

# Continuations in Ruby

Jeremy D. Frens

Calvin College  
Computer Science Department

4 December 2007

## Copyright Notice

*This work is licensed under the Creative Commons Attribution-Share Alike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.*

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

# Definitions

- A *continuation* is...

# Definitions

- A *continuation* is...
  - ...whatever's next to compute.

# Definitions

- A *continuation* is...
  - ...whatever's next to compute.
  - ...whatever's stacked up (on the runtime stack).

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

## Example

```
class Integer
  def factorial
    if self == 0
      1
    else
      self * (self-1).factorial
    end
  end
end
```

Compute 5.factorial.

```
5 * 4.factorial
```

## Example

```
class Integer
  def factorial
    if self == 0
      1
    else
      self * (self-1).factorial
    end
  end
end
```

Compute 5.factorial.

5 \* 4 \* 3.factorial

## Example

```
class Integer
  def factorial
    if self == 0
      1
    else
      self * (self-1).factorial
    end
  end
end
```

Compute 5.factorial.

5 \* 4 \* 3 \* 2.factorial

## Example

```
class Integer
  def factorial
    if self == 0
      1
    else
      self * (self-1).factorial
    end
  end
end
```

Compute 5.factorial.

```
5 * 4 * 3 * 2 * 1.factorial
```

## Example

```
class Integer
  def factorial
    if self == 0
      1
    else
      self * (self-1).factorial
    end
  end
end
```

Compute 5.factorial.

5 \* 4 \* 3 \* 2 \* 1 \* 0.factorial

# Continuation Passing Style (CPS)

- The programmer takes explicit control of the continuation.

## Continuation Passing Style (CPS)

- The programmer takes explicit control of the continuation.
- Events in the future are passed as an explicit parameter.

## Continuation Passing Style (CPS)

- The programmer takes explicit control of the continuation.
- Events in the future are passed as an explicit parameter.
- Can use whatever datastructure you want—procedures, blocks, hand-crafted.

## Factorial CPSed

```
class Integer
  def factorial_cps(k = lambda { |v| v })
    if self == 0
      k.call(1)
    else
      (self-1).factorial_cps(lambda { |v|
        k.call(self * v)
      })
    end
  end
end
```

In terms of the stack, `5.factorial_cps` is now:

```
5.factorial_cps(???)
```

## Factorial CPSed

```
class Integer
  def factorial_cps(k = lambda { |v| v })
    if self == 0
      k.call(1)
    else
      (self-1).factorial_cps(lambda { |v|
        k.call(self * v)
      })
    end
  end
end
```

In terms of the stack, `5.factorial_cps` is now:

```
4.factorial_cps(???)
```

## Factorial CPSed

```
class Integer
  def factorial_cps(k = lambda { |v| v })
    if self == 0
      k.call(1)
    else
      (self-1).factorial_cps(lambda { |v|
        k.call(self * v)
      })
    end
  end
end
```

In terms of the stack, 5.factorial\_cps is now:

```
3.factorial_cps(???)
```

## Factorial CPSed

```
class Integer
  def factorial_cps(k = lambda { |v| v })
    if self == 0
      k.call(1)
    else
      (self-1).factorial_cps(lambda { |v|
        k.call(self * v)
      })
    end
  end
end
```

In terms of the stack, `5.factorial_cps` is now:

```
2.factorial_cps(???)
```

## Factorial CPSed

```
class Integer
  def factorial_cps(k = lambda { |v| v })
    if self == 0
      k.call(1)
    else
      (self-1).factorial_cps(lambda { |v|
        k.call(self * v)
      })
    end
  end
end
```

In terms of the stack, 5.factorial\_cps is now:

```
1.factorial_cps(???)
```

## Factorial CPSed

```
class Integer
  def factorial_cps(k = lambda { |v| v })
    if self == 0
      k.call(1)
    else
      (self-1).factorial_cps(lambda { |v|
        k.call(self * v)
      })
    end
  end
end
```

In terms of the stack, 5.factorial\_cps is now:

```
0.factorial_cps(???)
```

## Lying “Animation” (Bonus Slide)

The “animation” of `5.factorial_cps` hides two things.

- First, stack frames *are* built up, but they only have to return a value.

## Lying “Animation” (Bonus Slide)

The “animation” of `5.factorial_cps` hides two things.

- First, stack frames *are* built up, but they only have to return a value.
- Second, the ??? hides *a lot*, but it’s worth working out because it would probably explain everything.

## No Free Lunch

- Those ???s hide procedures created on each recursion.

