Introduction to ACL2

A few starting remarks: ACL2 can be run from the department linux machines. It is
  /isd/users/software/ACL2/acl2-8.2/saved_acl2
Give that command and you should get:

```
...
ACL2 Version 8.2.  Level 1.  Cbd "/ubc/cs/home/m/mrg/".
System books directory "/ubc/cs/research/isd/users/software/ACL2/acl2-8.2/books/".
Type :help for help.
Type (good-bye) to quit completely out of ACL2.

ACL2 !>
```

You can then follow along with the examples here.

You can also install ACL2 on your own machine – linux or osx. If you're running Windows, do not despair. The standard solution is to install a virtual machine such as VirtualBox and set up a linux machine there. Then, go to:
  http://www.cs.utexas.edu/users/moore/acl2/v8-2/HTML/installation/installation.html
and follow the instructions. Building ACL2 itself is fairly quick and painless. Once you have a basic ACL2, you really need the standard books (step 4 in the instructions). Certifying the books is fairly slow and painless. Just let it run overnight, and all should be well. Why does it take so long? First, because there are lots of collections of useful theorems. Second, because ACL2 *proves* each theorem when certifying the book – nothing is trusted; so you are protected against getting a bogus theorem corrupting your theorem prover.

There are many more resources available from the ACL2 home page:
  http://www.cs.utexas.edu/users/moore/acl2/index.html
Check out the tutorials:
  http://www.cs.utexas.edu/users/moore/acl2/v8-2/combined-manual/index.html?topic=ACL2____ACL2-TUTORIAL
The xdoc documentation,
  http://www.cs.utexas.edu/users/moore/acl2/v8-2/combined-manual/index.html?topic=SMT____SMTLINK
is handy, but like typical API documentation, it's more useful once you already have some familiarity with the language. With that in mind, I'll give a few, narrated examples here.

1 Sum

The sum program from the March 17 lecture:

```
(defun sum (n)
 (if (zp n) ;; (zp n) true iff n is not an integer or n <= 0
     0      ;; base case
     (+ n (sum (- n 1))) ;; recursive case
))
```

You can get the code from   http://www.cs.ubc.ca/~mrg/cs513/2019-2/src/sum.lisp
(sum n) computes the sum of the first n natural numbers. If you already are familiear with list, you can jump down to the proof below. Here, I'll give a very terse summary:

(f arg1 arg2 ... argn): Call function f with arguments arg1, arg2, ..., argn. For example:

```
ACL2 !> (+ 2 3)
5
ACL2 !> (+ 2 (* 3 4))
14
```

`(defun f (arg1 arg2 ... argn) body)`: Define a function named `f` with `n` arguments. `body` is an expression that is the value returned by `f`.

`(if cond then-expr else-expr)`: a conditional expression. Note: lisp is functional, and it is more natural to use recursion than some "loop" syntax. In the `sum` function, the `then-expr` is `0`; it's the base-case for the recursion. The `else-expr` is `(+ n (sum (- n 1)))`; it's the recursive case.

`(zp n)`: true iff n is not an integer or `n` ≤ 0.

We can cut-and-paste this code into ACL2, and it should print a bunch of stuff ending with:

```
 SUM
ACL2 !>
```

If you get some error message instead, check to make sure that you did the cut-and-paste correctly, and then ask for help. If you type in the definition and ACL2 just hangs; it is likely that your missing a right parenthesis. You can try adding one right-parenthesis at a time (with a newline after each one), or just hit `<ctrl>-C` which should get you back to the `ACL2 !>` prompt and try again.

