Consistently Adding Primitive Recursive Definitions in ACL2

by

John Cowles
University of Wyoming

`defpun`

A macro for consistently introducing “partial functions” into CAL2.

Described in Pete & J’s paper, **Partial Functions in ACL2**, at *ACL2 Workshop 2000*.

One of many cases handled by **`defpun`** is when the “defining equation” is *tail recursive*.

Tail Recursion

Let `test`, `base`, and `st` be arbitrary unary functions.

There always is at least one ACL2 function `f` that satisfies

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

Tail Recursion Construction

Pete & J construct a *tail recursive* function `f` in ACL2:

1. Define `stn` so that `(stn x n)` computes (st^{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.

Tail Recursion Construction

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))
```

Any constant would do in place of `nil` in this definition.

Tail Recursion Construction

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

ACL2 verifies that this `f` satisfies the *tail recursive* equation

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

Primitive Recursion

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*.

Primitive Recursion

This definition of *primitive recursive* is inspired by the primitive recursive definitions studied in Theory of Computation courses:

For previously defined functions, k and h, on the non-negative integers, define f by

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

Here $\vec{x} = x_1, ..., x_n$.

Primitive Recursion

Extend Pete & J's tail recursive construction to *many, but not all*, primitive recursive defining equations.

Primitive Recursion

There are `h`'s for which **no** ACL2 function `f` satisfies the *primitive recursive* defining equation:

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

Example

No ACL2 function `g` satisfies this *primitive recursive* equation

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

Here

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

Primitive Recursion

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

A sufficient (but not necessary) condition on `h` for the existence of `f` is that `h` have a right fixed point.

That is, there is some `c` such that
`(h x c) = c`.

Primitive Recursion Construction

Modify Pete & J's tail recursion construction.

Construct a *primitive recursive* function `f` in ACL2:

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

   (Same as for tail recursion.)

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.

   (Same as for tail recursion.)

Primitive Recursion Construction

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)))
```

Primitive Recursion Construction

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

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

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

Example

A right fixed point for `h` is *not necessary* for some primitive recursive definitions.

The ACL2 function `fix` **satisfies** this *primitive recursive* equation

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

Here

- `(test x)` is `(equal x 0)`,
- `(base x)` is `0`,
- `(h x y)` is `(+ 1 y)` [**no** fixed point], and
- `(st x)` is `(- x 1)`.

`defpr`

A macro for consistently introducing *primitive recursive* equations into ACL2.

In an `encapsulate`, carry out the *Primitive Recursion Construction*:

- `f` is constrained only by

```
(defthm
    generic-primitive-recursive-f
    (equal (f x)
           (if (test x)
               (base x)
               (h x (f (st x)))))).
```

- `h` is constrained to have a right fixed point, `(h-fix)`.

- `test`, `base`, and `st` are unconstrained.

`defpr`

Given the required fixed point, the `defpr` macro

- recognizes a primitive recursive definition, and

- generates a *functional instance* of `generic-primitive-recursive-f` to produce a witness to the desired primitive recursive equation.

Example

No ACL2 function `g` satisfies this *primitive recursive* equation

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

The **problem**: `cons` has no right fixed point.

Example

The **problem**: `cons` has no right fixed point.

Provide a right fixed point by the following:

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

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

Example

```
(defpr
   g (x)
   (declare (xargs :fixpt (cons-fix)))
   (if (equal x 0)
       nil
       (cons$ nil (g (- x 1)))))
```

produces an ACL2 solution for `g`:

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

Note use of `XARGS` keyword `:fixpt` to give the *required fixed point*.

Example

Any fixed point will do.

Multiplication already has a right fixed point, namely 0:

$$\texttt{(* x 0)} = 0.$$

```
(defpr
   fact (x)
   (declare (xargs :fixpt 0))
   (if (equal x 0)
       1
       (* x (fact (- x 1)))))
```

produces an ACL2 solution for `fact`:

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

Note: ACL2 accepts 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

This succeeds: (a primitive recursive definition)

```
(defpr
   f (x)
   (declare (xargs :fixpt 0))
   (if (equal x 0)
       1
       (* (f (- x 1))
          (f (- x 1)))))
```

This fails: (**not** a primitive recursive definition)

```
(defpr
   f1 (x)
   (declare (xargs :fixpt 0))
   (if (equal x 0)
       1
       (* (f1 (- x 1))
          (f1 (+ x 1)))))
```

Example
with parameters.

```
(defpr
   k (a b)
   (declare (xargs :fixpt 0))
   (if (equal b 0)
       1
       (* a b (k a (- b 1)))))
```

Note: On the non-negative integers

$$\texttt{(k a b)} = \mathsf{a}^{\mathsf{b}} \cdot \mathsf{b}!$$

Example

Tail recursion is a special case.

The function, `Id-2-2`, defined by

$$\texttt{(Id-2-2 x1 x2)} = \texttt{x2}$$

is used for `h`.

Any constant can be used for the fixed point.

```
(defpr
   tail-f (x)
   (declare (xargs :fixpt nil))
   (if (tail-test x)
       (tail-base x)
       (Id-2-2 x (tail-f (tail-st x)))))

(defthm
   tail-f-is-tail-recursive
   (equal (tail-f x)
          (if (tail-test x)
              (tail-base x)
              (tail-f (tail-st x)))))
```

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`.