Computational Continuations

John Quigley
www.jquigley.com
jquigley@jquigley.com

Chicago Linux
September 2007

Definition

Question: What is a continuation?

A continuation represents the rest of a computation at any given point in the computation.

The 'rest of the computation' means control state, or the data structures and code needed to complete a computation.

The 'data structure' is often the stack, and the code is a pointer to the current instruction. Or, this could all be heap allocated.

Definition

Question: What is a continuation?

A continuation represents the rest of a computation at any given point in the computation.

The 'rest of the computation' means control state, or the data structures and code needed to complete a computation.

The 'data structure' is often the stack, and the code is a pointer to the current instruction. Or, this could all be heap allocated.

Definition

Question: What is a continuation?

A continuation represents the rest of a computation at any given point in the computation.

The 'rest of the computation' means control state, or the data structures and code needed to complete a computation.

The 'data structure' is often the stack, and the code is a pointer to the current instruction. Or, this could all be heap allocated.

Definition

Question: What is a continuation?

A continuation represents the rest of a computation at any given point in the computation.

The 'rest of the computation' means control state, or the data structures and code needed to complete a computation.

The 'data structure' is often the stack, and the code is a pointer to the current instruction. Or, this could all be heap allocated.

History

Most languages have facilities for manipulating the continuation of a computation step.

Early imperative languages provided the GOTO – or setjmp(3) in C – which would force the computation to continue at some designated label.

In the 1970's, additional control patterns were added like function returns, loop exits and iteration breaks.

Complex examples from this era include Simula 67's coroutines, Icon's generators and Prolog's backtracking.

History

Most languages have facilities for manipulating the continuation of a computation step.

Early imperative languages provided the GOTO – or setjmp(3) in C – which would force the computation to continue at some designated label.

In the 1970's, additional control patterns were added like function returns, loop exits and iteration breaks.

Complex examples from this era include Simula 67's coroutines, Icon's generators and Prolog's backtracking.

History

Most languages have facilities for manipulating the continuation of a computation step.

Early imperative languages provided the GOTO – or setjmp(3) in C – which would force the computation to continue at some designated label.

In the 1970's, additional control patterns were added like function returns, loop exits and iteration breaks.

Complex examples from this era include Simula 67's coroutines, Icon's generators and Prolog's backtracking.

History

Most languages have facilities for manipulating the continuation of a computation step.

Early imperative languages provided the GOTO – or setjmp(3) in C – which would force the computation to continue at some designated label.

In the 1970's, additional control patterns were added like function returns, loop exits and iteration breaks.

Complex examples from this era include Simula 67's coroutines, Icon's generators and Prolog's backtracking.

Availability

Only a few programming languages provide full, unrestrained access to continuations.

Scheme was the first production system, first providing 'catch,' and then call-with-current-continuation, or call/cc.

It continues to provide the most robust and systematic implementation.

Availability

Only a few programming languages provide full, unrestrained access to continuations.

Scheme was the first production system, first providing 'catch,' and then call-with-current-continuation, or call/cc.
It continues to provide the most robust and systematic implementation.

Availability

1. **Parrot**: Continuation
2. **Ruby**: callcc
3. **Scheme**: call-with-current-continuation, or call/cc
4. **Smalltalk**: Continuation currentDo:

In any language which supports closures, it is possible to manually implement call/cc!

This is a common strategy in Haskell.

Availability

1. **Parrot**: Continuation
2. **Ruby**: callcc
3. **Scheme**: call-with-current-continuation, or call/cc
4. **Smalltalk**: Continuation currentDo:

In any language which supports closures, it is possible to manually implement call/cc!

This is a common strategy in Haskell.

Availability

1. **Parrot**: Continuation
2. **Ruby**: callcc
3. **Scheme**: call-with-current-continuation, or call/cc
4. **Smalltalk**: Continuation currentDo:

In any language which supports closures, it is possible to manually implement call/cc!

This is a common strategy in Haskell.

