Reasoning About Programs

Panagiotis Manolios
Northeastern University

March 29, 2012
Version: 62

4

Definitions and Termination

4.1 The Definitional Principle

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

```
(defunc f(x)
  :input-contract ic
  :output-contract oc
  body)
```

then ACL2s adds the definitional axiom

$$\texttt{ic} \;\Rightarrow\; \texttt{(f x)} = \texttt{body}$$

and the contract theorem

$$\texttt{ic} \;\Rightarrow\; \texttt{oc}$$

We now more carefully examine what happens when you define functions.

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

In most languages, one is allowed to write functions such as the following:

```
(defunc f(x)
  :input-contract (natp x)
  :output-contract (natp (f x))
  (+ 1 (f x)))
```

This is a nonterminating recursive function.

There has been no reason for you to write nonterminating functions in your previous classes or in this class, but you had the ability to do it.

Suppose we add the axiom

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{(f x)} = \texttt{(+ 1 (f x))} \tag{4.1}$$

Then, using the axiom, ACL2s can prove the contract theorem

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{(natp (f x))} \tag{4.2}$$

This is unfortunate because we now get a contradiction, *i.e.*, we can prove `nil` in ACL2s, all because we added the definitional axiom for `f` (4.1).

Here is how to derive a contradiction. First, notice that the following is an obvious arithmetic fact.

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{x} \neq \texttt{x} + 1 \tag{4.3}$$

ACL2s can prove this directly.

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

If we instantiate (4.3), we get

$$\texttt{(natp (f x))} \;\Rightarrow\; \texttt{(f x)} \neq \texttt{(+ 1 (f x))} \tag{4.4}$$

Together with (4.2), we have

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{(f x)} \neq \texttt{(+ 1 (f x))} \tag{4.5}$$

Putting (4.1) and (4.5) gives us:

$$\texttt{(natp x)} \;\Rightarrow\; \texttt{nil} \tag{4.6}$$

But, now instantiating (4.6) with `((x 1))` gives us:

t
= { (4.6) }
(natp 1) ⇒ nil
= { Evaluation }
nil

As we have seen, once we have `nil`, we can prove anything. Therefore, this nonterminating recursive equation introduced unsoundness. The point of the definitional principle in ACL2s is to make sure that new function definitions do not render the logic unsound. For this reason, ACL2s does not allow you to define nonterminating functions.

Presumably, any reasonable language will prevent you from writing non-terminating functions. However, no widely used language provides this capability, because checking termination is *undecidable*: no algorithm can always correctly determine whether a function definition will terminate on all inputs that satisfy the input contract.

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

Consider:

```
(defunc f (x)
  :input-contract t
  :output-contract t
  (f x))
```

This leads to the definitional axiom:

$$\texttt{(f x)} = \texttt{(f x)}$$

This cannot possibly lead to unsoundness since it follows from the reflexivity of equality.

Question: can terminating recursive equations introduce unsoundness?

Consider:

```
(defunc f (x)
  :input-contract t
  :output-contract t
```

```
y)
```

This leads to the definitional axiom:

$$\texttt{(f x)} = \texttt{y} \tag{4.7}$$

Which causes problems, *e.g.*,

```
  t
= { Instantiation of (4.7) with ((y t) (x 0)) }
  (f 0)
= { Instantiation of (4.7) with ((y nil) (x 0)) }
  nil
```

We got into trouble 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 ACL2s by defining functions, then the logic stays sound.

That's what the *definitional principle* does.

Definitional Principle for ACL2s

The definition

```
(defunc f (x1 ... xn)
  :input-contract ic
  :output-contract oc
  body)
```

is *admissible* provided:

1. `f` is a new function symbol, *i.e.*, there are no other axioms about it. Functions are admitted in the context of a *history*, a record of all the built-in and defined functions in a session of ACL2s.

   Why do we need this condition? Well, what if we already defined `app`? Then we would have two definitions. What about redefining functions? That is not a good idea because we may already have theorems proven about `app`. We would then have to throw them out and any other theorems that depended on the definition of `app`. ACL2s allows you to undo, but not redefine.

2. The $\texttt{x}_i$ are distinct variable symbols.

   Why do we need this condition? If the variables are the same, say `(defunc f (x x) ...)`, then what is the value of `x` when we expand `(f 1 2)`?

3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no variables freely other than the $\texttt{x}_i$;

   Why? Well, we already saw that global variables can lead to unsoundness. When we say that `body` is a term, we mean that it is a legal expression in the current history.

4. The function is terminating. As we saw, nontermination can lead to unsoundness.

There are also two other conditions that I state separately.

5. `ic` ⇒ `oc` is a theorem.

6. The body contracts hold under the assumption that `ic` holds.

If admissible, the logical effect of the definition is to:

1. Add the *Definitional Axiom* for `f`: `ic` ⇒ `[(f` x_1 `...` x_n`) = body]`.

2. Add the *Contract Theorem* for `f`: `ic` ⇒ `oc`.

But, how do we prove termination?

