



EECS 490 – Lecture 10

Continuations

1

Announcements

- Project 2 due Fri 10/6 at 8pm

Agenda

- Restricted Continuations
- First-Class Continuations

Review: First-Class Entities

- We use *entity* to denote something that can be named in a program
 - Other terms also used: *citizen*, *object*
 - Examples: types, functions, data objects, values
- A *first-class entity* is an entity that supports all operations generally available to other entities
 - e.g. can be assigned to a variable, passed to or returned from a function



	C++	Java	Python	Scheme
Functions	<u>sort of</u>	no	<u>yes</u>	<u>yes</u>
Types	<u>no</u>	no	<u>yes</u>	<u>no</u>
Control	<u>no</u>	no	no	<u>yes</u>

Continuations

$x + 1 + \text{foo}()$

- A *continuation* represents the control state of a program
 - The sequence of active functions
 - Code location within each active function
 - Intermediate results
- A continuation can be *invoked* to return control to a previous state
- Only control state is restored, not state of data

Continuation Analogy

Say you're in the kitchen in front of the refrigerator, thinking about a sandwich [sic]. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it. :-)

— Luke Palmer

Types of Continuations

- A language may provide restricted forms of continuations that can only be invoked at specific times
 - Subroutines (i.e. functions)
 - Coroutines
 - Exceptions
 - Generators
- Some languages have first-class continuations that can be stored in a variable and invoked at arbitrary times

Subroutines

- A subroutine involves transfer of control between a caller and a callee
- Before control is transferred to the callee, the state of the caller, i.e. its continuation, must be saved
 - Intermediate results stored in caller's activation record
 - Information about how to return control to caller stored in callee's activation record
- Upon completion of call, caller's continuation invoked

```
def foo(x):  
    print(x - 1 + bar(x))
```

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

Continuation
of foo()
invoked

Continuation of foo()
must be saved
before call to bar()

Abrupt Termination

- In some languages the caller's continuation is only invoked when the callee completes normally
- Other languages allow early termination of a call, also called *abrupt termination*, with a return statement

```
def foo(x):
```

```
→ return x
```

```
# dead code
```

```
⚡ if x < 0:
```

```
    bar(x)
```

```
    baz(x)
```

Invoke caller's
continuation

Code never
reached

Control vs. Data State

- A continuation only represents control state, so invoking it does not restore the state of data

```
def outer():
```

```
    x = 0
```

```
    def inner():
```

```
        nonlocal x
```

```
        x += 1
```

Invoke continuation
of outer()

```
    inner()
```

```
    print(x) # prints 1
```

Value of x is
not restored

Coroutines

- Generalize subroutines to allow multiple routines to invoke each other's continuations

```
var q := new queue
```

```
coroutine produce  
  loop
```

```
    while q is not full  
      create some new items  
      add the items to q
```

```
    yield to consume
```

```
coroutine consume  
  loop
```

```
    while q is not empty  
      remove some items from q  
      use the items
```

```
    yield to produce
```

This is pseudocode.

10/16/17

Exceptions

- ▶ Allow control to be passed to a function further up in the call chain, rather than just the direct caller

```
def foo(x):  
    try:   
        bar(x)  
    except:  
        print('Exception')
```

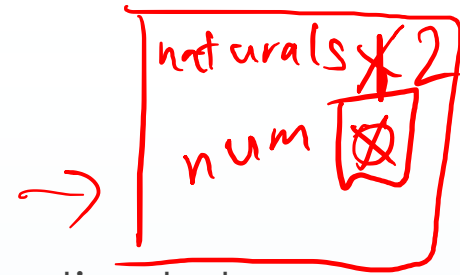
Save continuation of
foo(), add exception
handler to handler stack

```
def bar(x):  
    baz(x)
```

```
def baz(x):  
    raise Exception
```

Invoke continuation
of foo(), run
exception handler

Generators



- ▶ Like a subroutine, but allow execution to be paused and resumed
- ▶ Also called *semicoroutine*
 - ▶ Generator can be resumed by any caller
 - ▶ However, generator can only yield execution to caller that invoked it

```
def naturals():  
    num = 0  
    while True:  
        yield num  
        num += 1
```

Red lightning bolt arrows point to the `def`, `while`, and `yield` keywords in the code snippet.

Pause execution
and yield an
item to caller

A blue arrow points from this text box to the `yield num` line in the code snippet.

Generators and Iterators

- In Python, generators implement the same interface as an iterator
- Often simpler to write generator than a class that implements the iterator interface

```
def naturals():  
    num = 0  
    while True:  
        yield num  
        num += 1
```

```
>>> numbers = naturals()  
>>> next(numbers)  
0  
>>> next(numbers)  
1  
>>> next(numbers)  
2
```

Finite Generators

- A finite generator automatically raises a `StopIteration` exception when it completes
 - Used by a for loop to determine the end of an iterator

```
def range2(start, stop, step = 1):  
    while start < stop:  
        yield start  
        start += step
```

```
>>> values = range2(0, 5, 3)  
>>> next(values)  
0  
>>> next(values)  
3  
>>> next(values)  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
StopIteration
```

```
>>> for i in range2(0, 4):  
...     print(i)  
...  
0  
1  
2  
3
```

Generator Expressions

- Similar to list comprehensions, but produce a generator instead

```
def naturals():  
    num = 0  
    while True:  
        yield num  
        num += 1
```

```
>>> negatives = (-i for i in naturals() if i != 0)  
>>> next(negatives)  
-1  
>>> next(negatives)  
-2  
>>> next(negatives)  
-3
```

Generator
expression

- ▶ We'll start again in five minutes.

