Whirlwind tour of Scala Native

Denys Shabalin, LAMP/EPFL

October 13, 2016

Scala?

- First appeared on January 20, 2004.
- Has grown to become a prominent JVM language.

Scala?

- *“At the root, the language’s scalability is the result of a careful integration of object-oriented and functional language concepts.”*

http://www.scala-lang.org/what-is-scala.html

Scala?

- Key insight: *functional programming is cool and we should have more of that in object-oriented languages.*

- Support for first-class functions, higher kinded types, pattern matching, local type inference, tuples, immutable collections etc etc.

- While preserving tight interoperability with Java.

JVM-only?

- There have been quite a few mixed-results experiments that tried to make it work off the JVM over the years:
  - Scala .NET (2011, Miguel Garcia)
  - Scala GWT (2012, Grzegorz Kossakowski)
  - Scala LLVM (2013, Geoff Reedy)

Scala.js

- First successful alternative platform: Scala.js (2013, Sébastien Doeraene)

- In-house incremental whole-program optimising AOT compiler to JavaScript.

Scala.js

- Introduced it's own tree-based IR as a byte-code format.

- Advanced whole-program tree shaking and plethora of Scala-specific optimisations (e.g. aggressive closure-biased inliner.)

- Support for subset of JDK libraries.

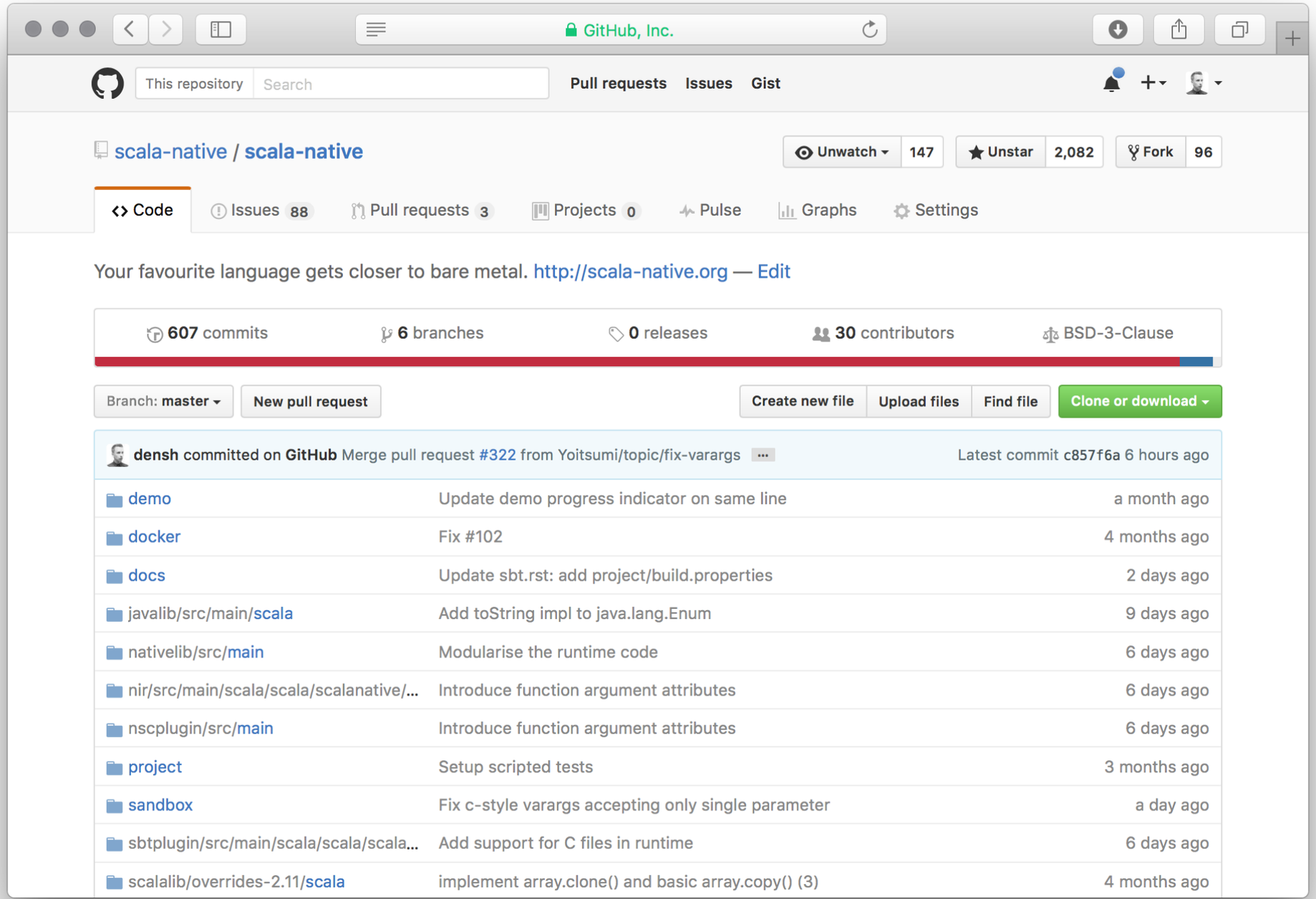
- Seamless interoperability with JavaScript.

