Metascala

A tiny DIY JVM

https://github.com/lihaoyi/Metascala

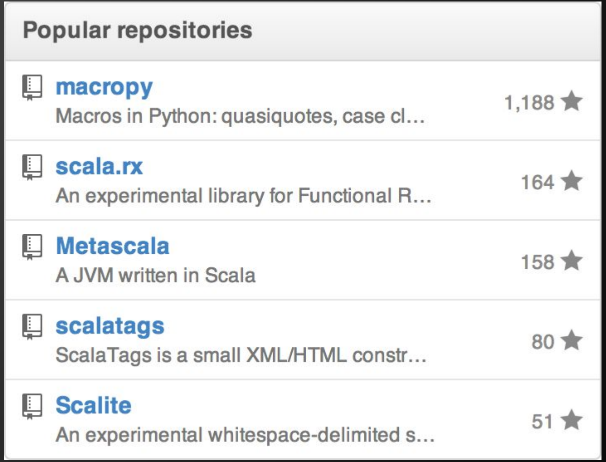
Li Haoyi
haoyi@dropbox.com
Scala Exchange
2nd Dec 2013

Who am I?

Li Haoyi

Write Python during the day

Write Scala at night



Popular repositories

macropy Macros in Python: quasiquotes, case cl...	1,188 ★
scala.rx An experimental library for Functional R...	164 ★
Metascala A JVM written in Scala	158 ★
scalatags ScalaTags is a small XML/HTML constr...	80 ★
Scalite An experimental whitespace-delimited s...	51 ★

What is Metascala?

- A JVM
- in 3000 lines of Scala
- Which can load & interpret java programs
- And can interpret itself!

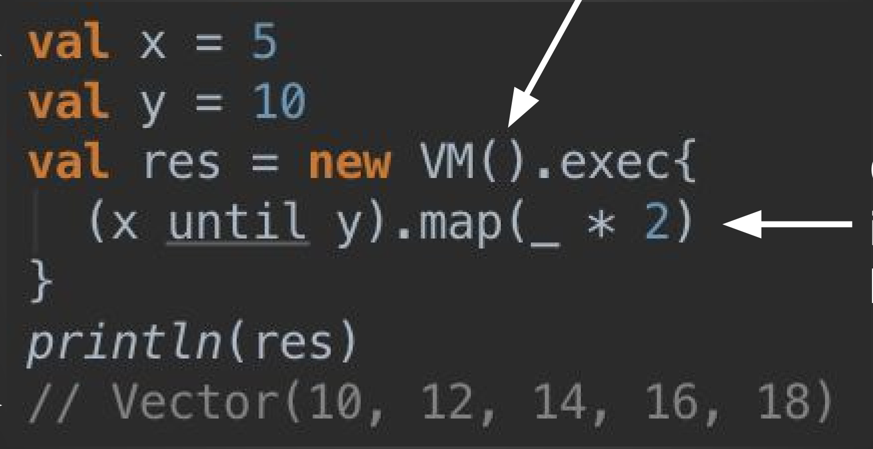
Size comparison

● Metascala:	~3,000 lines
● Avian JVM:	~80,000 lines
● OpenJDK:	~1,000,000 lines

Basic Usage

Create a new metascala VM
Plain Old Java Object

Captured variables are serialized into VM's environment



```
val x = 5
val y = 10
val res = new VM().exec{
  (x until y).map(_ * 2)
}
println(res)
// Vector(10, 12, 14, 16, 18)
```

