Logic and Computation	Lecture 17
CSU 290 Spring 2009 (Pucella)	Wednesday, Feb 25, 2009

Induction

A few weeks ago, we saw that `(app NIL y)` was always a true list, at least when `y` was itself a true list. In other words, we could prove, quite easily, that `(true-listp y)` $\Longrightarrow$ `(true-listp (app NIL y))`.

What about `(app x NIL)`, is that always a true list? By inspect and informally thinking about it, you should be able to convince yourself that `(true-listp (app x NIL))` is valid, that is, true for all choices of `x`. But can we prove it using what we've seen until now?

Suppose we wanted to prove `(true-listp (app x NIL))`. How would we go about it? Unfortunately, we don't know anything about `x`, so expanding `app` or every `true-listp` does not lead to anything we can really simplify. The one thing we learned that may be useful here is case analysis; we could do a case analysis on whether `(endp x)` is true or not.

We end up with the following two formulas to prove: `(endp x)` $\Longrightarrow$ `(true-listp (app x NIL))` and $\neg$`(endp x)` $\Longrightarrow$ `(true-listp (app x NIL))`. The first one is easy to prove (I'm not even going to bother isolating the context):

`(endp x)` $\Longrightarrow$ `(true-listp (app x NIL))`

by definition of ***app****,* ***(endp x)****, and if axiom*

`(endp x)` $\Longrightarrow$ `(true-listp NIL)`

by definition of ***true-listp***

`(endp x)` $\Longrightarrow$ `(if (endp NIL) (= NIL NIL) (true-listp (cdr NIL)))`

by endp axiom

`(endp x)` $\Longrightarrow$ `(= NIL NIL)`

by reflexivity of `=` *and propositional reasoning*

Easy. The second one is more interesting:

$\neg$`(endp x)` $\Longrightarrow$ `(true-listp (app x NIL))`

by definition of ***app****, assumption, and if axiom*

$\neg$`(endp x)` $\Longrightarrow$ `(true-listp (cons (car x) (app (cdr x) NIL)))`

by definition of ***true-listp***

```
¬(endp x) ⟹ (if (endp (cons (car x) (app (cdr x) NIL)))
                (= (cons (car x) (app (cdr x) NIL)) NIL)
                (true-listp (cdr (cons (car x) (app (cdr x) NIL)))))
```

by endp axiom, if axiom, and cdr axiom

```
¬(endp x) ⟹ (true-listp (app (cdr x) NIL))
```

We're kind of stuck here, since we don't know anything about `(cdr x)`. No problem, you say—let's do the same thing we did at the beginning. Let's do a case analysis on whether `(endp (cdr x))` is true or not. Try it. You'll see that you can probe the first case easily, and the second case will have you end up trying to establish that `(true-listp (app (cdr (cdr x)) NIL))`.

Do you notice a pattern here? The proof by case analysis ends up looking very much like trying to unwind the recursion! Now, and this is where termination comes in to help us, because `app` is admissible, we know that it terminates on all inputs, meaning, in particular that we know that this unwinding will eventually stop and we'll hit the base case where `endp` is true. But, and this is a bit of a problem, we cannot tell how deep we can go—after all, that depends on `x`, and we know nothing about `x`.

To help us here, we need a new proof rule, that resembles case analysis, but says something more. That proof rule is the induction rule (here, for true lists). Here is what it looks like:

> **Induction Rule (for true lists)**: If F is a formula, `x` a variable in F whose intended domain is true lists, and σ is a substitution that, among others, sends variable `x` to `(cdr x)`, then we can rewrite a formula F into
>
> $$\texttt{(endp x)} \implies F \quad \wedge \quad (\neg\texttt{(endp x)} \wedge F/\sigma) \implies F$$
>
> where F/σ is instance of formula F obtained by performing the substitution σ.

The two formulas `(endp x)` $\implies F$ and $(\neg$`(endp x)` $\wedge F/\sigma) \implies F$ are sometimes called the *proof obligations* of the induction.

Thus, for instance, consider the formula `(true-listp (app x NIL))`.

