# Contributions to the Theory of Tail Recursive Functions

John Cowles      Ruben Gamboa
Computer Science Department
University of Wyoming
{cowles,ruben}@cs.uwyo.edu

**Abstract**

It is shown that the existence of an unique total function satisfying a tail recursive definitional axiom ensures the recursion always halts. This is in contrast to the general case, when the adjective tail need not apply to the recursion: The existence of an unique total function satisfying a (general) recursive definitional axiom need not force the recursion to always terminate.

A similar result is shown to have application to Tail Recursive Interpreters.

The result reported in [1] about Knuth's generalization of McCarthy's 91 Function is obtained in a different way, as a corollary of more general results about reflexive tail recursive functions.

## Introduction

Tail recursive definitional axioms have desirable properties not enjoyed by arbitrary recursive definitional axioms. Foremost among these properties is consistency of the axiom. To ensure consistency, ACL2's definitional principle requires that the recursion in a proposed definitional axiom satisfy an appropriate measure conjecture. In [5], P. Manolios and J S. Moore show it is always consistent to add a tail recursive definitional axiom (even when the recursion does not satisfy any appropriate measure conjecture).

ACL2's definitional principle ensures more than consistency. Satisfaction of an appropriate measure conjecture means that the recursion always halts and that implies there is one and only one total function satisfying the definitional axiom. The converse implication, in general, does not hold. That is, the existence of an unique total function satisfying a recursive definitional axiom does not force the recursion to always terminate.

However, the afore mentioned converse implication does hold for tail recursive definitional axioms:The existence of an unique total function satisfying a tail recursive definitional axiom does mean the recursion always halts.

## Tail Recursion

What is tail recursion? A function is said to be tail recursive if its definition is tail recursive. The definition of a function `f` is tail recursive provided there is at least one recursive call to `f` in the *body* of the definition and each such recursive call to `f` is tail recursive.

Here is what it means for a recursive call to be tail recursive in a definition such as this one:

```
(defun f (x₁ ... xₙ)
  body)
```

Assume *body* contains no macros or lambda applications. That is, expand all macros in *body* and reduce the lambda applications by $\beta$-reduction. Think of the expanded *body* as an expression tree. A recursive call of `f` in *body* is tail recursive just in case these two conditions are met.

1. The call to `f` is not on the test branch of any `if`.
2. On any branch containing the call to `f`, only `if` may appear above the call to `f`.

### Examples.

-
  ```
  (defun f (x)
    (if (f x)
        x
        x))
  ```
  The call to `f` in this *body* violates the first condition above, so the call is not tail recursive.

-
  ```
  (defun f (x)
    (if (zp x)
        1
        (* x
           (f (- x 1)))))
  ```
  The call to `f` in this *body* violates the second condition above (`*` appears above `f` in the expression tree), so the call is not tail recursive.

-
  ```
  (defun A (x y)
    (declare
     (xargs :guard
            (and (natp x)
                 (natp y))))
    (if (zp x)
        (+ y 1)
        (if (zp y)
            (A (- x 1) 1)
            (A (- x 1)
               (A x
                  (- y 1))))))
  ```
  There are three calls to `A` in this *body*. The call `(A (- x 1) 1)` and the outer call in `(A (- x 1)(A x (- y 1)))` are both tail recursive. The inner call `(A x (- y 1))` is not tail recursive because the outer call to `A` appears above this inner call in the expression tree.

-
  ```
  (defun M91 (x)
    (declare
     (xargs :guard
            (integerp x)))
    (if (> x 100)
        (- x 10)
        (M91
         (M91 (+ x 11)))))
  ```
  There are two recursive calls to `M91` in this *body*. The outer call in `(M91 (M91 (+ x 11)))` is tail recursive. The inner call `(M91 (+ x 11))` is not tail recursive because the outer call to `M91` appears above this inner call in the expression tree.

-
  ```
  (defun 3x+1 (x)
    (declare
     (xargs :guard (natp x)))
    (if (<= x 1)
        x
        (if (evenp x)
            (3x+1 (/ x 2))
            (3x+1
             (+ (* 3 x) 1)))))
  ```
  The two calls to `3x+1` in this *body* are both tail recursive calls.