Availability

1. **Parrot**: Continuation
2. **Ruby**: callcc
3. **Scheme**: call-with-current-continuation, or call/cc
4. **Smalltalk**: Continuation currentDo:

In any language which supports closures, it is possible to manually implement call/cc!

This is a common strategy in Haskell.

Availability

1. **Parrot**: Continuation
2. **Ruby**: callcc
3. **Scheme**: call-with-current-continuation, or call/cc
4. **Smalltalk**: Continuation currentDo:

In any language which supports closures, it is possible to manually implement call/cc!

This is a common strategy in Haskell.

Availability

1. **Parrot**: Continuation
2. **Ruby**: callcc
3. **Scheme**: call-with-current-continuation, or call/cc
4. **Smalltalk**: Continuation currentDo:

In any language which supports closures, it is possible to manually implement call/cc!

This is a common strategy in Haskell.

Motivation

The fixation on call/cc and on the gritty details of its implementation and semantics has greatly obscured the simplicity and elegance of continuations.

The motivation for this presentation is to present continuations in a simple and intuitive way.

Motivation

The fixation on call/cc and on the gritty details of its implementation and semantics has greatly obscured the simplicity and elegance of continuations.

The motivation for this presentation is to present continuations in a simple and intuitive way.

Function Return

Traditionally, a function returns a value, e.g.:

function return

```
def foo(x):
    return x+1
```

This leaves implicit where this value is to be returned to.

Function Return

Traditionally, a function returns a value, e.g.:

function return

```
def foo(x):
    return x+1
```

This leaves implicit where this value is to be returned to.

Explicit Return

The core idea of continuations is to make this behavior explicit by adding a continuation argument.

Instead of 'returning' the value, the function 'continues' with the value by giving it as an argument to the continuation.

'continued' function

```
def foo(x,c):
    c(x+1)
```

Explicit Return

The core idea of continuations is to make this behavior explicit by adding a continuation argument.

Instead of `returning' the value, the function `continues' with the value by giving it as an argument to the continuation.

`continued' function

```
def foo(x,c):
    c(x+1)
```

Explicit Return

The core idea of continuations is to make this behavior explicit by adding a continuation argument.

Instead of ‘returning’ the value, the function ‘continues’ with the value by giving it as an argument to the continuation.

‘continued’ function

```
def foo(x,c):
    c(x+1)
```

Explicit Return

With this view, a function never ‘returns’ – instead it ‘continues.’

And it is for this reason, continuations have sometimes been described as gotos with arguments.

This idea is the basis of CPS, or Continuation Passing Style:

- Function signature gets extra ‘continuation’ argument
- Function doesn’t return value, instead passes it on as the continuation argument

Explicit Return

With this view, a function never 'returns' – instead it 'continues.'

And it is for this reason, continuations have sometimes been described as gotos with arguments.

This idea is the basis of CPS, or Continuation Passing Style:

1. Function signature gets extra 'continuation' argument
2. Function doesn't return value, instead passes it on to the continuation argument

Explicit Return

With this view, a function never 'returns' – instead it 'continues.'

And it is for this reason, continuations have sometimes been described as gotos with arguments.

This idea is the basis of CPS, or Continuation Passing Style:

1. Function signature gets extra 'continuation' argument
2. Function doesn't return value, instead passes it on as the continuation argument

Explicit Return

With this view, a function never `returns' – instead it `continues.'

And it is for this reason, continuations have sometimes been described as gotos with arguments.

This idea is the basis of CPS, or Continuation Passing Style:

1. Function signature gets extra `continuation' argument
2. Function doesn't return value, instead passes it on as the continuation argument

Explicit Return

With this view, a function never `returns' – instead it `continues.'

And it is for this reason, continuations have sometimes been described as gotos with arguments.

