Logic and Computation	**Lecture 18**
CSU 290 Spring 2009 (Pucella)	**Thursday, Feb 26, 2009**

First off, let me add a *coda* to the last lecture. Recall that we looked at a new proof rule, the induction rule for true lists, that let us take a formula and rewrite it into two formulas (the proof obligations of the induction), the first of which corresponding, intuitively, to the base case of the recursion, and the other corresponding to the recursive case of the recursion.

Last time we applied induction to formulas of the form `exp` only, including `(= exp exp')`. What about implications, though?

Consider the (valid) formula `(true-listp x)` $\Longrightarrow$ `(= (app x NIL) x)`. We don't know enough about `x` to push a proof through (someone, knowing that `x` is a true list doesn't provide much information to go on on) If we naively apply the induction rule, we get the following as a first proof obligation:

$$\texttt{(endp x)} \Longrightarrow \texttt{(true-listp x)} \Longrightarrow \texttt{(= (app x NIL) x)}.$$

This is not a great form to work with, but thankfully, we can use propositional reasoning to yield the more useful:

$$\texttt{(endp x)} \wedge \texttt{(true-listp x)} \Longrightarrow \texttt{(= (app x NIL) x)}.$$

This is easy to prove, so I'll let you prove it.

Similar deal with the second proof obligation. From the induction principle, we get

$$\begin{array}{l}\neg\texttt{(endp x)} \wedge (\texttt{(true-listp (cdr x))} \Longrightarrow \texttt{(= (app (cdr x) NIL) (cdr x))}) \\ \quad \Longrightarrow \texttt{(true-listp x)} \Longrightarrow \texttt{(= (app x NIL) x)}.\end{array}$$

Again, some propositional reasoning brings the above to the more reasonable

$$\begin{array}{l}\neg\texttt{(endp x)}\wedge \\ (\texttt{(true-listp (cdr x))} \Longrightarrow \texttt{(= (app (cdr x) NIL) (cdr x))}) \wedge \\ \texttt{(true-listp x)} \\ \quad \Longrightarrow \texttt{(= (app x NIL) x)}.\end{array}$$

Let's prove this one. First, let C be the context in the above formula, which contains three formulas:

A1 is $\neg$`(endp x)`

A2 is `(true-listp (cdr x))` $\Longrightarrow$ `(= (app (cdr x) NIL) (cdr x))`

A3 is `(true-listp x)`

And using this context we need to prove: (I'm leaving the $C \Longrightarrow$ part implicit...)

```
(= (app x NIL) x)
```

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

```
(= (cons (car x) (app (cdr x) NIL)) x)
```

...

We're sort of stuck here, as we would like to replace `(app (cdr x) NIL)` by `(cdr x)`, but A2 tells us that we can do that only if we know that `(true-listp (cdr x))` is true. We don't know yet that it is true; what we do know is that `(true-listp x)` is true, that's A3. But, and this is the key point, we can reason about what's in the context, and A1 and A3 together give us something interesting. In fact, it is easy to prove that `(endp x)` $\Longrightarrow$ `(= (true-listp x) (true-listp (cdr x)))`:

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

by definition of **true-listp**, *the assumption, and if axiom*

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

by reflexivity of `=` *and propositional reasoning*

Cool, so in the context, we have A1, so by Modus Ponens, we can add

A4 is `(= (true-listp x) (true-listp (cdr x)))`

to the context. Thus, we can substitute in the context, and get the slight variant

A5 is `(true-listp x)` $\Longrightarrow$ `(= (app (cdr x) NIL) (cdr x))`

in the context. And this one now we can use. So, continuing the original proof where we left off:

```
(= (cons (car x) (app (cdr x) NIL)) x)
```

by A5, because A3 is true

```
(= (cons (car x) (cdr x)) x)
```

by cons axiom, because A1 is true

```
(= x x)
```

by reflexivity of `=`

You can just learn the general shape of the proof obligations when the formula to be proved using the induction rule for true lists is of the form $C \Longrightarrow F$:

`(endp x)` $\wedge\, C \quad \Longrightarrow \quad F$

and

$$\neg\texttt{(endp x)} \wedge [(C \Longrightarrow F)/\sigma] \wedge C \quad \Longrightarrow \quad F.$$

Induction for Other Data Definitions

I argued last time that there was a relationship between induction and recursion, that one is in some sense the other side of the coin of the other.

In fact, the relation is deeper than that, and reveals interesting connections, which is useful to understand how to do induction over other forms of data.

