Assuming Termination

David Greve
Rockwell Collins Advanced Technology Center
Cedar Rapids, IA
dagreve@rockwellcollins.com

ABSTRACT

A requirement for the admission of a recursive function definition in ACL2 is a proof that the recursion terminates. Establishing termination involves identifying a well-founded relation and an appropriate measure function that decreases according to the well-founded relation with each recursive call. Depending on the domain this process may prove difficult, unnecessary or even impossible. Manolios and Moore introduced a method for admitting tail-recursive function definitions that does not require the specification of a measure. Their method, however, does not produce an induction scheme. We present an extension of their method that enables the admission of arbitrary recursive function definitions and their associated induction schemes augmented with a termination check and justified by a partial measure. We demonstrate the use of this method by defining the tarai function and proving by induction its unwinding under the assumption that it terminates.

Categories and Subject Descriptors

F.3.3 [**Studies of Program Constructs**]: Program and Recursion Scheme; F.4.1 [**Mathematical Logic**]: Recursive Function Theory

General Terms

Recursion, Termination

Keywords

ACL2, Partial Functions, Tail-Recursion, Continuations

1. MOTIVATION

A requirement for the admission of a recursive function definition in ACL2 is a proof that the recursion terminates. Establishing termination involves identifying a well-founded relation and an appropriate measure function that decreases according to the well-founded relation with each recursive call. Depending on the domain this process may prove difficult, unnecessary or even impossible. The difficulty can be experienced by attempting to admit a simple function that does nothing more than count to 10. Termination proofs are unnecessary when reasoning about computations that take place in a fixed finite number of steps, computations such as the symbolic execution of a specific straight line code segment. Finally, termination proofs may be impossible for models of certain programming languages[3, 6], as a general proof for those models would require a solution to the halting problem.

Even without a general termination argument there are still many interesting program properties that can be established under an assumption of termination. Such properties are commonly referred to as *partial correctness* results. We describe an extension of Manolios' and Moore's `defpun` that enables the admission of arbitrary recursive function definitions and their associated induction schemes augmented with a termination check and justified by a partial measure. This scheme allows interesting partial correctness results to be verified inductively. We believe that these techniques will be particularly useful for reasoning about models of operational semantics for programming languages.

The remainder of this paper is organized in the following manner. Section 2 introduces an extension of `defpun`, called `defminterm`, that constructs a partial, well-founded induction scheme (measure) for tail-recursive function definitions under the assumption that the recursion terminates. Section 3 describes how continuations can be used to construct tail-recursive implementations for a specific class of recursive functions and demonstrates that, by admitting such implementations using defminterm, it is possible to prove that the tail-recursive implementations satisfy their original definitional equations when the functions terminate. Section 4 shows how the techniques of Section 3 can be used to admit an arbitrary recursive function and prove that it satisfies its definitional equation under the assumption that it terminates. Section 5 illustrates the admission of the tarai function and proves its unwinding theorem under the assumption that it terminates. Finally, Section 6 concludes the paper and identifies areas for future work.

2. DEFMINTERM

A tail-recursive function is one in which any recursive call of the function along any execution path within the function body appears only as the outermost (last) unconditional operation[2]. Manolios and Moore[5] introduced a method for admitting tail-recursive function definitions without the use

of a measure. Their method, however, does not produce an induction scheme.

The `defminterm` macro leverages the techniques of Manolios and Moore to provide a means of introducing a partial, well-founded induction scheme (measure) for a tail-recursive function under the assumption that the recursion terminates. The termination check is added to the function body and the function is admitted as a simple definition using the partial measure as a justification of well-foundedness. Manolios and Moore present the following as a generic model of tail-recursive functions.