This idea is the basis of CPS, or Continuation Passing Style:

1. Function signature gets extra `continuation' argument
2. Function doesn't return value, instead passes it on as the continuation argument

More On CPS

You'll quickly realize that CPS also unfolds all nested expressions. An example:

nested return value

```
def baz(x,y):
    return 2*x+y
```

In the continuation passing style, even primitive operators such as * or + take an extra continuation argument.

More On CPS

You'll quickly realize that CPS also unfolds all nested expressions. An example:

nested return value

```
def baz(x,y):
    return 2*x+y
```

In the continuation passing style, even primitive operators such as * or + take an extra continuation argument.

More On CPS

You'll quickly realize that CPS also unfolds all nested expressions. An example:

nested return value

```
def baz(x,y):
    return 2*x+y
```

In the continuation passing style, even primitive operators such as * or + take an extra continuation argument.

More On CPS

We can simulate this with the following definitions:

simulated primitives

```
def add(x,y,c): c(x+y)
def mul(x,y,c): c(x*y)
```

CPS would transform the baz() function into:

cps transformation

```
def baz(x,y,c):
    mul(2,x,lambda v,y=y,c=c: add(v,y,c))
```

More On CPS

We can simulate this with the following definitions:

simulated primitives

```
def add(x,y,c): c(x+y)
def mul(x,y,c): c(x*y)
```

CPS would transform the baz() function into:

cps transformation

```
def baz(x,y,c):
    mul(2,x,lambda v,y=y,c=c: add(v,y,c))
```

Wrap Up

Continuations are as low-level as it gets.

Continuations are the functional expression of the GOTO statement, and the same caveats apply.

Continuations can quickly result in code that is difficult to follow: the programmer must maintain the invariants of control and continuations by hand.

Even hard-core continuation fans don't use them directly except as means to implement better-behaved abstractions.

Wrap Up

Continuations are as low-level as it gets.

Continuations are the functional expression of the GOTO statement, and the same caveats apply.

Continuations can quickly result in code that is difficult to follow: the programmer must maintain the invariants of control and continuations by hand.

Even hard-core continuation fans don't use them directly except as means to implement better-behaved abstractions.

Wrap Up

Continuations are as low-level as it gets.

Continuations are the functional expression of the GOTO statement, and the same caveats apply.

Continuations can quickly result in code that is difficult to follow: the programmer must maintain the invariants of control and continuations by hand.

Even hard-core continuation fans don't use them directly except as means to implement better-behaved abstractions.

Wrap Up

Continuations are as low-level as it gets.

Continuations are the functional expression of the GOTO statement, and the same caveats apply.

Continuations can quickly result in code that is difficult to follow: the programmer must maintain the invariants of control and continuations by hand.

Even hard-core continuation fans don't use them directly except as means to implement better-behaved abstractions.

Control Patterns

Continuations can be used to implement very advanced control flow patterns of varying rigidity:

1. fibers
2. iterators
3. coroutines

Control Patterns

Continuations can be used to implement very advanced control flow patterns of varying rigidity:

1. fibers
2. iterators
3. coroutines

Control Patterns

Continuations can be used to implement very advanced control flow patterns of varying rigidity:

1. fibers
2. iterators
3. coroutines

Control Patterns

Continuations can be used to implement very advanced control flow patterns of varying rigidity:

1. fibers
2. iterators
3. coroutines

Generators

The most basic form of a continuation is a subroutine call.

Definition: A generator is a special subroutine that can be used to control loop iteration behavior.

A generator looks like a function, but behaves like an iterator.

Generators add two new abstract operations on top of the subroutine: "suspend" and "resume".

Generators

The most basic form of a continuation is a subroutine call.

Definition: A generator is a special subroutine that can be used to control loop iteration behavior.

A generator looks like a function, but behaves like an iterator.

Generators add two new abstract operations on top of the subroutine: "suspend" and "resume".

Generators

The most basic form of a continuation is a subroutine call.

Definition: A generator is a special subroutine that can be used to control loop iteration behavior.

A generator looks like a function, but behaves like an iterator.

