

get off my tail



rethinking tail calls on the jvm

What is a tail call?

```
def f() = g()
```

A call performed as the final action of a procedure.

What is tail recursion?

```
def odd(x) =  
  if (x == 0) true else even(x-1)
```

```
def even(x) =  
  if (x == 0) false else odd(x-1)
```

When a tail call might lead to the same function being called later in the call chain.

What is tail self-recursion?

```
def factorial(x) = {  
  def loop(x,y) =  
    if (x == 0) y else loop(x-1, x*y)  
  loop(x,1)  
}
```

When a function calls itself as its final action.

All of these should be
as fast as a GOTO

But they're not :(

None of them should
blow the stack

But they do :(

the state of play



Java

- No support for tail calls
- Dead on arrival



Scala

- Self-recursive tail calls optimised using a *goto*
- Must be a final method, or a local function
- *@tailrec* annotation as a safety net



Clojure

- Supports self-recursive tail calls
- *recur* special form required to trigger optimisation



Kawa

- Supports general tail calls when enabled with a compiler flag
- Self-recursion optimised with a *goto*
- Some mutual recursion optimised with *goto*
- Uses trampolining for everything else (slow)
- At least it doesn't blow the stack



F#

- Not JVM
- Self-recursion optimised with a *goto*
- All other tail calls use the .NET *.tail* opcode
- .NET tail call actually slower than a standard call due to extra security checks
- At least it doesn't blow the stack



“Folklore states that GOTO statements are cheap, while procedure calls are expensive.

This myth is largely a result of poorly designed language implementations.”

– Guy Steele (1977)

A photograph of two dogs running on a grassy field. On the left, a small, fluffy tan and white dog is captured mid-leap, its front paws extended forward and its mouth open in a happy expression. It wears a dark blue collar with a silver chain. On the right, a larger dog with black, tan, and white fur is running towards the right, its body low to the ground and its tongue hanging out. It wears a green harness and a blue leash that trails behind it. The background is a blurred chain-link fence and a grey wall.

let's try

Notation

$e ::= v \mid v(vs) \mid$
 $\text{let } x = v \text{ in } e \mid$
 $\text{let } x = v(vs) \text{ in } e \mid$
 $\text{letrec } fs \text{ in } e_s \mid$
 $\text{if } v \text{ then } e_1 \text{ else } e_2$

$f ::= x(xs) = e$

$v ::= x \mid c$

$xs ::= x, xs \mid \dots$

$vs ::= v, vs \mid \dots$

$fs ::= f; fs \mid \dots$

$x ::= \text{variable}$

$c ::= \text{constant}$

Code generation

- When generating a JVM method
 - If a function is only ever called in tail position
 - And is only called by the JVM method being generated
 - Then the function becomes a block called by a *goto*
 - Otherwise it becomes its own JVM method

Self recursion

```
letrec factorial(x) =  
  letrec loop(x0,r0) =  
    if x0 then  
      let r1 = mul(r0, x0)  
        x1 = sub(x0, 1)  
      in loop(x1, r1)  
    else  
      r0  
  in loop(x, 1)  
in ...
```



```
static int factorial(int x) {  
  int loop_x0 = x;  
  int loop_r0 = 1;  
  goto loop;  
  
loop:  
  if (loop_x0 != 0) {  
    int r1 = loop_r0 * loop_x0;  
    int x1 = loop_x0 - 1;  
    loop_x0 = x1;  
    loop_r0 = r1;  
    goto loop;  
  } else {  
    return loop_r0;  
  }  
}
```


Mutual recursion

```
letrec
```

```
  odd(x) =  
    if x then  
      let x1 = sub(x, 1)  
      in even(x1)  
    else  
      0
```

```
  even(x) =  
    if x then  
      let x1 = sub(x, 1)  
      in odd(x1)  
    else  
      1
```

```
in odd(91)
```



```
static boolean odd(int x) {  
  if (x != 0) {  
    int x1 = x - 1;  
    return even(x1);  
  } else {  
    return false;  
  }  
}
```

```
static boolean even(int x) {  
  if (x != 0) {  
    int x1 = x - 1;  
    return odd(x1);  
  } else {  
    return true;  
  }  
}
```

```
static boolean f() {  
  return odd(91);  
}
```


we can do
better



A person wearing a green and white striped shirt is holding an orange tabby cat up by its front paws. The cat is looking directly at the camera with a neutral expression. The person's arms are visible, and they are holding the cat's paws in a way that makes it look like the cat is giving a thumbs up. The background is a dark, patterned surface. The text "drop the lambdas" is overlaid in white, lowercase letters across the middle of the image.

drop the lambdas

Lambda dropping

letrec

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in even(x1)  
  else  
    0
```

