Flow Control

Pavel Strnad
based on slides of Michal Pise

Types of Flow Control

- goto
- functions
- closures
- coroutines
- continuations

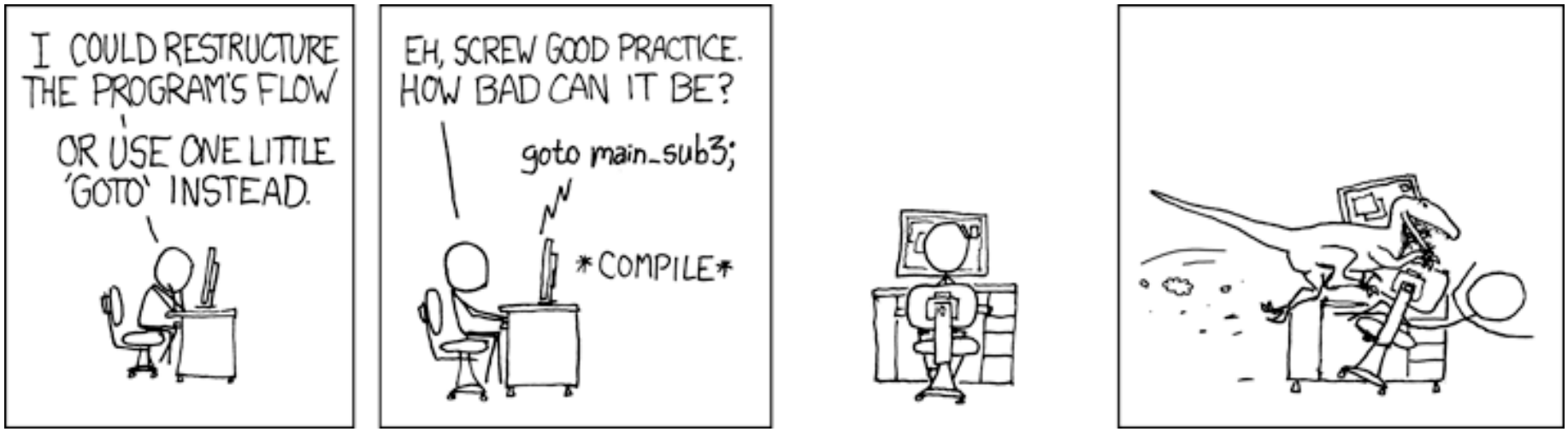
Goto

- goto sometimes called the Evil of programming
- programming standards disallow goto statement
- still implemented in many languages (C, C++, BASIC, reserved word in Java with no effect)

Goto Example

```
#include<stdio.h>
#include<conio.h>
void main() {
      if(1) {
         printf(”GOTO");
         goto A; }
        else {
         A:
         printf(" IS EVIL!!!");
}
getch();
}
```

Goto Reality



source: http://xkcd.com/292/

Functions

- better than using goto ☺
- can be found in all types of languages
- procedures vs. functions
- real functions does not have side effects
- imperative languages vs. functional languages

Fibonacci in C++

Function without side effects

```
#include
fibonacci(int iterations) {
    int first = 0, second = 1; // seed values
    for (int i = 0; i < iterations - 1; ++i) {
        int sum = first + second;
        first = second;
        second = sum;
    }
    return first;
}

int main() {
    std::cout << fibonacci(10) << "\n";
    return 0;
}
```

Function with side effects

```
#include <iostream>
int first = 0, second = 1; // seed values
fibonacci(int iterations) {
    for (int i = 0; i < iterations - 1; ++i) {
        int sum = first + second;
        first = second;
        second = sum;
    }
    return first;
}

int main() {
    std::cout << fibonacci(10) << "\n";
    return 0;
}
```

Fibonacci in Haskell

```
-- Fibonacci numbers, functional style
-- describe an infinite list based on the recurrence relation for Fibonacci numbers

fibRecurrence first second = first : fibRecurrence second (first + second)

-- describe fibonacci list as fibRecurrence with initial values 0 and 1

fibonacci = fibRecurrence 0 1

-- describe action to print the 10th element of the fibonacci list

main = print (fibonacci !! 10)

-- or oneliner

fibonacci2 = 0:1:zipwith (+) fibonacci2 (tail fibonacci2)
```

Fibonacci in Erlang

```
-module(fibonacci).
-export([start/1]).

%% Fibonacci numbers in Erlang
start(N) -> do_fib(0,1,N).
do_fib(_,B,1) -> B;
do_fib(A,B,N) -> do_fib(B,A
+B,N-1).
```

Fibonacci in Common Lisp

```
(defun fib (n &optional (a 0) (b 1))
    (if (= n 0)
         a
        (fib (- n 1) b (+ a b))))

fib(10)
```

Closures

- A closure is a function that references variables bound in its lexical environment.
- Depending on the language, it may or may not be able to change their values.
- A variable may therefore exist even outside of its scope.