First-Class Continuations

- Many functional languages allow the current continuation to be captured in an explicit data structure
- Continuation can be passed as a parameter, returned, saved as a variable, etc.
- Depending on the language, the continuation may be invoked only once or an arbitrary number of times

(define call/cc call-with-current-continuation)

call/cc

- In Scheme, the `call-with-current-continuation` procedure, often abbreviated `call/cc`, creates an object representing the current continuation
- It then calls another procedure with the continuation as the argument

(call-with-current-continuation <procedure>)

- The called procedure can invoke the continuation, return it, discard it, etc.
 - If the procedure returns normally, the `call/cc` call evaluates to its result

Must be a one-argument procedure

```
> (+ 1 (call/cc (lambda (cc) 3)))
4
```

(+ 1 [])

Invoking a Continuation

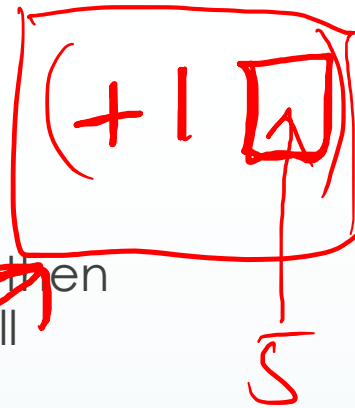
- A continuation is invoked with a value, which then becomes the "return value" of the `call/cc` call

```
> (+ 1 (call/cc (lambda (cc) (cc 5) 3)))  
6
```

Call to `call/cc`
replaced with
value 5

Continuation
invoked with
value 5

Not
evaluated



Storing a Continuation

- Allows a continuation to be invoked multiple times

```
> (define var (call/cc (lambda (cc) cc)))  
var  
> (define cont var)  
cont  
> (cont 3)  
var  
3  
> (cont 4)  
var  
4
```

Becomes
(define var 3)

Becomes
(define var 4)

Example: Factorial

- Continuation that multiplies a number by the factorial of another number:

~~(define cont '())~~

(define (factorial n)

(if (= n 0)

(call/cc (lambda (cc)
 (set! cont cc)
 1))

(* n (factorial (- n 1)))))

continuation

4
3

```
> (factorial 3)
6
> (cont 1)
6
> (cont 3)
18
> (factorial 5)
120
> (cont 4)
480
```

Emulating Call and Return

- Scheme does not provide abrupt termination, but we can emulate it with continuations
- We need:
 - A data structure to explicitly represent the call stack
 - A mechanism for calling a procedure while saving the caller's continuation
 - A mechanism for returning from a procedure by invoking the continuation of the caller
- For simplicity, we will only implement this for one-argument procedures

Call Stack

- A standard stack data structure using a list
- We need set! to modify the structure

```
(define call-stack '())
```

```
(define (push-call call)  
  (set! call-stack (cons call call-stack)))
```

```
(define (pop-call)  
  (let ((caller (car call-stack)))  
    (set! call-stack (cdr call-stack))  
    caller))
```


Call and Return

- The `return` procedure just pops off the caller's continuation from the stack and invokes it

```
(define (return value)
  ((pop-call) value))
```

- The `call` procedure must push the current continuation on the stack and then call the target procedure

```
(define (call func x)
  (call/cc (lambda (cc)
             (push-call cc)
             (func x))))
```

Using Call and Return

- We can now use `call` and `return`:

```
(define (foo x)
  (if (< x 10)
      (return x)
      (let ((y (- x 10)))
        (return (+ x (/ x y))))
      (some more stuff here))
```

```
(define (bar x)
  (return (- (call foo x)))
  (dead code))
```

```
> (+ 1 (call foo 3))
4
> (+ 1 (call foo 20))
23
> (+ 2 (call bar 3))
-1
> (+ 2 (call bar 20))
-20
```

Yin-Yang Puzzle

- Prints out unary representations of the natural numbers

```
(let* ((yin
        ((lambda (cc) (display "@") cc)
         (call/cc (lambda (c) c))))
       (yang
        ((lambda (cc) (display "*") cc)
         (call/cc (lambda (c) c)))))
  (yin yang))
```

```
@*@*@**@***@****@*****@*****@*****@*****@*****@*****@*****@
*****@*****@*****@*****@*****@*****@*****@*****@*****@
***@*****@*****@*****@*****@*****@*****@*****@*****@
*****@*****@*****@*****@*****@*****@*****@*****@*****@
*****@*****@*****@*****@*****@*****@*****@*****@*****@
*****@*****@*****@*****@*****@*****@*****@*****@*****@
*****@*****@*****@*****@*****@*****@*****@*****@*****@
**@*****@*****@*****@*****@*****@*****@*****@*****@
```

Continuations and Goto

- First-class continuations are often criticized for the same reasons as goto, since they allow unstructured transfer of control
- As with goto, continuations should be used judiciously
 - Implementing more restricted forms of control transfer such as exceptions
 - Adhering to conventions as in continuation-passing style