Closure's class file is given to VM to load/parse/execute

Result is extracted from VM into host environment

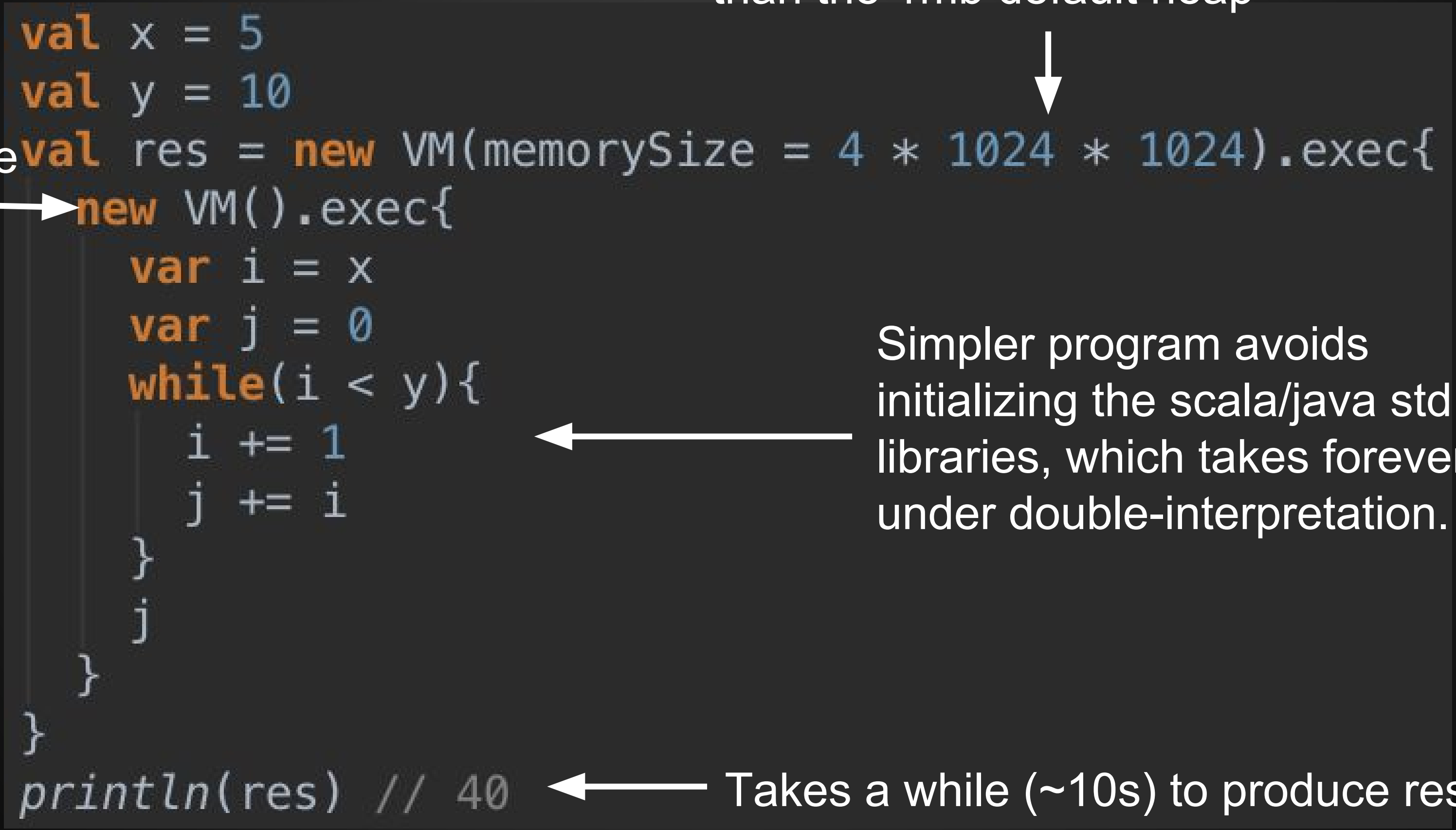
No global state

Any other classes necessary to evaluate the closure are loaded from the current Classpath

It's Metacircular!

Need to give the outer VM more than the 1mb default heap

VM inside a VM!



```
val x = 5
val y = 10
val res = new VM(memorySize = 4 * 1024 * 1024).exec{
  new VM().exec{
    var i = x
    var j = 0
    while(i < y){
      i += 1
      j += i
    }
    j
  }
}
println(res) // 40
```

Simpler program avoids initializing the scala/java std libraries, which takes forever under double-interpretation.

Takes a while (~10s) to produce result

Limitations

- Single-threaded
- Limited IO
- Slowww

Performance Comparison

- OpenJDK: 1.0x
- Metascala: ~100x
- Meta-Metascala: ~10000x

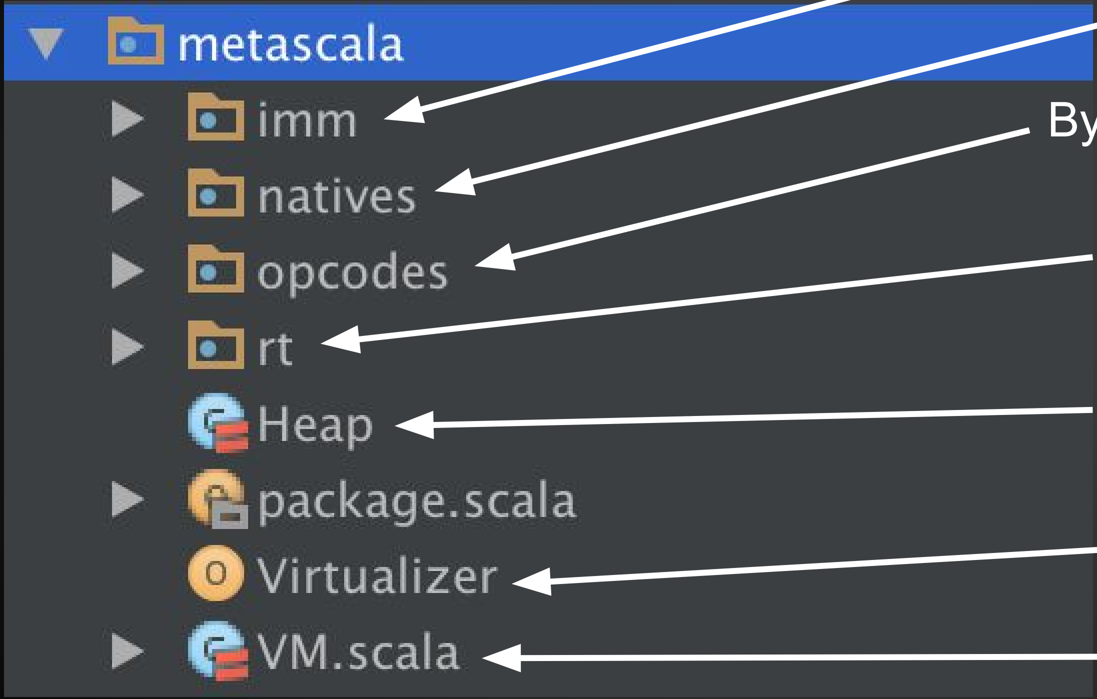
Why Metascala?

- Fun to explore the innards of the JVM
- An almost-fully secure Java runtime!
- Small size makes fiddling fun

Why Metascala?

