Peter Dillinger

Arithmetic in Proofs

Let us try to prove something that requires some arithmetic. Assuming these definitions of MIN and MAX:

```
(equal (min x y)
       (if (<= x y) x y))

(equal (max x y)
       (if (<= x y) y x))
```

Let us try to prove

```
(equal (- (+ x y)
          (max x y))
       (min x y))

<= { Def min, Def max }

(equal (- (+ x y)
          (if (<= x y) y x))
       (if (<= x y) x y))
```

Now we can use a strategy we learned about recently, Case Analysis:

Case 1: Assume (<= x y)

```
<= { Assumption (<= x y), IF axiom }
  (equal (- (+ x y)
            y)
         x)
```

Well, ACL2 has axioms concerning arithmetic, and most everything you know from high school mathematics can be proven from those axioms. Doing so can be very hard and beyond the scope of this course, so we allow you to use mathematics you know from high school in proofs.

For example, we know that in mathematics, (x + y) - y = x, so we are allowed to use that in writing ACL2 proofs. This might lead one to believe we could just go to

```
<= { Arithmetic }
  t
```

However, there's a problem. The original conjecture is not a theorem! Consider this example:

```
(let ((x nil)
      (y nil))
  (equal (- (+ x y)
            (max x y))
         (min x y)))
```

But ACL2 arithmetic treats non-numbers as 0, right? So what's the problem? Let's see:

```
(equal (- (+ nil nil)
          (max nil nil))
       (min nil nil))
```

Well, (max nil nil) and (min nil nil) are both nil, because they don’t necessarily return numbers, just one of their arguments.

```
(equal (- 0
          nil)
       nil)
=
(equal 0 nil)
=
nil
```

So you are allowed to use arithmetic reasoning, but you need to be sure you are working with numbers, because non-numbers might not satisfy arithmetic equations in ACL2, as we just saw.

If the conjecture were

```
(implies (and (integerp x)
              (integerp y))
         (equal (- (+ x y)
                   (max x y))
                (min x y))
```

then we could prove that like so:

Assumptions: (integerp x) (integerp y)

```
(equal (- (+ x y)
          (max x y))
       (min x y))

<= { Def min, Def max }

(equal (- (+ x y)
          (if (<= x y) y x))
       (if (<= x y) x y))

Case 1: Assume (<= x y)

<= { Assumption (<= x y), IF axiom }
  (equal (- (+ x y)
            y)
         x)
<= { Arithmetic }
  t

Case 2: Assume (not (<= x y))

<= { Assumption (<= x y), IF axiom }
  (equal (- (+ x y)
            x)
         y)
<= { Arithmetic }
  t
```

Introduction to Induction

Suppose we want to prove

```
  (>= (len x) 0)
```

Let's try to start proving it:

```
<= { Def len }
  (>= (if (endp x)
          0
          (+ 1 (len (cdr x))))
      0)
```

Case 1: assume (endp x)

```
<= { (endp x), IF }
  (>= 0 0)
<= { Arith }
  t
```

Case 2: assume (not (endp x))

```
<= { (not (endp x)), IF }
  (>= (+ 1 (len (cdr x)))
      0)
```

What do we do from here? We need to know something about (len (cdr x))
in order to prove this. For example, if we knew that (>= (len (cdr x)) 0),
then we could finish the proof by arithmetic reasoning.

But (>= (len (cdr x)) 0) is almost exactly like what we're trying to prove.
It seems like we are hopelessly stuck trying to prove this.

====

Let us consider a method of proof from Mathematics that will come to play a
central role in ACL2 proofs. Consider the following function:

```
  f(x) = { 0           if x = 0
         { x + f(x-1)  if x > 0 and an integer
```

What does this function correspond to? Well, what is f(3)?

```
  f(3) = 3 + f(2)
       = 3 + 2 + f(1)
       = 3 + 2 + 1 + f(0)
       = 3 + 2 + 1 + 0
```

Actually, f(x) is the sum of all the natural numbers up to and including x.
Some may recall a "closed form" formula for this:

```
  f(x) = x (x + 1) / 2
```

If we want to prove that the recursive definition of f satisfies this formula,
we would need to prove that it holds for all natural numbers, and for
problems like this, we don't get very far without INDUCTION.

If you want to prove something holds for all natural numbers, induction says
that you only need to
 - prove that it holds for 0
 - prove that it holds for x UNDER THE ASSUMPTION that it holds for x - 1.

Call the first one the BASE CASE and the second the INDUCTIVE STEP.

For example, the inductive step says that if it's true for 3, it must be true for 4. If it's true for 13, it must be true for 14. Etc.

If someone tries to claim there is a natural number, n, that makes the proposition false, we can present this argument:
- use base case proof to show it is true for 0
- instantiate inductive step to show if it is true for 0, it is true for 1
- instantiate inductive step to show if it is true for 1, it is true for 2
- instantiate inductive step to show if it is true for 2, it is true for 3
- ...
- instantiate inductive step to show if it is true for n-1, it is true for n

Because we can make an argument like this for any n, induction on natural numbers allows us to conclude from base case and inductive step theorems that a proposition holds for any natural number.

Let us continue our mathematical example. Let us prove the base case,

```
f(0) = 0 (0 + 1) / 2
     = 0
```

which agrees with the recursive definition. (I used mathematical reasoning you should be familiar with.)

Let us consider the inductive step, which says that we should assume

```
f(x-1) = (x - 1) x / 2
```

and use that to prove

```
f(x) = x (x + 1) / 2
```

where, according to its definition, f(x) = x + f(x-1).

Let us start with that definition and replace f(x-1) by what it is assumed to be equal to: (x^2 is x squared)

```
f(x) = x + f(x - 1)

     = x + ((x - 1) x / 2)

           x^2    x
     = x + --- - ---
           2      2

        x    x^2
     = --- + ---
        2     2

     = (x + x^2) / 2

     = x (x + 1) / 2
```

So I have shown f(x) = x (x + 1) / 2 assuming the same is true for the next smaller value of x. That concludes the mathematical proof by induction. We have proven that

```
f(x) = x (x + 1) / 2
```

for all natural numbers x, and with f defined recursively as above.

Induction in ACL2

ACL2 also has induction. We can use induction on natural numbers, but we can also use induction on lists. Here's the idea:

Base case: Proof that the proposition holds for all atoms (non-conses)

Inductive step: Assume we are given a cons and that the proposition holds for the CDR of that cons. Prove it holds for that cons structure.

The idea is that we can divide up the ACL2 universe as follows:

- Atoms (including t, nil, numbers, strings, etc.)
- Conses whose CDR is an atom
- Conses whose CDR of the CDR is an atom
- Conses whose CDR of the CDR of the CDR is an atom
- ...

Take what we wanted to prove at the beginning, (>= (len x) 0). If we prove it holds for the first division,

```
(implies (endp x)
         (>= (len x) 0))
```

and then prove that it holds for any division assuming it holds for the previous (the CDR):

```
(implies (and (consp x)
              (>= (len (cdr x)) 0))
         (>= (len x) 0))
```

Then we can conclude it holds for all the divisions of the universe, which is the whole universe. From the base case and inductive step, we can conclude the proposition is true:

```
(>= (len x) 0)
```

Later we will learn more about how this this works more generally, and, in more detail, how to apply it.