Defpun, Termination and Continuations

David Greve
November, 2007

Defminterm

- Defpun
  - Partial Function Specification of Tail-Recursive Functions
    - No need to prove termination
  - Proofs
    - Require Termination
    - Concrete : Symbolic Simulation
    - Implicit : Assumptions

```
(defpun foo (x)
  (if (done x) (base x)
    (foo (step x))))
```

```
(equal (foo x)
  (if (done x) (base x)
    (foo (step x))))
```

- Defminterm
  - Extends Defpun Concept
    - Defines Termination ..
    - Defines Measure ..
    - .. as partial functions
  - Uses these to **defun** function
  - Proofs
    - Explicitly Assume Termination
    - Employ Induction
    - Scheme Suggested by definition

```
(defminterm foo (x)
  (if (done x) (base x)
    (foo (step x))))
```

```
(defun foo (x)
  (declare (xargs :measure
                (foo_measure x)))
  (if (and (foo_terminates x)
           (not (done x)))
      (foo (step x))
    (base x)))
```

Termination

```
(defun foo-pun-stn (x n)
   (if (zp n) x
     (foo-pun-stn (step x) (1- n))))

(defchoose foo-pun-fch (n) (x)
  (done (foo-pun-stn x n)))
```

Standard defpun mechanism

```
(defun foo_terminates (x)
  (done (foo-pun-stn x (foo-pun-fch x))))

(defthm foo_terminates_prop
     (and
       (implies (not (done x))
                    (iff (foo_terminates x)
                         (foo_terminates (step x))))
       (implies (done x)
                   (foo_terminates x)))
```

Termination

```
(defun foo-pun-stn (x n)
   (if (zp n) x
     (foo-pun-stn (step x) (1- n))))

(defchoose foo-pun-fch (n) (x)
  (done (foo-pun-stn x n)))

(defun foo_terminates (x)
  (done (foo-pun-stn x (foo-pun-fch x))))

(defthm foo_terminates_prop
     (and
       (implies (not (done x))
                    (iff (foo_terminates x)
                         (foo_terminates (step x))))
       (implies (done x)
                   (foo_terminates x)))
```

Not function, termination.

Termination

```
(defun foo-pun-stn (x n)
   (if (zp n) x
     (foo-pun-stn (step x) (1- n))))

(defchoose foo-pun-fch (n) (x)
  (done (foo-pun-stn x n)))

(defun foo_terminates (x)
  (done (foo-pun-stn x (foo-pun-fch x))))
```

```
(defthm foo_terminates_prop
     (and
       (implies (not (done x))
                     (iff (foo_terminates x)
                          (foo_terminates (step x))))
       (implies (done x)
                    (foo_terminates x)))
```

Opens in a nice way, following recursion.

Measure

```
(defun foo_measure-stn (x r n)
  (if (or (zp n) (done x)) r
    (foo-xun-stn (step x) (1+ r) (1- n))))
```

Defined in a similar manner.

```
(defun foo_measure (x)
  (foo_measure-stn x 0 (foo-pun-fch x)))

(defthm natp_measure
  (natp (foo_measure x)))

(defthm open-foo_measure
   (implies (foo_terminates x)
                 (equal (foo_measure x)
                           (if (done x) 0
                             (1+ (foo_measure (step x)))))))
```

Measure

```
(defun foo_measure-stn (x r n)
  (if (or (zp n) (done x)) r
    (foo-xun-stn (step x) (1+ r) (1- n))))

(defun foo_measure (x)
  (foo_measure-stn x 0 (foo-pun-fch x)))

(defthm natp_measure
  (natp (foo_measure x)))

(defthm open-foo_measure
   (implies (foo_terminates x)
               (equal (foo_measure x)
                          (if (done x) 0
                            (1+ (foo_measure (step x)))))))
```

Defined in conjunction with termination.

