Definitional Equations in ACL2

Sandip Ray

Department of Computer Science
University of Texas at Austin

Email: `sandip@cs.utexas.edu`
web: `http://www.cs.utexas.edu/users/sandip`

Joint work with J Strother Moore

Background

ACL2 provides a **definitional principle** to introduce new (recursive) function definitions:

```
(defun factorial (x)
  (if (zp x) 1
   (* x (factorial (- x 1)))))
```

The effect is to extend the current ACL2 theory with a new axiom:

```
(equal (factorial x)
       (if (zp x) 1
         (* x (factorial (- x 1)))))
```

Background

ACL2 provides a **definitional principle** to introduce new function definitions:

```
(defun factorial (x)
  (if (zp x) 1
   (* x (factorial (- x 1)))))
```

To admit a function, we must show that there is an ordinal measure that decreases along every recursive call.

1. `(o-p (nfix x))`
2. `(implies (not (zp x)) (o< (nfix (- x 1)) (nfix x)))`

Background

ACL2 provides a **definitional principle** to introduce new function definitions:

```
(defun factorial (x)
  (if (zp x) 1
   (* x (factorial (- x 1)))))
```

To admit a function, we must show that there is an ordinal measure that decreases along every recursive call.

This ensures that there exists one **unique** function satisfying the equation.

Background

ACL2 provides a **definitional principle** to introduce new function definitions:

```
(defun factorial (x)
  (if (zp x) 1
   (* x (factorial (- x 1)))))
```

To admit a function, we must show that there is an ordinal measure that decreases along every recursive call.

This ensures that there exists one **unique** function satisfying the equation.

It also justifies the use of induction based on the induction scheme suggested by the function.

Definitional Equation

In some cases, we might be only interested in the definitional equation (not the induction axioms).

Then it should be possible to introduce more defining equations (as long as we do not also introduce the corresponding induction axioms).

Definitional Equation

In some cases, we might be only interested in the definitional equation (not the induction axioms).

We can do this by using `encapsulation`.

Encapsulation

Encapsulation allows us to introduce function symbols by specifying certain properties (or **constraints**).

Soundness requires that we show some witnessing function that satisfies the postulated constraints.

If we want to introduce a definitional equation in ACL2 we can show that there is **some** function satisfying this equation.

Using encapsulation

```
(encapsulate
  (((triple-rev *) => *))
  (local (defun triple-rev (x) ...))
  (defthm triple-rev-def
    (equal (triple-rev x)
           (cond ((endp x)  nil)
                 ((endp (cdr x)) (list (car x)))
                 (t (let* ((b@c (cdr x))
                           (c@rev-b (triple-rev b@c))
                           (rev-b (cdr c@rev-b))
                           (b (triple-rev rev-b))
                           (a (car x))
                           (a@b (cons a b))
                           (rev-b@a (triple-rev a@b))
                           (c (car c@rev-b))
                           (c@rev-b@a (cons c rev-b@a)))
                         c@rev-b@a))))))
```

Using encapsulation

Using encapsulation we can introduce alternative efficient defining equations for the same function.

Using `mbe`, we can then use these alternative definitions for execution if desired.

Encapsulation goodies

But we can achieve more things!

- We can show that there are generic classes of defining equations that can be introduced in the ACL2 logic.

One example is tail-recursive equations.

```
(equal (f x) (if (test x) (base x) (f (recur x))))
```

Manolios and Moore (2000) showed that any tail-recursive equation can be introduced in ACL2.

In this Talk

We will consider more general defining equations:

```
(equal (f x)
       (if (test x)
           (base x)
         (wrap x (f (recur x)))))
```

We will investigate one sufficient condition under which this equation is admissible.

Non-triviality

Not all such equations should be admissible.

Consider the equation:

```
(equal (nils x)
       (if (equal x 0) nil (cons nil (nils (- x 1)))))
```

Non-triviality

Not all such equations should be admissible.

Consider the equation:

```
(equal (nils x)
       (if (equal x 0) nil (cons nil (nils (- x 1)))))
```

But consider the following equation:

```
(equal (num x)
       (if (equal x 0) 0 (+ 1 (num (- x 1)))))
```

Is this axiom also inconsistent?