- **Fun to explore the innards of the JVM**
- An almost-fully secure Java runtime!
- Small size makes fiddling fun

Quick Tour



Immutable Defs: ~380 loc

Native Bindings: ~650 loc

Bytecode SSA transform: ~650 loc

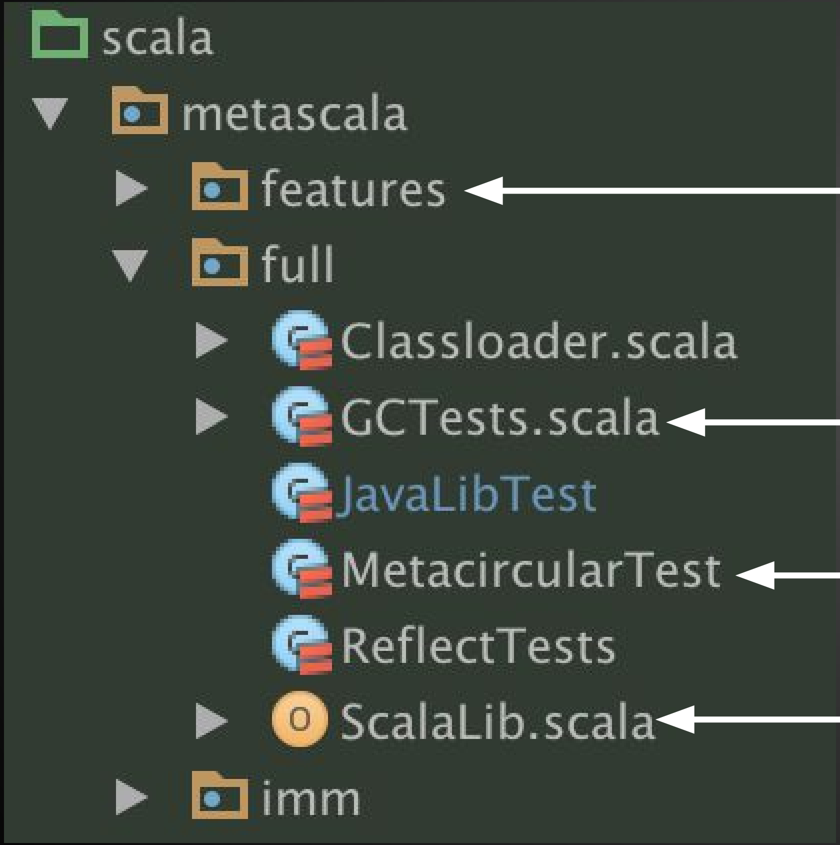
Runtime data structures: 820 loc

Binary heap & Copying GC: 132 loc

DIY scala-pickling: 132 loc

“This is a VM”: 243 loc

Quick Tour: Tests



```
scala
▼ metascala
  ▶ features
  ▼ full
    ▶ Classloader.scala
    ▶ GCTests.scala
      JavaLibTest
      MetacircularTest
      ReflectTests
    ▶ ScalaLib.scala
  ▶ imm
```

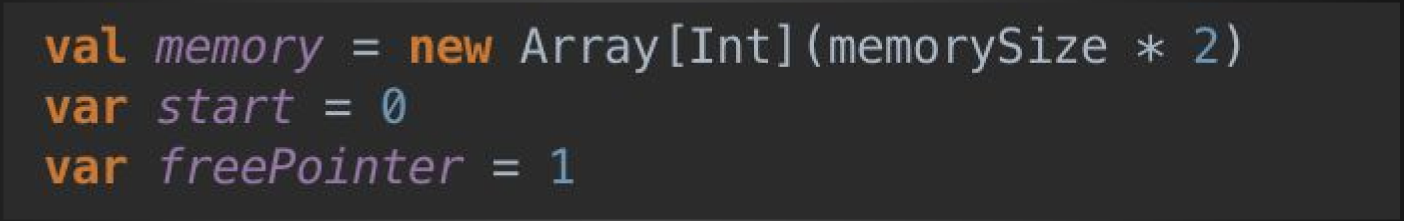
Tests for basic Java features

GC Fuzz-tests

Test Metacircularity!

Scala std lib usage

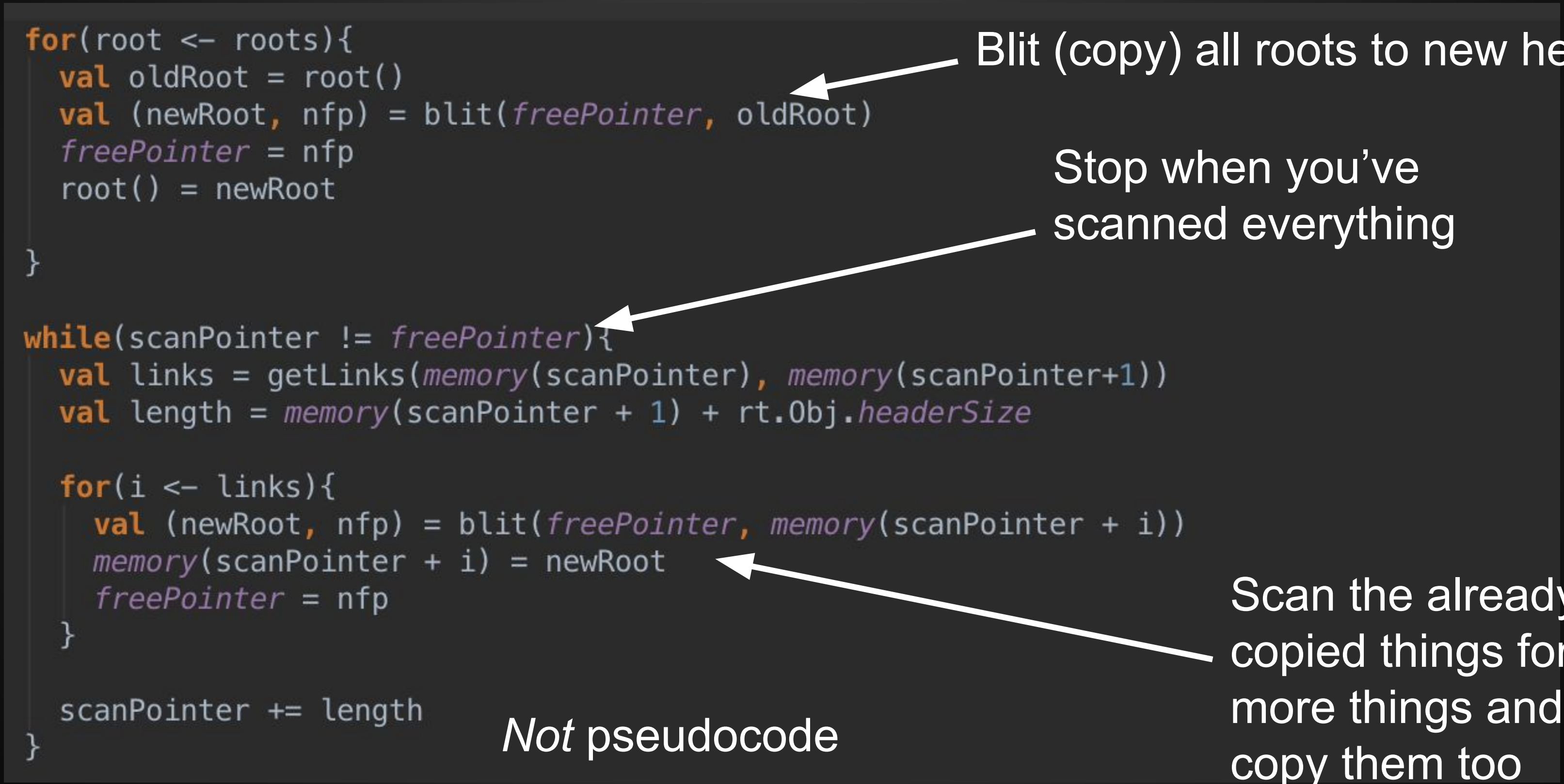
What’s a Heap?



```
val memory = new Array[Int](memorySize * 2)
var start = 0
var freePointer = 1
```