Measure

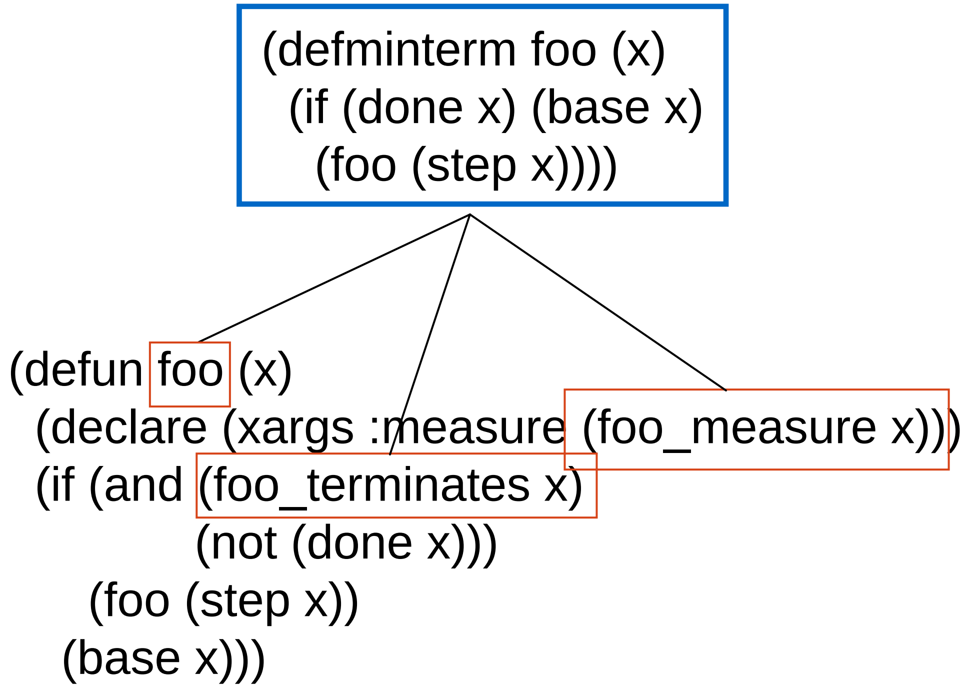
```
(defun foo_measure-stn (x r n)
  (if (or (zp n) (done x)) r
    (foo-xun-stn (step x) (1+ r) (1- n))))

(defun foo_measure (x)
  (foo_measure-stn x 0 (foo-pun-fch x)))

(defthm natp_measure
  (natp (foo_measure x)))

(defthm open-foo_measure
   (implies (foo_terminates x)
                 (equal (foo_measure x)
                             (if (done x) 0
                               (1+ (foo_measure (step x)))))))
```

Measure

```
(defun foo_measure-stn (x r n)
  (if (or (zp n) (done x)) r
    (foo-xun-stn (step x) (1+ r) (1- n))))

(defun foo_measure (x)
  (foo_measure-stn x 0 (foo-pun-fch x)))

(defthm natp_measure
  (natp (foo_measure x)))

(defthm open-foo_measure
   (implies (foo_terminates x)
                (equal (foo_measure x)
                             (if (done x) 0
                               (1+ (foo_measure (step x)))))))
```

Operation commutes when recursion terminates.

Defminterm



```
(defminterm foo (x)
  (if (done x) (base x)
    (foo (step x))))
```

```
(defun foo (x)
  (declare (xargs :measure (foo_measure x)))
  (if (and (foo_terminates x)
           (not (done x)))
      (foo (step x))
    (base x)))
```

Defminterm

```
(defminterm foo (x)
  (if (done x) (base x)
    (foo (step x))))
```

```
(defun foo (x)
  (declare (xargs :measure (foo_measure x)))
  (if (and (foo_terminates x)
             (not (done x)))
      (foo (step x))
    (base x)))
```

```
(:induction foo)
```

Tail-Recursion and Continuations

Tail Recursion: Not a property of a function, a property of an implementation.

```
(equal (foo arg)
        (if (done arg) (base arg)
         (add arg (foo (step arg)))))
```

Not Tail Recursive

Tail-Recursion and Continuations

```
(equal (foo arg)
       (if (done arg) (base arg)
         (add arg (foo (step arg)))))

(defminterm foo-stk (arg stk)
  (if (and (done arg) (not (consp stk))) (base arg)
    (if (done arg) (let ((value (base arg)))
                     (let ((arg (car stk)))
                       (add arg value)))
      (foo-stk (step x) (cons arg stk)))))
```

Continuation enables tail recursion.

Tail-Recursion and Continuations

```
(equal (foo arg)
       (if (done arg) (base arg)
         (add arg (foo (step arg)))))

(defminterm foo-stk (arg stk)
  (if (and (done arg) (not (consp stk))) (base arg)
    (if (done arg) (let ((value (base arg)))
                     (let ((arg (car stk)))
                       (add arg value)))
      (foo-stk (step x) (cons arg stk)))))
```

Continuation enables tail recursion.

Tail-Recursion and Continuations

```
(equal (foo arg)
           (if (done arg) (base arg)
             (add arg (foo (step arg)))))

(defminterm foo-stk (arg stk)
  (if (and (done arg) (not (consp stk)) (base arg)
    (if (done arg) (let ((value (base arg)))
                            (let ((arg (car stk)))
                              (add arg value)))
       (foo-stk (step x) (cons arg stk))))
```

Continuation enables tail recursion.

Tail-Recursion and Continuations

```
(equal (foo arg)
           (if (done arg) (base arg)
             (add arg (foo (step arg)))))

(defminterm foo-stk (arg stk)
  (if (and (done arg) (not (consp stk)) (base arg)
    (if (done arg) (let ((value (base arg)))
                         (let ((arg (car stk)))
                           (add arg value)))
       (foo-stk (step x) (cons arg stk))))
```

Continuation enables tail recursion.

