CS2800 Fall 2010 - Lecture 22

8 November 2010

1 On Termination

Lets by example see why termination in ACL2 is so important. A Mersenne prime is a prime number that is equal to 2n - 1 for some natural number n. Not all numbers that are 2n - 1 are primes. 3 is a Mersenne prime. 7 is also. 15 is not. 31 is.

An unsolved problem in mathematics is whether there are an infinite number of Mersenne primes. Let's write a function that generates them. We can ask ACL2 to include something that already defines what a prime number is with a predicate PRIMEP and check a few examples:

```
(include-book "quadratic-reciprocity/euclid" :dir :system)
(check= (primep 6) nil)
(check= (primep 7) t)
(check= (primep 6563) t)
```

Now let's write a function that takes the n in 2n - 1 and finds the next Mersenne prime for a larger n:

```
(defun next-mprime (n)
  (let ((v (- (expt 2 (+ 1 n)) 1)))
    (if (primep v)
        v
      (next-mprime (+ 1 (nfix n))))))
```

Basically, it checks whether 2n+1 - 1 is prime. If it is, it returns that value. If not, it makes the recursive call to keep checking larger natural numbers.

Programming mode accepts this definition but Intermediate mode does not. For now, we will force Intermediate mode to accept it. Do not concern yourself with the details:

```
(set-termination-method :measure)
(skip-proofs
  (defun next-mprime (n)
    (let ((v (- (expt 2 (+ 1 n)) 1)))
      (if (primep v)
          v
        (next-mprime (+ 1 (nfix n)))))))
```

Do you recall in a former lecture how we formalized that there is no largest integer? We will do something similar here, asking ACL2 to prove that for any integer n, there is a Mersenne prime greater than 2n - 1:

```
(thm (implies (posp n)
              (primep (next-mprime n))))
```

If I ask ACL2 to prove this, IT PASSES! Wait!? Did I, with the help of ACL2, just solve an unsolved math problem? The answer is no. ACL2 rejected the function definition because it could not prove that the function terminates. Remember how we require functions to be total and terminate on all inputs? Well, when we leave Programming mode, that requirement is enforced by ACL2 attempting to prove termination of functions before it accepts them.

We have just seen how important termination is. The `skip-proofs` told ACL2 to just believe us that the function terminates, and using that assumption, it was able to prove that there are an infinite number of Mersenne primes.

Why exactly is termination so important? It establishes a connection between **computational functions** and **mathematical functions**. When we write a function in a programming language, it might not return a value for some inputs because it would run indefinitely on those inputs. Mathematical functions, on the other hand, have no notion of *computing* an output from an input. They are like infinite look-up tables from inputs to outputs.

The ACL2 logic assumes functions can be treated as mathematical total functions. Requiring termination on all inputs guarantees this is the case.

Here is an example of a non-terminating function that causes serious problems:

```
(defun bad (x)
  (not (bad x)))
```

Non-terminating functions can potentially cause our logic to break. Recall that any theory becomes inconsistent/unsound, if we can derive `nil`, since `nil` can derive anything, we can use it to derive both a formula and its negation, which means we derived a contradiction in our theory, rendering our theory bogus. Lets try to derive `nil` in a theory where the above definition has been admitted:

```
nil
⇐ {Prop. Deduction((and p (not p)) = nil)  }
(and (bad x) (not (bad x)))
⇐ {Def. bad }
(and (bad x) (bad x))
⇐ {Prop. deduction }
(bad x)
⇐ {Prop. Deduction(p = (or p p)}
(or (bad x) (bad x))
⇐ {def. bad }
(or (bad x) (not (bad x)))
⇐ {Prop. deduction }
t
```

2 Termination helps give rise to two Principles

Termination is the reason why we can equate the call of a function with its body, since if this was not the case, there would be certain calls of the function which never terminate and never give an answer, in which case it does not satisfy the definition of a mathematical total function, which for any input returns an answer.