## No Free Lunch

- Those ???s hide procedures created on each recursion.
- We're trading heap space for stack space.

## No Free Lunch

- Those ???s hide procedures created on each recursion.
- We're trading heap space for stack space.
  - More space?

## No Free Lunch

- Those ???s hide procedures created on each recursion.
- We're trading heap space for stack space.
  - More space?
  - Lose special stack instructions?

# Outline

- 1 Continuation Passing Style
  - Factorial
  - **Fibonacci**
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

# Fibonacci in Ruby

```
class Integer
  def fibonacci
    if self < 2
      1
    else
      (self-1).fibonacci + (self-2).fibonacci
    end
  end
end
```

## Fibonacci in Ruby

```
class Integer
  def fibonacci
    if self < 2
      1
    else
      (self-1).fibonacci + (self-2).fibonacci
    end
  end
end
```

Two recursive calls...

## Fibonacci in Ruby

```
class Integer
  def fibonacci
    if self < 2
      1
    else
      (self-1).fibonacci + (self-2).fibonacci
    end
  end
end
```

Two recursive calls... two continuations!

## CPSed Fibonacci in Ruby

```
class Integer
  def fibonacci_cps(k = lambda { |v| v })
    if self < 2
      k.call(1)
    else
      (self-1).fibonacci_cps(lambda { |v|
        (self-2).fibonacci_cps(lambda { |w|
          k.call(v + w)
        })
      })
    end
  end
end
```

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

## Why Would You CPS a Method?

- You get complete control of what happens next.

## Why Would You CPS a Method?

- You get complete control of what happens next.
  - What if you never invoked `continuation.call()`?

## Why Would You CPS a Method?

- You get complete control of what happens next.
  - What if you never invoked `continuation.call()`?
- You can save a continuation somewhere.

## Why Would You CPS a Method?

- You get complete control of what happens next.
  - What if you never invoked `continuation.call()`?
- You can save a continuation somewhere.
  - *In a session!*

## Why Would You CPS a Method?

- You get complete control of what happens next.
  - What if you never invoked `continuation.call()`?
- You can save a continuation somewhere.
  - *In a session!*
- You can generate assembly code from CPSed code.

## Why Would You CPS a Method?

- You get complete control of what happens next.
  - What if you never invoked `continuation.call()`?
- You can save a continuation somewhere.
  - *In a session!*
- You can generate assembly code from CPSed code.
  - Some compilers do (but not mine).

# Scheme Has an Advantage

```
(define factorial
  (lambda (n)
    (cond [(zero? n) 1]
          [else (* n (factorial (- n 1)))])))

(define factorial-cps
  (lambda (n k)
    (cond [(zero? n) (k 1)]
          [else (factorial-cps (- n 1)
                                (lambda (v) (* n v)))])))
```

Is the recursive call the *last* thing done in the general case?

## Compute the Stack Space

- How much stack space does `1000.factorial` take up?

## Compute the Stack Space

- How much stack space does `1000.factorial` take up?
- `1000.factorial_cps`?

## Compute the Stack Space

- How much stack space does `1000.factorial` take up?
- `1000.factorial_cps`?
- `(factorial 1000)`?

## Compute the Stack Space

- How much stack space does `1000.factorial` take up?
- `1000.factorial_cps`?
- `(factorial 1000)`?
- `(factorial-cps 1000 (lambda (v) v))`?

# Tail Recursion

- Tail recursion:

# Tail Recursion

- Tail recursion:
  - At most one function call, and it is the last thing done.

# Tail Recursion

- Tail recursion:
  - At most one function call, and it is the last thing done.
  - Elimination: re-use the current stack frame for that function call.

# Tail Recursion

- Tail recursion:
  - At most one function call, and it is the last thing done.
  - Elimination: re-use the current stack frame for that function call.
- Languages:

# Tail Recursion

- Tail recursion:
  - At most one function call, and it is the last thing done.
  - Elimination: re-use the current stack frame for that function call.
- Languages:
  - Scheme interpreters and compilers are *required* to eliminate tail recursion.

# Tail Recursion

- Tail recursion:
  - At most one function call, and it is the last thing done.
  - Elimination: re-use the current stack frame for that function call.
- Languages:
  - Scheme interpreters and compilers are *required* to eliminate tail recursion.
  - GCC is (at times) tail recursive.