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

Our generic theory also begins with this premise. We define a clocked recursive function, `stn`, with the same step behavior as our target function. We then witness a clock function that will run the clocked recursion to a terminal state, if possible.

```
(defun stn (x n)
  (if (zp n) x
    (stn (st x) (1- n)))))

(defchoose fch (n) (x)
  (test (stn x n)))
```

A companion predicate, `term`, is defined to check the final state of the `stn` function after executing `fch` steps to see if `test` is, in fact, true in the end. Effectively this predicate tells us whether our target function terminates on the given input.

```
(defun term (x) (test (stn x (fch x))))
```

Interestingly, it is possible to prove that `term` satisfies a defining equation with exactly the same recursive structure as our desired equation[1].

```
(defthm open-term
  (equal (term x)
         (if (test x) t
           (term (st x)))))
```

Rather than attempting to witness the desired function directly at this point, as with `defpun`, our approach is to witness a measure function with exactly the same recursive structure as the desired recursive definition. The existence of such a measure function will then allow the desired tail-recursive function to be admitted as a simple definition in ACL2.

[1]We assume throughout that `(test x)` is Boolean

```
(equal (measure x)
       (if (test x) 0
         (1+ (measure (st x)))))
```

Unfortunately the desired measure function is not tail-recursive and the `defpun` methodology works only for tail-recursive functions. It is not difficult, however, to construct a tail-recursive function that has essentially the same behavior as our desired measure function.

```
(equal (measure-tail x r)
       (if (test x) r
         (measure-tail (st x) (1+ r)))))
```

To satisfy this defining equation we repeat the process of introducing a clocked recursion and define the tail-recursive measure function in terms of this recursion evaluated at `fch`, the same clock function defined above.

```
(defun measure-tail-stn (x r n)
  (if (zp n) r
    (measure-tail-stn (st x) (1+ r) (1- n))))

(defun measure-tail (x r)
  (measure-tail-stn x r (fch x)))
```

We then prove that `measure-tail` satisfies its defining equation. Our desired measure function is now defined in terms of this tail-recursive function:

```
(defun measure (x) (measure-tail x 0))
```

The critical step in the proof that this function satisfies the desired defining equation is the following statement that the tail-recursive function commutes over increment:

```
(equal (measure-tail x (1+ r))
       (1+ (measure-tail x r))
```

This proof would be easy if `measure-tail` had an associated induction scheme, but it does not. Completing this proof using partial functions is possible, but it requires an assumption about the termination of the recursion. We add this using the `term` predicate discussed above. Assuming termination, enables us to prove:

```
(defthm open-measure
 (implies
    (term x)
    (equal (measure x)
           (if (test x) 0
             (1+ (measure (st x)))))))
```

Adding the termination predicate to the recursive test of our desired function now enables the admission of the following definition in ACL2:

```
(defun f (x)
  (declare (xargs :measure (measure x)))
  (if (or (not (term x)) (test x)) (base x)
    (f (st x)))))
```

Note that this definition suggests an ACL2 induction scheme.

The `defminterm` macro builds on this generic infrastructure in the same manner as does `defpun`[2]. The theory exported from an event such as:

```
(defminterm f (x)
  (if (test x) (base x)
    (f (st x))))
```

includes the function symbols `f_terminates` and `f_measure` with the same signatures as `f` as well as a definition for `f` augmented with a check for termination and justified by `f_measure`. These functions satisfy the following theorems:

```
(defthm f-property
  (equal (f x)
         (if (or (test x)
                 (not (f_terminates x))
             (base x)
           (f (st x)))))

(defthm f_measure-property
  (implies
    (f_terminates x)
    (equal (f_measure x)
           (if (test x) 0
             (1+ (f_measure (st x)))))))
```

Note that it would be trivial for `defminterm` to witness functions satisfying `f-property` and `f_measure-property` if `f_terminates` always evaluated to `nil`. We can test that this is not the case by proving the following theorem about `f_terminates`:

```
(defthm f_terminates-property
  (equal (f_terminates x)
         (if (test x) t
           (f_terminates (st x)))))
```

This theorem ensures that `f_terminates` is non-trivial. In fact, it guarantees that, if there exists a value of `x` satisfying `(test x)`, there exists an `x` such that `f_terminates` is not `nil`. The fact that the original function, the measure function and the termination function all share the same recursive structure is crucial in inductive proofs where it is necessary to relieve the termination assumption in the inductive hypothesis.