Non-triviality

Not all such equations should be admissible.

Consider the equation:

```
(equal (nils x)
       (if (equal x 0) nil (cons nil (nils (- x 1)))))
```

But consider the following equation:

```
(equal (num x)
       (if (equal x 0) 0 (+ 1 (num (- x 1)))))
```

Is this axiom also inconsistent?

No.

A Little History

On February 21, 2004, Vinod Vishwanath gave a presentation on defining a sequential simplifier in ACL2.

Quick Note: I do not quite know what a sequential simplifier is.

His definition was of the form

```
(equal (f x)
       (if (test x)
           (base x)
         (wrap x (f (recur x)))))
```

But the recursive equation was not terminating!

Some Observations

We can of course artificially terminate the equation by recurring up to a fixed upper bound.

```
(defun fn (x n)
  (if (or (test x) (zp n)) (base x)
   (wrap x (fn (recur x) (- n 1)))))
```

Some Observations

We can of course artificially terminate the equation by recurring up to a fixed upper bound.

```
(defun fn (x n)
  (if (or (test x) (zp n)) (base x)
   (wrap x (fn (recur x) (- n 1)))))
```

Suppose we can then prove that for each x there is some “large enough” n beyond which we need not bother to bound.

Some Observations

We can of course artificially terminate the equation by recurring up to a fixed upper bound.

```
(defun fn (x n)
  (if (or (test x) (zp n)) (base x)
   (wrap x (fn (recur x) (- n 1)))))
```

Suppose we can then prove that for each x there is some “large enough” n beyond which we need not bother to bound.

```
(natp (clock x))
(implies (and (natp n) (>= n (clock x)))
         (equal (fn x (+ n 1)) (fn x n)))
```

Some Observations

We can of course artificially terminate the equation by recurring up to a fixed upper bound.

```
(defun fn (x n)
  (if (or (test x) (zp n)) (base x)
   (wrap x (fn (recur x) (- n 1)))))
```

Suppose we can then prove that for each x there is some “large enough” n beyond which we need not bother to bound.

```
(natp (clock x))
(implies (and (natp n) (>= n (clock x)))
         (equal (fn x (+ n 1)) (fn x n)))
```

Then is it ok to introduce the original axiom for `f`?

Some Observations

We can of course artificially terminate the equation by recurring up to a fixed upper bound.

```
(defun fn (x n)
  (if (or (test x) (zp n)) (base x)
   (wrap x (fn (recur x) (- n 1)))))
```

Suppose we can then prove that for each x there is some “large enough” n beyond which we need not bother to bound.

```
(natp (clock x))
(implies (and (natp n) (>= n (clock x)))
         (equal (fn x (+ n 1)) (fn x n)))
```

Then is it ok to introduce the original axiom for `f`?

Yes.

The `defpun` Intuition

Consider the proof that every tail-recursive definition can be introduced.

```
(defstub test (*) => *)
(defstub base (*) => *)
(defstub recur (*) => *)
```

We want to introduce the axiom:

```
(equal (f x)
       (if (test x)
           (base x)
         (f (recur x))))
```

The `defpun` Intuition

The recipe **(Manolios and Moore, 2000)**:

Consider the bounded version of `f`:

```
(defun fn (x n)
  (if (or (test x) (zp n)) (base x)
    (fn (recur x) (- n 1))))
```

The `defpun` Intuition

Now choose a large enough n if such an n exists.

```
(defun recur-n (x n)
  (if (zp n) x
     (recur-n (recur x) (- n 1))))

(defun-sk f-terminates (x)
  (exists n (test (recur-n x n))))
```

Then define `f` as follows:

```
(defun f (x)
  (if (f-terminates x) (fn x (f-terminates-witness x))
    42))
```

The `defpun` Proof

The definition:

```
(defun f (x)
  (if (f-terminates x) (fn x (f-terminates-witness x))
    42))
```

The theorem:

```
(equal (f x)
       (if (test x)
           (base x)
         (f (recur x))))
```

Complications in Our Case

```
(equal (f x)
       (if (test x) (base x)
         (wrap x (f (recur x)))))
```

We do not know that eventually `test` becomes true, but only that the bounded version stabilizes.