Fig 1. A Heap

What’s a Garbage Collector?



```
for(root <- roots){
  val oldRoot = root()
  val (newRoot, nfp) = blit(freePointer, oldRoot)
  freePointer = nfp
  root() = newRoot

}

while(scanPointer != freePointer){
  val links = getLinks(memory(scanPointer), memory(scanPointer+1))
  val length = memory(scanPointer + 1) + rt.Obj.headerSize

  for(i <- links){
    val (newRoot, nfp) = blit(freePointer, memory(scanPointer + i))
    memory(scanPointer + i) = newRoot
    freePointer = nfp
  }

  scanPointer += length
}
```

Blit (copy) all roots to new heap

Stop when you’ve scanned everything

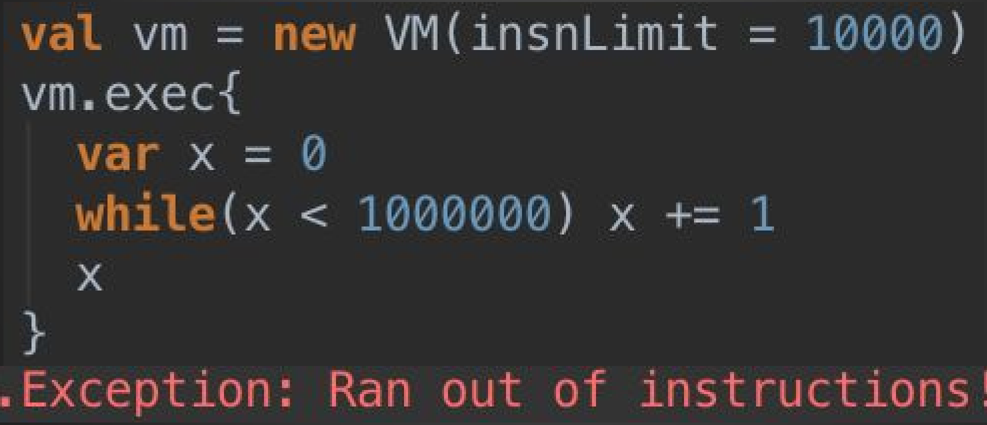
Scan the already copied things for more things and copy them too

Not pseudocode

Why Metascala?

- Fun to explore the innards of the JVM
- <u>**An almost-fully secure Java runtime!**</u>
- Small size makes fiddling fun

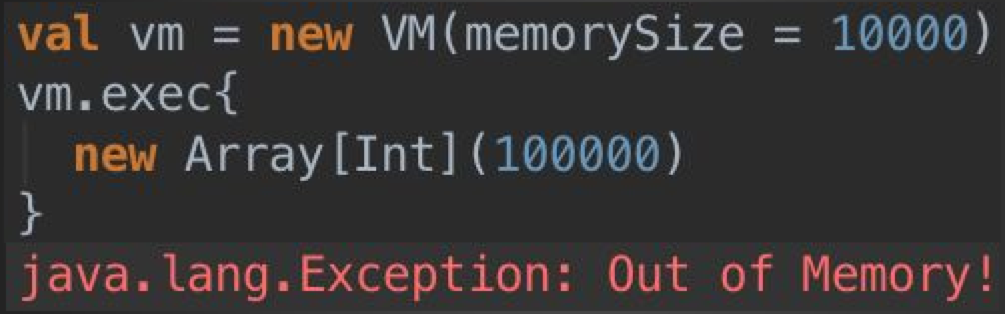
Limited Instruction Count



```
val vm = new VM(insnLimit = 10000)
vm.exec{
  var x = 0
  while(x < 1000000) x += 1
  x
}
```

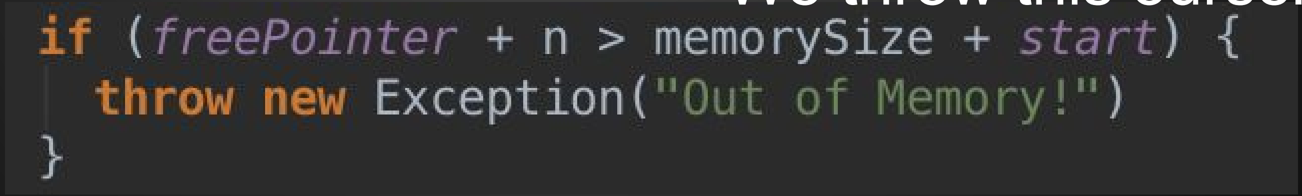
```
java.lang.Exception: Ran out of instructions! Limit: 10000
```

And Limited Memory!



