Consistently Adding Primitive Recursive Definitions in ACL2

John Cowles
Department of Computer Science
University of Wyoming
Laramie, Wyoming 82071
cowles@uwyo.edu

February 25, 2002

Abstract

In [2, 3], P. Manolios and J Moore show that a tail recursive defining equation for a new function can always be consistently added to ACL2. This is done by "constructing" a function that satisfies the proposed tail recursive defining equation. Their construction is extended to many *primitive recursive* defining equations. This extends the known recursive schemes that can be consistently introduced into ACL2's logic. Exactly what is meant by "primitive recursive" and the exact restrictions placed on the definitions are explained below.

1 Tail Recursion

P. Manolios and J Moore [2, 3] describe a macro named `defpun` for consistently introducing "partial functions" into ACL2. One of the many cases handled by `defpun` is when the "defining equation" is tail recursive: Let `test`, `base`, and `st` be arbitrary unary functions. There always is an ACL2 function `f` that satisfies

```
(equal (f x)
       (if (test x)
           (base x)
           (f (st x)))).
```

As shown by Manolios and Moore, such a function `f` can be constructed, in ACL2, as follows:

1. Define `stn` so that `(stn x n)` computes ($\texttt{st}^{\texttt{n}}$ `x`).
2. Use `defchoose` to define a Skolem (witnessing) function `fch` so that `(fch x)` is an `n` such that `(test (stn x n))` holds, if such an `n` exists. If no such `n` exists, then ACL2 knows nothing about the value of `(fch x)`. If `(test (stn x (fch x)))` holds, then `(fch x)` need not be the smallest `n` such that `(test (stn x n))` holds.
3. Define a version of `f`, called `fn`, with an extra "clock-like" input parameter, `n`, that ensures termination:

```
(defun fn (x n)
  (declare (xargs :measure (nfix n)))
  (if (or (zp n) (test x))
      (base x)
      (fn (st x) (1- n)))).
```

4. Finally define `f`:

```
(defun f (x)
  (if (test (stn x (fch x)))
      (fn x (fch x))
      nil))
```

It is not difficult to believe (as ACL2 verifies) that `f` also satisfies the tail recursive equation

```
(equal (f x)
       (if (test x)
           (base x)
           (f (st x)))).
```

For this construction, any constant would do in place of `nil` in the definition of `f`.

2 Primitive Recursion

The class of all primitive recursive functions on the nonnegative integers is often discussed in courses on the Theory of Computation [1]. This class of functions is closed under primitive recursive definitions: If k and h are primitive recursive functions, then so is the function f defined by the equations

$$\begin{aligned} f(x_1, ..., x_n, 0) &= k(x_1, ..., x_n) \\ f(x_1, ..., x_n, t+1) &= h(t, f(x_1, ..., x_n, t), x_1, ..., x_n). \end{aligned}$$

Reducing the number of input parameters to one and slightly generalizing this form of recursive definition leads to the following definition: Let `h` be a binary function. A function `f` satisfying an equation of the form

```
(equal (f x)
       (if (test x)
           (base x)
           (h x (f (st x)))))
```

is called *primitive recursive*, for the purposes of this paper.

The Manolios-Moore construction for any tail recursive defining equation can be extended, for many `h`'s, to the case of a primitive recursive defining equation. But, as shown by Manolios and Moore, there are `h`'s for which no ACL2 function `f` exists that satisfies the primitive recursive defining equation. Their example, showing such an `f` need not exist, is

```
(equal (g x)
       (if (equal x 0)
           nil
           (cons nil (g (- x 1))))),
```

so in this case `(test x)` is `(equal x 0)`, `(base x)` is `nil`, `(h x y)` is `(cons nil y)`, and `(st x)` is `(- x 1)`.

A sufficient (but not necessary) condition on `h` for the existence of `f` is that `h` have a right fixed point, i.e., there is some `c` such that `(h x c) = c`. The tail recursion construction given above need only be modified, in steps 3 and 4, to accommodate the additional function `h` in the primitive recursive defining equation:

1. Define `stn` so that `(stn x n)` computes ($\texttt{st}^{\texttt{n}}$ `x`).

2. Use `defchoose` to define a Skolem (witnessing) function `fch` so that `(fch x)` is an `n` such that `(test (stn x n))` holds, if such an `n` exists.