Let's prove something

This will be the standard, introduction to induction example. We want to prove the closed-form for sum:

$$\sum_{j=0}^{N} j = \frac{N(N+1)}{2}$$

We state this theorem in ACL2 as:

```
(defthm closed-form-for-sum
 (implies (natp n)
          (equal (sum n)
                 (/ (* n (+ n 1)) 2))))
```

We can cut-and-paste this code into ACL2, and get:

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

Subgoal *1/4'
(IMPLIES (AND (NOT (ZP N))
              (EQUAL (SUM (+ -1 N))
                     (+ (* 1/2 N)
                        (* -1/2 N)
                        (* (+ -1 N) 1/2 N)))
              (<= 0 N))
         (EQUAL (+ N (SUM (+ -1 N)))
         (+ (* 1/2 N) (* N 1/2 N))))

ACL2 Error in ( DEFTHM CLOSED-FORM-FOR-SUM ...):  See :DOC failure.
```

What happened? The "`Subgoal *1/4'`" says that ACL2 has broken the proof into smaller pieces, i.e. subgoals. `*1/4'` is the name for this subgoal. The `*1` means that ACL2 is attempting a proof by induction. The `4` means that the induction proof has four cases. The single `'` means that the goal has been "simplified" by one pass (one-pass for one code') through the rewriter. ACL2 has some simple rules that it tries first (e.g. substituting equals-for-equals). If it can't prove the goal with the simple methods, it tries progressively more "powerful" reasoning methods. Of course, ACL2 is incomplete – it can't prove every theorem in first-order logic (Gödel's Incompleteness Theorem). Some of these more "powerful" methods replace the goal with something that implies the goal, and ACL2 can fail due to over strengthening. That's why it tryis the simple methods first. For all of this trying, ACL2 failed. What should we do?

The issue here is that ACL2 "knows" the definitions of basic lisp functions such as `+`, `*`, and `if`, but it doesn't know the rules of high-school algebra. Fortunately, other users have already proven collections of theorems about algebra so we don't have to. We tell ACL2:

```
ACL2 !> (include-book "arithmetic/top" :dir :system)

Summary
Form:  ( INCLUDE-BOOK "arithmetic/top" ...)
Rules:  NIL
Time:  0.27 seconds (prove:  0.00, print:  0.00, other:  0.27)
"/Users/mrg/src/acl2/acl2-8.yan/books/arithmetic/top.lisp"
```

and we try the theorem again. Just cut-and-paste it into ACL2 again, and we get

```
Prover steps counted:  569
CLOSED-FORM-FOR-SUM
```

Think about how much more fun your introductory discrete-math course would have been if you had been able to use ACL2 to do those tedious proofs for you!

How did ACL2 prove this. We can scroll back and find

```
Goal''
(IMPLIES (NATP N)
         (EQUAL (SUM N)
                (+ (* 1/2 N) (* 1/2 N N))))

1 (Goal'') is pushed for proof by induction.
```

The `Goal''` part means that the original theorem has endured two passes through the rewriter. Basically, ACL2 has now used all the algebra that it knows from `arithmetic/top` and can't make any more progress. The proof narrative continues with

> Perhaps we can prove *1 by induction. One induction scheme is suggested by this conjecture.
>
> We will induct according to a scheme suggested by `(SUM N)`. . . .

We were expecting a proof by induction. What does "a scheme suggested by `(SUM N)`" mean? If we scroll back to where `sum` was defined, we find:

> The admission of `SUM` is trivial, using the relation `O<` (which is known
> to be well-founded on the domain recognized by `O-P`) and the measure
> `(ACL2-COUNT N)`.

This says that ACL2 proved that `(sum x)` always terminates, no matter what the value of `x` is when `sum` is called. It does this by using the function `acl2-count n` to map `x` to a natural number. Note that if `x` is a natural number, then `(equal (acl2-count n) n)`. Want to be sure this is true? Just try:

```
ACL2 !> (thm (implies (natp n) (equal (acl2-count n) n)))
Q.E.D.
...
Proof succeeded.
ACL2 !>
```

If a recursive function, `f`, terminates, then it defines a well-founded-relation on its arguments. One way to think of this is to consider the tree of recursive calls made when evaluating `(f arg1 arg2 ... argn)`. If `h` is the height of this tree, then we can define