[2]Note that `defminterm` supports a somewhat larger class of function bodies than `defpun`. In particular it supports nested let bindings and multiple tail-recursive calls on different execution branches.

We have described a generic theory that permits the construction of a termination predicate and a partial measure that can be used to admit arbitrary tail-recursive function definitions. We now consider how to extend this principle to arbitrary recursive functions.

3. CONTINUATIONS

Tail recursion is a structural property, a property of how a function is implemented or expressed, not a functional property. Every computable function has a tail-recursive implementation. This follows from the fact that a Turing machine has a tail-recursive implementation and therefore any program implementable on a Turing machine also has a tail-recursive implementation. The proof that such an implementation satisfies a particular defining equation, however, is generally possible only when the function terminates[3]. The `defminterm` macro provides a convenient means of expressing the termination assumption and a partial induction scheme that is valid under that assumption.

Methods exist for transforming generic recursive functions into tail-recursive implementations. It could be argued that the most common such method is compilation[4]. A more semantically self-contained method involves the use of *continuations*[8]. A continuation is a means of saving execution state so that is can be resumed from that point at a later time. Continuations can be used to transform generic recursive functions into tail-recursive functions. A particularly simple continuation can be used on a specific class of reflexive, recursive functions. A reflexive, recursive function is one in which one recursive call is nested within another. Consider the following generic model of a reflexive, recursive function `frr`:

```
(equal (frr x)
       (if (test x) (base x)
         (let ((value (frr (st x))))
           (frr (op x value)))))
```

While the outermost operation in the recursive branch is a tail-recursive call, a recursive call also appears as an inner form where its result is bound to `value` and is subsequently operated upon by `op`. Thus this function is not strictly tail-recursive. It is possible, however, to construct a tail-recursive implementation of this function through the use of a particularly simple continuation.

In general incorporating continuations into a function implementation requires the addition of at least one function argument to use for storage of continuation state (the continuation stack). It also requires the addition of continuation processing functionality. This functionality is functionality not found in the original function that is included solely to store continuations on the continuation stack and to retrieve and evaluate them at appropriate times.

[3]The `open-measure` proof above was one simple illustration of this phenomena.

[4]A microprocessor, like a Turing machine, has a tail-recursive implementation. Therefore a compiled program being evaluated by a microprocessor implementation constitutes a tail-recursive implementation of the program.

In our example a new function argument, `stk`, is introduced as a push-down stack to store continuations. On each recursive branch, `(not (test x))`, the implementation computes the innermost recursive operation just as it normally would. However, at this point continuation storage functionality, `(cons x stk)`, is inserted to postpone the outer operation by pushing a continuation (in this case, just the argument to the function) onto the continuation stack. The base branch is also augmented with continuation retrieval functionality to check for pending operations on the continuation stack, `(consp stk)`, and if they exist, to restore the saved value `(let ((x (car stk)) (stk (cdr stk))) ..)` and complete the computation started in the recursive branch. The binding to `value` has been preserved here to hint at how this transformation might be unwound.

```
(defminterm frr-imp (x stk)
       (if (test x)
          (if (not (consp stk))
              ;; If there are no pending
              ;; continuations, finish
              (base x)
            ;; Otherwise complete the
            ;; innermost recursion and
            ;; then pop the continuation.
            (let ((value (base x))
              (let ((x   (car stk))
                    (stk (cdr stk)))
                ;; Compute outermost call
                (frr-imp (op x value) stk))))
         ;; Compute the innermost call
         ;; and push a continuation.
         (frr-imp (st x) (cons x stk))))
```

Note that the outermost function applications in the two recursive branches are both recursive calls and that there are no other recursive calls within the body of `frr-imp`. `frr-imp` is, therefore, a tail-recursive function. Being tail-recursive we now know that we can witness the defining equation. However, despite the preservation of the `value` binding, it is far from obvious that this is an implementation of our original function. Furthermore, the proof of this fact requires induction which is not provided by `defpun`. Witnessing a tail-recursive implementation using `defminterm`, however, provides an induction scheme which is useful under the additional assumption that the recursion terminates.