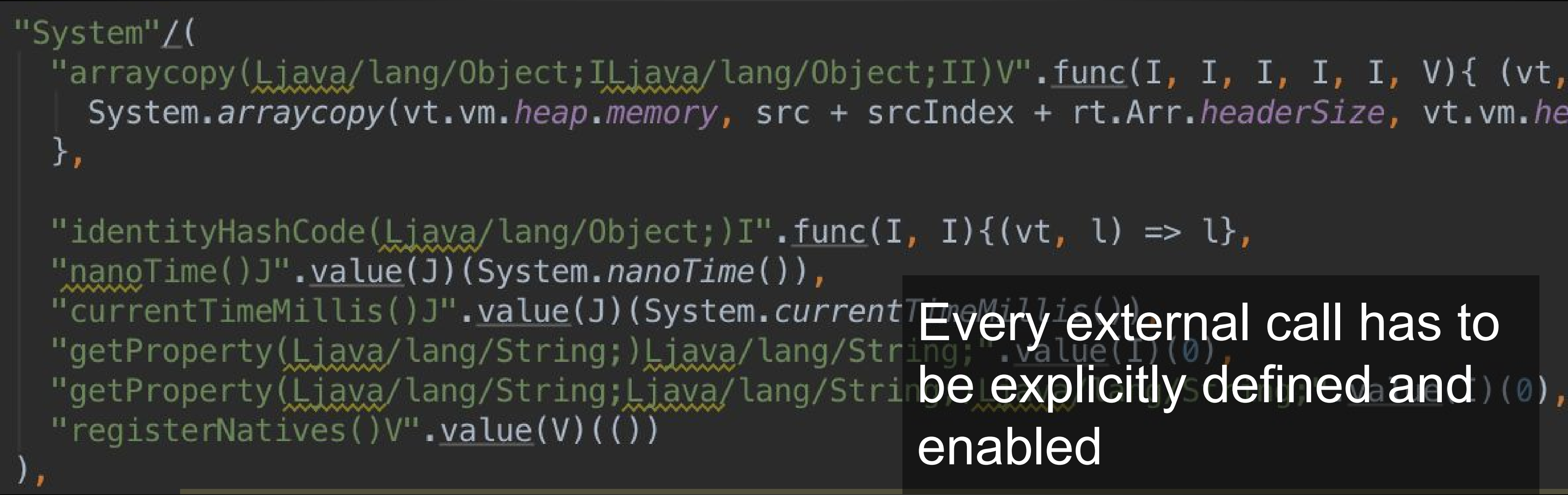
```
val vm = new VM(memorySize = 10000)
vm.exec{
  new Array[Int](100000)
}
java.lang.Exception: Out of Memory!
```

Not an OOM Error!
We throw this ourselves



```
if (freePointer + n > memorySize + start) {
  throw new Exception("Out of Memory!")
}
```

Explicitly defined capabilities



```
"System"/(
  "arraycopy(Ljava/lang/Object;ILjava/lang/Object;II)V".func(I, I, I, I, I, V){ (vt,
    System.arraycopy(vt.vm.heap.memory, src + srcIndex + rt.Arr.headerSize, vt.vm.he
  },

  "identityHashCode(Ljava/lang/Object;)I".func(I, I){(vt, l) => l},
  "nanoTime()J".value(J)(System.nanoTime()),
  "currentTimeMillis()J".value(J)(System.currentTimeMillis()),
  "getProperty(Ljava/lang/String;)Ljava/lang/String;".value(I)(0),
  "getProperty(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;".value(I)(0),
  "registerNatives()V".value(V)(())
),
```

Every external call has to be explicitly defined and enabled

Security Characteristics

- Finite instruction count
- Finite memory
- Well-defined interface to outside world
- Doesn’t rely on Java Security Model at all!