Closures in Java

- Java can emulate Closures by anonymous classes
- Closures were planned for Java 7, but were discarded in release

Closures in Java

```
public static void main(String[] args) {
  final JFrame frame = new JFrame();
  frame.addWindowListener(
    new WindowAdapter() {
      public void windowClosing(WindowEvent e){
        frame.dispose();
      }
    });
  frame.setVisible(true);
}
```

Coroutine

- A coroutine is a subroutine generalization with multiple entry points for suspending and resuming execution at certain locations.
- Invocation of a coroutine is not stateless—context and environment are withheld.

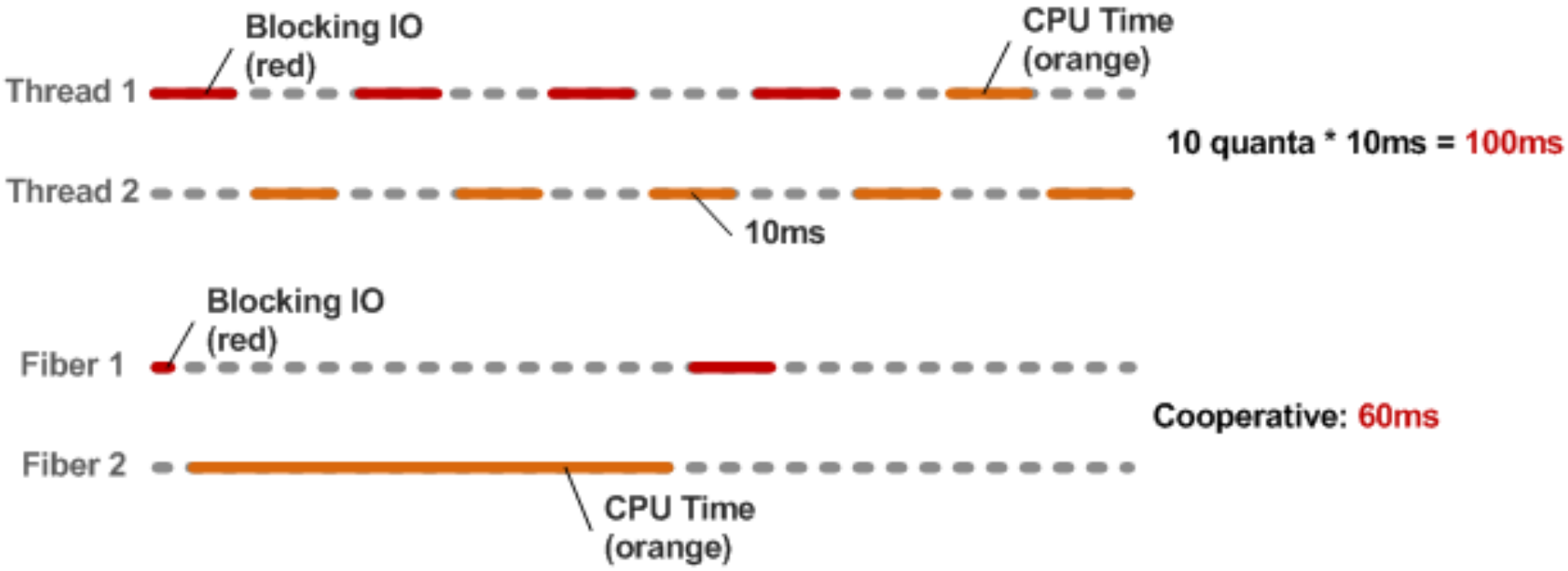
Coroutine Example in Java (yield does not exist in Java)

```
class Set {
  int[] contents; ...
  class SetIterator {
    int next() {
      for (int i = 0; i < contents.length; i++)
        yield contents[i];
        throw new RuntimeException();
    }
...
  }
}
```

Coroutine Example in Ruby

```
fibonacci = Fiber.new do
  i, j = 0, 1
  while(true)
    Fiber.yield(i)
    i,j = i+j,j
  end
end


100.times do
  value = fibonacci.resume
  puts value
end
```

Fibers vs. Threads in Ruby



Continuations

- Continuation is the rest of the executed computation.
- You can imagine it as a place in the function where we can return (many times!).
- Every program can be rewritten in this mannor.