```
even(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in odd(x1)  
  else  
    1
```

in odd(91)



letrec

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in letrec  
      even(y) =  
        if y then  
          let y1 = sub(y, 1)  
          in odd(y1)  
        else  
          1  
      in even(x1)  
  else  
    0
```

in odd(91)

Mutual recursion (again)

letrec

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in letrec  
      even(y) =  
        if y then  
          let y1 = sub(y, 1)  
          in odd(y1)  
        else  
          1  
    in even(x1)  
  else  
    0
```

in odd(91)



```
static boolean f() {  
  int odd_x, even_y;  
  odd_x = 91;  
  goto odd;
```

```
odd:  
  if (odd_x != 0) {  
    int x1 = odd_x - 1;  
    even_y = x1;  
    goto even;  
  } else {  
    return false;  
  }
```

```
even:  
  if (even_y != 0) {  
    int y1 = even_y - 1;  
    odd_x = y1;  
    goto odd;  
  } else {  
    return true;  
  }  
}
```


WOW



Mutual recursion (non tail)

letrec

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in even(x1)  
  else  
    0
```

```
even(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in odd(x1)  
  else  
    1
```

```
in let z = odd(91)  
    w = even(92)  
in and(z, w)
```



```
static boolean odd(int x) {  
  if (x != 0) {  
    int x1 = x - 1;  
    return even(x1);  
  } else {  
    return false;  
  }  
}
```

```
static boolean even(int x) {  
  if (x != 0) {  
    int x1 = x - 1;  
    return odd(x1);  
  } else {  
    return true;  
  }  
}
```

```
static boolean f() {  
  boolean z = odd(91);  
  boolean w = even(92);  
  return z && w;  
}
```



specialise

Specialisation

letrec

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in even(x1)  
  else  
    0
```

```
even(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in odd(x1)  
  else  
    1
```

```
in let z = odd(91)  
    w = even(92)  
in and(z, w)
```



letrec

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in odd_even(x1)  
  else  
    0
```

```
odd_even(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in odd(x1)  
  else  
    1
```

...

Specialisation

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in odd_even(x1)  
  else  
    0
```

```
odd_even(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in odd(x1)  
  else  
    1
```

```
even(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in even_odd(x1)  
  else  
    1
```

```
even_odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in even(x1)  
  else  
    0
```

Lambda dropping

```
odd(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in letrec  
      odd_even(y) =  
        if y then  
          let y1 = sub(y, 1)  
          in odd(y1)  
        else  
          1  
    in odd_even(y1)  
  else  
    0
```

```
even(x) =  
  if x then  
    let x1 = sub(x, 1)  
    in letrec  
      even_odd(y) =  
        if y then  
          let y1 = sub(y, 1)  
          in even(y1)  
        else  
          0  
    in even_odd(y1)  
  else  
    1
```

Mutual recursion (done!)

```
static boolean odd(int odd_x) {
odd:
    if (odd_x != 0) {
        int x1 = odd_x - 1;
        odd_even_y = x1;
        goto odd_even;
    } else {
        return false;
    }

odd_even:
    if (odd_even_y != 0) {
        int y1 = odd_even_y - 1;
        odd_x = y1;
        goto odd;
    } else {
        return true;
    }
}
```

```
static boolean even(int even_x) {
even:
    if (even_x != 0) {
        int x1 = even_x - 1;
        even_odd_y = x1;
        goto even_odd;
    } else {
        return true;
    }

even_odd:
    if (even_odd_y != 0) {
        int y1 = even_odd_y - 1;
        even_x = y1;
        goto even;
    } else {
        return false;
    }
}
```


We have full support for direct tail calls!



such tail

wow

very speed



indirect calls

“in F# on .NET (which supports tail calls) there is really nice support for asynchronous programming that depends on tail calls to avoid the stack increasing when you swap between different asynchronous handlers and lightweight software threads.”

– Rowan Davies

We still can't do that :(

A photograph of a person with reddish hair, wearing a green t-shirt and blue jeans, captured mid-jump on a blue trampoline. The trampoline is set up on a green lawn. In the background, there is a green metal shed, a large green bush, and several tall evergreen trees under a bright sky. The text "resort to trampolines?" is overlaid in white on the right side of the image.

resort to
trampolines?

Trampolines

```
interface Cont {  
    Cont invoke();  
}  
  
class Foo implements Cont {  
    Cont invoke() {  
        return new Bar(1, 2, 3);  
    }  
}  
  
class Bar implements Cont {  
    ...  
}
```

```
static void trampoline(Cont k) {  
    while (k != null) {  
        k = k.invoke();  
    }  
}
```

Slow! :(

Direct tails calls should
always be fast and efficient

Even on the JVM!

Acknowledgements

- Patryk Zadarnowski - for many long conversations on optimising administrative normal form in the context of the JVM and in particular for pointing me towards the technique of lambda dropping.
- Conrad Parker - for great constructive feedback on earlier incarnations of the talk.