CSU290 Lecture Notes Lecture 16 23 Oct 2008

Peter Dillinger

Today, I will introduce a higher-level proof notation so that we don't get bogged down in the details of complete formal proofs, but we should be sure that each step we make could be filled in with a formal proof.

We will consider an example problem and use this definition

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

which introduces the axiom

```
  (equal (app x y)
         (if (endp x)
           y
           (cons (car x)
                 (app (cdr x) y))))
```

We will also use these axioms:

```
; Consp-nil
(equal (consp nil)
       nil)

; Consp-cons
(equal (consp (cons a b))
       t)

; Car-cons
(equal (car (cons a b))
       a)

; Cdr-cons
(equal (cdr (cons a b))
       b)

; If-true
(implies x
         (equal (if x y z)
                y))

; If-false
(equal (if nil y z)
       z)
```

Prove

```
  (equal (app (cons a b) c)
         (cons a (app b c)))
```

We will prove this by working backwards--starting with the original conjecture and iteratively reduce the conjecture to simpler and simpler formulas. When I write

<=

I am indicating that we can conclude the top formula from the bottom one by some axioms and rules of inference. I describe with ones I used in the curly braces. However, instantiation and equals for equals are going to be used so often, that they don't really need to be mentioned, except when the particular instantiation might be unclear.

```
  (equal (app (cons a b) c)
         (cons a (app b c)))
<= { Defn of APP }
  (equal (if (endp (cons a b))
             c
             (cons (car (cons a b))
                   (app (cdr (cons a b)) c)))
         (cons a (app b c)))
<= { Defn of ENDP, Car-cons, Cdr-cons }
  (equal (if (not (consp (cons a b)))
             c
             (cons a (app b c)))
         (cons a (app b c)))
<= { Consp-cons }
  (equal (if (not t)
             c
             (cons a (app b c)))
         (cons a (app b c)))
<= { Evaluate (not t) }
  (equal (if nil
             c
             (cons a (app b c)))
         (cons a (app b c)))
<= { If axiom }
  (equal (cons a (app b c))
         (cons a (app b c)))
<= { Equality axiom }
  t
```

So now we have shown that in the current theory, T implies our original conjecture. Recall from boolean logic that true -> p is the same as p. Thus, if true -> p is true, p is true.

Note that in there, I appealed to a new rule of inference we can use: Execution (or Evaluation). Suppose we want to prove that 1999 is a prime number. If we have defined a predicate for prime numbers, PRIMEP, that would be

```
  (primep 1999)
```

But this conjecture has no free variables, we could determine whether it is always true or not by just evaluating it. In fact, if we give ACL2

```
  (thm (primep 1999))
```

then it will prove it by evaluation. So if we encounter part of a formula with no free variables, we can evaluate it to get it in its simplest form. For example, we can evaluate (not t) to get nil.

It turns out we can adopt another strategy to prove the same theorem above,

```
  (equal (app (cons a b) c)
         (cons a (app b c)))
```

by utilizing something we know about equality. ACL2 has axioms

```
  (implies (and (equal a b)          (Transitivity of equality)
                (equal b c))
           (equal a c))
```

```
  (implies (equal a b)               (Symmetry of equality)
           (equal b a))
```

So in fact, if we show that (app (cons a b) c) is equal something, which is equal to something, ..., which is equal to (cons a (app b c)), then we know that (app (cons a b) c) equals (cons a (app b c)).

Here's how that proof would go:

```
   (app (cons a b) c)
= { Defn of APP }
   (if (endp (cons a b))
     c
     (cons (car (cons a b))
           (app (cdr (cons a b)) c)))
= { Defn of ENDP, Car-cons, Cdr-cons }
   (if (not (consp (cons a b)))
     c
     (cons a (app b c)))
= { Consp-cons }
   (if (not t)
     c
     (cons a (app b c)))
= { Evaluate (not t) }
   (if nil
     c
     (cons a (app b c)))
= { If axiom }
    (cons a (app b c))
```

That allows us to conclude (app (cons a b) c) equals (cons a (app b c)).

Now let us prove

```
  (implies (endp x)
           (equal (app (app x y) z)
                  (app x (app y z))))
```

You might recognize the

```
  (equal (app (app x y) z)
         (app x (app y z)))
```

part as an associativity property. So we are proving a special case of the associativity of APP.

Notice we are proving an equality, but an equality that is within an implication (implies). The way we will typically prove an implies is to make the hypotheses of the implication assumptions and use those in proving the conclusion. Since in this case the conclusion is an equality, we will follow our method of proving equalities. But first, let's write our assumptions:

```
 Assumptions:   (endp x)
```

Then we would start by figuring out what

```
 (app (app x y) z)
```

is equal to, etc. But let's talk a little more about assumptions & lemmas first.

