

1 Announcements

Exam tomorrow

2 Review

Last time: equational proofs in ACL2.

For example,

1. $(\text{endp } x) \Rightarrow (\text{len } (\text{app } (x \ y))) = (+ (\text{len } x) (\text{len } y))$
2. $(\text{consp } x) \wedge (\text{len } (\text{app } (\text{cdr } x) \ y)) = (+ (\text{len } (\text{cdr } x)) (\text{len } y))$
 $\Rightarrow (\text{len } (\text{app } (x \ y))) = (+ (\text{len } x) (\text{len } y))$

We can also prove the kinds of theorems you've seen in discrete math:

```
(defun sum (n)
  (if (zp n)
      0
      (+ n (sum (- n 1)))))
```

We can prove that $(\text{sum } n) = n(n+1)/2$, which is formalized as:

```
(implies (natp n)
  (equal (sum n)
    (/ (* n (+ n 1)) 2)))
```

by induction. How? Base case: $(\text{zp } n)$; induction step $\neg(\text{zp } n)$, *i.e.*, n is a natural number > 0 and above holds for $n-1$.

```
(zp n)  $\wedge$  (natp n)
 $\Rightarrow$  (sum n) = (/ (* n n+1) 2)

n>0  $\wedge$  ((natp n-1)  $\Rightarrow$  (sum n-1) = (/ (* n-1 n) 2))  $\wedge$  (natp n)
 $\Rightarrow$  (sum n) = (/ (* n n+1) 2)
```

3 Today: The Definitional Principle

We've already seen that when you define a function, say

```
(defun f(x) body)
```

then ACL2 adds the axiom

```
(f x) = body
```

Today, we'll more carefully examine what happens when you define functions.

First, let's see why we have to examine anything at all.

In 211, and using ACL2s in programming mode, you were allowed to write functions such as the following:

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

This is a nonterminating recursion.

Note that you are not able to write nonterminating functions if you use the basic design recipe. Furthermore, there has been no reason for you to write nonterminating functions in 211 or in this class, but you had the ability to do it and, presumably, a reasonable language would have prevented you from doing so.

In fact, once we leave programming mode, you will not be allowed to write such functions in ACL2s, since function defined in ACL2 *have* to be terminating.

Suppose we add the axiom

```
(f x) = (+ 1 (f x))
```

Then what?

We get a contradiction, since in ACL2s, we can prove $x \neq x + 1$.

```
(thm (not (equal x (1+ x))))
```

That is, in ACL2s, we can prove `nil`. How?

Instantiate $x \neq x + 1$ using substitution $((x (f x)))$. That, with the axiom for `f` gives us `nil`.

What's so bad about that?

Consider `t=nil`; we proved that they are different last time.

Here's a proof of it:

1. `nil` \Rightarrow `t=nil` { propositional logic; false implies anything }
2. `nil` { above theorem }
3. `t=nil` { MP, 1,2 }

So, some nonterminating recursive equations introduce unsoundness. Therefore, ACL2s does not allow you to define nonterminating functions.

Question: does every non-terminating recursive equation introduce unsoundness?

No. Consider $(f x) = (f x)$. You can prove such things in ACL2, since every function satisfies this (it's the Leibniz axiom); you just can't define a function this way.

But, can some terminating recursive equations introduce unsoundness?

Well, yes.

Consider:

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

Now, here is a proof of `nil`:

1. $(f\ 1) = t$ { Instantiation }
2. $(f\ 1) = \text{nil}$ { Instantiation }
3. $t = \text{nil}$ { Equality }
4. nil { Since we proved $t \neq \text{nil}$ }

But this happened because we allowed a “global” variable. It will turn out that we can rule out bad terminating equations with some simple checks.

So, modulo some checks we are going to get to soon, terminating recursive equations do not introduce unsoundness, because we can prove that if a recursive equation can be shown to terminate then there exists a function satisfying the equation.

The above discussion should convince you that we need a mechanism for making sure that when users add axioms to ACL2 by defining functions, then the logic stays sound.

That’s what the *definitional principle* does. We’ll study that next time!