The proof of correctness hinges on the following commuting property of `frr-imp`:

```
(equal (frr-imp x (cons y stk))
       (frr-imp (op y (frr-imp x nil)) stk))
```

However, a proof of this property is possible only when `frr-imp` terminates. The commuting property (with an appropriate hypothesis) is an instance of the following generalization which we prove by induction over `(frr-imp x a)` assuming that `(frr-imp_terminates x a)`:

```
(defthm fff-imp-unwind-helper
  (implies
    (frr-imp_terminates x a)
    (equal (frr-imp x (append a (cons y stk)))
           (frr-imp (op y (frr-imp x a)) stk)))

(defthm frr-imp-unwind
  (implies
    (frr-imp_terminates x a)
    (equal (frr-imp x (cons y stk))
           (frr-imp (op y (frr-imp x nil)) stk)))
  :hints (("Goal" :use
            (:instance frr-imp-unwind-helper
              (a nil)))))
```

We now simply define `frr` and `frr_terminates` as:

```
(defun frr (x) (frr-imp x nil))

(defun frr_terminates (x)
  (frr-imp_terminates x nil))
```

And demonstrate that `frr` satisfies our defining equation when it terminates.

```
(defthm reflexive-recursive-f
  (implies
   (frr_terminates x)
   (equal (frr x)
          (if (test x) (base x)
            (let ((value (frr (st x))))
              (frr (op x value)))))
```

We have studied how a specific style of continuation can be used construct tail-recursive implementations that witness the defining equations for a specific class of reflexive, recursive functions under the assumption of termination. We now show how this result can be used to witness the defining equations for arbitrary recursive function definitions under similar assumptions.

4. A GENERIC INTERPRETER

The reflexive, recursive functions considered above were amenable to a particularly simple continuation implementation involving only a stack and primitive storage and retrieval mechanisms. In order to support arbitrary recursive function definitions, however, the continuation mechanism must be substantially more complex. The mechanism we use is so complex that the continuation processing implementation is, itself, recursive. This recursion is *in addition* to the natural recursion of the function being implemented. Nonetheless, we will show that this continuation model ultimately leads to a tail-recursive implementation capable of witnessing an arbitrary recursive function definition.

4.1 The Isolated Interpreter

We begin by isolating the functionality of the generic continuation interpreter from the recursive functionality that we are modeling. A self-contained generic continuation interpreter that can be used to demonstrate this is presented

below. In this function the term `(foo foo-args)` represents a call to the recursive function we are attempting to model. We assume that `foo` already exists to simplify our exposition and to isolate the behavior of the interpreter. We further assume that the function `foo-step` represents behavior specific to the body of the recursive function definition. The behavior of this function will be tailored to program the interpreter for a given application.

```
(defun gen-cont (args pc spec vals)
  (declare (xargs :measure (acl2-count spec)))
  (if (not (consp spec)) (foo-step pc args vals)
    (let ((npc   (caar spec))
          (nspec (cdar spec)))
      (let ((foo-args
              (gen-cont args npc nspec nil)))
        (let ((foo-value (foo foo-args)))
          (let ((vals (acons npc foo-value vals)))
            (gen-cont args pc (cdr spec) vals))))))
```

Explaining the behavior of the generic continuation processor is best done by examining the process used to program it for a particular defining equation. The process begins by deconstructing the body of the recursive function definition. Our methodology views every recursive function call within the body of the defining equation as a continuation cut point. Continuation cut points partition the body of the defining equation into continuation fragments and each continuation fragment is associated with a unique identifier that we call a program counter (`pc`). The innermost boundary of a continuation fragment is either an input variable or the result of a recursive call. The outermost boundary of a continuation fragment is always a recursive call unless that fragment is the final, outermost fragment. In our simple reflexive, recursive example we identified only one continuation cut point, the innermost recursive call, which partitioned the body of the function into two continuation fragments. The innermost boundary of the first continuation fragment was the function argument and the outermost boundary was the innermost recursive call. The innermost boundary of the second continuation fragment was the result of the innermost recursive call and its outermost boundary was the outermost recursive call.