Lemma: a theorem (which might be an axiom) that is used in another proof.

Assumption: a formula that we assume is true in proving something concerning the same variables.

The key difference is that when an assumption refers to a variable such as x, that must refer to the same x as in the formula we're proving--if it has an x. For example, in the proof we're about to work on, the (endp x) assumption is clearly not a theorem--because it's not true for all x--but we can use it as an assumption in showing

```
(equal (app (app x y) z)
       (app x (app y z)))
```

so that we can overall conclude that that formula is true under the assumption:

```
(implies (endp x)
         (equal (app (app x y) z)
                (app x (app y z))))
```

On the other hand, a lemma such as

```
(equal (endp x)
       (not (consp x)))
```

holds for all x. Thus,

THE PRACTICAL DIFFERENCE BETWEEN ASSUMPTIONS AND LEMMAS IS THAT WE CAN INSTANTIATE LEMMAS BUT NOT ASSUMPTIONS.

So we can use an instantiation of the ENDP definitions, such as

```
(equal (endp (cons a b))
       (not (consp (cons a b))))
```

but we cannot instantiate the assumption (endp x) to conclude

```
(endp (cons a b))
```

because that would allow us to conclude NIL, which (if you know your boolean logic well enough) would allow us to use propositional deduction to conclude anything! (false -> anything is true. Thus, it's valid to deduce anything from NIL.)

Let's end the digression and get back to solving the problem:

Assumptions: (endp x)

```
  (app (app x y) z)
= { Defn of APP }
  (app (if (endp x)
           y
           (cons (car x) (app (cdr x) y)))
       z)
= { Assumption (endp x) & IF axiom }
  (app y z)
```

At this point, there doesn't seem to be much to do, because if we open up the definition of this APP, we would need to know something about y to make progress. It turns out it's possible for us to complete the proof from here, but it would involve going from something simple to something more complicated, which is unnatural. Let us now work from the other side of the conjectured equality, with the same assumption:

```
   (app x (app y z))
= { Defn of APP }     ; Be sure you use the right instantiation!
   (if (endp x)
     (app y z)
     (cons (car x)
           (app (cdr x) (app y z))))
= { Assumption (endp x) & IF axiom }
   (app y z)
```

Ok. To complete the proof, I will connect the dots:

```
(implies (endp x)
         (and (equal (app (app x y) z)          ; first equality proof above
                     (app y z))
              (equal (app x (app y z))          ; second equality proof above
                     (app y z))))
=> { Equality }
(implies (endp x)
         (equal (app (app x y) z)
                (app x (app y z))))
```

==========

Now for another proof:

```
(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))))
```

Assumptions:
```
(consp x)
(equal (app (app (cdr x) y) z)
       (app (cdr x) (app y z))))
```

Start with left side of equality:

```
   (app (app x y) z)
= { Defn APP }
   (app (if (endp x)
            y
            (cons (car x)
                  (app (cdr x) y)))
        z)
= { Defn endp }
   (app (if (not (consp x))
            y
            (cons (car x)
                  (app (cdr x) y)))
        z)
= { Assumption (consp x), Not, IF axiom }
   (app (cons (car x)
              (app (cdr x) y))
        z)
```

Now we will leave that as is and work on the right side of the equality:

```
   (app x (app y z))
= { Defn APP }
   (if (endp x)
     (app y z)
```

```
     (cons (car x)
           (app (cdr x) (app y z))))
= { Defn endp }
   (if (not (consp x))
     (app y z)
     (cons (car x)
           (app (cdr x) (app y z))))
= { Assumption (consp x), Not, IF axiom }
   (cons (car x)
         (app (cdr x) (app y z)))
= { Assumption (equal (app (app ...)) (app (cdr ...))) }
   (cons (car x)
         (app (app (cdr x) y) z))
```

Now how to we show that

```
   (app (cons (car x)
              (app (cdr x) y)) z)
```

is equal to

```
   (cons (car x)
         (app (app (cdr x) y) z))
```

thus making (app x (app y z)) equal to (app (app x y) z) under the assumptions?

Do you recall us proving that

```
  (equal (app (cons a b) c)
         (cons a (app b c)))
```

? Let us call this theorem Lemma 1. We can use an instance of this to finish:

```
   (cons (car x)
         (app (app (cdr x) y) z))
= { Lemma 1 }
   (app (cons (car x)
              (app (cdr x) y)) z)
```

Now under the assumptions, we have shown

```
(equal (app (app x y) z)
       (app (cons (car x)
                  (app (cdr x) y)) z))
```

and

```
(equal (app x (app y z))
       (app (cons (car x)
                  (app (cdr x) y)) z))
```

Therefore,

```
(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))))
```