of the theorems that we proved by hand in class, and see how ACL2 proves them. As we'll see, the proofs that the prover comes up with are essentially those that we came up with as well. The differences are "in the leaves", that is, at the level of the simplest axioms—ACL2 will use slightly different terminology than we have been using, but the effect will be just the same.

Onward. First thing we proved was a bunch of theorems about `app`. So let's define `app`:

```
ACL2 > (defun app (x y)
         (if (endp x)
           y
           (cons (car x) (app (cdr x) y))))

The admission of APP is trivial, using the following CCMs:
(APP CCG (ACL2-COUNT X)).    We observe that the type of APP is described
by the theorem (OR (CONSP (APP X Y)) (EQUAL (APP X Y) Y)).  We used
primitive type reasoning.

Summary
Form:  ( DEFUN APP ...)
Rules: NIL
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)
 APP
```

The output here is just to tell you that ACL2 managed to prove that `app` terminates. Remember that only admissible functions can be used, and an admissible function has to terminate.

The first few theorems we wanted to prove were straightforward.

```
ACL2 > (thm (= (app NIL y) y))

By the simple :definition = we reduce the conjecture to

Goal'
(EQUAL (APP NIL Y) Y).
```

```
But simplification reduces this to T, using the :definition APP, the
:executable-counterpart of CONSP and primitive type reasoning.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: ((:DEFINITION =)
        (:DEFINITION APP)
        (:EXECUTABLE-COUNTERPART CONSP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)

Proof succeeded.
```

You see that this proof is a simple proof that uses the definition of `app` and basically some simplification—in class, we used the endp and if axioms, here, ACL2 uses what it calls *executable counterpart*, which is something equivalent. (Roughly, this means that ACL2, when presented with an expression that contains only values, not variables, can actually *execute* the expression to simplify it down to a value, and replaces the expression by the resulting value.)

We can generalize this result:

```
ACL2 > (thm (implies (endp x) (= (app x y) y)))

By the simple :definitions = and ENDP we reduce the conjecture to

Goal'
(IMPLIES (NOT (CONSP X))
         (EQUAL (APP X Y) Y)).

But simplification reduces this to T, using the :definition APP and
primitive type reasoning.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: ((:DEFINITION =)
```

```
        (:DEFINITION APP)
        (:DEFINITION ENDP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)

Proof succeeded.
```

Again, an essentially trivial proof. (Just like it was on paper, frankly.)

I will refer you to the reference book on ACL2, page 102, for a breakdown of what it is that ACL2 actually does when trying to prove a formula. It really has a single way of proving formulas, and it follows a bunch of steps in order, trying everything until it either gets stuck or manages to prove the formula. This is called the waterfall model.

Basically, proving a formula can be broken down into trying to prove a bunch of *goals*. Initially, there is a single goal, the formula to prove. As we saw when we looked at induction, say, some proofs require breaking a formula down into other formulas to be proved, called proof obligations. This corresponds to replacing a goal (the formula) by two other goals (the proof obligations). Some of those goals can be further broken down into other goals, although we haven't encountered such examples in class.

So at any point during the proof, there is a pool of goals to be proved. ACL2 will take a formula out of that pool, and attempt to work on it using one of the following, in order:

(1) simplification: trying to rewrite the formula into a simpler formula using either an axiom or a previously proved theorem;

(2) destructor elimination: this step corresponds to replacing, in a context where we know `(consp x)`, every `(car x)` by `x1` and every `(cdr x)` by `x2` and every other `x` by `(cons x1 x2)`, and eliminating `(consp x)` from the context;

(3) use of equivalences: this lets the prover reason about equivalence relations you may have defined (which we haven't seen);

(4) generalization: trying to replace the formula by a more general formula;

(5) elimination of irrelevance: trying to identify formulas in the context that do not seem to play a role in the formula to be proved, and removing them;

(6) induction: trying to identify an induction scheme and derive new goals based on the derived proof obligations.

After applying the first step that works, the result of applying the step is sent back to the pool of goals (and often picked right up again for more work).

Note that generalization and elimination of irrelevance require modifying the meaning of a formula, and can be problematic because at that point ACL2 may end up trying to prove something false. We shall do our upmost to keep the prover from generalizing—basically by proving theorems that can be used to simply and avoid the need to generalize. More on this later.

The next thing we proved was the first theorem that required an induction.

```
ACL2 > (thm (true-listp (app x NIL)))

^^^ Checkpoint Goal ^^^

Name the formula above *1.

Perhaps we can prove *1 by induction.  One induction scheme is suggested
by this conjecture.

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

Subgoal *1/2
(IMPLIES (AND (NOT (ENDP X))
              (TRUE-LISTP (APP (CDR X) NIL)))
         (TRUE-LISTP (APP X NIL))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP X)
              (TRUE-LISTP (APP (CDR X) NIL)))
         (TRUE-LISTP (APP X NIL))).

But simplification reduces this to T, using the :definition APP, primitive
type reasoning and the :type-prescription rule APP.
```

```
Subgoal *1/1
(IMPLIES (ENDP X)
         (TRUE-LISTP (APP X NIL))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP X))
         (TRUE-LISTP (APP X NIL))).

But simplification reduces this to T, using the :definition APP and
the :executable-counterpart of TRUE-LISTP.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: ((:DEFINITION APP)
        (:DEFINITION ENDP)
        (:DEFINITION NOT)
        (:EXECUTABLE-COUNTERPART TRUE-LISTP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION APP)
        (:TYPE-PRESCRIPTION APP))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)

Proof succeeded.
```

You see that induction required the determination of which induction scheme to use (here, the usual one), and the system came up with two proof obligations, which it called `*1/1` and `*1/2`, which became the new goals replacing the original goal `*1`. It then tries (and succeeds) in proving those two goals, and each of those are pretty simple.