## Tail Recursive Functions

Let test, base, and step be unary functions. Consider the following proposed tail recursive definition.

```
(defun f (x)
  (if (test x)
      (base x)
      (f (step x))))
```

Since this recursive call to f is simple and explicitly given, it is possible to be explicit and very precise about the meanings of the following with respect to this proposed definition:

- A total function satisfies the defining tail recursion axiom for this definition.
- The tail recursion in this definition terminates for a given input.
- The tail recursion in this definition satisfies a measure conjecture.

It possible to state these concepts in ACL2. Therefore proofs of the theorems (but not the propositions nor the corollaries) given below were mechanically verified using ACL2.

A total ACL2 function f is said to satisfy the defining tail recursion axiom for the proposed definition provided the following is true about every x.

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

P. Manolios and J S. Moore's defpun paper [5] shows that there is always at least one total ACL2 function that satisfies the defining tail recursion axiom for any such proposed tail recursive definition.

The tail recursion in the above proposed definition is said to terminate for a given x provided the following holds $\exists \mathtt{n}(\mathtt{test}(\mathtt{step}^{\mathtt{n}}\,\mathtt{x}))$. The tail recursion in the above proposed definition is said to always halt provided the tail recursion terminates for all x.

The tail recursion in the above proposed definition is said to satisfy a measure conjecture provided there is a well-founded binary relation rel, on the set of objects recognized by some predicate mp, and a measure m satisfying

```
(and (mp (m x))
     (implies (not (test x))
              (rel (m (step x))
                   (m x))))
```

The binary relation `rel` is well-founded on the set of objects recognized by `mp` just in case there is a `rel`-order-preserving function `fn` that embeds objects recognized by `mp` into ACL2's ordinals:

```
(and (implies (mp x)(O-p (fn x)))
     (implies (and (mp x)
                   (mp y)
                   (rel x y))
              (O< (fn x)(fn y))))
```

In ACL2 Version 2.9, `O-p` recognizes the ordinals up to epsilon-0 and `O<` is the well-founded less-than relation on those ordinals.

**Theorem 1** *The following are equivalent for any function with a tail recursive definition like that for* `f`*.*

1. *The recursion satisfies a nonnegative-integer-valued measure conjecture.*
2. *The recursion satisfies a measure conjecture.*
3. *The recursive defining axiom is satisfied by an unique total function.*
4. *The recursion always halts.*

**Proof.** Clearly *1* ⇒ *2*.

*2* ⇒ *3*. Assume the recursion in the definition of `f` satisfies a measure conjecture. Show that any two functions, say `f` and `g`, that satisfy the defining tail recursive axiom for `f` are equal:

Assume `f` and `g` satisfy these equations.

```
(equal (f x)                    (equal (g x)
       (if (test x)                    (if (test x)
           (base x)                        (base x)
           (f (step x))))                  (g (step x))))
```

[The equation involving `g` is what is meant by "`g` satisfies the defining tail recursive axiom for `f`."]

Use the induction suggested by the definition of `f` to prove `(equal (f x)(g x))`. The base case is `(test x)` $\Rightarrow$ `(equal (f x)(g x))`. The induction step, `(not (test x))` $\Rightarrow$ `(equal (f x)(g x))`, follows from the induction hypothesis, `(not (test x))` $\Rightarrow$ `(equal (f (step x))(g (step x)))`.

*3* $\Rightarrow$ *4*. Assume the recursive defining axiom for `f` is satisfied by an unique total function. Now closely follow the construction in Manolios and Moore's `defpun` paper [5]: Define a "clocked" version of `f`.

```
(defun
  f_n (x n)
  (declare (xargs :measure (nfix n)))
  (if (or (zp n) (test x))
      (base x)
      (f_n (step x)(- n 1))))
```

Manolios and Moore use `f_n` to construct a total function that satisfies the recursive defining axiom for `f`. Slightly modify their construction to define two apparently different functions that satisfy the recursive defining axiom for `f`.