Other than termination there are a few other minor constraints for a function definition to be admitted in ACL2:

2.1 Definitional Principle

A function definition, `(defun f (`$v_1\ v_2 \dots\ v_n$`) body)` is admissible if

- `f` is a new function symbol, meaning it has not yet been defined
- $v_1, v_2, \dots v_n$ are distinct variable symbols
- `body` is a valid ACL2 expression, possibly calling `f` recursively, and no free variables other than $v_1, v_2, \dots, v_n$
- ACL2 is able to prove the function terminates on all inputs

If admissible, the definition is accepted, meaning it can be used and executed in code and adds a new axiom to the current logical theory:
`(equal (f `$v_1\ v_2 \dots\ v_n$`) body)`
The textbook on ACL2 covers in depth how the user can prove functions terminating in ACL2, but we will not cover that in this class. We will depend on a system referred to as *CCG termination analysis* for proving termination automatically. It is built into the ACL2s's *Intermediate* session mode, among others. So you dont have to worry about termination in most cases, but in some cases you might have to rethink and change your definition to make it terminating.

2.2 Induction Principle

We have seen three Rules of Inference till now, Instantiation, Equals for Equals, and Propositional Deduction.

The fourth and final major Rule of Inference[1] in ACL2 comes from the Induction Principle. Each admitted (recursive) function adds an **Induction scheme**(which can be **applied** to a formula to get a rule of inference) to the ACL2 Logic.

And the Induction scheme since it is generated from a function definition, exactly follows the recursive structure(template) of that function.

[1]Actually a rules of inference scheme which can be **applied**

For example, if the following function[2] is admitted to the ACL2 logic:

```
(defun f (n)
  (if (zp n)
      (g1 n)
    (if (evenp n)
        (g2 n)
      (if (> n 0)
          (h1 (f (- n 1))
              (f (floor n 2)))
        (h2 (f (+ n 1)))))))
```

Then the definitional principle of ACL2 says a definitional axiom is added and the Induction principle of ACL2 says the following induction scheme is generated:

```
(implies (zp n)
         φ) [Base Case 1]

(implies (and (not (zp n)
              (evenp n)))
         φ) [Base case 2]

(implies (and (not (zp n))
              (not (evenp n))
              (> n 0)
              φ|σ:((n (− n 1))
              φ|σ:((n (floor n 2)))
         φ)  [Induction Step 1]

(implies (and (not (zp n))
              (not (evenp n))
              (not (> n 0))
              φ|σ:((n (+ n 1)))
         φ)  [Induction Step 2]

⇒ {Induction based on f}
 φ
```

[2](floor n) = $\lfloor \frac{n}{2} \rfloor$

So every function gives rise to a rule of of inference scheme(Induction Scheme) which can be *applied* to any ϕ, i.e any ACL2 formula.

$\phi|_{\sigma:((n\ (-\ n\ 1))}$in the above example is called an **Induction Hypothesis**. Notice that Induction step 1 has 2 induction hypotheses, and the induction step 2 has one induction hypothesis. Induction Step 1 has two induction hypotheses since the body of the function `f` had two recursive calls under the if-branch condition corresponding to that Induction Step.

For example if we need to apply the above induction scheme to a formula, say `(natp (f n))`. Then we have the following rule of inference:

```
(implies (zp n)
         (natp (f n))) [Base Case 1]

(implies (and (not (zp n)
              (evenp n)))
         (natp (f n))) [Base case 2]

(implies (and (not (zp n))
              (not (evenp n))
              (> n 0)
              (natp (f (- n 1)))
              (natp (f (floor n 2))))
         (natp (f n)))  [Induction Step 1]

(implies (and (not (zp n))
              (not (evenp n))
              (not (> n 0))
              (natp (f (+ n 1))))
         (natp (f n)))  [Induction Step 2]

⇒ {Induction based on f}
(natp (f n))
```

So the above Induction Scheme is just a rule of inference, it says, if the base cases are true and the induction steps are true, then by the reason: "induction on `f`", we can deduce `(natp (f n))`.