Generators add two new abstract operations on top of the subroutine: "suspend" and "resume".

Generators

The most basic form of a continuation is a subroutine call.

Definition: A generator is a special subroutine that can be used to control loop iteration behavior.

A generator looks like a function, but behaves like an iterator.

Generators add two new abstract operations on top of the subroutine: "suspend" and "resume".

Coroutines

Coroutines add only one new abstract operation: transfer.

'Transfer' names a coroutine to transfer to, and gives a value to deliver to it.

When A transfers to B, it acts like a generator 'suspend'.

Coroutines are an achingly natural way to model independent objects that interact with feedback.

A UNIX pipeline is suggestive of their full power.

Coroutines

Coroutines add only one new abstract operation: transfer.

'Transfer' names a coroutine to transfer to, and gives a value to deliver to it.

When A transfers to B, it acts like a generator 'suspend'.

Coroutines are an achingly natural way to model independent objects that interact with feedback.

A UNIX pipeline is suggestive of their full power.

Coroutines

Coroutines add only one new abstract operation: transfer.

'Transfer' names a coroutine to transfer to, and gives a value to deliver to it.

When A transfers to B, it acts like a generator 'suspend'.

Coroutines are an achingly natural way to model independent objects that interact with feedback.

A UNIX pipeline is suggestive of their full power.

Coroutines

Coroutines add only one new abstract operation: transfer.

'Transfer' names a coroutine to transfer to, and gives a value to deliver to it.

When A transfers to B, it acts like a generator 'suspend'.

Coroutines are an achingly natural way to model independent objects that interact with feedback.

A UNIX pipeline is suggestive of their full power.

Coroutines

Coroutines add only one new abstract operation: transfer.

'Transfer' names a coroutine to transfer to, and gives a value to deliver to it.

When A transfers to B, it acts like a generator 'suspend'.

Coroutines are an achingly natural way to model independent objects that interact with feedback.

A UNIX pipeline is suggestive of their full power.

Continuations

Give the pedagogical structure so far, you're primed to view continuations as enhancements of coroutines.

Continuations aren't more elaborate than coroutines, they're simpler!

Indeed they're simpler than generators, and even a simpler "regular call."

Continuations

Give the pedagogical structure so far, you're primed to view continuations as enhancements of coroutines.

Continuations aren't more elaborate than coroutines, they're simpler!

Indeed they're simpler than generators, and even a simpler "regular call."

Continuations

Give the pedagogical structure so far, you′re primed to view continuations as enhancements of coroutines.

Continuations aren′t more elaborate than coroutines, they′re simpler!

Indeed they′re simpler than generators, and even a simpler "regular call."

Continuations

This is what makes continuations so confusing at first: they're a different basis for all call-like behavior.

Generators and coroutines are variations on what you already know; continuations challenge your fundamental view of the programming universe.

Continuations

This is what makes continuations so confusing at first: they're a different basis for all call-like behavior.

Generators and coroutines are variations on what you already know; continuations challenge your fundamental view of the programming universe.

Subroutines

Let's look at Python.

When Python makes a call, it allocates a frame object. When a subroutine returns, it decrefs the frame and it goes away.

Attached to that frame:

1. locals, or a map of name:object bindings
2. evaluation stack for holding temps and dynamic block-nesting info
3. offset to current byte code instruction, relative to start of code object's immutable bytecode vector

Subroutines

Let's look at Python.

When Python makes a call, it allocates a frame object. When a subroutine returns, it decrefs the frame and it goes away.

Attached to that frame:

1. locals, or a map of name:object bindings
2. evaluation stack for holding temps and dynamic block-nesting info
3. offset to current byte code instruction, relative to start of code object's immutable bytecode vector

Subroutines

Let's look at Python.

When Python makes a call, it allocates a frame object. When a subroutine returns, it decrefs the frame and it goes away.

Attached to that frame:

1. locals, or a map of name:object bindings
2. evaluation stack for holding temps and dynamic block-nesting info
3. offset to current byte code instruction, relative to start of code object's immutable bytecode vector

Subroutines

Let's look at Python.