Theorem 1. `(true-listp (app x NIL))`

Proof. The induction rule (on variable `x`) gives us the following proof obligations:

$$\texttt{(endp x)} \implies \texttt{(true-listp (app x NIL))}$$

and

$$\neg\texttt{(endp x)} \wedge \texttt{(true-listp (app (cdr x) NIL))} \implies \texttt{(true-listp (app x NIL))}.$$

We already prove the first one above, and the second one is not that much harder now.

To prove it, let $C = \neg$`(endp x)` $\wedge$ `(true-listp (app (cdr x) NIL))` be the context, where A1 is $\neg$`(endp x)` and A2 is `(true-listp (app (cdr x) NIL))`:

$C \Longrightarrow$ `(true-listp (app x NIL))`

by definition of ***app****, A1, and if axiom*

$C \Longrightarrow$ `(true-listp (cons (car x) (app (cdr x) NIL)))`

by definition of ***true-listp****, endp axiom, if axiom*

$C \Longrightarrow$ `(true-listp (cdr (cons (car x) (app (cdr x) NIL))))`

by cdr axiom

$C \Longrightarrow$ `(true-listp (app (cdr x) NIL))`

by A2

□

Intuitively, this is what the induction rule for true lists says: in order to prove a formula F where a variable `x` of the formula ranges over true lists, it suffices to prove F in the case where `x` is an `endp`, to prove F in the case where `x` is not an `endp`, and where additionally we know that formula F holds for the rest of `x`.

(This is actually very similar to the argument that you use to come up with a recursive function. Remember why recursion works, for true lists: you write a recursive function by giving a base case in which the function immediately returns a result, and then you write the recursive case *assuming that your function works correctly for the* ***cdr*** *of the input.* This works when the functions terminates. The exact same thing happens for induction: to prove F, we prove that it is true for the base case of the input, and then we prove it is true for the inductive case of the input, where we *assume that the formula is true for the* ***cdr*** *of the input.* Recursion and induction go hand in hand, as we'll see in more detail in the next few weeks.)

The main difficulty, when applying the induction rule, is to determine which variable to do induction on. In fact, that induction goes hand in hand with recursion is the key here: to be useful, induction should be over a variable that controls the recursion in one of the function in the formula. For instance, consider the following theorem.

Theorem 2. `(= (rev (app y z)) (app (rev z) (rev y)))`

Proof. To prove this, we pretty much have to use induction because no theorem applies here, and we don't know anything about either `y` or `z`. But induction on what? Looking at the first `app`, we see that `y` is the variable driving the recursion, meaning that `y` is a good

candidate to do induction on. Here are the proof obligations you get for that application of the induction rule:

$$\texttt{(endp y)} \Longrightarrow \texttt{(= (rev (app y z)) (app (rev z) (rev y)))}$$

and

$$\neg\texttt{(endp y)} \wedge \texttt{((= (rev (app (cdr y) z)) (app (rev z) (rev (cdr y)))))} \Longrightarrow \texttt{(= (rev (app y z)) (app (rev z) (rev y)))}.$$

Ans the proof of those proof obligations you had to come up with for the midterm. □

Finally, a more complicated formula is associativity of `app`.

Theorem 3. `(= (app a (app b c)) (app (app a b) c))`

Proof. Again, induction, because no theorem really applies and we don't know anything about `a`, `b`, or `c`. So which variable do we do induction on? `a` is the only variable that appears on both sides of the `=` in a position where it controls the recursion, so it seems like a good candidate. The proof obligations we get are:

$$\texttt{(endp a)} \Longrightarrow \texttt{(= (app a (app b c)) (app (app a b) c))}$$

and

$$\neg\texttt{(endp a)} \wedge \texttt{(= (app (cdr a) (app b c)) (app (app (cdr a) b) c))} \Longrightarrow \texttt{(= (app a (app b c)) (app (app a b) c))}$$

and we already proved those before—Theorems 2 and 3 in Lecture 14. □

Note that we have applied induction to formulas of the form `exp` only, including `(= exp exp')`. What about implications, though? For next lecture, consider the proof obligations you get for the formula `(true-listp x)` $\Longrightarrow$ `(= (app x NIL) x)`, and see whether you can "massage" them into a form in which we can prove them.