Van Wijngaarden Transformation

- Provide each procedure declaration with an extra parameter – specified **label** – and insert at the end of its body a goto statement leading to that formal parameter.
- label the statement following a procedure statement and provide that label as the corresponding extra actual parameter.

CPS Transformation Example

Sample program

```
void p(char x)
{
    putchar(x);
}


main()
{
    p('a');
    p('b');
}
```

Step 1: goto

```
void p(char x,
       void *k)
{
    putchar(x);
    goto *k;
}

main()
{
    p('a', &&L1);
L1: p('b', &&L2);
L2: exit();
}
```

Step 2: non-returning functions

```
void p(char x,
       void (*k)())
{
    putchar(x);
    (*k)();
}

void L1(){p('b', &exit);}

main()
{
    p('a', &L1);
}
```

Figure 1: An example of van Wijngaarden's CPS transformation, applied to a C function `p`

Continuation Passing Style

- When a program is written using continuation-passing style, its functions never return.
- Instead, they invoke functions that represent the rest of the computation (continuations).
- It can be viewed as a **goto statement with parameters**.

Call with Current Continuation

- Imagine that we can label a sub-expression in an expression.
- This label marks the place in the expression.
- Than we can have escape operator **goto** to jump to this place.

callcc

$$100 + (\texttt{L:}\ \ (10 + ((\texttt{goto L})\, 1)))$$

```
        sum = 1;
        goto L;
        sum += 10;
L:      sum += 100;
```

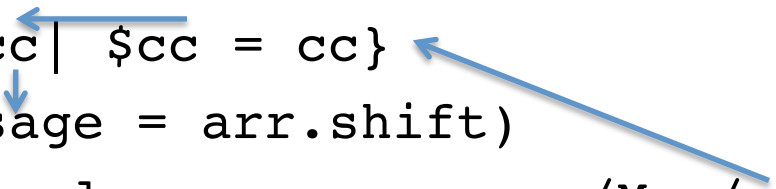
Example callcc in GNU C

```
void *label_as_result()
{
    return &&L;
 L: printf("Jumped back into the function. \n");
}

main()
{
    void *p;
    p = label_as_result();
    printf("The function returned; now jump back into it.\n");
    goto *p;
}
```

Figure 2: Jumping back into a function in Gnu C

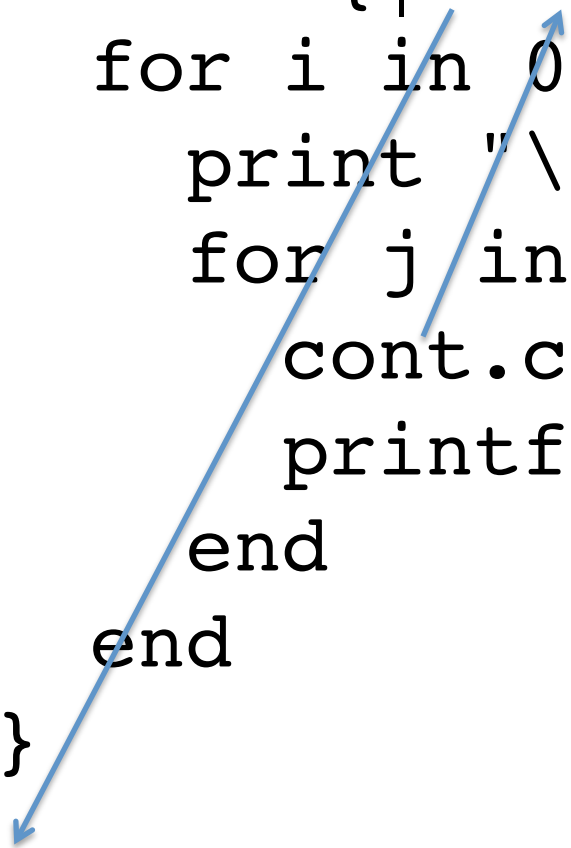
Example calcc in Ruby

```
require "continuation”

arr = [ "Freddie", "Herbie", "Ron", "Max", "Ringo" ]
callcc{|cc| $cc = cc}
puts(message = arr.shift)
$cc.call unless message =~ /Max/
```



Example callcc in Ruby

```
callcc {|cont|
   for i in 0..4
      print "\n#{i}: "
      for j in i*5...(i+1)*5
         cont.call() if j == 17
         printf "%3d", j
      end
   end
}
```



Bedtime reading

Hayo Thielecke. Continuations, functions and jumps. SIGACT News 30, 2 (June 1999). 33–42. http://doi.acm.org/10.1145/568547.568561