Programming Paradigms

Control Flow (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

Overview

- Expression Evaluation
- Structured and Unstructured Control Flow 
- Selection
- Iteration
- Recursion

Control Flow with `gotos`

- **Most assembly languages: Control flow via conditional and unconditional jumps**
- **Early PLs: `goto` statements**
  - Jump to a statement label
  - Target label can be anywhere in the code

Example

```
// C code
int a = 10;
my_label: do {
    if(a == 12) {
        a = a + 1;
        goto my_label;
    }
    printf("%d\n", a);
    a++;
} while(a < 15);
```

Example

```
// C code
int a = 10;
my_label: do {
    if(a == 12) {
        a = a + 1;
        goto my_label;
    }
    printf("%d\n", a);
    a++;
} while(a < 15);
```

Output:

10

11

13

14

Quiz: Goto Hell

```
// C code
int result = 0;
int number = 3;
one : for (int i = 0; i < number; ++i)
{
three:
    result += i;
    goto two;
}
goto one;
two : if (result < 2)
{
    goto three;
}
printf("%d\n", result);
```

What does this code print?

Quiz: Goto Hell

```
// C code
int result = 0;
int number = 3;
one : for (int i = 0; i < number; ++i)
{
three:
    result += i;
    goto two;
}
goto one;
two : if (result < 2)
{
    goto three;
}
printf("%d\n", result);
```

What does this code print?

Nothing! It never terminates.

Beyond `gotos`

- ***Go To Statement Considered Harmful***
  article by Edsger Dijkstra (CACM, 1968)
- **Instead: Structured control flow**
- **Express algorithms with**
  - Sequencing
  - Selection
  - Iteration

Avoiding `gotos`

Use case of `goto`

- Jump to end of subroutine
- Escape from middle of loop
- Propagate to surrounding context

Structured control flow alternative

- `return` statement
- `break` and `continue` statements
- Exceptions

Continuations

- **Generalization of `gotos`**
- **Powerful language feature:**
  Allows programmer to define new control flow constructs
  - Exceptions
  - Iterators
  - Coroutines
  - etc.

Continuations (2)

- **High-level definition: Context in which to continue execution**
- **Low-level definition: Three parts**
  - Code address (where to continue)
  - Referencing environment (for resolving names)
  - Another continuation (to use when code returns)

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end


v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end
```

Creates a continuation, i.e.,

execution will continue here

```
v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end
```

d is a reference to the continuation

```
v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end


v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

`foo` gets called and calls itself two more times

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end

v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Jumps into context captured by `c` and makes `callcc` appear to return `i`

Example

```
# Ruby code
def foo(i ,c)
    printf("start %d; ", i)
    if i < 3
        foo(i+1, c)
    else c.call(i)
    end
    printf "end %d; ", i
end
v = callcc{ |d| foo(1, d) }
printf "got %d\n", v
```

Code prints:

start 1; start 2; start 3; got 3

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

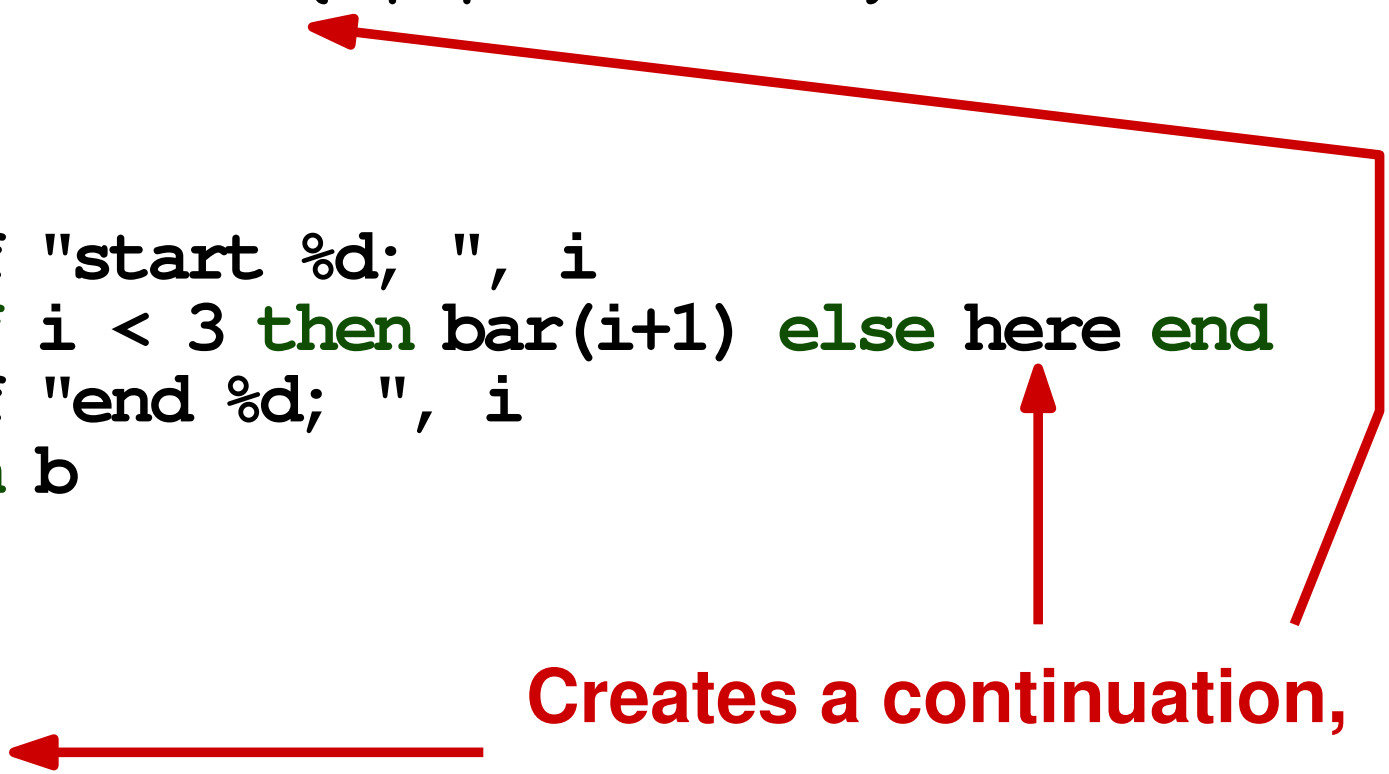
`bar` gets called and calls itself two more times