Our generic continuation processor computes the body of a recursive function definition by incrementally executing continuation fragments. The process begins at the inputs of the function and proceeds from the innermost cut points to the outermost cut points, executing one continuation fragment at a time, until reaching the outermost fragment. The continuation processor for our reflexive, recursive example behaved in exactly the same way. However, in general, multiple inner recursive calls contribute values to subsequent recursive calls. This suggests the need to track which continuation is active at a given point in the computation and the ability to select for execution one of several different continuation fragments in response to that information. Our generic continuation processor uses the program counter (the `pc` argument) to track program fragments and `foo-step` to select one for execution.

In our simple reflexive, recursive example we had a single continuation cut point. Multiple continuation points necessitate a means of storing and recalling multiple intermediate continuation results. In our generic continuation processor the value produced by each continuation fragment is associated with the program counter for that fragment and stored in the data binding environment (the `vals` argument). Subsequent access to the value computed by a given continuation fragment is also provided using this binding environment.

A feature present in the generic interpreter but not found in our reflexive recursive example is the `spec` argument. To assist in explaining the `spec` argument, and to make concrete our discussion on continuation fragments, we consider the following defining equation. Note that it contains a range of possible recursive patterns, from `test1`'s simple tail-recursive pattern to the final reflexive, doubly recursive pattern.

```
(equal (foo args)
 (cond
  ((test0 args) (next0a args))
  ((test1 args) (foo (next1a args))
  ((test2 args) (op2a (foo (next2a args))))
  ((test3 args)
   (op3a (foo (op3b (foo (next3a args))))))
  ((test4 args) (op4a (foo (next4a args))
                      (foo (next4b args))))
  (t (op5a (foo (op5b (foo (next5a args))
                      (foo (next5b args))
      )))))
```

Viewing each recursive call as a cut point, we define and label the various cut point fragment. Each fragment is modified by replacing any recursive calls to `foo` on the innermost boundaries of the fragments with references to the value binding environment, employing the program counter associated with the fragment responsible for computing the result. Using these fragments a simple non-recursive function, `(fn-step pc args vals)`, is defined that accepts as input a program counter (`pc`), the argument to `foo` (`args`) and a binding environment (`vals`). This function executes the cut point fragment associated with `pc` to compute either the argument for the recursive call of `foo` constituting the outer boundary of the selected continuation segment or the final result of the function body. In this example, the program counter 9 identifies the outermost continuation fragment and 0 designates the fragment that computes the arguments of the recursive call of `foo` when `(test1 args)` is true, ie: `(next1a args)`.

```
(defun key-val (key vals)
  (cdr (assoc key vals)))

(defund foo-step (pc args vals)
  (case pc
        (0 (next1a args))
        (1 (next2a args))
        (2 (next3a args))
        (3 (op3b (key-val 2 vals)))
```

```
(4 (next4a args))
(5 (next4b args))
(6 (next5a args))
(7 (next5b args))
(8 (op5b (key-val 6 vals)
         (key-val 7 vals)))
(9 (cond
    ((test0 args)
     (next0a args))
    ((test1 args)
     (key-val 0 vals))
    ((test2 args)
     (op2a (key-val 1 vals)))
    ((test3 args)
     (op3a (key-val 3 vals)))
    ((test4 args)
     (op4a (key-val 4 vals)
           (key-val 5 vals)))
    (t (op5a (key-val 8 vals)))))
))
```

The `spec` (or program spec) argument to our generic continuation interpreter specifies which continuation fragments to evaluate and in which order. A program spec is an abstract model of the data dependency relationships between the various continuation fragments. It is implemented as a recursive list data structure whose first element is a program counter (representing a continuation fragment) and whose remaining elements are program specs. A program spec containing only a program counter is a leaf and it depends only on the arguments to the recursive function. A program spec containing other program specs represents a continuation fragment that depends upon the values of other continuation fragments. Program specs are evaluated from the inside out, a fact that drives the recursive structure of `gen-cont`. To more[5] accurately reflect the recursive behavior of the function being modeled, the program spec varies based on the value of `args`. A function for computing the program spec for our labeling of `foo` is defined as follows:

```
(defun foo-spec (args)
  (cond
    ((test0 args) nil)
    ((test1 args) ((0)))
    ((test2 args) ((1)))
    ((test3 args) ((3 (2))))
    ((test4 args) ((4) (5))
    (t            ((8 (6) (7))))))
```

