Trampolining Architectures

Steven E. Ganz
Blue Vector Systems

Daniel P. Friedman
Indiana University

Mitchfest
August 23, 2009

Computations are Represented with Monads

Multithreaded Systems Shouldn’t have to be “One Size Fits All”

- Multithreaded systems are not all equal
- But they may all have something in common
- Trampolining architectures bring out the commonality

What is Trampolined Style?

- A transformation of programs
- Inserts *bounce* forms to interrupt threads
- Threads are interchanged by a scheduler

Factorial Trampolined

- Given an initial procedure:

```
(define fact
   (lambda (n)
      (if (zero? n)
         1
         (* n (fact (sub1 n))))))
```

- the trampolined version is:

```
(define fact-tramp
  (lambda (n)
    (if (zero? n)
        (unit 1)
        (m-let (v ((bounce
                    (lambda ()
                      (fact-tramp (sub1 n)))))))
          (unit (* n v))))))
```

- and it can be run as

```
(trampoline
   (lambda ()
      (fact-tramp m)))
```

What is a Trampolining Architecture?

- A framework for providing powerful and efficient non-cooperative multiprocessing
- Provides an implementation for programs that have been converted to trampolined style

Trampolining Architecture

Defines the operators:

- unit
- extend

  (m-let (?var ?rhs) ?body) $\Rightarrow$

  ((extend (lambda (?var) ?body)) ?rhs)
- bounce
- trampoline

The Simplest Architecture: Pogo

```
(define-record done (value))
(define-record doing (thunk))

(define unit make-done)
(define bounce
  (lambda (thunk)
    (lambda ()
      (make-doing thunk))))
```

```
(define extend
  (lambda (recvr)
    (lambda (comp)
      (record-case comp
        (done (value) (recvr value))
        (doing (thunk)
          (make-doing
            (compose (extend recvr)
              thunk)))))))
```

```
(define trampoline
  (lambda (thunk)
    (letrec ((tramp
              (lambda (thread)
                (record-case thread
                 (done (value)
                  (stream-cons value
                   (make-empty-stream)))
                 (doing (thunk)
                  (tramp (thunk)))))))
      (tramp (make-doing thunk)))))
```

- (trampoline (lambda () (fact-tramp 5)))

[120]

Dynamic Threads

- (trampoline
  (lambda ()
    (spawn
      (lambda () (fact-tramp -1))
      (lambda () (fact-tramp 5)))))

[120 ***]

```
(define unit (compose unit_List make-done))

(define bounce
   (lambda (thunk)
      (lambda ()
         (unit_List (make-doing thunk)))))
```

```
(define extend
  (lambda (recvr)
    (extend_List
      (lambda (thread)
        (record-case thread
          (done (value) (recvr value))
          (doing (thunk)
           ((bounce
              (compose (extend recvr) thunk)))))))))
```

```
(define die (lambda () '()))

(define spawn
   (lambda (thunk1 thunk2)
      (list (make-doing thunk1)
            (make-doing thunk2))))
```

```
(define fib-asynch
   (lambda (n)
      (if (< n 2)
        (begin
           (set! acc (add1 acc))
           (die))
        (spawn
           (lambda () (fib-asynch (- n 1)))
           (lambda () (fib-asynch (- n 2)))))))
```

- (define acc 0)

- (stream-car
  (trampoline
    (lambda ()
      (spawn
        (lambda () (fact-tramp 9))
        (lambda () (fib-asynch 10))))))

362880

- acc

89

Fair Threads

```
(define-record multi (siblings))

(define unit make-done)

(define bounce
   (lambda (thunk)
      (lambda ()
         (make-doing thunk))))
```

```
(define extend
  (lambda (recvr)
    (Y (lambda (p)
         (lambda (thread)
           (record-case thread
             (done (value) (recvr value))
             (doing (thunk)
               (make-doing
                 (compose (extend recvr) thunk)))
             (multi (siblings)
               (make-multi ((List p) siblings)))))))))
```

- (define acc 0)

- (stream-car
  (trampoline
    (lambda ()
      (spawn
        (lambda () (fact-tramp 9))
        (lambda () (fib-asynch 10))))))