A very simple first idea is to use what are called measure functions. These are functions from the parameters of the function at hand into the natural numbers, so that we can prove that on every recursive call the function terminates. Let's try this with `app`. What is a measure function for `app`?

How about the length of `x`? So, the measure function is `(len x)`.

Measure Function Definition: `m` is a measure function for `f` if all of the following hold.

1. `m` is an admissible function defined over the parameters of `f`;

2. `m` has the same input contract as `f`;

3. `m` has an output contract stating that it always returns a natural number; and

4. on every recursive call, `m` applied to the arguments to that recursive call decreases, under the conditions that led to the recursive call.

Here then is a measure function for `app`:

```
(defunc m (x y)
  :input-contract (and (listp x) (listp y))
  :output-contract (natp (m x y))
  (len x))
```

This is a non-recursive function, so it is easy to admit. Notice that we do not use the second parameter. That is fine and it just means that the second parameter is not needed for the termination argument.

Next, we have to prove that `m` decreases on all recursive calls of `app`, under the conditions that led to the recursive call. Since there is one recursive call, we have to show:

```
(implies (and (listp x)
              (listp y)
              (not (endp x)))
         (< (m (rest x) y) (m x y)))
```

which is equivalent to:

```
(implies (and (listp x)
              (listp y)
              (not (endp x)))
         (< (len (rest x)) (len x)))
```

which is a true statement.

More examples:

```
(defunc rev (x)
  :input-contract (listp x)
  :output-contract (listp (rev x))
  (if (endp x)
      nil
    (app (rev (rest x)) (list (first x)))))
```

Is this admissible? It depends if we defined `app` already. Suppose `app` is defined as above. What is a measure function?

`len`.

What about:

```
(defunc drop-last (x)
  :input-contract (listp x)
  :output-contract (listp (drop-last x))
  (if (equal (len x) 1)
      nil
    (cons (first x) (drop-last (rest x)))))
```

No. We cannot prove that it is non-terminating, *e.g.*, when `x` is `nil`, what is `(rest x)`?

Exercise 4.1 *Define* `drop-last` *using the design recipe.*

```
(defunc drop-last (x)
  :input-contract (listp x)
  :output-contract (listp (drop-last x))
  (cond ((endp x) nil)
        ((endp (rest x)) nil)
        (t (cons (first x) (drop-last (rest x))))))
```

Exercise 4.2 *What is a measure function for* `drop-last`*?*

What about the following function?

```
(defunc prefixes (l)
  :input-contract (listp l)
  :output-contract (listp (prefixes l))
  (cond ((endp l) '( () ))
        (t (cons l (prefixes (drop-last l))))))
```

Is `prefixes` admissible?

Yes. It satisfies the conditions of the definitional principle; in particular, it terminates because we are removing the last element from `l`.

Exercise 4.3 *What is a measure function for* `prefixes`*?*

Does the following satisfy the definitional principle?

```
(defunc f (x)
  :input-contract (integerp x)
  :output-contract (integerp (f x))
  (if (equal x 0)
      1
    (+ 1 (f (- x 1)))))
```

No. It does not terminate.

What went wrong?

Maybe we got the input contract wrong. Maybe we really wanted natural numbers.

```
(defunc f (x)
  :input-contract (natp x)
  :output-contract (integerp (f x))
  (if (equal x 0)
      1
    (+ 1 (f (- x 1)))))
```

Another way of thinking about this is: What is the largest type that is a subtype of `integer` for which `f` terminates? Or, we could ask: What is the largest type for which `f` terminates?

But, maybe we got the input contract right. Then we used the wrong design recipe:

```
(defunc f (x)
  :input-contract (integerp x)
  :output-contract (integerp (f x))
  (cond ((equal x 0) 1)
        ((> x 0) (+ 1 (f (- x 1))))
        (t (+ 1 (f (+ x 1))))))
```

Now `f` computes the absolute value of `x` (in a very slow way).

The other thing that should jump out at you is that the output contract could be `(natp (f x))` for all versions of `f` above.

4.2 Admissibility of common recursion schemes

We examine several common recursion schemes and show that they lead to admissible function definitions.

The first recursion scheme involves recurring down a list.

```
(defunc f (x1 ... xn)
  :input-contract (and ... (listp xi) ...)
  :output-contract ...
  (if (endp xi)
      ...
      (... (f ... (rest xi) ...) ...)))
```

The above function has n parameters, where the i^{th} parameter, x_i is a list. The function recurs down the list x_i. The ...'s in the body indicate non-recursive, well-formed code, and (rest x_i) appears in the i^{th} position.

We can use (len x_i) as the measure for any function conforming to the above scheme:

```
(defunc m (x1 ... xn)
  :input-contract (and ... (listp xi) ...)
  :output-contract (natp (m x1 ... xn))
  (len xi))
```

That m is a measure function is obvious. The non-trivial part is showing that

(listp x_i) $\wedge$ (not (endp x_i)) $\Rightarrow$ (len (rest x_i)) $<$ (len x_i)

which is easy to see.