Study this example carefully, look at how the induction scheme is being generated by exactly following the function definition. Basically, for each

branch of the body of the function, if the branch has a non-recursive call, its a base case, for each such branch we have a *base case*:
`(implies` *branch-condition* ϕ`)`
for each branch which has one or more recursive calls, we have an *induction step*:

```
(implies (and branch-condition
              [Induction Hypotheses for each recursive call])
         φ)
```

Each Induction Hypothesis is of the form $\phi|_\sigma$, where the σ, is given by the argument expressions in the recursive function call.

Why does Induction principle hold, why is it valid? It works because of the following argument:
Since any admitted function in ACL2 is total and terminating, then for any n-tuple(i.e. function's arity is n) from the ACL2 universe, it either satisfies the base case, or takes a finite number of computation steps to reach the base case. Similarily if we want to show ϕ is true, for any n-tuple, either the base case holds, or a finite number of valid induction steps take the formula to the base case, thus ϕ holds for ALL objects in the universe.

Lets look at an example of an induction proof.

Look at next page.

3 Example of an Induction Proof

Given the following definitions:

```
;tlp : All -> Bool
(defun tlp (x)
"Recognize true lists"
  (if (endp x)
    (equal x nil)
    (tlp (cdr x))))

; in: all true-list -> boolean
(defun in (x A)
  "Returns boolean indicating whether the first argument is a
   inber  of the second argument list."
  (if (endp A)
      nil
    (if (= x (car A))
      t
      (in x (cdr A)))))
;subset: true-list true-list -> boolean
(defun subset (A B)
 "All members of first set are in the second argument(set)"
 (if (endp A)
     t
   (and (in (car A) B)
        (subset (cdr A) B))))
```

Prove:
`(subset A A)`

Since we dont know any lemmas about subset, and subset is a recursive definition, alas we have no choice but the prove this formula by induction. Now we have a choice, we can choose any induction scheme in ACL2(corresponding to any admitted function). But we should choose the induction scheme that fits the recursive structure of subset(the only function in the formula). Since subset recursive structure just follows the recursive structure of the true-list data definition, so we can either use the induction scheme generated by `tlp` or `subset`. But both of them generate the exact same induction scheme, so it doesnt matter which is chosen, lets choose the induction scheme generated by `tlp`. If we choose to do the proof by induction in `tlp`, to prove `(subset A A)`, all we need to prove is the base case and the induction step of the induction scheme of `tlp` applied to the formula `(subset A A)`:

So we have two subgoals:
Base Case:
`(implies (endp A) (subset A A))`

```
CONTEXT
A1: (endp A)
```

```
(subset A A)
⇐ { def. subset, A1 }
 t
```

Induction Step:

```
(implies (and (not (endp A))
              (subset (cdr A) (cdr A)))
         (subset A A))
```

```
CONTEXT
A1: (not (endp A))
A2: (subset (cdr A) (cdr A))
```

```
(subset A A)
⇐ { def. subset, A1 }
 (and (in (car A) A)
      (subset (cdr A) A))
⇐ { def in, A1}
 (and (or (= (car A) (car A))
          (in (car A) (cdr A)))
      (subset (cdr A) A))
```

```
⇐ {equality and Prop. Deduction }
 (subset (cdr A) A)
⇐ { cons axiom, A1, def. endp }
 (subset (cdr A) (cons (car A) (cdr A)))
⇐ { Instantiate Lemma subset-cons-element, A2 }
 t
```

where Lemma subset-cons-element is:

```
(implies (subset A B)
         (subset A (cons x B)))
```

Coming up with general lemmas, is an art and is the most creative and interesting part of ACL2 proofs. It signifies big jumps in your proofs, that in your mind, you think is correct, but you cant formalize it in a few steps, in which case, you should come up with the most general lemma you can think of and then instantiate it to complete your current proof. After you finish your current proof, unless and untill you prove the lemma you used in your proof, the proof is not complete, it is still a proof, but under the assumption that the lemma you used is a theorem, which might not be the case.

In the next class we will see the proof for this lemma and also the technique of how to do induction proofs for formulas which are implications.Time permitting I will also talk about proofs about tail-recursive functions.