CS2800 Fall 2010 - Lecture 20

1st November 2010

Equivalently simpler proofs

Recall that an implication is of this form:
`(implies A C)`
where A is called the antecedent, or hypothesis, or assumption of the formula, and C is called its conclusion.

Given the following definition:

```
(defun mul (n m)
 (if (zp n)
     0
   (+ m (mul (- n 1) m))))
```

Lets Prove some cases of the conjecture that this slow multiplication gives the same result as the normal multiplication in ACL2(*):

```
(and (implies (zp n)
              (implies (and (natp n) (natp m))
                       (= (mul n m)
                          (* n m))))
     (implies (and (not (zp n))
                   (implies (and (natp n) (natp m))
                            (= (mul (- n 1) m)
                               (* (- n 1) m))))
              (implies (and (natp n) (natp m))
                             (= (mul n m)
                                (* n m)))))
```

Now to prove this formula, our proof will carry this huge formula around, which is very inconvenient. So we will break it down to (multiple) simple proof(s). First of all notice that, a conjuction can always be proved by proving the subformulas. We call them subgoals.

Subgoal 1:
```
(implies (zp n)
               (implies (and (natp n) (natp m))
                        (= (mul n m)
                           (* n m))))
```

Which can be furthur simplified,using the tautology:

$$P \to (R \to C) \equiv P \wedge R \to C$$

i.e. if you see an implication in the conclusion of the top-level implication, you can push its antecedents into conjunction of the top-level antecedents.

```
(implies (and (zp n)
               (natp m)  ;pushed here
               (natp n)) ;pushed here
          (= (mul n m) (* n m)))
```

Which mneans we can just prove `(= (mul n m) (* n m))` under the context(list of assumptions) which consists of three formulas from the antecendent(i.e. `(zp n) (natp m) (natp n)`).

Subgoal 2 similarily can be simplified to:

```
(implies (and (not (zp n))
               (implies (and (natp n) (natp m))
                              (= (mul (- n 1) m)
                                 (* (- n 1) m)))
               (natp n)  ;pushed here
               (natp m)) ;pushed here
          (= (mul n m)
             (* n m)))))
```

Now all we need to prove is LHS:`(mul n m)` is equal to RHS:`(* n m)` under the context which has 4 assumptions. Note: Second assumption in the context is called an Induction Hypothesis, which you will understand later. By the way, it is incorrect to simplify the Induction hypothesis furthur in this case using just Prop Deduction(tautology only helps push the antecents of the conclusion to the antecendents of the top-level implication.

We will later see how to deal with this situation in proofs by Induction.

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))
```

$\Leftarrow$ { *Def min, Def max* }

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

$\Leftarrow$ {`Case-analysis`}

Split into 2 sub-proofs(one for each case):

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

Context
CA1: `(<= x y)`

$\Leftarrow$ { *CA1, IF-true 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

$\Leftarrow$ { `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)
= { evaluate }
(equal 0 nil)
= { evaluate }
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:

Context
A1: `(integerp x)`
A2: `(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))
```

Since we cannot decide the "ifs" in the above formula, we have no choice but to do case-analysis:

By Case analysis we have 2 cases:

Case 1

Extra Context 1

```
CA1: (<= x y)
```

```
⇐ { CA1, IF axiom }
  (equal (- (+ x y)
            y)
         x)
⇐ { Arithmetic, A1, A2 }
  t
```

Case2

Extra Context 2

```
CA2: (not (<= x y))
```

```
⇐ { CA2, IF axiom }
  (equal (- (+ x y)
            x)
         y)
⇐ { Arithmetic, A1, A2 }
  t
```

Introduction to Induction

Which means in the last homework, we needed another lemma which is:

```
(natp (len x))
```

Let's try to start proving it:

```
⇐ { def. len }
  (natp (if (endp x)
        0
        (+ 1 (len (cdr x)))))
```

Lets try to do Case Analysis, since we have no way of "deciding" the if in the body of len:

Case 1 (assume (endp x))

```
CA1: (endp x)
```

```
⇐ { def. len, CA1 }
  (natp 0)
⇐ { Evaluation }
  t
```

Case 2 (assume (not (endp x)))

```
CA2: (not (endp x))
```

```
⇐ { def. len, CA2 }
  (natp (+ 1 (len (cdr x))))
```

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 `(natp (len (cdr x)))` then we could finish the proof by arithmetic reasoning.