## Tail Recursion

- Tail recursion:
  - At most one function call, and it is the last thing done.
  - Elimination: re-use the current stack frame for that function call.
- Languages:
  - Scheme interpreters and compilers are *required* to eliminate tail recursion.
  - GCC is (at times) tail recursive.
  - *It appears that Ruby is not.*

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

## Accumulator Passing Style

```
class Integer
  def factorial_aps(accumulator = 1)
    if self == 0
      accumulator
    else
      (self-1).factorial_aps(accumulator * self)
    end
  end
end
```

- Continuation has simple arithmetic in it.

## Accumulator Passing Style

```
class Integer
  def factorial_aps(accumulator = 1)
    if self == 0
      accumulator
    else
      (self-1).factorial_aps(accumulator * self)
    end
  end
end
```

- Continuation has simple arithmetic in it.
- Still tail recursive.

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1?

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1? 3?

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1? 3? 3?

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1? 3? 3? 4?

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1? 3? 3? 4? 5?

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1? 3? 3? 4? 5? 6?

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1? 3? 3? 4? 5? 6? error?

## What Does This Return?

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

1? 3? 3? 4? 5? 6? error?

Answer: 4

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:
  - The current continuation is  $1 + v$ .

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:
  - The current continuation is  $1 + v$ .
  - Bind it to `k`.

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:
  - The current continuation is  $1 + v$ .
  - Bind it to `k`.
- When `k` is invoked:

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:
  - The current continuation is  $1 + v$ .
  - Bind it to `k`.
- When `k` is invoked:
  - The current continuation is now  $1 + 2 + v$ .

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:
  - The current continuation is  $1 + v$ .
  - Bind it to `k`.
- When `k` is invoked:
  - The current continuation is now  $1 + 2 + v$ .
  - *Toss out that current continuation!*

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:
  - The current continuation is  $1 + v$ .
  - Bind it to `k`.
- When `k` is invoked:
  - The current continuation is now  $1 + 2 + v$ .
  - *Toss out that current continuation!*
  - Use the continuation bound to `k`.

# Analysis of Call-with-Current-Continuation

```
1 + callcc do |k|  
  2 + k.call(3)  
end
```

- When `callcc` is called:
  - The current continuation is  $1 + v$ .
  - Bind it to `k`.
- When `k` is invoked:
  - The current continuation is now  $1 + 2 + v$ .
  - *Toss out that current continuation!*
  - Use the continuation bound to `k`.
  - Bind its argument to the place-holder:  $v \leftarrow 3$ .

## Try Again and Again

```
1 + callcc do |k|  
  k.call(2) + 3  
end
```

## Try Again and Again

```
1 + callcc do |k|  
  k.call(2) + 3  
end
```

```
1 + callcc do |k|  
  2 + 3  
end
```

## Try Again and Again

```
1 + callcc do |k|  
  k.call(2) + 3  
end
```

```
1 + callcc do |k|  
  2 + 3  
end
```

```
1 + callcc do |k|  
  k.call(2) + k.call(3)  
end
```

## Saving Up for a Rainy Day

```
r = nil
1 + callcc do |k|
  r = k
  2 + k.call(3)
end
r.call(3)
r.call(-2)
r.call(66)
```

## Saving Up for a Rainy Day

```
r = nil
1 + callcc do |k|
  r = k
  2 + k.call(3)
end                # => 4
r.call(3)
r.call(-2)
r.call(66)
```

## Saving Up for a Rainy Day

```
r = nil
1 + callcc do |k|
  r = k
  2 + k.call(3)
end                # => 4
r.call(3)         # => 4
r.call(-2)
r.call(66)
```

## Saving Up for a Rainy Day

```
r = nil
1 + callcc do |k|
  r = k
  2 + k.call(3)
end                # => 4
r.call(3)          # => 4
r.call(-2)         # => -1
r.call(66)
```

## Saving Up for a Rainy Day

```
r = nil
1 + callcc do |k|
  r = k
  2 + k.call(3)
end                # => 4
r.call(3)         # => 4
r.call(-2)       # => -1
r.call(66)       # => 67
```

## Practical Uses?

- Throw, catch.

## Practical Uses?

- Throw, catch.
- Raise, rescue.

## Practical Uses?

- Throw, catch.
- Raise, rescue.
- Early exits for special data.

## Practical Uses?

- Throw, catch.
- Raise, rescue.
- Early exits for special data.
- Confuse students on programming-languages tests.