Use `defchoose` to let `(n_ch x)` be an `n` such that $(\texttt{test}(\texttt{step}^{\texttt{n}}\texttt{x}))$, if such an `n` exists. The value of `n_ch` is not specified otherwise.

```
(defun g (x)                        (defun h (x)
  (if (test (step^(n_ch x) x))        (if (test (step^(n_ch x) x))
      (f_n x (n_ch x))                    (f_n x (n_ch x))
      nil))                               t))
```

It should be fairly obvious that both `g` and `h` satisfy the defining axiom for `f`. Since there is exactly one function satisfying the defining axiom for `f`, it must be the case that `g` = `h`, which means that $\forall \texttt{x}(\texttt{test}(\texttt{step}^{(\texttt{n\_ch x})}\texttt{x}))$.

*4* $\Rightarrow$ *1*. Assume $\forall x \exists \texttt{n}(\texttt{test}(\texttt{step}^{\texttt{n}}\ \texttt{x}))$. Let `m(x)` be the least nonnegative integer `n` such that $(\texttt{test}(\texttt{step}^{\texttt{n}}\ \texttt{x}))$. Then whenever `(not (test x))`, it is the case that `(< (m (step x))(m x))`.

This theorem suggests one way to show that the famous "$3x+1$" function always terminates on all natural number inputs: It is sufficient to show the defining axiom

```
(equal (3x+1 x)
       (if (<= x 1)
           x
           (3x+1 (if (evenp x)
                     (/ x 2)
                     (+ (* 3 x) 1)))))
```

is satisfied by only one total function on the nonnegative integers. The termination of this function on all nonnegative integer inputs remains an open problem.

The following propositions show how much of **Theorem 1** holds for recursive definitions that may not be tail recursive.

**Proposition 1** *The following are equivalent for any function with a recursive definition.*

*1. The recursion satisfies a nonnegative-integer-valued measure conjecture.*

*2. The recursion satisfies a measure conjecture.*

*4. The recursion always halts.*

**Proof.** Clearly *1* $\Rightarrow$ *2*.

Since all descending chains, of elements related by a well-founded relation, are finite; *2* $\Rightarrow$ *4*.

*4* $\Rightarrow$ *1*. Assume the recursion always halts. Then the Canonical Measure (essentially the minimal stack depth required for computing the value of the function on a given input, using the body of the recursive definition.) described by M. Kaufmann and J S. Moore, in [2], is a nonnegative-integer-valued measure.

**Proposition 2** *The following implications hold for any function with a recursive definition.*
*Each of these*

1. *The recursion satisfies a nonnegative-integer-valued measure conjecture.*

2. *The recursion satisfies a measure conjecture.*

4. *The recursion always halts.*

*implies*

3. *The recursive defining axiom is satisfied by an unique total function.*

**Proposition 3** *The following implications could fail for any function with a recursive definition.*

3. *The recursive defining axiom is satisfied by an unique total function.*

*implies each of these*

1. *The recursion satisfies a nonnegative-integer-valued measure conjecture.*

2. *The recursion satisfies a measure conjecture.*

4. *The recursion always halts.*

**Proof.** The equation

```
(equal (f x)
       (if (f x)
           x
           x))
```

is satisfied by only one total function, namely the identity function, but the recursion suggested by the equation does not terminate nor satisfy any measure conjecture.

**Theorem 2** *Let* `a` *and* `b` *be constants. Suppose that the only constraint on the function* `f` *that mentions* `f` *is the defining tail recursive axiom for* `f`*. If ACL2 can prove* `(equal (f a) b)`*, then ACL2 can also prove, that the recursion for* `f` *terminates on input* `a`*.*

**Proof.** Assume `(equal (f a) b)` is a theorem.

Once more the construction in Manolios and Moore's `defpun` paper [5] is closely followed: Define a "clocked" version, such as `f_n` from above, of `f`. Choose any constant `c` such that $c \neq b$.

Use `f_n` and `c` to construct a total function that satisfies the recursive defining axiom for `f`.

```
(defun fc (x)
  (if (test (step^(n_ch x) x))
      (f_n x (n_ch x))
      c))
```

Once again, it should be fairly obvious that `fc` satisfies the defining axiom for `f`. That is, the following holds.

```
(equal (fc x)
       (if (test x)
           (base x)
           (fc (step x))))
```

By functional instantiation, `(equal (fc a) b)` is a theorem. Since $c \neq b$, it follows from the definition of `fc` that ($\mathtt{test}(\mathtt{step}^{(\mathtt{n\_ch\ a})}\mathtt{a})$).