Combining these definitions with the definition of `gen-cont` we can construct an implementation that mimics the behavior of the body of `foo` as follows:

```
(defun foo-body-imp (args)
  (let ((spec (foo-spec args)))
    (gen-cont args 9 spec nil)))
```

We can now prove, through repeated expansion of the definition of `gen-cont`, that this function is equal to the original body of `foo`'s defining equation:

```
(defthm foo-body-imp-proof
  (equal (foo-body-imp args)
    (cond
      ((test0 args) (next0a args))
      ((test1 args) (foo (next1a args))
      ((test2 args)
       (op2a (foo (next2a args))))
      ((test3 args)
       (op3a (foo (op3b (foo (next3a args))))))
      ((test4 args)
       (op4a (foo (next4a args))
             (foo (next4b args))))
      (t
       (op5a (foo (op5b (foo (next5a args))
             (foo (next5b args))
        ))))))
```

This demonstrates that our generic continuation interpreter, `gen-cont`, in conjunction with an appropriate `fn-step` and program spec, can be used to construct a function that implements the behavior of a specific recursive function body. We further claim that this technique is amenable to automation and can be applied to arbitrary recursive function bodies.

4.2 Interpreter Implementation

While our generic interpreter is capable of emulating arbitrary recursive function bodies, it does not constitute a tail-recursive witness for such functions. To accomplish that we must connect the definitions of `foo` and `gen-cont` and do so in a tail-recursive fashion. We begin by developing a tail-recursive implementation of `gen-cont`. Note that `gen-cont`, as defined above, is a reflexive recursive function. We studied in the previous section how a simple continuation can be used to transform reflexive, recursive functions into tail-recursive implementations. Applying these techniques to `gen-cont` and accounting for 4 arguments produces the following results:

```
(defun pop4-stk (stk)
  (let ((top (car stk)))
    (let ((stk (cdr stk)))
      (mv (car top) (cadr top)
          (caddr top) (cadddr top) stk))))

(defun push4-stk (args pc spec vals stk)
  (cons (list args pc spec vals) stk))

(defminterm gen-cont-imp (args pc spec vals stk)
 (if (not (consp spec))
  (let ((value (foo-step pc args vals)))
   (if (consp stk)
    (mv-let (args pc spec vals stk) (pop4-stk stk)
```

[5]The manner in which the generic interpreter implements the body of the recursive function influences the form of the termination predicate. Our implementation identifies recursive guards up to the point of the first (innermost) recursive call. We do not support recursive guards that depend upon the results of previous recursive calls (reflexive recursive guards).

```
    (let ((foo-value (foo value)))
     (let ((vals (acons (caar spec) foo-value
                        vals)))
       (gen-cont-imp args pc (cdr spec) vals stk))))
   value))
 (let ((stk (push4-stk args pc spec vals stk)))
  (gen-cont-imp args (caar spec) (cdar spec)
                nil stk)
    )))
```

As with the example in the previous section, this implementation can be unwound under the assumption that it terminates. The next step is connecting the definitions of `gen-cont-imp` and `foo`. We do this by simply replacing the call of `foo` in the body of `gen-cont-imp` as prescribed by `foo-body-imp` from above and including an `stk` value of `nil`:

```
(equal (foo args)
  (let ((spec (foo-spec)))
    (gen-cont-imp args 9 spec nil nil)))
```