- Still some holes…

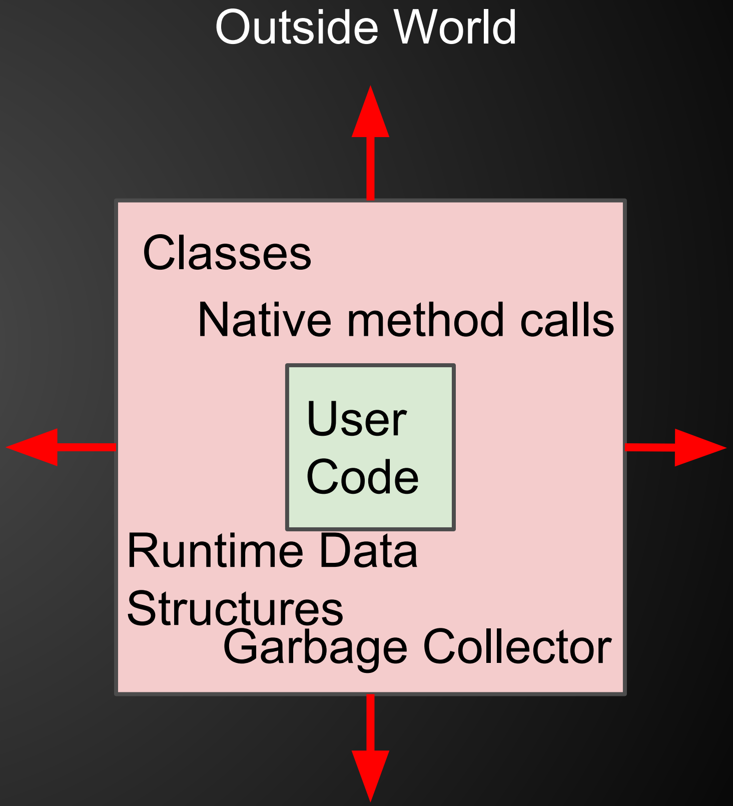
Security Holes

- Classloader can read from anywhere
- Time spent classloading not accounted
- Memory spent on classes not accounted
- GC time not accounted
- “native” methods’ time/memory not accounted

Basic Problem

User code resource consumption is bounded

VM's runtime resource usage can be made to grow arbitrarily large

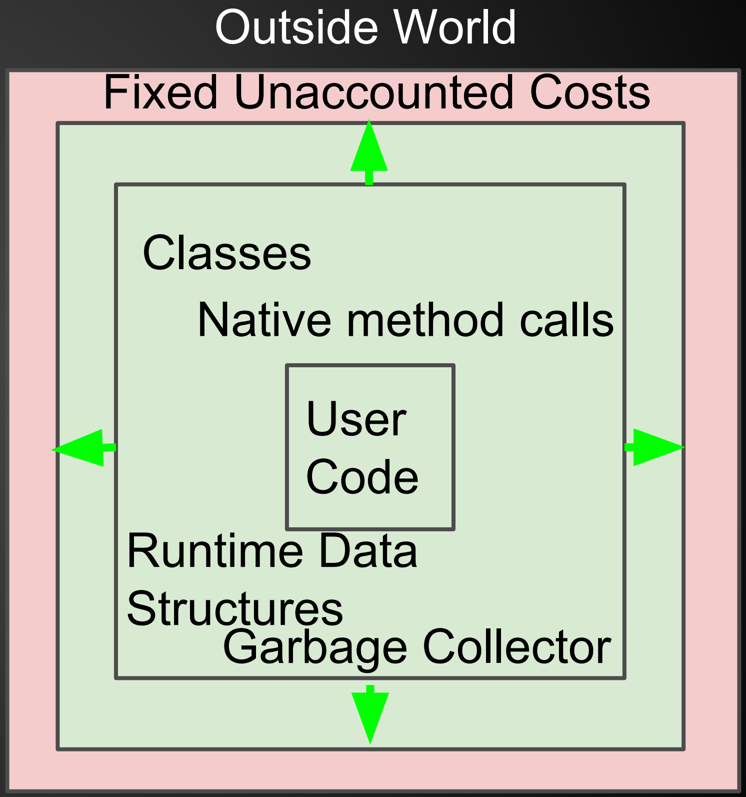


Possible Solution

Put a VM Inside a VM!

Works,

... but 10000x slowdown

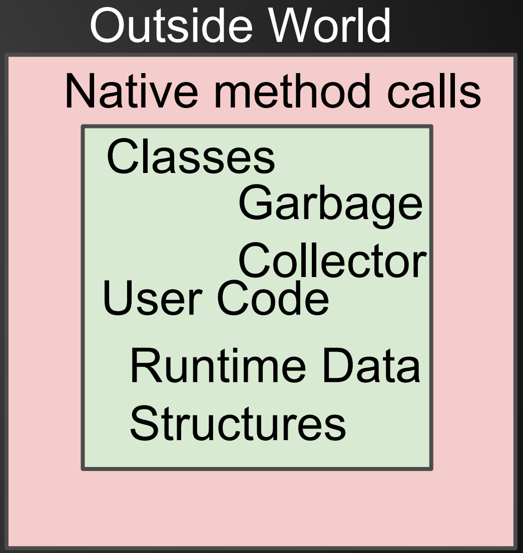


Another Possible Solution

Move more components into virtual runtime

Difficult to bootstrap correctly

WIP



Why Metascala?

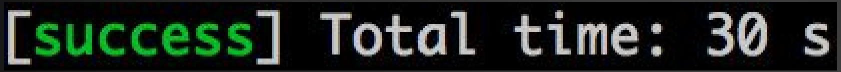
- Fun to explore the innards of the JVM
- An almost-fully secure Java runtime!
- **Small size makes fiddling fun**