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

Simple Mathematical Induction Recap

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)?

$$\begin{aligned} f(3) &= 3 + f(2) \\ &= 3 + 2 + f(1) \\ &= 3 + 2 + 1 + f(0) \\ &= 3 + 2 + 1 + 0 \end{aligned} \tag{1}$$

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:

$$f(x) = \frac{x(x+1)}{2}$$

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 is:

Base Case Prove that it holds for 0

Induction Step Prove that it holds for x **under the assumption** that it holds for x - 1 .

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.

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,

$$\begin{aligned} f(0) &= \frac{0(0+1)}{2} \\ &= 0 \end{aligned}$$

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

$$f(x-1) = \frac{(x-1)x}{2}$$

and use that to prove

$$f(x) = \frac{x(x+1)}{2}$$

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

$$\begin{aligned}
f(x) &= x + f(x-1) \\
&= x + \frac{(x-1)x}{2} \\
&= x + \frac{x^2}{2} - \frac{x}{2} \\
&= \frac{x}{2} + \frac{x^2}{2} \\
&= \frac{(x+x^2)}{2} \\
&= \frac{x(x+1)}{2}
\end{aligned}$$

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

$$f(x) = \frac{x(x+1)}{2}$$

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

Introduction to Induction in ACL2

ACL2 also has induction. We can use induction on natural numbers, but we can also use induction on true-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
  - ...
  -
  -
  - and so on
```

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

```
  (implies (endp x)
           (natp (len x)))
```

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

```
  (implies (and (consp x)
                (natp (len (cdr x)))
           (natp (len x))))
```

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:

```
  (natp (len x))
```

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

Note that the induction scheme follows the definition of a true-list:

```
(defun tlp (x)
 (if (endp x)
     (equal x nil) ;BASE CASE
   (tlp (cdr x)))) ;INDUCTION STEP
```

Lets go into more detail.

Suppose we want to prove

```
(true-listp (app x nil))
```

The problem we run into if we try to prove this directly is that we can never cover all the cases of all possible values of x. We would prove it for `(endp x)`, `(endp (cdr x))`, `(endp (cdr (cdr x)))`, etc. and never finish. Induction to the rescue! It allows us to prove more interesting properties about recursive functions.

As we saw before, we can deduce (prove) by induction the conjecture above by proving the **base case**:

```
(implies (endp x)
         (true-listp (app x nil)))
```

and the **induction step**:

```
(implies (and (not (endp x))
              (true-listp (app (cdr x) nil)))
         (true-listp (app x nil)))
```

The reason this is sufficient to conclude the original conjecture is that we could take any element of the ACL2 universe and either the base case is applicable, or some number of applications of the induction step reduce it to the base case, which is true.

Does this idea sound familiarthat no matter which element of the ACL2 universe is given, a finite number of applications will eventually get to a base case? That's right: *induction is tightly linked to termination of functions* – especially in ACL2.

Example

Example Let's consider an example:

```
(defun fact (x)
  (if (zp x)
    1
    (* x (fact (- x 1)))))
```

Let say we want to prove `(>= (fact x) 1)`. Here are the proof obligations of the **induction scheme** given by `fact` for this formula:

```
(>= (fact x) 1)
⇐ { Induction based on fact }
(and (implies (zp x) (>= (fact x) 1))
     (implies (and (not (zp x))
                   (>= (fact (- x 1)) 1))
              (>= (fact x) 1)))
```

Because that formula is an AND of two formulas, if we prove both of those independently, we can use propositional deduction to conclude the AND of those two. Basically, to prove `(>= (fact x) 1)` by induction based on `fact`, you can prove two formulas independently and consider yourself finished:
base case: `(implies (zp x) (>= (fact x) 1))`

inductive step:

```
(implies (and (not (zp x))
              (>= (fact (- x 1)) 1))
         (>= (fact x) 1)))
```

Suppose I try to come up with a value for which `(>= (fact x) 1)` is not true, despite the base case and inductive steps being true. For simple demonstration purposes, let's choose x = 2 and see why it must be true for that value:

```
(>= (fact 2) 1)
⇐ { Inductive step }
(and (not (zp 2))
     (>= (fact (- 2 1)) 1))
⇐ { Evaluation, Prop. ded. }
(>= (fact 1) 1)
```

⇐ { *Inductive step* }
`(and (not (zp 1))`
`     (>= (fact (- 1 1)) 1))`
⇐ { *Evaluation, Prop. ded.* }
`(>= (fact 0) 1)`
⇐ { *Base case* }
`(zp 0)`
⇐ { *Eval* }
`t`

Next time we will see why termination is so important in ACL2, and how it gives rise to both the definitional principle and the induction principle of ACL2. I will also make the definitions more formal.