```
(f-measure arg1 arg2 ... argn) -> h
```

Clearly,

- The measure of the arguments to `f` is a natural number.
- Each recursive call made by `(f arg1 ...)` has a smaller measure than that the measure of `arg1`, ...
- When the measure is 0, code f returns.

This makes `f-measure` a well-founded relation, which means we can perform proof by induction. For `sum`, the well-founded relation is `O<` which is the ordering of the ordinal numbers. We could take a fun tangent into ordinals and transfinite-set theory, but not in this lecture.

Once we have a well-founded relation, we have an induction scheme. Basically, we need to prove the conjecture with the assumption that the conjecture holds for all values of the variables that have a smaller measure. For the base-cases (the non-recursive cases), there are no cases with smaller measure; so we have to prove it directly. For the inductive cases (where `f` will make a recursive call), we get to assume that the conjecture holds for the values returned by these recursive calls. With this, ACL2 sets up the induction proof:

```
(AND (IMPLIES (AND (NOT (ZP N)) (:P (+ -1 N)))
              (:P N))
     (IMPLIES (ZP N) (:P N)))
```

Where `(:P N)` is short-hand for `Goal''`, i.e.

```
(IMPLIES (NATP N)
    (SUM N(EQUAL
    (+ (* 1/2 N) (* 1/2 N N))))
```

We recognize that
`(IMPLIES (AND (NOT (ZP N)) (:P (+ -1 N))) (:P N))`
is the "induction step", and `(IMPLIES (ZP N) (:P N)))` is the "base case". ACL2 generates three subgoals from this. Subgoal `*1/3` is the induction step. Subgoals `*1/1` and `*1/2` are the base-case – it splits into two because of the compound hypothesis of `(zp n)` and `(natp n)`. Details provide upon request, but I'll skip them here to keep this as an "introduction".

2 Quick Sort

You can get the code from http://www.cs.ubc.ca/~mrg/cs513/2019-2/src/qsort.lisp
I also borrowed some code (see the comments) for a definition of permutatons and proofs about permutations.
http://www.cs.ubc.ca/~mrg/cs513/2019-2/src/perm.lisp
The function `(qsort x)` sorts the list `x` using the Quick Sort algorithm. I wrote a helper function `(split x pivot)` that returns two lists: the elements of `x` that are ≤ pivot, and the elements that are greater than pivot. The return values are of the form `(mv yle ygt)`
where `mv` is a function to return multiple-values. I call `split` as `(mv-let (yle ygt) (split x pivot) expr)`
which binds `yle` to the first return value of `split` and `ygt` to the second return value. I thought this would make it easier to show that split preserves permutation, but this was an unnecessary precaution – the quick sort implementation from 2002 (see my comments in `perm.lisp`) had separate functions for returning the ≤ list and the > list, and their proofs worked without hints.

Let's look briefly at `qsort` itself.

```
(defun qsort (x)
 (cond ((atom x) x)
       ((atom (cdr x)) x)
       (t (mv-let (yle ygt) (split (cdr x) (car x))
                  (append (qsort yle) (cons (car x) (qsort ygt)))))))
```

We might be temted to call `split` on `x` instead of `(cdr x)`. Then, we wouldn't need to `(cons (car x) ...)` onto `(qsort ygt)`. That would avoid some clutter. Let's try it:

```
(defun qsort2 (x)
 (cond ((atom x) x)
       ((atom (cdr x)) x)
       (t (mv-let (yle ygt) (split x (car x))
                  (append (qsort yle) (qsort ygt))))))
```

I did, and ACL2 said:

> ACL2 Error in `(DEFUN QSORT2 ...)`: The proof of the measure conjecture for `QSORT2` has failed. See `:DOC` failure.
>
> ******* FAILED ********

I could work through the failed subgoal, but the issue is that if `x` has duplicate entries, then `qsort2` will reach a place where `(split x (car x))` returns `x` and `nil`, and the code recurses forever. Let's go back to `qsort` as defined above. ACL2 proves termination without any help. Now we want to prove that `qsort` sorts correctly.