When Python makes a call, it allocates a frame object. When a subroutine returns, it decrefs the frame and it goes away.

Attached to that frame:

1. locals, or a map of name:object bindings
2. evaluation stack for holding temps and dynamic block-nesting info
3. offset to current byte code instruction, relative to start of code object's immutable bytecode vector

Subroutines

Let's look at Python.

When Python makes a call, it allocates a frame object. When a subroutine returns, it decrefs the frame and it goes away.

Attached to that frame:

1. locals, or a map of name:object bindings
2. evaluation stack for holding temps and dynamic block-nesting info
3. offset to current byte code instruction, relative to start of code object's immutable bytecode vector

Generators

Generators are a trivial extension on what Python does with subroutines.

When a generator suspends, it's just like a return, except we decline to decref the frame. That's it!

The locals, and where we are in the computation, aren't thrown away.

A 'resume,' then, consists of restarting the frame at its next bytecode instruction, with the locals and eval stack retained.

Generators

Generators are a trivial extension on what Python does with subroutines.

When a generator suspends, it's just like a return, except we decline to decref the frame. That's it!

The locals, and where we are in the computation, aren't thrown away.

A 'resume,' then, consists of restarting the frame at its next bytecode instruction, with the locals and eval stack retained.

Generators

Generators are a trivial extension on what Python does with subroutines.

When a generator suspends, it's just like a return, except we decline to decref the frame. That's it!

The locals, and where we are in the computation, aren't thrown away.

A 'resume,' then, consists of restarting the frame at its next bytecode instruction, with the locals and eval stack retained.

Generators

Generators are a trivial extension on what Python does with subroutines.

When a generator suspends, it's just like a return, except we decline to decref the frame. That's it!

The locals, and where we are in the computation, aren't thrown away.

A 'resume,' then, consists of restarting the frame at its next bytecode instruction, with the locals and eval stack retained.

Coroutines

Coroutines are much harder to implement than generators.

'Transfer' names who next gets control, while generators always return to their (unnamed) caller.

A generator simply "pops the stack" when it suspends, while a coroutine's flow need not be stack-like (and often is not).

Coroutines

Coroutines are much harder to implement than generators.

‘Transfer’ names who next gets control, while generators always return to their (unnamed) caller.

A generator simply "pops the stack" when it suspends, while a coroutine’s flow need not be stack-like (and often is not).

Coroutines

Coroutines are much harder to implement than generators.

'Transfer' names who next gets control, while generators always return to their (unnamed) caller.

A generator simply "pops the stack" when it suspends, while a coroutine's flow need not be stack-like (and often is not).

Coroutines

In Python, this is a coroutine-killer, because the C stack gets intertwingled.

The Python coro implementation uses threads under the covers (where capturing pieces of the C stack isn't a problem).

Coroutines

In Python, this is a coroutine-killer, because the C stack gets intertwingled.

The Python coro implementation uses threads under the covers (where capturing pieces of the C stack isn't a problem).

Continuations

A continuation is like what a coro would be if you could capture its resumption state at any point, and assign that to a variable.

We can say it adds an abstract operation "capture," which snapshots the program counter, call stack, and the 'block stack.'

Continuations

A continuation is like what a coro would be if you could capture its resumption state at any point, and assign that to a variable.

We can say it adds an abstract operation "capture," which snapshots the program counter, call stack, and the 'block stack.'

Continuations

The snapshot would be taken at the point of continuation invocation, and would be packaged into a first-class ‘object.’

In a pure vision, a continuation can be captured anywhere (even in the middle of an expression), and any continuation can be invoked from anywhere else.

Continuations

The snapshot would be taken at the point of continuation invocation, and would be packaged into a first-class 'object.'

In a pure vision, a continuation can be captured anywhere (even in the middle of an expression), and any continuation can be invoked from anywhere else.