362880

- acc

0

Recycling Threads

```
(define unit
  (lambda (value)
    (lambda (thread)
      (to-done! thread)
      (done-value-set! thread value)
      thread)))

(define bounce
  (lambda (thunk)
    (lambda ()
      (lambda (thread)
        (to-doing! thread)
        (doing-proc-set! thread thunk)
        thread))))
```

```
(define extend
  (lambda (recvr)
    (B (Y (lambda (p)
            (lambda (thread)
              (cond
                ((done? thread)
                 (let ((value (done-value thread)))
                   (to-doing! thread)
                   (doing-proc-set! thread
                     (lambda () (recvr value)))))
                ((doing? thread)
                 (doing-proc-set! thread
                   (compose (extend recvr) (doing-proc thread))))
                ((multi? thread)
                 (multi-sibs-set! thread ((List p) (multi-sibs thread)))))
              thread))))))
```

```
  (compose (extend recvr) unit)
= (lambda (value)
    (lambda (thread)
      ((Y (lambda (p)
            (lambda (thread)
              (cond
                ((done? thread) (let ((value (done-value thread)))
                                  (to-doing! thread)
                                  (doing-proc-set! thread (lambda () (recvr value)))))
                ((doing? thread) (doing-proc-set! thread
                                   (compose (extend recvr) (doing-proc thread))))
                ((multi? thread) (multi-sibs-set! thread ((List p) (multi-sibs thread)))))
              thread)))
       (begin (to-done! thread)
              (done-value-set! thread value)
              thread))))
=  (lambda (value)  ((bounce (lambda () (recvr value)))))
!= (lambda (value) (recvr value))
=  recvr
```

So What is a Trampolining Architecture?

$\approx \triangleq = \cup \{$(bounce, id), (id, bounce)$\}$

$\Rightarrow$

[$\forall$*recvr*.(compose (extend *recvr*) unit) $\approx$ *recvr* $\wedge$

(extend unit) $\approx \mathrm{id}_{M\rho\alpha} \wedge$

$\forall$*f, g*.(extend (compose (extend *f*) *g*)) $\approx$
(compose (extend *f*) (extend *g*)) $\wedge$

$\forall$*comp,recvr.*
(compose (extend *recvr*) (bounce (lambda () *comp*))) $\approx$
(bounce (lambda () ((extend *recvr*) *comp*)))]

$\wedge$

$\exists$ *i* >0.(compose trampoline bounce^i) = trampoline

Communicating Threads

See the paper

Logic Programming

```
(define unit (compose unit_List make-done))

(define bounce
   (lambda (goal)
      (lambda (subst)
         (unit_List (make-doing goal subst)))))
```

```
(define interleave
  (lambda (recvr)
    (extend_List
      (lambda (thread)
        (record-case thread
          (done (subst) (recvr subst))
          (doing (goal subst)
            ((bounce (compose (interleave goal) recvr))
              subst)))))))
```

```
(define succeed unit)
(define fail (lambda (subst) '()))

(define any
  (lambda (goal1 goal2)
    (lambda (subst)
      (list (make-doing goal1 subst)
            (make-doing goal2 subst)))))

(define all
  (lambda (goal1 goal2)
    (compose (interleave goal2) goal1)))
```

```
(define trans
  (lambda (op)
    (lambda (goal)
      (op goal
          (bounce (lambda (subst)
                    (((trans op) goal) subst)))))))

(define any+ (trans any))
(define all+ (trans all))

(rewrites-as (gletrec ((x g1) …) g2)
             (letrec ((x (bounce g1)) …) g2))
```

- (trampoline (any+ succeed))

[() ...]

- (trampoline (all+ fail))

[]

- (trampoline (any (any+ fail) succeed))

[()]

- (trampoline (all (any+ fail) fail))

[]

- (trampoline (all (any+ succeed) fail))

[]

Trampolining and Reflection

- bounce provides Reification

```
(bounce
  (lambda (subst)
      (if (pred subst) comp1 comp2))
  subst)
```

- For reflecting, translate as “simple”

```
(m-let (newSubst (modifySubst subst))
   (let ((subst newSubst))
      comp))
```

So What is a Trampolining Architecture?

$\approx \triangleq = \cup \{$(bounce, id), (id, bounce)$\}$

$\Rightarrow$

$[\forall$*recvr*.(compose (extend *recvr*) unit) $\approx$ *recvr* $\wedge$

(extend unit) $\approx \mathrm{id}_{M\rho\alpha} \wedge$

$\forall$*f, g*.(extend (compose (extend *f*) *g*)) $\approx$
(compose (extend *f*) (extend *g*)) $\wedge$

$\forall \bar{x}$, *comp,recvr.*
(compose (extend *recvr*) (bounce (lambda ($\bar{x}$) *comp*))) $\approx$
(bounce (lambda ($\bar{x}$) ((extend *recvr*) *comp*)))]

$\wedge$

$\exists$ *i* >0.(compose trampoline bounce^i) = trampoline

Multithreaded Computations are Represented with “Relative” Monads

Most Relevant References

- Trampolining
  (Filinski 1999; Claessen 1999;
  Ganz, Friedman, Wand 1999; Harrison 2006)
- Small Bisimulations (used in proofs)
  (Koutavas, Wand 2006)
- Fairness in Scheduling (Nagle 1987)
- Interleaving in Logic Programs
  (Kiselyov, Shan, Friedman, Sabry 2005)
- Exposing Implementation Through Reflection
  (Smith 1982; Friedman, Wand 1984)
- First-class Substitutions (Pientka 2008)

Conclusion

Trampolining Architectures:

- can provide useful and efficient implementations of multithreaded systems
- can provide a form of thread-level reflection
- need not be monads in the category of the programming language, but are closely related to monads