Live Demo

Ugliness

- Long compile times
- Nasty JVM Interface
- Impossible Debugging

Long compile times

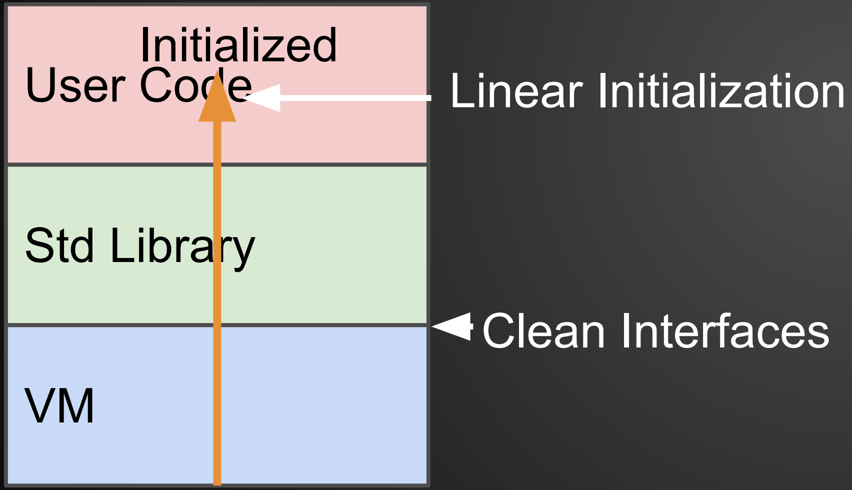
- 

```
[success] Total time: 30 s
```

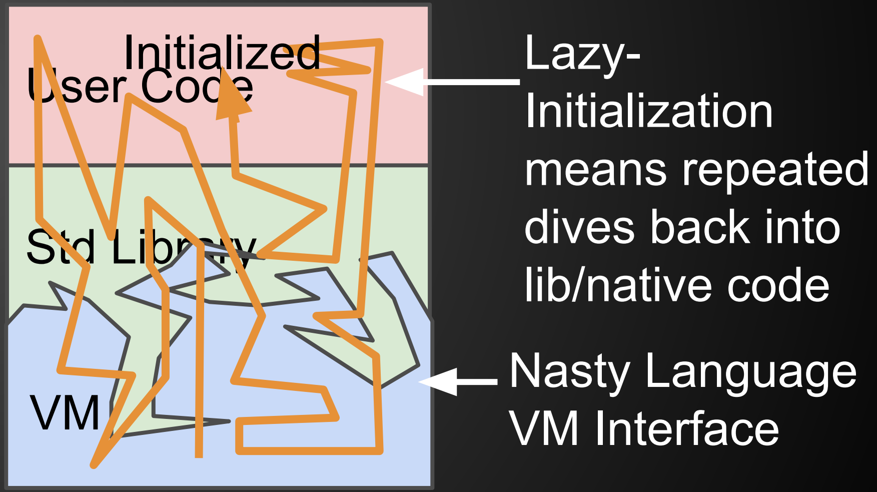
- 100 lines/s
- Twice as slow (50 lines/s) on my older machine!

Nasty JVM Interface

Ideal World

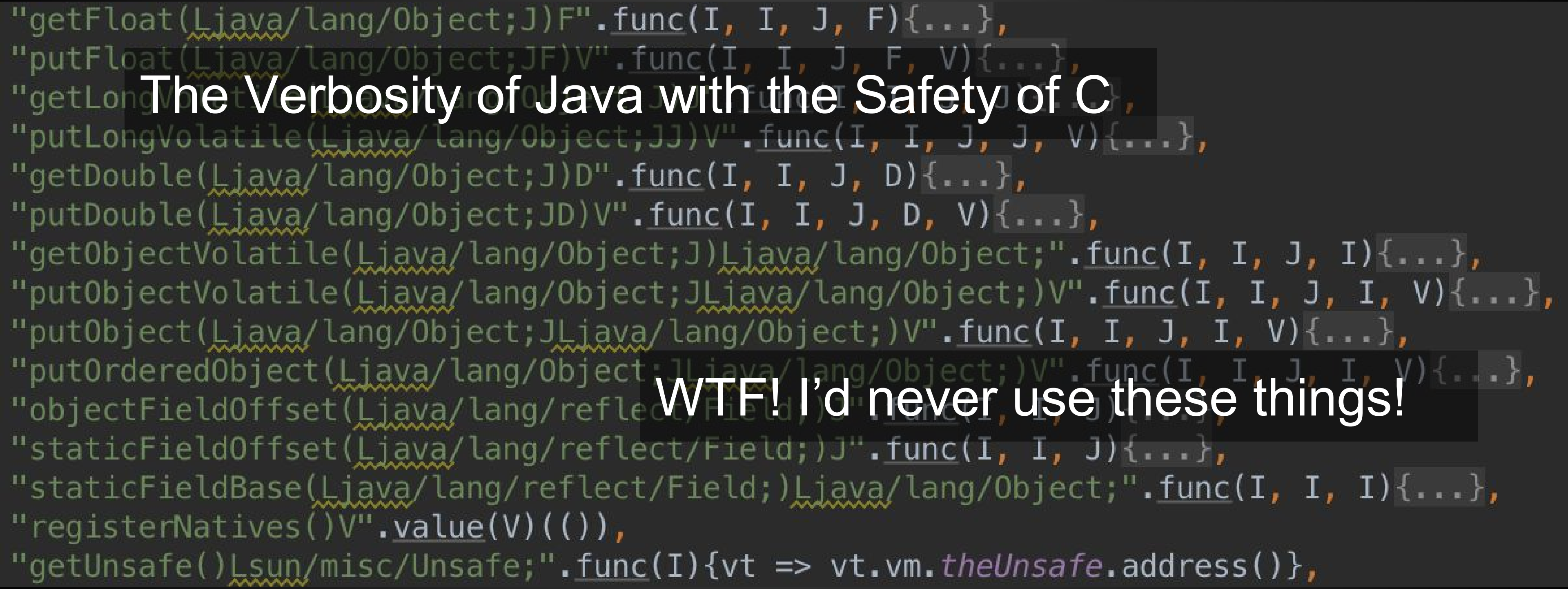


Real World



Initialized
User Code
Std Library
VM
Lazy-Initialization means repeated dives back into lib/native code
Nasty Language VM Interface