Note also that the proof includes *checkpoints*, which roughly speaking correspond to interesting spots in the proof. Here, the only interesting spot is where we perform induction. We will see other interesting checkpoints next lecture.

```
ACL2 > (defthm true-listp-app
         (= (true-listp (app x y)) (true-listp y)))
```

```
By the simple :definition = we reduce the conjecture to

Goal'
(EQUAL (TRUE-LISTP (APP X Y))
       (TRUE-LISTP Y)).
^^^ Checkpoint Goal' ^^^

Name the formula above *1.

Perhaps we can prove *1 by induction.  Two induction schemes are suggested
by this conjecture.  However, one of these is flawed and so we are
left with one viable candidate.

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

Subgoal *1/2
(IMPLIES (AND (NOT (ENDP X))
              (EQUAL (TRUE-LISTP (APP (CDR X) Y))
                     (TRUE-LISTP Y)))
         (EQUAL (TRUE-LISTP (APP X Y))
                (TRUE-LISTP Y))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP X)
              (EQUAL (TRUE-LISTP (APP (CDR X) Y))
                     (TRUE-LISTP Y)))
         (EQUAL (TRUE-LISTP (APP X Y))
                (TRUE-LISTP Y))).

But simplification reduces this to T, using the :definitions APP and
```

```
TRUE-LISTP, primitive type reasoning and the :rewrite rule CDR-CONS.

Subgoal *1/1
(IMPLIES (ENDP X)
         (EQUAL (TRUE-LISTP (APP X Y))
                (TRUE-LISTP Y))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP X))
         (EQUAL (TRUE-LISTP (APP X Y))
                (TRUE-LISTP Y))).

But simplification reduces this to T, using the :definition APP and
primitive type reasoning.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM TRUE-LISTP-APP ...)
Rules: ((:DEFINITION =)
        (:DEFINITION APP)
        (:DEFINITION ENDP)
        (:DEFINITION NOT)
        (:DEFINITION TRUE-LISTP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION APP)
        (:REWRITE CDR-CONS))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)
 TRUE-LISTP-APP
```

Same business as above, a standard proof by induction, knocked out pretty simply. Note we use `(defthm NAME F)` to prove the theorem. As with `(thm F)`, this attempts to prove formula `F`. But if the proof succeeds, then the resulting theorem is *remembered* with name `NAME`, and can subsequently be used in proofs.

This is both useful and not completely trivial:

```
ACL2 > (defthm app-assoc
         (= (app (app x y) z) (app x (app y z))))

By the simple :definition = we reduce the conjecture to

Goal'
(EQUAL (APP (APP X Y) Z)
       (APP X (APP Y Z))).
^^^ Checkpoint Goal' ^^^

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  Subsumption reduces that number to two.
However, one of these is flawed and so we are left with one viable
candidate.

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

Subgoal *1/2
(IMPLIES (AND (NOT (ENDP X))
              (EQUAL (APP (APP (CDR X) Y) Z)
                     (APP (CDR X) (APP Y Z))))
         (EQUAL (APP (APP X Y) Z)
                (APP X (APP Y Z)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP X)
              (EQUAL (APP (APP (CDR X) Y) Z)
                     (APP (CDR X) (APP Y Z))))
         (EQUAL (APP (APP X Y) Z)
```

```
                (APP X (APP Y Z)))).

But simplification reduces this to T, using the :definition APP, primitive
type reasoning and the :rewrite rules CAR-CONS and CDR-CONS.

Subgoal *1/1
(IMPLIES (ENDP X)
         (EQUAL (APP (APP X Y) Z)
                (APP X (APP Y Z)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP X))
         (EQUAL (APP (APP X Y) Z)
                (APP X (APP Y Z)))).

But simplification reduces this to T, using the :definition APP and
primitive type reasoning.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM APP-ASSOC ...)
Rules: ((:DEFINITION =)
        (:DEFINITION APP)
        (:DEFINITION ENDP)
        (:DEFINITION NOT)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION APP)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, proof tree: 0.00, other:
   0.00)
 APP-ASSOC
```

Again, a proof by induction, and the proofs of the obligations here are a bit longer, but involve only simplifications.

So, as you can see, all of these proofs are straightforward, and more importantly, they correspond to the proofs that you end up writing by hand.

Next time, we will see proofs that are more involved, and for which ACL2 needs a bit of guidance.