Consider

```
(defun test (x) nil)
(defun wrap (x y) y)
(defun recur (x) x)
```

Our Proof

Let us recount what we have:

```
1. (fn x n) = (if (or (zp n) (test x))
                  (base x)
                (wrap x (fn (recur x) (- n 1))))
```

We also have an upper bound:

```
T0: (natp n) /\  n >= (clock x) ==>  (fn x n+1) = (fn x n)
```

We can define c such that:

```
T1: (natp (c x))
T2: 0 < (c x)   ==>  (fn x (c x)) /= (fn x (c x)-1)
T3: (natp n) /\ n >= (c x) ==> (fn x n+1) = (fn x n)
```

The function we are seeking is:

```
(f x) = (fn x (c x))
```

The Proof

To prove:

```
  (f x) = (if (test x) (base x)
              (wrap x (f (recur x))))
```

```
Or,

  (fn x (c x)) = (if (test x) (base x)
                     (wrap x (fn (recur x) (c (recur x)))))
```

Some Lemmas:

```
L1: (natp i) /\ (natp j) /\ i >= (c x)  =>  (fn x i) = (fn x (c x))
L2. 0 < (c x) ==> (c x) - 1 <= (c (recur x))
```

L1 is trivial by induction.

Proof of `L2`

Observation: If `(c x) > 0`, then `(test x)` does not hold. Otherwise,

```
o For all nats i, j (fn x i) = (fn x j) = (base x)
o Therefore [T1, T2] (c x) = 0
```

Assume `(c x) - 1 > (c (recur x))`:

```
  (fn x (c x))
=   [ ~(test x), 0 < (c x) ]
  (wrap x (fn (recur x) (- (c x) 1)))
=   [T3 ((recur x) for x), assumption]
  (wrap x (fn (recur x) (c (recur x))))
=   [ (c x) - 2 >= (c (recur x)), T3]
  (wrap x (fn (recur x) (- (c x) 2)))
=   [Definition]
  (fn x (- (c x) 1))
```

This is contradiction to **T2**.

The Main Proof

```
  (fn x (c x)) = (if (test x) (base x)
                     (wrap x (fn (recur x) (c (recur x)))))
```

The proof is by case analysis.

```
Case 1  :  (c x) = 0
Case 1.1: (test x)
LHS = RHS = (base x)
Case 1.2:  (not (test x))
 RHS
=  [ ~(test x) ]
 (wrap x (fn (recur x) (c (recur x))))
=  [ (natp (c (recur x))), ~(test x)]
 (fn x (+ (c (recur x)) 1))
=  [L1, (c x) = 0, (natp (c (recur x)))]
  (fn x 0)
= LHS
```

The Main Proof, Cont'd

```
Case 2: (c x) > 0
 (fn x (c x))
= [L2, T3]
  (fn x (1+ (c (recur x))))
= [Definition, ~ (test x) since (c x) > 0]
 (wrap x (fn (recur x) (c (recur x))))
= [ ~(test x)]
  (if (test x) (base x)
    (wrap x (fn (recur x) (c (recur x)))))
```

An Interesting Aside

Suppose the stabilization condition was instead:

```
 (defun recur-n (x n)
   (if (test x) x
     (if (zp n) x
       (recur-n (recur x) (1- n)))))
 (defthm clock-natp (natp (clock-fn x)))
 (defthm test-eventually-is-true
   (test (recur-n x (clock-fn x))))
```

Then we can actually do a `defun`.

```
(defun f (x)
  (declare (xargs :measure ...))
  (if (test x) (base x)
    (wrap x (f (recur x)))))
```

Trivial Exercise: Come up with a measure given the above conditions.

Other Work on Primitive Recursive Admission

Cowles showed the following condition to be sufficient.

```
(exists c (equal (wrap x c) c))
```

He shows that Manolios-Moore construction works for that case.

Thus the following axiom is admissible.

```
(equal (factorial x)
       (if (equal x 0) 1
         (* x (factorial (- x 1)))))
```

Other Work on Primitive Recursive Admission

But the following will not be, although it is consistent:

```
(equal (num x)
       (if (equal x 0) 0
         (+ 1 (num (- x 1)))))
```

Question: Is there a more systematic way to find sufficient conditions for generalized primitive recursive equations?