Scala Native

- Whole-program optimising AOT compiler to native code built on top of LLVM.

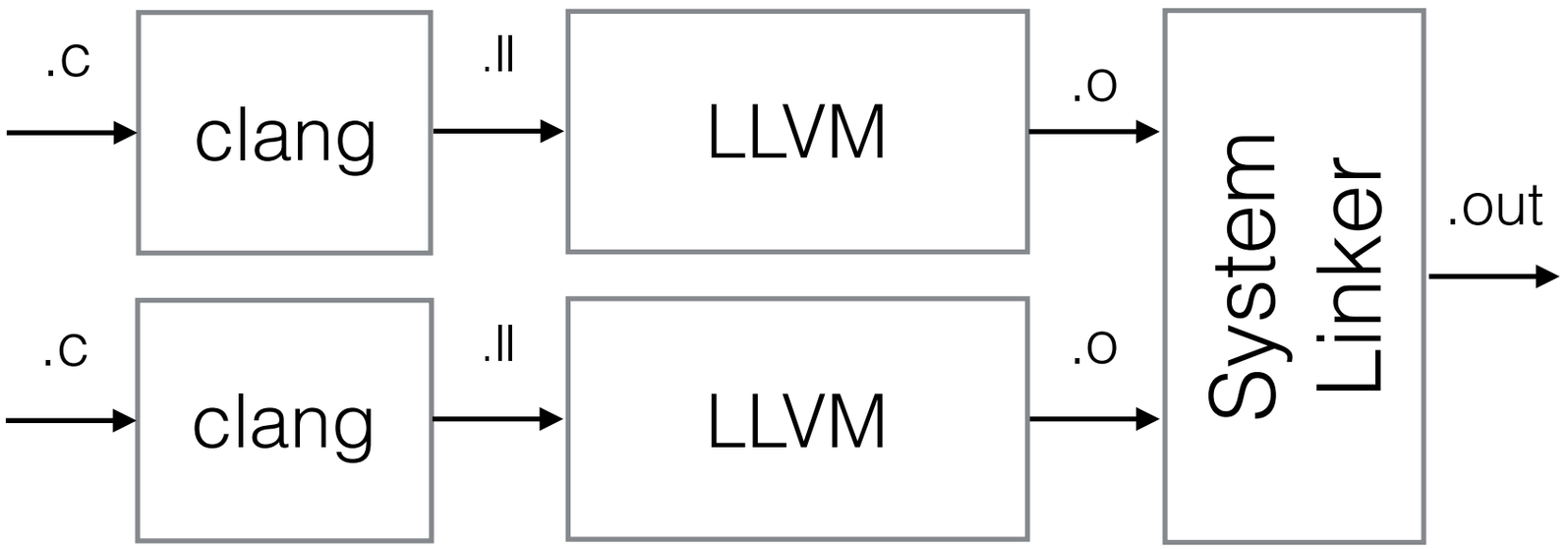
- Easy interoperability with C ABI-compatible code.