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

## Early Exit for Zero—The Problem

```
class Array
  def product
    p = 1
    each do |x|
      p = p * x
    end
    p
  end
end
```

## Early Exit for Zero—The Problem

```
class Array
  def product
    p = 1
    each do |x|
      p = p * x
    end
    p
  end
end
```

What about `(0..1000).to_a.product?`

## Early Exit for Zero—Solution

```
class Array
  def product
    callcc do |k|
      p = 1
      each do |x|
        k.call(0) if x.zero?
        p = p * x
      end
      p
    end
  end
end
```

# Beware Premature Optimization!

- If you have data with lots of zeros in it, good!

## Beware Premature Optimization!

- If you have data with lots of zeros in it, good!
- Otherwise, you've added an extra **zero?** test for every element.

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 **Call with Current Continuation**
  - Example and Analysis
  - Practical Use for Early Exit
  - **Fibonacci Again**
  - Tormenting Students
- 3 Lame Ending

## Coroutines (Ruby 1.9)

```
fib = Fiber.new do
  x, y = 0, 1
  loop do
    Fiber.yield y
    x,y = y,x+y
  end
end
```

- `Fiber.yield` saves the current continuation and returns its argument.

## Coroutines (Ruby 1.9)

```
fib = Fiber.new do
  x, y = 0, 1
  loop do
    Fiber.yield y
    x,y = y,x+y
  end
end
```

- `Fiber.yield` saves the current continuation and returns its argument.
- `Fiber#resume` resumes the coroutine until the `yield` and returns that result.

## Coroutines (Ruby 1.9)

```
fib = Fiber.new do
  x, y = 0, 1
  loop do
    Fiber.yield y
    x,y = y,x+y
  end
end
```

- `Fiber.yield` saves the current continuation and returns its argument.
- `Fiber#resume` resumes the coroutine until the `yield` and returns that result.
- `So 20.times { puts fib.resume }...`

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

## What Gets Returned?

```
callcc { |k| k.call(1) }  
1 + callcc { |k| 1 + k.call(0) }  
1 + callcc { |k| 1 + k.call(k.call(2)) }
```

## What Gets Returned?

```
callcc { |k| k.call(1) }           # => 1  
1 + callcc { |k| 1 + k.call(0) }  
1 + callcc { |k| 1 + k.call(k.call(2)) }
```

## What Gets Returned?

```
callcc { |k| k.call(1) }           # => 1  
1 + callcc { |k| 1 + k.call(0) }   # => 1  
1 + callcc { |k| 1 + k.call(k.call(2)) }
```

## What Gets Returned?

```
callcc { |k| k.call(1) }           # => 1  
1 + callcc { |k| 1 + k.call(0) }   # => 1  
1 + callcc { |k| 1 + k.call(k.call(2)) } # => 3
```

# Scheme Monstrosity and Wrong Ruby Translation

```
(add1 (call/cc
      (call/cc
        (lambda (k)
          (cons (lambda (k) (k 0))
                (k (lambda (k) (k 1))))))))))
```

```
1 + callcc do |kk|
  callcc do |k|
    [kk.call(0), k.call(kk.call(1))]
  end
end
```

Ruby's `callcc` requires a block; Scheme will take any old closure.

# Scheme Monstrosity and Wrong Ruby Translation

```
(add1 (call/cc
      (call/cc
        (lambda (k)
          (cons (lambda (k) (k 0))
                (k (lambda (k) (k 1)))))))))) ;; => 2
```

```
1 + callcc do |kk|
  callcc do |k|
    [kk.call(0), k.call(kk.call(1))]
  end
end
```

Ruby's `callcc` requires a block; Scheme will take any old closure.

# Scheme Monstrosity and Wrong Ruby Translation

```
(add1 (call/cc
      (call/cc
        (lambda (k)
          (cons (lambda (k) (k 0))
                (k (lambda (k) (k 1)))))))))) ;; => 2
```

```
1 + callcc do |kk|
  callcc do |k|
    [kk.call(0), k.call(kk.call(1))]
  end
end # => 1
```

Ruby's `callcc` requires a block; Scheme will take any old closure.

# Outline

- 1 Continuation Passing Style
  - Factorial
  - Fibonacci
  - Why, Oh, Why?
  - Is a Lambda Necessary?
- 2 Call with Current Continuation
  - Example and Analysis
  - Practical Use for Early Exit
  - Fibonacci Again
  - Tormenting Students
- 3 Lame Ending

## No Really...

...this is my lame ending.

## Less Lame (Bonus Slide)

Check out my blog at <http://jdfrens.blogspot.com/> called “Programming During Recess”. I hope to blog a bit about continuations and Ruby there.