## Tail Recursive Interpreters

This section starts by closely following a similar section in Manolios and Moore's `defpun` paper [5].

An important class of tail recursive functions consists of the "state machine interpreters" traditionally used in ACL2 to give operational semantics. We consider one such interpreter, called WyoM1. WyoM1 was used at the University of Wyoming while teaching a class, inspired by a similar class at the University of Texas, on formalizing the Java Virtual Machine in ACL2. WyoM1 is very similar to an interpreter known as M2 at UT.

A WyoM1 state is a pair consisting of a call stack and a list of function definitions. The call stack is a stack of frames, each frame corresponding to an activation of some defined function. A frame contains a program counter, the code for the function, bindings for the formal and local variables of the function, and an operand stack. Each function definition contains the name, list of formal arguments, and list of instructions for some function.

Here is the definition for a recursive function `fact` implementing factorial.

```
(defconst *fact-def*
  '(fact (n)
         (load n)       ;;  0
         (ifgt 3)       ;;  1
         (push 1)       ;;  2
         (ret)          ;;  3
         (load n)       ;;  4
         (load n)       ;;  5
         (push 1)       ;;  6
         (sub)          ;;  7
         (call fact)    ;;  8
         (mul)          ;;  9
         (ret)))        ;; 10
```

Let `step` be the single-step state transition function for WyoM1. So `(step s)` is the state produced by executing the instruction indicated by the program counter in the top frame of the call stack of state `s`.

The "clocked" interpreter for WyoM1 is

```
  (defun run (s n)
    (if (zp n)
        s
        (run (step s)(- n 1))))
```

An interpreter without a clock for WyoM1 is given below by `run-w`. `(Run-w s)` runs WyoM1, starting with state `s`, to termination, if a halted state can be reached by repeated steps. The value of `(run-w s)` on states that do not terminate is not specified.

```
  (defun haltedp (s)
    (equal s (step s)))

  (defpun run-w (s)
    (if (haltedp s)
        s
        (run-w (step s))))
```

The interpreter without a clock for WyoM1, `run-w`, can be used to state and prove, in ACL2, the following WyoM1 program correctness result.

First WyoM1 function definitions are given for `sq` which squares its input and `max` which returns the maximum of its two inputs.

```
(defconst *sq-def*
  '(sq (n)
       (load n)
       (dup)
       (mul)
       (ret)))
```

```
(defconst *max-def*
  '(max (x y)
    (load x)
    (load y)
    (sub)
    (ifle 3)
    (load x)
    (ret)
    (load y)
    (ret)))
```

Let `s` be the following state described by specifying its top (and only) frame and list of function definitions.

```
  (modify nil
         :pc 0
         :locals local-vars
         :stack s0
         :program '((load x)      ;; 0
                    (call sq)     ;; 1
                    (call fact)   ;; 2
                    (load x)      ;; 3
                    (call fact)   ;; 4
                    (call sq)     ;; 5
                    (call max)    ;; 6
                    (store y)     ;; 7
                    (halt))       ;; 8
         :defs (list *sq-def*
                     *max-def*
                     *fact-def*)).
```

Let `x` be the value of the variable `'x` in `(locals s)`. If `x` is a nonnegative integer and `s` is run to termination, then WyoM1 ends in the following state described by indicating how the state `s` is modified.

```
  (modify s
          :pc 8
          :locals (bind 'y (MAX (! (SQ x))
                                (SQ (! x)))
                         (locals s)))
```