Tail-Recursion and Continuations

```
(equal (foo arg)
           (if (done arg) (base arg)
             (add arg (foo (step arg)))))

(defminterm foo-stk (arg stk)
  (if (and (done arg) (not (consp stk)) (base arg)
    (if (done arg) (let ((value (base arg)))
                            (let ((arg (car stk)))
                              (add arg value)))
       (foo-stk (step x) (cons arg stk))))

(defthm foo-stk-prop
  (implies
    (foo-stk_terminates arg stk)
    (equal (foo-stk_terminates arg (cons x stk))
             (add x (foo-stk_terminates arg stk)))))
```

Tail-Recursion and Continuations

```
(equal (foo arg)
           (if (done arg) (base arg)
             (add arg (foo (step arg)))))

(defminterm foo-stk (arg stk)
  (if (and (done arg) (not (consp stk)) (base arg)
    (if (done arg) (let ((value (base arg)))
                        (let ((arg (car stk)))
                          (add arg value)))
       (foo-stk (step x) (cons arg stk))))

(defthm foo-stk-prop ◄──────── (:induction foo-stk)     Proof by
  (implies                                               Induction
    (foo-stk_terminates arg stk)
    (equal (foo-stk_terminates arg (cons x stk))
               (add x (foo-stk_terminates arg stk)))))
```

Complex Recursions

```
(defun foo (arg)
  (if (done arg) (base arg)
    (add arg (foo (foo (step arg))))))

(defun foo-stk (arg stk)
  (if (and (done arg) (not (consp stk))) (base arg)
    (if (base arg) (let ((value (base arg)))
                              (met ((pc arg stk) (pop-stk stk))
                                      (cond
                                       ((equal pc :first)
                                        (foo value (push :second arg stk)))
                                       (t
                                        (add arg value)))))
      (foo (step arg) (push :first arg stk)))))
```

Complex Recursions

```
(defun foo (arg)
  (if (done arg) (base arg)
    (add arg (foo (foo (step arg))))))

(defun foo-stk (arg stk)
  (if (and (done arg) (not (consp stk))) (base arg)
    (if (base arg) (let ((value (base arg)))
                          (met ((pc arg stk) (pop-stk stk))
                               (cond
                                ((equal pc :first)
                                 (foo value (push :second arg stk)))
                                (t
                                 (add arg value)))))
      (foo (step arg) (push :first arg stk)))))
```

Unwinding Continuations

```
(defun foo_terminates (arg)
  (foo-stk_terminates arg nil))

(defun foo (arg)
  (if (foo_terminates arg)
      (foo-stk arg nil)
    (base arg)))

(defthm foo_def
  (equal (foo arg)
             (if (and (not (base arg))
                      (foo_terminates arg)
                 (add arg (foo (step arg)))
               (base arg))))
```

While implementation

```
(defminterm run-stk (stmt mem stk)
 (if (and (base stmt mem) (not (consp stk)))
     (base-computation stmt mem)
   (if (base stmt mem)
           (let ((mem (base-computation stmt mem)))
             (run-stk (car stk) mem (cdr stk)))
     (case (op stmt)
           (if       (if (zerop (evaluate (arg1 stmt) mem))
                              (run-stk (arg2 stmt) mem stk)
                            (run-stk (arg3 stmt) mem stk)))
           (while    (run-stk (arg2 stmt) mem (cons stmt stk)))
           (sequence (run-stk (arg1 stmt) mem (cons (arg2 stmt) stk)))))))

(defun run (x y)
  (run-stk x y nil))
```

While solution

```
(defthm run_spec
  (equal (run stmt mem)
                (if (run_terminates stmt mem)
                    (case (op stmt)
                      (skip     (run-skip stmt mem))
                      (assign   (run-assignment stmt mem))
                      (if       (if (zerop (evaluate (arg1 stmt) mem))
                                         (run (arg2 stmt) mem)
                                       (run (arg3 stmt) mem)))
                      (while    (if (zerop (evaluate (arg1 stmt) mem))
                                         mem
                                       (run stmt
                                                 (run (arg2 stmt) mem))))
                      (sequence (run (arg2 stmt)
                                          (run (arg1 stmt) mem)))
                      (otherwise mem))
                  mem))
  :rule-classes nil)
```

Future Work

- Continuations Suggest General Solution
  - “Compile” spec into tail-recursive implementation
  - Proove unwinding theorem
- Anticipated Result
  - Admit arbitrary function specifications
    - Under assumption of termination
      - Can we do better?
  - Executable body
    - Matching spec exactly
  - Termination predicate
    - Use as an assumption
    - Prove under appropriate conditions
  - Measure
    - Admit similar recursive schemes
  - Induction
    - Enable interesting proofs