Note that this transformation results in a definition of `gen--cont-imp` that is self-contained (meaning that it does not reference `foo`) but, once again, the definition is reflexive recursive. Once again, however, we can apply continuations to construct a tail-recursive implementation, `gen-cont-imp--imp`. In the interest of brevity we omit the details of this transformation. Suffice it to say that the implementation will have yet another stack and additional logic to detect and apply the continuations from that stack appropriately. However, because it is tail-recursive, it can be admitted using `defminterm` and, under the assumption that it terminates, the continuation can be unwound to prove that it is equivalent to `gen-cont-imp`. The function `foo` can be defined directly in terms of `gen-cont-imp-imp`:

```
(defun foo (args)
 (let ((spec (foo-spec args)))
  (gen-cont-imp-imp args 9 spec nil nil nil)))
```

This completes a logical chain of events associating the definition of `foo` with a tail-recursive function implementing our generic interpreter `gen-cont` that satisfies the original defining equation for `foo`.

5. DEF::UN

The process described above has been automated and implemented in a macro called def::un. The basic form of `gen-cont-imp-imp` remains the same for all recursive function definitions. The only aspects of `gen-cont-imp-imp` specific to the function being modeled are `foo-step` and `foo-spec`. A generic proof that `gen-cont-imp-imp` implements `gen-cont-imp` has been constructed and is functionally instantiated with each new instance. The proof that `gen-cont-imp` implements `gen-cont` is actually skipped and the symbolic simulation proof that `gen-cont` satisfies the recursive function body, which will be different for every function body, is actually performed using `gen-cont-imp`. This entire process is designed to be completely automatic and should be completely transparent to the user.

The theory exported from an event such as:

```
(def::un tarai (x y z)
  (cond
   ((> x y)
     (tarai
       (tarai (1- x) y z)
       (tarai (1- y) z x)
       (tarai (1- z) x y)))
   (t y)))
```

actually includes `tarai` , `tarai_terminates` and `tarai_-measure`. These functions satisfy the following properties:

```
(defthm tarai-property
  (equal (tarai (x y z)
    (cond
     ((and (> x y)
           (tarai_terminates x y z))
      (tarai
        (tarai (1- x) y z)
        (tarai (1- y) z x)
        (tarai (1- z) x y)))
     (t y))))

(defthm tarai_terminates-property
  (equal (tarai_terminates x y z)
    (cond
      ((> x y)
        (and (tarai_terminates (1- x) y z)
             (tarai_terminates (1- y) z x)
             (tarai_terminates (1- z) x y)
             (tarai_terminates
               (tarai (1- x) y z)
               (tarai (1- y) z x)
               (tarai (1- z) x y))))
      (t t))))

(defthm tarai_measure-property
  (implies
    (tarai_terminates x y z)
    (equal (tarai_measure x y z)
      (cond
        ((> x y)
          (+ (tarai_overhead x y z)
             (tarai_measure (1- x) y z)
             (tarai_measure (1- y) z x)
             (tarai_measure (1- z) x y)
             (tarai_measure
               (tarai (1- x) y z)
               (tarai (1- y) z x)
               (tarai (1- z) x y))))
        (t 1)))))
```

Where `tarai_overhead` is a positive natural value that reflects the recursive overhead of the continuation interpreter implementation.

Using the assumption that `tarai` terminates, we can prove the unwinding theorem[1] automatically by induction:

```
(defthm tarai_unwinding
  (implies
    (tarai_terminates x y z)
    (equal (tarai x y z)
           (if (<= x y) y
             (if (<= y z) z
               x)))))
```

While the termination predicate `tarai_terminates` allows users to assume that the recursion terminates for specific inputs, `def::un` also defines a stronger predicate that uses quantification to enable the user to assume that the recursion always terminates. This stronger, nullary predicate is (`tarai_always_terminates`). Note that the termination predicate cannot be trivially `nil`. In fact, any arguments satisfying (`not (> x y)`) will cause the predicate to evaluate to true. We can also prove that tarai terminates on a specific set of constant arguments:

```
(defthm tarai_terminates_6_4_5
  (tarai_terminates 6 4 5)
  :hints (("Goal" :expand
           (:free (x y z)
             (tarai_terminates x y z)))))
```