Here `MAX`, `SQ`, and `!` are ACL2 functions implementing the usual maximum, squaring, and factorial functions. Here is the formal correctness result in ACL2.

```
(defthm prog-is-correct-with-run-w
  (let* ((s (modify nil
                    :pc 0
                    :locals local-vars
                    :stack s0
                    :program '((load x)      ;; 0
                               (call sq)     ;; 1
                               (call fact)   ;; 2
                               (load x)      ;; 3
                               (call fact)   ;; 4
                               (call sq)     ;; 5
                               (call max)    ;; 6
                               (store y)     ;; 7
                               (halt))       ;; 8
                    :defs (list *sq-def*
                                *max-def*
                                *fact-def*)))
         (x (binding 'x (locals s))))
    (implies (and (integerp x)
                  (>= x 0))
             (equal (run-w s)
                    (modify s
                            :pc 8
                            :locals (bind 'y (MAX (! (SQ x))
                                                  (SQ (! x)))
                                           (locals s))))))
  :hints . . .)
```

Remember that `(run-w s)` is not specified for those states `s` for which WyoM1 does not terminate. So for example, how do we know that for the state `s` initially given in the above `defthm` that WyoM1 actually halts and produces the modified state? Could it be that WyoM1 does not halt on `s` and the unspecified value of `(run-w s)` just happens to be the modified state given in the `defthm`?

The meta-theorem, **Theorem 2**, says that if ACL2 can prove `prog-is-correct-with-run-w`, then ACL2 can also prove there is a nonnegative integer `n` such that the statement of this theorem remains true when `(run-w s)` is replaced by `(run s n)`. The proof of **Theorem 2** is carefully followed using `haltedp` for `test`, `identity`, ie., `(identity x)` = `x` for `base`, `step` for `step`, `run` for `step`$^n$, the initial state `s` in the `defthm` for `a`, and the modified state for `b`.

```
(defthm prog-is-correct-with-run
  (let* ((s (modify nil
                    :pc 0
                    :locals local-vars
                    :stack s0
                    :program '((load x)      ;; 0
                               (call sq)     ;; 1
                               (call fact)   ;; 2
                               (load x)      ;; 3
                               (call fact)   ;; 4
                               (call sq)     ;; 5
                               (call max)    ;; 6
                               (store y)     ;; 7
                               (halt))       ;; 8
                    :defs (list *sq-def*
                                *max-def*
                                *fact-def*)))
         (x (binding 'x (locals s)))
         (n (nfix (nbr-steps-to-halt s))))
    (implies (and (integerp x)
                  (>= x 0))
             (equal (run s n)
                    (modify s
                            :pc 8
                            :locals (bind 'y (MAX (! (SQ x))
                                                  (SQ (! x)))
```

```
                                                    (locals s))))))
  :hints . . .)
```

where `nbr-steps-to-halt` is the choice function

```
(defchoose
  nbr-steps-to-halt (n)(s)
  (haltedp (run s n)))
```

## Reflexive Tail Recursion

If `test`, `base`, and `step` are already defined, then the `defpun` construction shows that the equation

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

is satisfiable by some total function. Manolios and Moore also consider the case when (step x) mentions `f`. Equations with nested recursive calls are sometimes called reflexive. Manolios and Moore show [5] that the problem of deciding if a reflexive tail recursive equation is satisfiable by some total function is undecidable. Since the problem is undecidable, there must be cases when no total function satisfies the given reflexive tail recursive equation.

ACL2 can verify the following two theorems.

**Theorem 3** *Let* `c` *be a positive integer and let* `test`*,* `base`*, and* `step` *be total functions such that*

-
  ```
  (implies (test (base x))
           (test x))
  ```

- `base` *and* `step` *commute:*

  ```
  (equal (base (step x))
         (step (base x)))
  ```

- *either the recursion with respect to* $\mathtt{base}^{(-\,\mathtt{c}\,1)} \circ \mathtt{step}$ *and* `test` *always halts OR it never halts when* `x` *satisfies* `(not (test x))`*:*

$$[\forall \mathtt{x} \exists \mathtt{n}(\mathtt{test}([\mathtt{base}^{(-\,\mathtt{c}\,1)} \circ \mathtt{step}]^{\mathtt{n}}\, \mathtt{x}))]$$
$$\vee \quad [\forall \mathtt{x} \forall \mathtt{n}((\mathtt{not}(\mathtt{test}\, \mathtt{x})) \Rightarrow (\mathtt{not}(\mathtt{test}([\mathtt{base}^{(-\,\mathtt{c}\,1)} \circ \mathtt{step}]^{\mathtt{n}}\, \mathtt{x}))))]$$