Scala Native

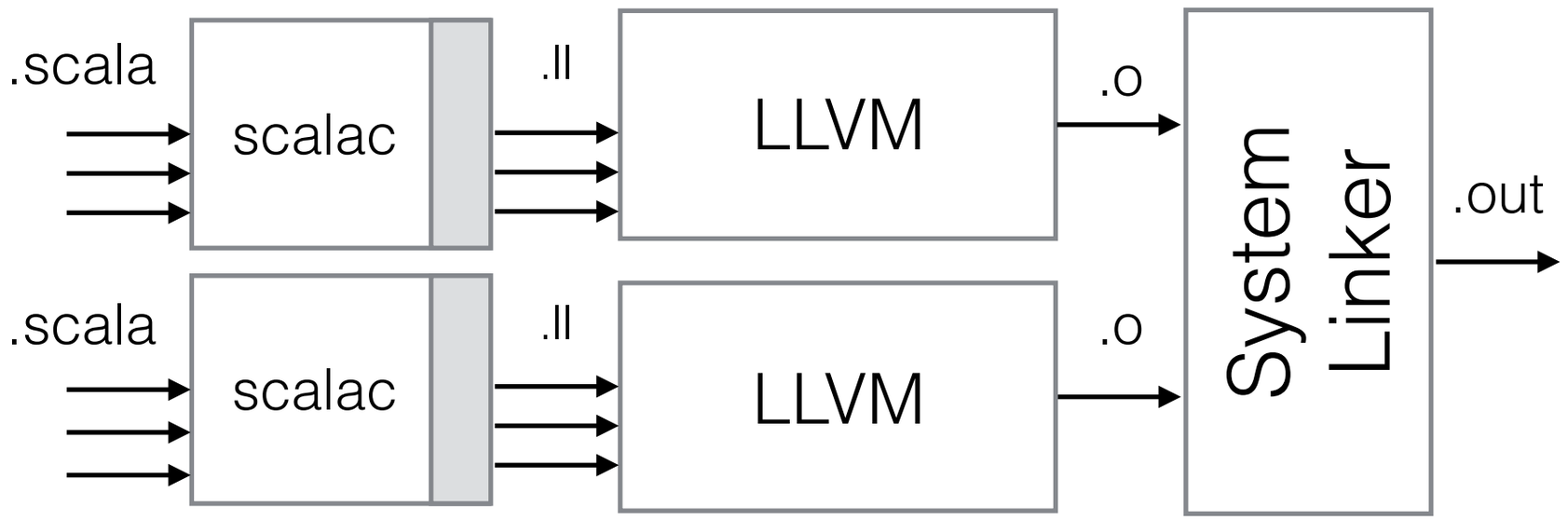


How does it work?

Clang



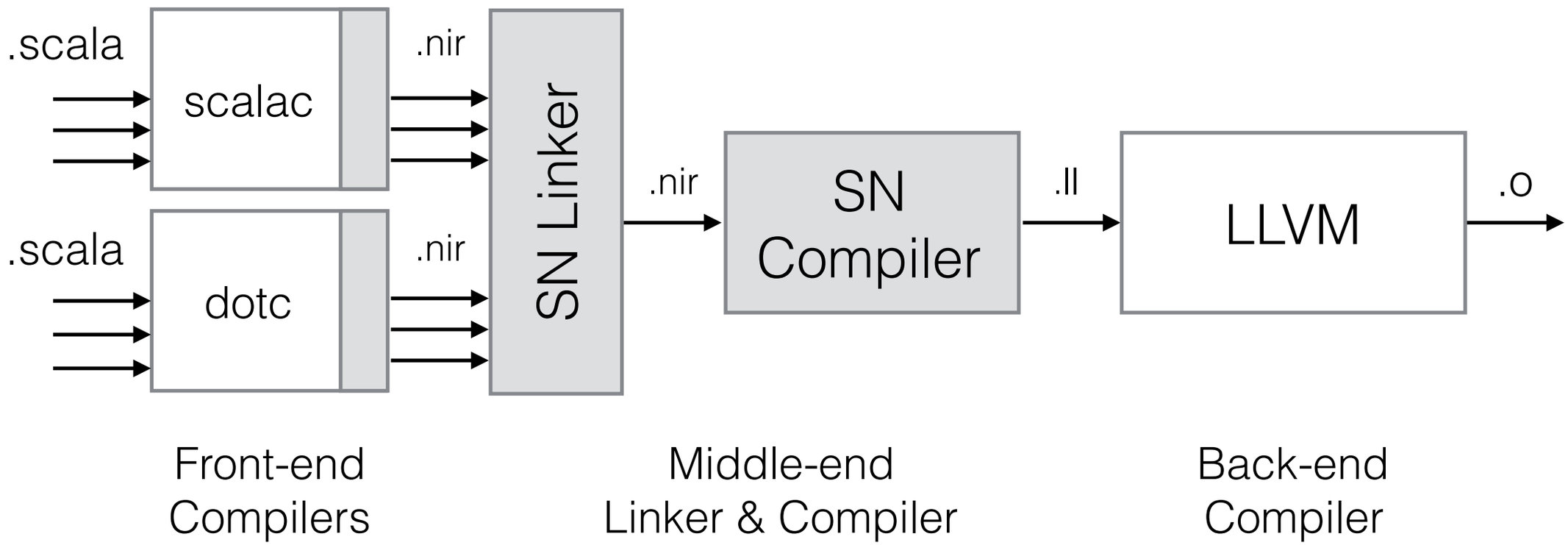
Scala LLVM


.scala
scalac
.ll
LLVM
.o
System
Linker
.out
.scala
scalac
.ll
LLVM
.o

Problems

- How do you portably distribute compiled code?
  - LLVM IR?
  - Native Object Files?
- How do you optimise features that LLVM doesn't really know about across module boundaries?