3. Define a version of `f`, called `fn`, with an extra "clock-like" input parameter, `n`, that ensures termination:

```
(defun fn (x n)
  (declare (xargs :measure (nfix n)))
  (if (or (zp n) (test x))
      (base x)
      (h x (fn (st x) (1- n)))))).
```

4. Finally define `f`: Here `(h-fix)` is a right fixed point for `h`.

```
(defun f (x)
  (if (test (stn x (fch x)))
      (fn x (fch x))
      (h-fix)))
```

   ACL2 verifies that `f` also satisfies the *primitive recursive* equation

```
(equal (f x)
       (if (test x)
           (base x)
           (h x (f (st x))))).
```

An example, also due Manolios and Moore, showing that `h` having a right fixed point is not necessary for the existence of `f` is

```
(equal (g1 x)
       (if (equal x 0)
           0
           (+ 1 (g1 (- x 1))))),
```

so `(test x)` is `(equal x 0)`, `(base x)` is `0`, `(h x y)` is `(+ 1 y)`, and `(st x)` is `(- x 1)`. The ACL2 function `fix` satisfies this primitive recursive equation. Here `(fix x)` returns `x` if `x` is an ACL2 number and returns 0 otherwise.

2.1 Examples

Example 1. Modified `cons` example.

This example illustrates what to do when `h` lacks a right fixed point. The "problem" with the defining equation, given above, for `g`, from the point of view of this paper, is that `cons` does not have a right fixed point, so one is provided by the following:

```
(defstub
  cons-fix () => *)
```

```
(defun
  cons$ (x y)
  (if (equal y (cons-fix))
      (cons-fix)
      (cons x y)))
```

Then the following primitive recursive equation has an ACL2 solution for g:

```
(equal (g x)
       (if (equal x 0)
           nil
           (cons$ nil (g (- x 1))))).
```

Example 2. Naive Factorial.

This example illustrates that any fixed point will do: Multiplication already has a right fixed point, namely zero. That is `(* x 0) = 0`. The following primitive recursive equation has an ACL2 solution for `fact`.

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

Note that ACL2's definitional principle would accept the definition that uses the zero-test idiom `(zp x)` in place of the test `(equal x 0)`:

```
(defun
  fact (x)
  (if (zp x)
      1
      (* x (fact (- x 1))))).
```

Example 3. Example with multiple input parameters.

This example illustrates, as shown by Manolios and Moore, that knowing how to deal with a defining equation of an unary function means that a defining equation of a function with multiple input parameters can also be dealt with systematically.

The following primitive recursive equation has an ACL2 solution for `k`.

```
(equal (k a b)
       (if (equal b 0)
           1
           (* a b (k a (- b 1)))))
```

First construct an unary version, `k1`, of `k`.

```
(defun
  k1-test (x)
  (let ((a (car x))
        (b (cadr x)))
       (equal b 0)))
```

```
(defun
  k1-base (x)
  (let ((a (car x))
        (b (cadr x)))
       1))
```

```
(defun
  k1-st (x)
  (let ((a (car x))
        (b (cadr x)))
       (list a (- b 1))))
```

```
(defun
  k1-h (x y)
  (let ((a (car x))
        (b (cadr x)))
       (* a b y)))
```

Using the primitive recursive version of the constructions outlined above allows the construction of an unary function `k1` such that

```
(equal (k1 x)
       (if (k1-test x)
           (k1-base x)
           (k1-h x (k1 (k1-st x))))).
```

Then `k` defined by

```
(defun
  k (a b)
 (k1 (list a b)))
```

has the desired property,

```
(equal (k a b)
       (if (equal b 0)
           1
           (* a b (k a (- b 1))))).
```

3 Conclusion

Recursive equations of the form

```
(equal (f x)
       (if (test x)
           (base x)
           (h x (f (st x)))))
```

are satisfiable in ACL2's logic whenever `h` has a right fixed point. Proving h has a right fixed point ensures the systematic construction of such a function `f`.

References

[1] M. Davis, R. Sigal, and E. Weyuker. *Computability, Complexity, and Languages.* Academic Press, second edition, 1994.

[2] P. Manolios and J S. Moore. Partial Functions in ACL2. In M. Kaufmann and J S. Moore, Editors, *ACL2 Workshop 2000 Proceedings*, October 30–31, 2000, University of Texas at Austin.

[3] P. Manolios and J S. Moore. Partial Functions in ACL2. January 2001, submitted for publication.