*Then there is a total function* `f` *that satisfies both the reflexive tail recursive equation*

```
(equal (f x)
       (if (test x)
           (base x)
           (f^c (step x))))
```

*and the simpler tail recursive equation*

```
(equal (f x)
       (if (test x)
           (base x)
           (f (base^(- c 1) (step x)))))
```

**Theorem 4** *Let* `c` *be a positive integer and let* `f`*,* `test`*,* `base`*, and* `step` *be total functions such that*

- `f` *is reflexive tail recursive:*

```
(equal (f x)
       (if (test x)
           (base x)
           (f^c (step x))))
```

-
  ```
  (implies (test (base x))
           (test x))
  ```

- `base` *and* `step` *commute:*

```
(equal (base (step x))
       (step (base x)))
```

- *recursion with respect to* `step` *and* `test` *always halts:*
  $\forall \mathtt{x} \exists \mathtt{n}(\mathtt{test}(\mathtt{step}^{\mathtt{n}}\, \mathtt{x}))$

*Then* `f` *also satisfies the simpler tail recursive equation*

```
(equal (f x)
       (if (test x)
           (base x)
           (f (base^(- c 1) (step x)))))
```

**Corollary 1 (Knuth [1, 3, 4])** *Let* `c` *be a positive integer and let* $\mathtt{a}, \mathtt{b} > 0, \mathtt{d}$ *be real numbers.*

1. *There is a total function on the reals satisfying the reflexive tail recursive equation*

   ```
   (equal (K x)
          (if (> x a)
              (- x b)
              (K^c (+ x d))))
   ```

2. *If* `(< (* (- c 1) b) d)` *then there is an unique function on the reals satisfying the above reflexive tail recursive equation for* `K`*.*

**Proof.** Let `(test x)` be `(> x a)`, `(base x)` be `(- x b)`, and `(step x)` be `(+ x d)`. Then $([\mathtt{base}^{(-\,\mathtt{c}\,1)} \circ \mathtt{step}]\ \mathtt{x})$ is `(+ x d (-(* (- c 1) b)))` and $([\mathtt{base}^{(-\,\mathtt{c}\,1)} \circ \mathtt{step}]^{\mathtt{n}}\ \mathtt{x})$ is `(+ x (* n (+ d (-(* (- c 1) b)))))`. If `(< (* (- c 1) b) d)`, then the recursion with respect to $\mathtt{base}^{(-\,\mathtt{c}\,1)} \circ$ `step` and `test` always halts; otherwise the recursion never halts when `x` satisfies `(not (test x))`. So by **Theorem 3**, there is a total function satisfying the reflexive tail recursive equation for `K`.

If `(< (* (- c 1) b) d)`, then `(> d 0)`, so the recursion with respect to `step` and `test` always halts. Then by **Theorem 4**, K must also satisfy that theorem's simpler recursive equation. Since `(< (* (- c 1)b) d)`, the recursion specified in simpler recursive equation always halts, so by **Theorem 1**, the simpler equation is satisfied by an unique function.

**Corollary 2** *There is an unique function on the reals satisfying the reflexive tail recursive equation for McCarthy's 91 function,*

```
(equal (M91 x)
       (if (> x 100)
           (- x 10)
           (M91 (M91 (+ x 11)))))
```

**Proof.** By the previous Corollary.

## References

[1] J.R. Cowles. Knuth's Generalization of McCarthy's 91 Function. In M. Kaufmann, P. Manolios, and J S. Moore, Editors, Computer-Aided Resoning: ACL2 Case Studies, pages 283–299. Kluwer Academic Press, 2000.

[2] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic, J. Automated Reasoning 26 (2001), 161–203.

[3] D.E. Knuth. Textbook Examples of Recursion. In V. Lifschitz, Editor, Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, pages 207–230. Academic Press, 1991.

[4] D.E. Knuth. Selected Papers on the Analysis of Algorithms. CSLI Publications, Distributed by Cambridge University Press, 2000. Chapter 22 is an update of [3].

[5] P. Manolios and J S. Moore. Partial functions in ACL2, J. Automated Reasoning 31 (2003), 107–127.