Logic and Computation — **Lecture 16**
CSU 290 Spring 2009 (Pucella) — **Monday, Feb 23, 2009**

First, some practice proof problems. Consider the following definitions:

```
(defun length (L)
   (if (endp L)
       0
       (+ 1 (length (cdr L)))))

(defun add1 (L)
   (if (endp L)
       NIL
       (cons (+ 1 (car L)) (add1 (cdr L)))))

(defun same-lengthp (L M)
   (if (endp L)
       (endp M)
       (if (endp M)
           (endp L)
           (same-lengthp (cdr L) (cdr M)))))
```

Prove the following theorems:

(a) `(endp x)` $\Longrightarrow$ `(= (length (add1 x)) (length x))`

(b) $($$\neg$`(endp x)` $\wedge$ `(= (length (add1 (cdr x))) (length (cdr x)))`$)$
  $\Longrightarrow$ `(= (length (add1 x)) (length x))`

(c) `(endp x)` $\Longrightarrow$ `(same-lengthp (add1 x) x)`

(d) $($$\neg$`(endp x)` $\wedge$ `(same-lengthp (add1 (cdr x)) (cdr x))`$)$
  $\Longrightarrow$ `(same-lengthp (add1 x) x)`

Note that (d) is a *bit* tricky. Not hard. There's just a particular axiom that you need to use that is not completely obvious, although I gave it to you way back when.

Termination and the Definitional Principle

Weve already seen that when you define a function, say

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

then that yields an axiom

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

to the theory, that is, the set of all axioms you can work with.

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

Lets see why we have to examine anything at all.

First, let's see why we harped on termination so much a while back, and consider a function such as

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

This is a nonterminating recursion, clearly. Now consider the corresponding axiom

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

There's a problem with that axiom. If `(= (f x) (+ 1 (f x)))` is provable, then we can derive that `(= 0 1)` is provable. But by the axioms of arithmetic, we know that ¬`(= 0 1)`. Thus, if we allow `(= (f x) (+ 1 (f x)))` as an axiom, we can prove both that `(= 0 1)` and that ¬`(= 0 1)`. In other words, we can prove the formula *false*, or put more shockingly, that *false* $\equiv$ *true*.

So we can prove that *false* $\equiv$ *true*. Why is that bad? Well, aside from the fact that it's clearly not true, once we've proved that, we can prove anything else! Given any formula F, we know from propositional logic that *false* $\Longrightarrow F$ is a valid formula. But then, given that we can prove *false*, then using Modus Ponens we can get F. And this is so *no matter what F is.* Once we can prove *false*, we can prove any formula. A logic that lets us prove *false* is called *unsound*, and any axiom that we add to the logic that lets us prove *false* is an *unsound axiom.* Unsound axioms are bad. An unsound logic

So, nonterminating functions like the one above can lead to unsound definitional axioms, and those are bad.[1]

But problems do not come only from nontermination—some terminating recursive functions also can give you definitional axioms that are unsound. Consider:

[1]Not all nonterminating recursive functions lead to definitional axioms that are unsound. For instance, `(defun f (x) (f x))` gives the axiom `(= (f x) (f x))` which is perfectly fine—indeed, it is just an instance of the Leibniz axiom for `=`.

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

Leading to the axiom `(= (f x) y)`.

This axiom is unsound. Indeed, Note that `(= (f x) 0)` $\wedge$ `(= (f x) 1)` is provable using that axiom (by using two instances of the axioms), and by transitivity of `=`, we can prove `(= 0 1)`, and as we saw above, this leads to being able to prove *false*.

What else can go wrong? Redefining functions can also lead to problems. Suppose we define a function

```
(defun f (x) 0)
```

which leads to the axiom `(= (f x) 0)`. If we later redefine

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

then we get the new axiom `(= (f x) 1)`, and with the old axiom we can derive that `(= 0 1)` from which, once again, we can prove *false*.

So how do we avoid introducing unsound axioms? We do so by restricting the definitions we allow.

Definition. *A 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 in the current history;*

(2) *The* ***xi*** *are distinct variable symbols;*[2]

(3) ***body*** *is a well-formed term—a legal expression using only functions already defined in the current history, including* ***f****—mentioning no variables freely other than the* ***xi****;*

(4) *the function is terminating.*

If a definition `(defun f (x1 ... xn) body)` is admissible, then the logical effect of the definition is to add a new axiom to our theory, called the *definitional axiom* for `f`:

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

So how do you check that a function terminates? First, in ACL2, the only thing we have to worry about is recursion (including, possibly, mutual recursion, that is, two or more functions that recursively call each other). Non-recursive functions always terminate.