However, proving that tarai terminates for a more general class of inputs requires the identification of an appropriate inductive measure[7].

```
(include-book "ordinals/lexicographic-ordering"
              :dir :system)

(defun m1 (x y z)
  (declare (ignore z))
  (if (<= x y) 0 1))

(defun m2 (x y z)
  (- (max (max x y) z) (min (min x y) z)))

(defun m3 (x y z)
  (- x (min (min x y) z)))

(defun tarai-measure (x y z)
  (llist (m1 x y z) (m2 x y z) (m3 x y z)))

(defun tarai-open (x y z)
  (if (<= x y) y
    (if (<= y z) z
      x)))

(defun tarai-induction (x y z)
  (declare (xargs :measure (tarai-measure x y z)
                  :well-founded-relation l<))
 (cond
  ((and (integerp x)
        (integerp y)
        (integerp z)
        (> x y))
   (list
    (tarai-induction (tarai-open (1- x) y z)
                     (tarai-open (1- y) z x)
                     (tarai-open (1- z) x y))
    (tarai-induction (1- x) y z)
    (tarai-induction (1- y) z x)
    (tarai-induction (1- z) x y)))
  (t y)))

(defthm tarai_terminates_proof
  (implies
    (and (integerp x)
         (integerp y)
         (integerp z))
    (tarai_terminates x y z))
  :hints (("Goal"
           :induct (tarai-induction x y z)
           :expand (tarai_terminates x y z))))
```

The `def::un` macro does for ACL2 what Alexander Krauss' construction does for Isabelle/HOL[4]. Krauss describes an automated tool based on inductive definitions for admitting partial recursive functions in HOL along with appropriate reasoning tools. In particular, like `def::un`, his technique expresses termination in a uniform manner and provides an induction principle which corresponds to the recursive structure of the admitted function. Krauss' method differs from ours in his use of higher order constructs, in particular the use of the definite description operator and the introduction of inductive sets as least fix-points via the Knaster-Tarski fixed-point theorem. Logically, however, the end results of the different techniques appear very similar. Krauss demonstrates the efficacy of his technique on the zero function:

```
      f 0 = 0
  f(n + 1) = f (f n)
```

He uses the assumption of termination to prove the unwinding theorem and then uses the unwinding theorem to prove termination. Our approach is similar.

```
(def::un f (n)
  (if (zp n) 0
    (f (f (1- n)))))

(defthm f-is-zero
  (implies
    (f_terminates n)
    (equal (f n) 0)))

(defun f-induction (n)
  (if (zp n) n
    (f-induction (1- n))))

(defthm f-termination_proof
  (f_terminates n)
  :hints (("Goal" :induct (f-induction n)
           :expand (f_terminates n))))
```

6. CONCLUSION

A method has been presented that extends the techniques employed in `defpun` to enable the admission of arbitrary recursive function definitions and their associated induction schemes augmented with a termination check and justified by a partial measure. The method has been implemented as a macro in ACL2 called `def::un` and this macro has been employed to define the reflexive, multiply recursive tarai function and prove by induction its unwinding theorem under the assumption that it terminates. The methodology also enabled us to separate the admission of the tarai function from a proof of its termination. We anticipate this technique being particularly useful for reasoning about models of operational semantics for programming languages.

7. REFERENCES

[1] Tom Bailey and John Cowles. Knuth's generalization of takeuchi's tarai function: Preliminary report. In *ACL2 Workshop 2000*, October 2000. available as University of Texas Dept. of CS TR 00-29.

[2] John Cowles and Ruben Gamboa. Contributions to the theory of tail recursive functions. In *Fifth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 '04)*, November 2004.

[3] John Cowles, David Greve, and William Young. The while-language challenge: First progress. In *ACL2 Workshop 2007*, November 2007.

[4] Alexander Krauss. Partial recursive functions in higher-order logic. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer, 2006.

[5] Panagiotis Manolios and J Strother Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31:2003, 2003.

[6] John Matthews. Deeply embedding cryptol in ACL2: A challenge problem, 2005. ACL2 Theorem Proving Seminar.

[7] J. Strother Moore. A mechanical proof of the termination of takeuchi's function. *Inf. Process. Lett.*, 9(4):176–181, 1979.

[8] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, 1980.