

1 Announcements

2 Review

Last time: talked about definitions and their relation to the logic. For example, if we allow the “definition”

```
(defun f(x)
  (+ 1 (f x)))
```

then the ACL2 logic becomes unsound. And, we saw that even non-recursive functions could cause problems. For example:

```
(defun f (x) y)
```

Also, renders the logic unsound. So, ACL2 has a mechanism for making sure that “definitions” are really definitions.

3 Today: The Definitional Principle

Definitional Principle:

The definition

```
(defun f (x1 ... xn)
  body)
```

is *admissible* provided:

1. **f** is a new function symbol, *i.e.*, there are no other axioms about it; (note that this happens in the context of a history)
(BTW, why do we need this? Well, what if we already defined **app**? Then we would have two definitions. What about redefining? We may already have theorems proven about **app**. We’ve have to throw them out. ACL2s allows you to undo, but not redefine.)
2. The x_i are distinct variable symbols;
(Why? If the variables are the same, say `(defun f (x x) body)` then what is `(f 1 2)`? ACL2 is total.)
3. *body* is a term, possibly using **f** recursively as a function symbol, mentioning no variables freely other than the x_i ; and
(Why? Well, we already saw that `(defun f(x) y)` can lead to unsoundness. *body* is a term means that it is a legal expression in the current history)

4. The function is terminating.

(Why? We saw that nontermination can lead to unsoundness.)

If admissible, the logical effect of the definition is to add a new axiom:

Definitional Axiom for f : $(f\ x_1 \dots x_n) = \text{body}$.

Examples:

```
(defun true-listp (x)
  (if (endp x)
      (equal x nil)
      (true-listp (cdr x))))
```

Is `true-listp` admissible? Yes

```
(defun rev (x)
  (if (endp x)
      nil
      (app (rev (cdr x)) (list (car x)))))
```

Is `rev` admissible? It depends if we defined `app` already.

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

After defining `app`, which is admissible, we can admit `rev`.

Now, *why* do the above terminate?

Well, think of it this way. Imagine partitioning the ACL2 universe into sections (sets) as follows:

1. atoms
2. conses of length 1
3. conses of length 2
4. conses of length 3
5. ...

How many sections are there? An infinite number. Coverage: does every element in the ACL2 universe fit in some section? Yes.

Since every element is in some section, if we call the function `true-listp` on that element, then either we are done (when it is an atom) or it will lead to a call of `true-listp` on some element with smaller length, so we get closer to the base case and in a way that guarantees we eventually get there. This is a nice property of natural numbers; we would say that they are *well-founded*.

Here's another way of seeing this, using a proof by contradiction. What does a proof by contradiction look like? We assume that what we are trying to prove, say φ , does not hold and derive a contradiction: $\varphi \Rightarrow \mathbf{false}$. Note that

by propositional logic, this is $\neg \mathbf{false} \Rightarrow \varphi$, which is $\mathbf{true} \Rightarrow \varphi$, which by MP is: φ . Another way to see this is that $\varphi \Rightarrow \mathbf{false}$ is $\neg \varphi \vee \mathbf{false}$, which is φ .

Why do people use proofs by counterexample? It seems like an elaborate way of proving φ . In some sense it is, but it is a nice technique to have in your arsenal because it often helps you focus on the goal: prove false.

OK, here we go. If `true-listp` is not terminating, then there is some set of elements in the ACL2 Universe for which `true-listp` never returns. Call this set A . If we sort elements in A by their length, then the first element, say s , has the smallest length. Let's see what `true-listp` will do to s . Well, if s has length 0, *i.e.*, is an atom, `true-listp` terminates right away, so s is a cons. The length of s , say l , is therefore > 0 ; now `true-listp` calls itself on `(cdr s)`, so since `true-listp` doesn't terminate on s , it doesn't terminate on `(cdr s)` either, so `(cdr s)` is in A , but then it's length is $l - 1$, which is $< l$, which *contradicts* the minimality of s .

So, now we come to *induction*.

If we want to prove a theorem, *e.g.*,

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

we can try equational reasoning.

```
(true-listp (app x nil))
= { Definition of app }
  (true-listp (if (endp x)
                  nil
                  (cons (car x) (app (cdr x) y))))
= { ??? }
  ???
```

Now, we can say, *informally*: well, x has some length, say l , so I can expand `app` that many times, and it would look like this:

```
(true-listp (cons (car x) (cons (cadr x) ... nil)))
```

which is just

```
(true-listp (list (car x) (cadr x) ... ))
```

and `list` always returns a true list.

That's the right idea. You should use your intuitions, but this is not a proof: "... " is too vague and ill-defined for our purposes.

Induction formalizes the above idea. It gives us two cases:

```
(endp x)  $\Rightarrow$   $\varphi$ 
```

and

```
[(consp x)  $\wedge$   $\varphi$ (( x (cdr x)))]  $\Rightarrow$   $\varphi$ 
```

Why is this a proof?

For the "same" reason `true-listp` terminates.

Suppose after proving the above two cases, we still have elements in the ACL2 Universe that don't satisfy φ . Let the set of such elements be A and let s

be an element in A with smallest length. Well, if s has length 0, *i.e.*, is an atom, then we proved directly that φ holds, so s is a cons. The length of s , say l , is therefore > 0 ; now by the definition of `true-listp` s is a true list iff the `(cdr s)` is; hence, since s is not a true list, then neither is `(cdr s)`, so `(cdr s)` is in A , but then it's length is $l - 1$, which is $< l$, which *contradicts* the minimality of s .

How did we come up this? We used the definition of `true-listp`!

So, notice a wonderful connection.

The data-function-induction (DFI) trinity:

1. Data definitions give rise to predicates recognizing such definitions. These predicates must be shown to terminate. (Otherwise they are inadmissible by the Definitional principle.) Their bodies give rise to a *recursion scheme*.
2. Functions over these data types are defined by using the *recursion scheme* as a template.
3. The *Induction Principle*: Proofs by induction involving such functions and data definitions use the same *recursion scheme* to generate proof obligations. Non-recursive cases are proven directly. For each recursive case, we assume the theorem under *any* substitutions that map the formals to arguments in that recursive call.

Next time, examples, and examples, and more examples!