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```



Creates a continuation, which gets stored in `c`

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

n is 2, therefore execution jumps to the continuation

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

We are here again!

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

We are here again!

Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

n is 1, therefore execution jumps to the continuation

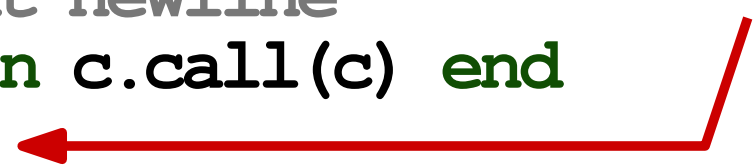
Another Example

```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

n is 0. We are finally done



Another Example

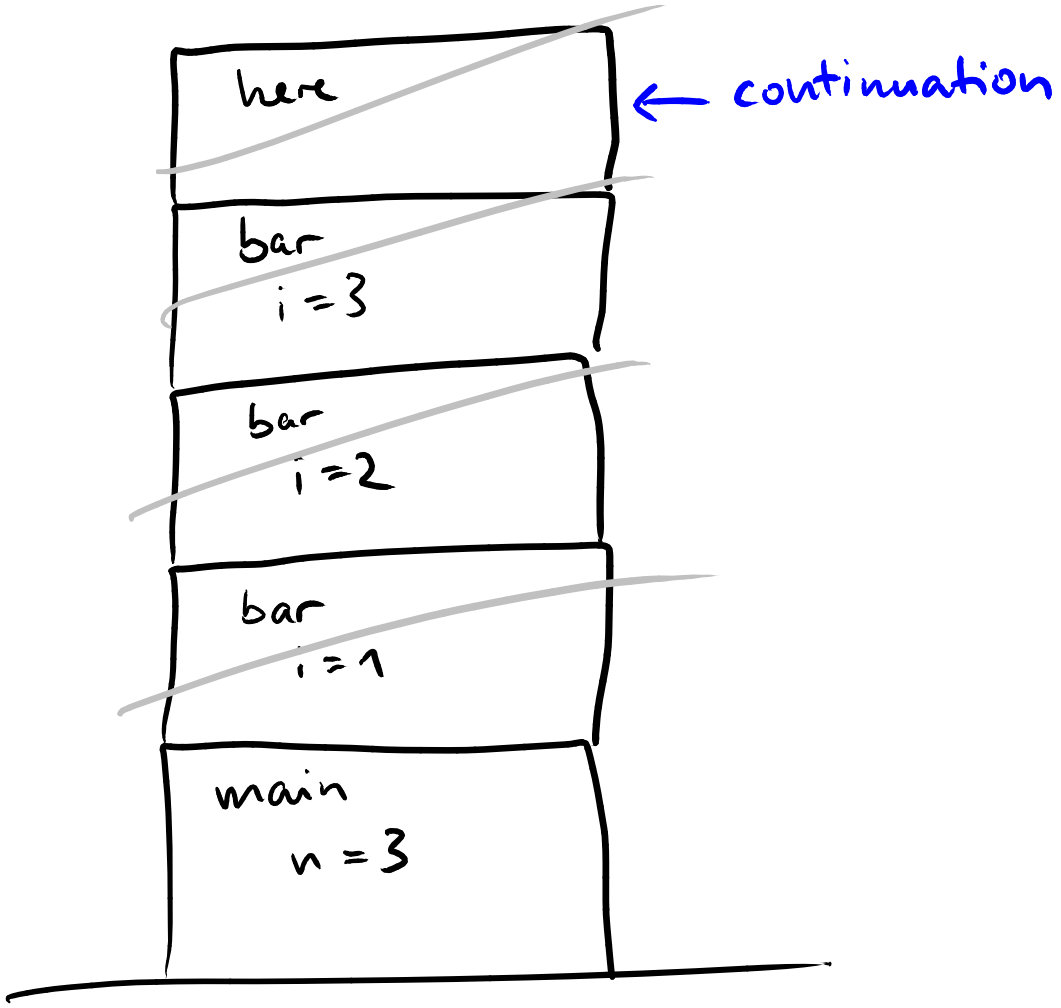
```
def here
    return callcc { |a| return a }
end

def bar(i)
    printf "start %d; ", i
    b = if i < 3 then bar(i+1) else here end
    printf "end %d; ", i
    return b
end

n = 3
c = bar(1)
n = n - 1
puts  # print newline
if n > 0 then c.call(c) end
puts "done"
```

Code prints:

```
start 1; start 2; start 3; end 3; end 2; end 1;
end 3; end 2; end 1;
end 3; end 2; end 1;
done
```



start 1
start 2
start 3
end 3
end 2
end 1
end 3
end 2
end 1
end 3
end 2
end 1
done