[2]Why? If the variables are the same, say `(defun f (x x) body)` then what does `(f 1 2)` mean anyways?

What about recursive functions? As you saw in 211, a recursive function is guaranteed to terminate if: (1) it has a base case that terminates immediately, and (2) every recursive call of the function "makes progress towards the base case". What this means exactly depends on the function at hand, or more precisely, on the kind of data that we are recursing over. In the case of true lists, the base case is often the empty list, and the recursive case, in order to make progress towards termination, should recursively call the function on a shorter list. A third point is ACL2 specific, as we saw at the beginning of the course: (3) the function should terminate no matter what the input is. Happily, if you use the design recipes from 211 adapted to ACL2, as we showed you earlier in the course, then recursive functions always terminate. That's one of the purposes of the design recipe: termination is obvious.

If you have functions that do not exactly follow the design recipe, then the argument for termination can be more subtle. Consider the following ACL2 function:

```
(defun foo (n)
   (cond ((zp n) 0)
         ((= n 1) 1)
         ((evenp n) (foo (/ n 2)))
         ((oddp n) (foo (+ n 1)))))
```

(Assume that we have correct definitions for `evenp` and `oddp`; also, there is no "else" clause to the `cond`, because it is easy to check that any input will make one of the conditions true.) I claim that this terminates for all inputs. Clearly, if the input is not a natural number, it terminates immediately. (Why?) Otherwise, can we argue that we are making progress towards the base cases (either 0 or 1)? For even inputs, the recursive call can easily be seen to make progress towards the base cases. When the input is odd, even though the recursive call uses a bigger number (that therefore does not immediately progresses towards the base case), `(foo (+ n 1))` will be called on an even number, which means that at the following iteration, `foo` will be invoked on `(/ (+ n 1) 2)`, which is smaller than `n`. Thus, even though we are not making progress towards the base case *at every step*, we are making progress towards the base case at either every step or every two steps, depending on whether the input is even or odd. That's actually sufficient to establish termination. (Why?)

However, this doesn't work for every function, even those that look similar to the above. One of the most famous examples is the following so-called Collatz function:

```
(defun collatz (n)
  (cond ((zp n) 0)
        ((= n 1) 1)
        ((evenp n) (collatz (/ n 2)))
        ((oddp n) (collatz (+ (* 3 n) 1)))))
```

It is unknown whether this function terminates on all inputs. Plot out a few calls for small values of n, and you'll get a sense for how long some of the iterations take. Termination has been checked for all natural numbers up to $10^1$6, but for all we know it fails to terminate on some input greater than $10^1$7. We just don't know enough about number theory to say anything else.

The fact that there are functions that we have no idea whether they terminate or not suggests that we do not know how to automatically check for termination. Indeed, it's worse than that. Turing showed that it is impossible to write a program that can take any function and automatically decide (correctly) if it terminates or not. This is a fundamental limitation of computation as we understand it.

The general argument needs more theory than we have, but I can give you a sense for why that is by considering the problem in Scheme. Here is the argument that it is impossible to write a Scheme function that correctly determines whether another Scheme function terminates. We argue by contradiction.

Suppose that we could write a function `(terminates? f x)` that returns true when `(f x)` terminates, and returns false when `(f x)` does not terminate. I don't care how `terminates?` is written—just suppose you managed to write it. I will now show that you get something completely absurd as a result.

In particular, I can now write the following perfectly legal scheme functions:

```
(define (loop-forever) (loop-forever))

(define (HP x)
  (if (terminates? HP x)
      (loop-forever)
      1))
```

My question: does `(HP 0)` terminate or not?

Let's see. It either terminates or doesn't. So let's consider both cases.

- If `(HP 0)` terminates, then by assumption `(terminates? HP 0)` must be true, and therefore, from the defintion of `HP`, `(HP 0)` must loop forever, i.e., not terminate, contradicting that `(HP 0)` terminates.

- If `(HP 0)` does not terminate, then by assumption `(terminates? HP 0)` must be false, and therefore by definition of `HP`, `(HP 0)` returns 1 immediately, i.e., it terminates, contradicting that `(HP 0)` does not terminate.

Because we get a contradiction no matter what, it must be that what we initially assumed was wrong, i.e., that `terminates?` works correctly. Because `terminates?` was completely arbitrary, the argument works no matter what the implementation of `terminates?` is. In other words, we cannot write a correct `terminates?` function in Scheme.

It turns out that this argument can be made to work for any programming language and even across programming languages, but you'll have to wait for CSU 390 to see that.