Assuming Termination

May 11, 2009

Rockwell Collins

Advanced Technology Center

Recursive Function definition in ACL2

- Requires a proof of termination
  - Identify a well-founded relation
  - And a measure that decreases with each recursive call
- Depending on domain
  - Difficult, Unnecessary or Impossible
- Defpun
  - Manolios and Moore
  - Admit tail-recursive definitions w/out measure
  - Does not provide an induction scheme

What was our objective?

- Admit functions without proving (by assuming) termination
- Prove properties about those functions inductively

Defminterm

- Leverages technique of Manolios and Moore
  - To introduce a partial measure
  - Under the assumption the recursion terminates

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

Defminterm: Termination Predicate

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

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

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

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

Defminterm: Partial Measure

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

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

(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)))

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

Defminterm: Partial Measure Proof

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

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

Defminterm

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

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

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

Reflexive Recursive Continuations

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

```
(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))))
```

```
(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)))
```

Reflexive Recursive Continuations

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

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

(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)))))))
```

Generic Recursive Functions

```
(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))))))))
```

- Pretend (foo x) already exists .. implement the body

The Generic Interpreter

```
(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)))))))
```

Programming the Generic Interpreter

```
(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)))))))
```

```
(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))))))
```

Programming the Generic Interpreter (Cont.)

```
(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))))))

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

Generic Interpreter Proof

```
(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))
             ))))))
```

Generic Interpreter Implementation

```
(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)
          (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))))
```

Apply Continuation Transformation ..

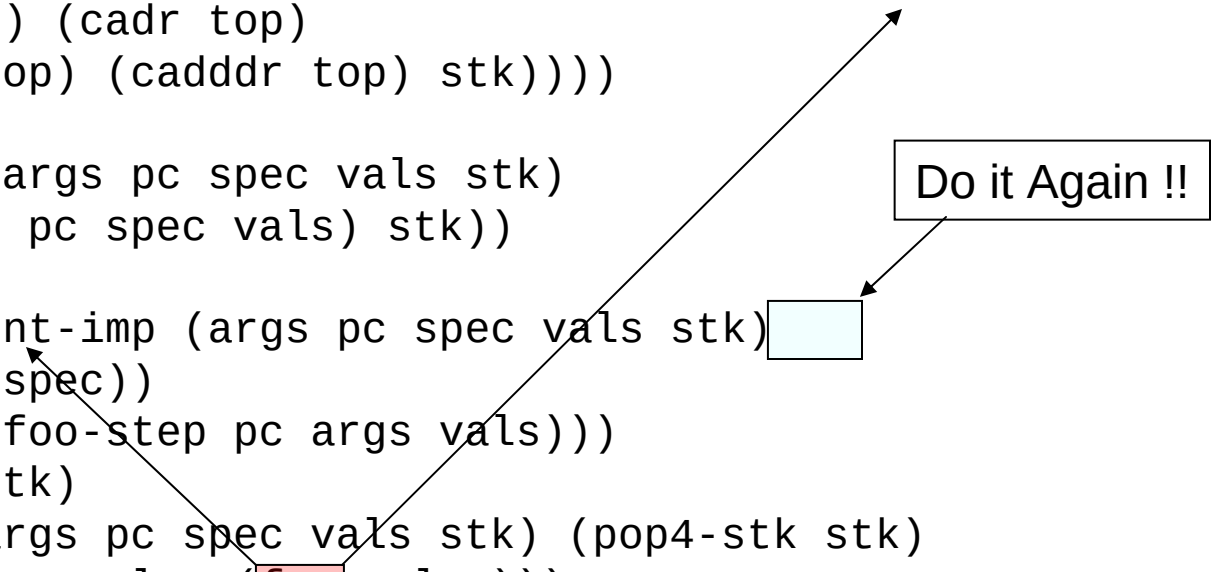
Generic Interpreter Implementation (Cont.)

```
(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)
          (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))))

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



Def::un Implementation

- Analyze Body of Recursive Function
  - Generate “foo-step” and “foo-spec”
  - Programs Generic Interpreter : gen-cont-imp-imp
  - A fully tail-recursive implementation!
- Unwind Generic Interpreter implementation (twice!)
  - Functional Instantiation (Same Every Time)
- Prove Generic Interpreter Implements body
  - Symbolic Simulation (Different Each Time)
- Actually Provides
  - Termination Predicate
  - Partial Measure

What kinds of problems can it solve?

- Admit Arbitrary Recursive Functions
  - Termination Predicate
  - Partial Measure

```
(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)))
```

- Support Proof by induction under termination assumption

Termination Predicate

```
(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))))
```

Partial Measure

```
(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)))))
```

Tarai Unwinding 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)))))
```

J's Tarai Measure

```
(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)))
```

J's Tarai Induction

```
(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)))
```

Proof of tarai_termination

```
(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))))
```

Actually, it always terminates

```
(defun m0 (x y z)
  (unique-nonnumbers x y z))

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

(defun m2 (x y z)
  (let ((rx (realpart x))
        (ry (realpart y))
        (rz (realpart z)))
    (- (max (max rx ry) rz) (min (min rx ry) rz))))

(defun m3 (x y z)
  (let ((rx (realpart x))
        (ry (realpart y))
        (rz (realpart z)))
    (- rx (min (min rx ry) rz))))

(defun m4 (x y z)
  (let ((ix (imagpart x))
        (iy (imagpart y))
        (iz (imagpart z)))
    (- ix (min (min ix iy) iz))))

(defun tarai-measure (d x y z)
  (let ((m0 (m0 x y z)))
    (let ((x (* d x)) (y (* d y)) (z (* d z)))
      (llist m0 (m1 x y z) (m2 x y z) (m3 x y z) (m4 x y z)))))
```

How could it be better?

- Improve termination computation
- Drop in Replacement for defun
- Executable definition
  - With executable termination guard

Something Like This ..

```
(mutual-recursion

  (defun ack (m n)
    (declare (xargs :guard (ack_terminates m n))
             (type integer m n))
    (if (= m 0) (1+ n)
      (if (and (> m 0) (= n 0)) (ack (1- m) n)
        (ack (1- m) (ack m (1- n))))))

  (defun ack_terminates (m n)
    (declare (type integer m n))
    (if (= m 0) (1+ n)
      (if (and (> m 0) (= n 0)) (ack_terminates (1- m) n)
        (and (ack_terminates m (1- n))
             (ack_terminates (1- m) (ack m (1- n)))))))
  )
```

Conclusion

- Developed an ACL2 book providing def::un
  - “replacement” for defun
  - Provides Termination predicate and Partial Measure
  - Supports inductive proofs (under assumption of termination)
- Demonstrated on Tarai
- Many enhancements possible
  - Clean up namespace clutter, re-architect library
  - Fix (weaken) termination predicate
  - Executable definition
    - with executable termination guard?