Scala Native



Front-ends (2x)

- One stable and battle-tested and one experimental and exciting.

- Parses, type-checks, infers types.

- Translates away all the functional features to corresponding object-oriented equivalents.

- Mostly the same codebase for JVM, JS and Native back-ends.

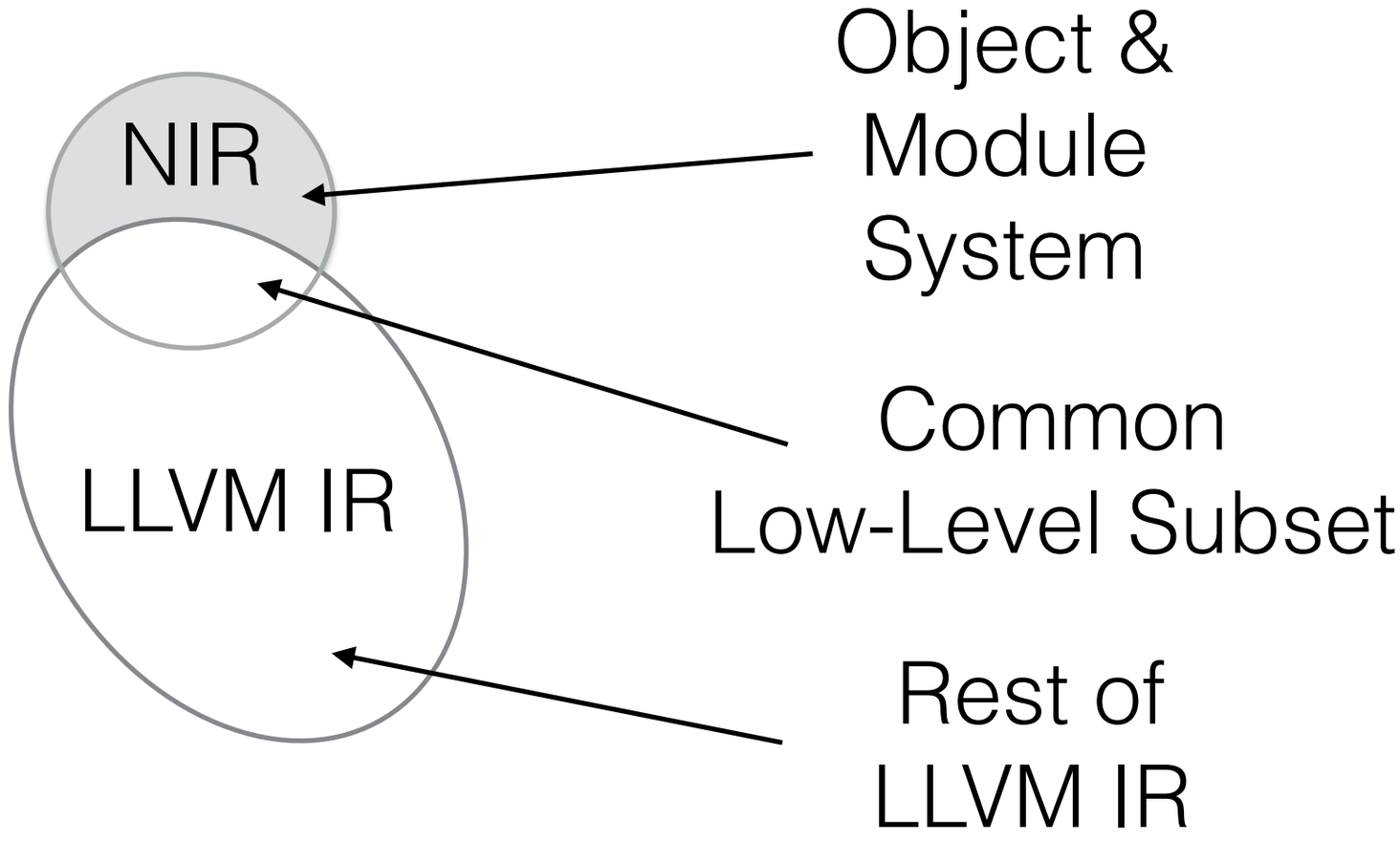
Middle-end

- Link and tree shake the whole program.
- Compile all the higher-level features away given the static knowledge of the remaining code.
- Single intermediate representation: NIR

NIR

- Typed high-level object-oriented SSA representation.
- Mixes high-level and low-level primitives.
- We took quite a bit of ideas from SIL.