Aside: Justifying Induction

One question you may have is why the induction rule actually makes sense. It looks a bit like magic. We get to assume stuff about what we want to prove. Compare to case analysis: I justified the "case analysis" step in a proof by showing that it corresponds to a four steps of propositional reasoning—in other words, we have case analysis as soon as we have propositional reasoning, it's just not obvious. Case analysis doesn't require any new idea. Induction, however, *cannot* be justified by showing that it's just propositional

reasoning in disguise. There is genuinely something new there. So how can we justify it is a good rule?

The proof is in the pudding. Recall that the main reason why we want to prove theorems is because a proof of F tells us that F is valid, and validity is what we really care about, since validity tells us something about how a formula behaves when we plug in real values.

So, our induction rule says that to prove F by induction, it suffices to prove the proof obligations obtained form the induction rule. Thus, to justify the induction rule, we need to argue that if indeed we prove the proof obligations (i.e., the proofs obligations are valid), then the original formula F is also valid.

Let's argue this for a specific example. (The general case is similar, just more cumbersome to write down.) Consider `(true-listp (app x NIL))`, the formula we started with.

I claim that if

$$\texttt{(endp x)} \implies \texttt{(true-listp (app x NIL))}$$

and

$$(\neg\texttt{(endp x)} \wedge \texttt{(true-listp (app (cdr x) NIL))} \implies \texttt{(true-listp (app x NIL))}$$

are valid, then `(true-listp (app x NIL))` itself is valid.

Here is the argument. First, recall that every ACL2 value is either an atom, or a cons-structure. Define the *size* of a value to 0 for an atom, or the length of the spine for a cons-structure. I want to show that `(true-listp (app x NIL))` is valid, that is, it is true no matter what value we put in for `x`.

Let A be the set of all values of `x` that make the formula `(true-listp (app x NIL))` false, that is,

$$A = \{a \mid \texttt{(true-listp (app } a \texttt{ NIL))} \text{ is false}\}$$

where a ranges over ACL2 values. Our claim, that `(true-listp (app x NIL))` is valid, corresponds to A being empty. (Why?)

To argue that A must indeed be empty, we go by contradiction, a common technique. We assume that A is not empty, and show that this leads to something absurb, meaning that that assumption was wrong, and that A must be empty, that is, `(true-listp (app x NIL))` is valid.

Therefore, assume that A is not empty. Because every value has a size, there is some value in A with minimal size in A. (There may be more than one, of course.) That is, there is some value a_m in A such that no other values in A have a smaller size. Note that the value a_m cannot be an atom, because one of our assumptions says `(endp x)` $\implies$ `(true-listp (app x NIL))` is valid, which says that every atom makes the formula true, meaning that no atom is in A. So the value a_m is not an atom, but it's a cons-structure, of some size $\mathit{size}(a_m) \geq 1$.

Okay, so because a_m is in A, we know that `(true-listp (app` a_m `NIL))` is false. Using the definition of `app`, and the fact that a_m is not an atom, meaning that `(endp` a_m`)` is

false, we get that `(true-listp (cons (car` a_m`) (app (cdr` a_m`) NIL)))` is false. Using the definition of `true-listp` and simplifying, we get that `(true-listp (app (cdr` a_m`) NIL))` is false. Ah, but then look what happens: this says that `(cdr` a_m`)` is a value in A, right? And moreover, by the way we defined size, we know that the size of `(cdr` a_m`)` is strictly less than the size of a_m—so we have an element of A, namely, `(cdr` a_m`)` of size strictly less than a_m, but then we *expressely* chose a_m in A with the property that no other value in A had smaller size. This is our contradiction. Thus, our initial assumption, that A is not empty, was wrong, and A must be empty. That is, `(true-listp (app x NIL))` must be valid.