So, this scheme is terminating. This is why all of the code you wrote in your beginning programming class that was based on lists terminates.

We can generalize the above scheme, *e.g.*, consider:

```
(defunc f (x1 x2)
  :input-contract (and (listp x1) (listp x2))
  :output-contract (listp (f x1 x2))
  (cond ((endp x1) x2)
        ((endp x2) x1)
        (t (list (f (rest x1) (rest x2))
                 (f (rest x1) (f (rest x1) (cons x2 x2)))))))
```

We now have three recursive calls and two base cases. Nevertheless, the function terminates for the same reason: len decreases.

```
(defunc m (x1 x2)
  :input-contract (and (listp x1) (listp x2))
  :output-contract (natp (m x1 x2))
  (len x1))
```

All three recursive calls lead to the same proof obligation:

(listp x_1) $\wedge$ (not (endp x_1)) $\wedge$ (not (endp x_2)) $\Rightarrow$ (len (rest x_1)) $<$ (len x_1)

Thinking in terms of recursion schemes and templates is good for beginners, but what really matters is termination. That is why recursive definitions make sense.

Let's look at one more interesting recursion scheme.

```
(defunc f (x1 ... xn)
  :input-contract (and ... (natp xi) ...)
  :output-contract ...
  (if (equal xi 0)
      ...
      (... (f ... (- xi 1) ...) ...)))
```

The above is a function of n parameters, where the i^{th} parameter, x_i is a natural number. The function recurs on the number x_i. The ...'s in the body indicate non-recursive, well-formed code, and (- x_i 1) appears in the i^{th} position.

We can use x_i as the measure for any function conforming to the above scheme:

```
(defunc m (x1 ... xn)
  :input-contract (and ... (natp xi) ...)
  :output-contract (natp (m x1 ... xn))
  xi)
```

That `m` is a measure function is obvious. The non-trivial part is showing that

(natp x_i) $\wedge$ (not (equal x_i 0)) $\Rightarrow$ (- x_i 1) $<$ x_i

which is easy to see.

So, this scheme is terminating. This is why all of the code you wrote in your beginning programming class that was based on natural numbers terminates.

Exercise 4.4 *We can similarly construct a recursion scheme for integers. Do it.*

4.3 Exercises

For each function below, you have to check if its definition is admissible, *i.e.*, it satisfies the definitional principle.

If the function does satisfy the definitional principle then:

1. Provide a measure that can be used to show termination.
2. Explain in English why the contract theorem holds.
3. Explain in English why the body contracts hold.

If the function does not satisfy the definitional principle then identify each of the 6 conditions above that are violated. Also, if the function is terminating, provide a measure function.

Exercise 4.5

```
(defunc f (x y)
  :input-contract (and (true-listp x) (natp y))
  :output-contract (true-listp (f x y))
  (cond ((equal y 0) nil)
        ((endp x) (list y))
        (t (f (cons y x) (- y 1)))))
```

Exercise 4.6 *Dead code example*

```
(defunc f (x y)
  :input-contract (and (natp x) (natp y))
  :output-contract (integerp (f x y))
  (cond ((equal x 0) 1)
        ((< x 0) (f -1 -1))
        (t (+ 1 (f (- x 1) y)))))
```

Notice that the second case of the `cond` above will never happen.

Below are some generative recursion examples.

Exercise 4.7

```
(defunc f (x y)
  :input-contract (and (integerp x) (natp y))
```

```
  :output-contract (integerp (f x y))
  (cond ((equal x 0) 1)
        ((< x 0) (f (+ 1 y) (* x x)))
        (t (+ 1 (f (- x 1) y)))))
```

Exercise 4.8

```
(defunc f (x y)
  :input-contract (and (true-listp x) (integerp y))
  :output-contract (natp (f x y))
  (cond ((endp x) y)
        (t (f (rest x) (+ 1 y)))))
```

Exercise 4.9

```
(defunc f (x y)
  :input-contract (and (true-listp x) (integerp y))
  :output-contract (natp (f x y))
  (cond ((and (endp x) (equal y 0))
         0)
        ((and (endp x) (< y 0))
         (+ 1 (f x (+ 1 y))))
        ((endp x)
         (+ 1 (f x (- y 1))))
        (t
         (+ 1 (f (rest x) y)))))
```

4.4 Final Comments

As we already mentioned, checking for termination is undecidable; Turing showed that. So, you can define functions that terminate, but that ACL2s can't prove terminating automatically. However, we expect that for the programs we ask you to write, ACL2s will be able to prove termination automatically. If not, send email and we will help you.

Exercise 4.10 *How would you write a program that checks if other programs terminate?*

By the way, remember "big-Oh" notation? It is connected to termination. How? Well if the running time for a function is $O(n^2)$, say, then that means that:

1. the function terminates; and
2. there is a constant c s.t. the function terminates within $c \cdot n^2$ steps, where n is the "size" of the input

The big-Oh analysis is just a refinement of termination, where we are not interested in only whether a function terminates, but also we want an upper bound on how long it will take.