From 211, you know that (recursive) data definitions and recursive functions that use those kind of data go hand in hand. In fact, the data definition gives you a design recipe for recursive functions, which is also sometimes called a *recursion scheme*. In fact, you can see the connection in another way. If I ask you to write down an ACL2 function that recognizes exactly those values that satisfy the data definition, than that recognizer uses the recursion scheme. Think of `true-listp`, which is a recognizer for true lists, and uses a recursion scheme which is just the good old design recipe for the data definition of true lists.

So we have the following relationship

```
          Data Definition
            /
Recursion Scheme
```

Think about true lists. The recursion scheme essentially gives you the induction rule, in the sense that it gives you the two proof obligations. The recursion scheme for true lists is:

```
(defun true-list-rec-scheme (x)
  (if (endp x)
      ...
      (... (true-list-rec-scheme (cdr x)) ...)))
```

The base case of the scheme gives you the proof rule of the base case `(endp x)` $\Longrightarrow F$, and the recursive case of the scheme gives you the proof rule for the inductive case, where the corresponding of the recursive call is the inductive hypothesis, that talks about the formula F but at `(cdr x)` instead of `x`—just like the recursive call talks about `(cdr x)` instead of `x`: $\neg$`(endp x)` $\wedge F/\sigma \Longrightarrow F$, where σ sends `x` to `(cdr x)`.

In fact, the relation forms one of the core trinities of Computer Science:

```
          Data Definition
            /        \
Recursion Scheme  --  Induction Rule
```

So, just like you have an induction rule for true lists, which corresponds to the recursion scheme for true lists, we have induction rules for other data definitions, such as trees, and the likes. Let's look at another data type for which we have a recursion scheme, namely natural numbers.

The recursion scheme (design recipe) for recursion over the natural numbers is the following one:

```
(defun nat-rec-scheme (n)
  (if (zp n)
      (... (nat-rec-scheme (- n 1)) ...)))
```

Which leads to the following:

> **Induction Rule (for natural numbers)**: If F is a formula, `n` a variable in F whose intended domain is natural numbers, and σ is a substitution that, among others, sends variable `n` to `(- n 1)`, then we can rewrite a formula F into
>
> $$\texttt{(zp n)} \Longrightarrow F \quad \wedge \quad (\neg\texttt{(zp n)} \wedge F/\sigma) \Longrightarrow F$$
>
> where F/σ is instance of formula F obtained by performing substitution σ.

Let's put this to use and prove a formula that I mentioned way at the beginning of the course, but never could prove. Recall the definition of factorial:

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

Theorem 1. `(>= (fact n) 0)`

Proof. We don't know anything about `n` except that it's intended to be a natural number, so let's do natural-number induction on `n`. We get the following two proof obligations:

$$\texttt{(zp n)} \Longrightarrow \texttt{(>= (fact n) 0)}$$

and

$$\neg\texttt{(zp n)} \wedge \texttt{(>= (fact (- n 1)) 0)} \quad \Longrightarrow \quad \texttt{(>= (fact n) 0)}.$$

The first proof obligation is easy to take care of:

`(zp n)` $\Longrightarrow$ `(>= (fact n) 0)`

by definition of **`fact`***, assumption, and if axiom*

`(zp n)` $\Longrightarrow$ `(>= 0 0)`

by arithmetic and propositional reasoning

The second proof obligation is not much harder. First, let's identify the context $C =$ `¬(zp n)` $\wedge$ `(>= (fact (- n 1)) 0)` and name the assumptions:

A1 = `¬(zp n)`

A2 = `(>= (fact (- n 1)) 0)`

and prove what we need to prove. First, note that from A1 we can derive that `(>= n 0)`.[1]

$C \Longrightarrow$ `(>= (fact n) 0)`

by definition of **`fact`***, A1, and if axiom*

$C \Longrightarrow$ `(>= (* n (fact (- n 1))) 0)`

by A2, arithmetic (if $r \geq 0, s \geq 0$ then $rs \geq 0$) and propositional reasoning

□

A different but still easy example:

```
(defun foo (n)
  (if (zp n)
      1
      (- (* 2 (foo (- n 1))) 1)))
```

Prove that `(<= (foo n) (expt 2 n))`, where `(expt a b)` computes a^b.

If you feel idle and unchallenge, determine if `(>= n 1)` $\Longrightarrow$ `(>= (foo n) (expt 2 (- n 1)))` is valid. If you think it is, give a proof (by induction on `n`), otherwise, can you modify the formula so that it is valid, and then prove it?

[1] To do this we need to know more about `zp` and its definition. For now, let's just assume that the formula `¬(zp n)` $\Longrightarrow$ `(>= n 0)` is valid. (It is.)