Java’s dirty little secret



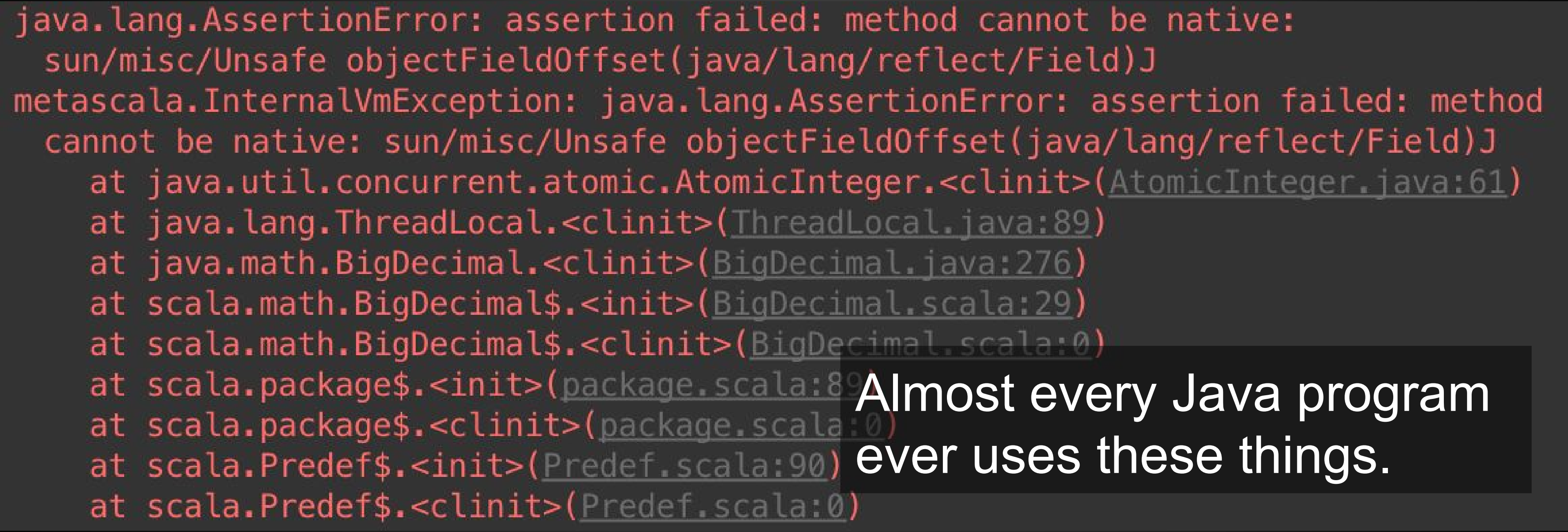
```
"getFloat(Ljava/lang/Object;J)F".func(I, I, J, F){...},
"putFloat(Ljava/lang/Object;JF)V".func(I, I, J, F, V){...},
"getLongVolatile(Ljava/lang/Object;J)J".func(I, I, J){...},
"putLongVolatile(Ljava/lang/Object;JJ)V".func(I, I, J, J, V){...},
"getDouble(Ljava/lang/Object;J)D".func(I, I, J, D){...},
"putDouble(Ljava/lang/Object;JD)V".func(I, I, J, D, V){...},
"getObjectVolatile(Ljava/lang/Object;J)Ljava/lang/Object;".func(I, I, J, I){...},
"putObjectVolatile(Ljava/lang/Object;JLjava/lang/Object;)V".func(I, I, J, I, V){...},
"putObject(Ljava/lang/Object;JLjava/lang/Object;)V".func(I, I, J, I, V){...},
"putOrderedObject(Ljava/lang/Object;JLjava/lang/Object;)V".func(I, I, J, I, V){...},
"objectFieldOffset(Ljava/lang/reflect/Field;)J".func(I, I, J){...},
"staticFieldOffset(Ljava/lang/reflect/Field;)J".func(I, I, J){...},
"staticFieldBase(Ljava/lang/reflect/Field;)Ljava/lang/Object;".func(I, I, I){...},
"registerNatives()V".value(V)(()),
"getUnsafe()Lsun/misc/Unsafe;".func(I){vt => vt.vm.theUnsafe.address()},
```

The Verbosity of Java with the Safety of C

WTF! I’d never use these things!

You probably do

What happens if you don’t have them



```
java.lang.AssertionError: assertion failed: method cannot be native:
  sun/misc/Unsafe objectFieldOffset(java/lang/reflect/Field)J
metascala.InternalVmException: java.lang.AssertionError: assertion failed: method
  cannot be native: sun/misc/Unsafe objectFieldOffset(java/lang/reflect/Field)J
    at java.util.concurrent.atomic.AtomicInteger.<clinit>(AtomicInteger.java:61)
    at java.lang.ThreadLocal.<clinit>(ThreadLocal.java:89)
    at java.math.BigDecimal.<clinit>(BigDecimal.java:276)
    at scala.math.BigDecimal$.<init>(BigDecimal.scala:29)
    at scala.math.BigDecimal$.<clinit>(BigDecimal.scala:0)
    at scala.package$.<init>(package.scala:89)
    at scala.package$.<clinit>(package.scala:0)
    at scala.Predef$.<init>(Predef.scala:90)
    at scala.Predef$.<clinit>(Predef.scala:0)
```

Almost every Java program ever uses these things.

Next Steps

- Maximize correctness
  - Implement Threads & IO
  - Fix bugs (GC, native calls, etc.)
- Solidify security characteristics
  - Still plenty of unaccounted-for memory/processing
  - Some can be hosted “within” VM itself
- Simplify Std-Lib/VM interface
  - Try using Android Std Lib?

Possible Experiments

- Native codegen instead of an interpreter?
  - Generate/exec native code through JNI
  - Heap is already a binary blob that can be easily passed to native code
- Bytecode transforms and optimizations?
  - Already in SSA form
- Continuations, Isolates, Value Classes?
- Port the whole thing to Scala.Js?

Metascala: a tiny DIY JVM

Ask me about:

- Single Static Assignment form
- Copying Garbage Collection
- sun.misc.Unsafe
- Warts of the .class file format
- Capabilities-based security
- Abandoned approaches