NIR


NIR
LLVM IR
Object &
Module
System
Common
Low-Level Subset
Rest of
LLVM IR

NIR

How to draw an owl

1.

2.

1. Draw some circles

2. Draw the rest of the owl

NIR

```
object Test {
  def main(args: Array[String]): Unit =
    println("Hello, world!")
}
```

NIR

```
module @Test$ : @java.lang.Object

def @Test$::init : (module @Test$) => unit {
  %src.1(%src.0: module @Test$):
    %src.2 = call[...] @java.lang.Object::init(%src.0)
    ret unit
}
```

```
def @Test$::main_class.ssnr.ObjectArray_unit : ... {
  %src.2(%src.0: module @Test$,
         %src.1: class @scala.scalanative.runtime.ObjectArray):
    %src.3 = module @scala.Predef$
    %src.4 = method[...] %src.3,
             @scala.Predef$::println_class.java.lang.Object_unit
    %src.5 = call[...] %src.4(%src.3, "Hello, world!")
    ret %src.5
}
```

Middle-end

- Most of NIR trivially maps to LLVM.

- Apart from a few problematic areas.

Pain points

- First-class lazy modules
- Virtual method dispatch
- Value types & boxing
- Compilation time
- Garbage Collection

First-class lazy modules

- No top-level members in Scala
- Modules can extend classes and traits
- Initialisation happens on first access

First-class lazy modules

```
object Test {
  def main(args: Array[String]): Unit =
    println("Hello, world!")
}
```

First-class modules

```
@"value.scala.Predef$" = global i8* zeroinitializer

// check if @value.scala.Predef == null
// if not return old value
// otherwise allocate & initialise new instance
define i8* @“load.scala.Predef$"() { ... }

define void @"Test$::main_class.ssnr.ObjectArray_unit"(i8* %src.0, i8* %src.1) {
src.2:
  %src.3 = call i8* () @"load.scala.Predef$"()
  call void (i8*, i8*) @"scala.Predef$::println_class.java.lang.Object_unit"(
     i8* %src.3, i8* bitcast ({ i8*, i32, i32, i32, i8* }* @"__const.1" to i8*))
  ret void
}
```

First-class modules

- Short-term solutions:
  - Don’t generate accessors for stateless objects with side-effect-free constructors
  - Use GVN over NIR to eliminate duplicate method loads

Under development by Hugo Kapp.

Virtual methods

- Analyse class hierarchy and detect effectively final methods
- Use vtables for class virtual method dispatch
- Use statically generated dispatch table for interface virtual method dispatch

Virtual methods

- On-going experiments:
- Off-line type profiling
- Online type profiling and inline caching in AOT setting with LLVM’s patch-points.

Under research by Martin Duhem.

Value types & boxing

```
object Test {
  def main(args: Array[String]) = {
    val elems = (1 to 10000).toArray
    val transformed = elems.map(_ * 2)
    transformed.foreach(println)
  }
}
```

- Allocates two integer boxes on every iteration in map
- Allocates an integer box on every iteration in foreach

Value types & boxing

- Simple solution:
  - Pack values into pointers (aka SMI).
  - 62 bits of space should be enough for everyone.
  - Boxing/unboxing is going to shuffle bits around without actually allocating anything.

Compilation time

- Solution:
  - Incremental & parallel middle-end.
  - Incremental back-end with ThinLTO.

“Parallel Incremental Whole-Program Optimizations for Scala.js”
by Sébastien Doeraene to appear in OOPSLA’16
http://lampwww.epfl.ch/~doeraene/publications/oopsla16-incremental-optimizer.pdf

Garbage collection

- SN currently uses Boehm GC.
- No open-source high-quality LLVM-compatible GCs available to date.
- Azul seems is upstreaming improvements in this area.

Garbage collection

- Likely solution: write our own.
- Pitfalls:
  - There is no ultimate solution to root extraction (statepoint vs gcroot vs conservative)
  - LLVM optimising things in GC-unsafe way (see strong GC references discussion for details)

Questions?