ESSENTIAL VS. ACCIDENTAL COMPLEXITY IN SCALA & DOTTY

TOWARDS A PUZZLER-FREE FUTURE?

ABOUT ANDREW

- A.P. but not A.P. Marki
- maintainer of Scala Puzzlers
- Infra Builder at x.ai
- we Magically Schedule Meetings, which is awesome!

ABOUT ANDREW

- A.P. but <u>not</u> A.P. Marki
- maintainer of Scala Puzzlers
- Infra Builder at x.ai
- we Magically Schedule Meetings, which is awesome!

ABOUT DMITRY

- github.com/darkdimius
- Doing a PhD at EPFL
- Previously worked on ScalaBlitz (up to 24x faster collections)
- Since March 2014, building Dotty under supervision of Martin
- Since March 2015, working on Dotty Linker and supervising others working on Dotty

What is this all about?

Scala has lots of flexible features and attempts to combine many powerful ideas

With great power comes great ~~respons~~ability to shoot oneself in the foot

```scala
val numbers = List("1", "2").toSet() + "3"
println(numbers) // "false3"
```

Today is not about head-scratching puzzles

(Although we may sneak the odd one in here or there)

Instead, we'd like to examine:

- What can we learn from these puzzlers?
- Are there any interesting groups or clusters we can detect?
- What does that say about the language?

Specifically: Which of these puzzlers are the result of essential vs. accidental complexity in the language?

And: Which, if any, might we be able to eliminate in future?

The end of Scala Puzzlers?

Don't worry, we're ready ;-)

dottypuzzlers.com

"ESSENTIAL VS. ACCIDENTAL COMPLEXITY"...QUOI?

Ideally, we would like to have our cake and eat it too

We would like a language that is expressive, elegant *and* avoids counterintuitive behaviour

Is this possible?

Or is it inevitable that, in a language with sufficient expressive power and a non-trivial feature set...

...there will always be interactions that result in behaviour that the developer does not expect?

Perhaps it makes more sense to talk about which areas of complexity are an unavoidable price to pay for the power a language offers...

...as compared to "weirdness" that is purely a result of specification or implementation quirks

...or of features that, in hindsight, caused more problems than they were worth

TL;DR

Essential complexity: counterintuitive behaviour that is an (almost) inevitable result of the interaction of desirable language features

Accidental complexity: surprising behaviour that results from implementation quirks, specification "grey areas", or language features that are no longer considered relevant

TIME FOR SOME (SEMI-)SCIENCE

A "PUZZLER CLUSTER ANALYSIS"

n = 66 puzzlers

- Object-orientation (~18%)
- JVM & Java interop (~14%)
- First-class functions (~11%)
- Type inference (~15%)
- Conciseness & syntax sugar (~21%)
- Collections (~19%)

OBJECT ORIENTATION

```
trait A {
  val foo: Int
  val bar = 10
  println("In A: foo: " + foo + ", bar: " + bar)
}

class B extends A {
  val foo: Int = 25
  println("In B: foo: " + foo + ", bar: " + bar)
}

class C extends B {
  override val bar = 99
  println("In C: foo: " + foo + ", bar: " + bar)
}

new C // prints "In A: ... bar: 0"
```

OBJECT ORIENTATION (2)

```
object Oh {
  def overloadA(u: Unit) = "I accept a Unit"
  def overloadA(u: Unit, n: Nothing) =
  "I accept a Unit and Nothing"
  def overloadB(n: Unit) = "I accept a Unit"
  def overloadB(n: Nothing) = "I accept Nothing"
}
println(Oh overloadA 99) // compiles
println(Oh overloadB 99) // does not compile
```

JVM & JAVA INTEROP

```
class A {
  type X // equivalent to X <: Any
  var x: X = _
}
class B extends A {
  type X = Int
}

val b = new B
println(b.x) // prints "null" (!)
val bX = b.x
println(bX)
```

JVM & JAVA INTEROP

```
class A {
  type X // equivalent to X <: Any
  var x: X = _
}
class B extends A {
  type X = Int
}

val a: A = new B
println(b.x)
```

JVM & JAVA INTEROP (2)

```
import collection.JavaConverters._
def fromJava: java.util.Map[String, java.lang.Integer] = {
  val map = new java.util.HashMap[String, java.lang.Integer]()
  map.put("key", null)
  map
}

// watch out here...Integer is not Int!
val map = fromJava.asScala.
  asInstanceOf[scala.collection.Map[String, Int]]
println(map("key") == null) // prints "true"
println(map("key") == 0) // prints "true"
```

JVM & JAVA INTEROP (3)

```
object XY {
  object X {
    val value: Int = Y.value + 1
  }
  object Y {
    val value: Int = X.value + 1
  }
}
// does not loop forever only due to JVM class initialization spec
println(if (math.random > 0.5) XY.X.value else XY.Y.value)
```

FIRST-CLASS FUNCTIONS

```
type Dollar = Int
final val Dollar: Dollar = 1
val a: List[Dollar] = List(1, 2, 3)

println(a map { a: Int => Dollar }) // a map (_ => 1)
println(a.map(a: Int => Dollar)) // a map a
```

TYPE INFERENCE

```
val (x, y) = (List(1, 3, 5), List(2, 4, 6)).
 zipped find (_._1 > 10) getOrElse (10) // throws a MatchError
Console println s"Found $x"
```

CONCISENESS & SYNTAX SUGAR

```scala
implicit class Padder(val sb: StringBuilder) extends AnyVal {
  def pad2(width: Int) = {
    // not the same as ... foreach { _ => sb append '*' }
    1 to width - sb.length foreach { sb append '*' }
    sb
  }
}

// greeting.length == 14
val greeting = new StringBuilder("Hello, kitteh!")
println(greeting pad2 20)

// farewell.length == 9
val farewell = new StringBuilder("U go now.")
println(farewell pad2 20)
```

CONCISENESS & SYNTAX SUGAR (2)

```
class A {
  // RHS becomes Predef.augmentString(_).toInt
  implicit val stringToInt = (_: String).toInt
  println("4" - 2)
}
class B {
  // RHS becomes stringToInt(_).toInt
  implicit val stringToInt: String => Int = _.toInt
  println("4" - 2)
}
new A()
new B()
```

CONCISENESS & SYNTAX SUGAR (3)

```
var x = 0
def counter = { x += 1; x }
def add(a: Int)(b: Int) = a + b

val adder1 = add(counter)(_) // does not capture the value of counter
val adder2 = add(counter) _ // captures the value of counter

println("x = " + x)
println(adder1(10))
println("x = " + x)
println(adder2(10))
println("x = " + x)
```

ENTER DOTTY

PUZZLERS OF THE PAST ARE PANDORA'S BOXES

WHEN NEWCOMERS OPEN THEM, THEY MAY RUN AWAY TERRIFIED

BUT SOME PEOPLE MADE USE OF THOSE PANDORA'S BOXES

CLOSING THE BOXES

For every box we close, we need to consider migration

CLOSING THE BOXES: OVERLOADING

- Removal would simplify life a lot for compiler developers
- People love this feature; this feature is here to stay

PUZZLERS AFFECT

- Type system
- Performance
- Memory usage
- Initialization order

CLOSING THE BOXES: TYPECHECKING - TYPE PROJECTIONS

- A feature that seriously breaks the code
- You can't trust the type system

CLOSING THE BOXES: TYPECHECKING - TYPE PROJECTIONS

```scala
trait C { type A }

type T = C { type A >: Any }
type U = C { type A <: Nothing }

type X = T & U

val y: X#A = 1
val z: String = y
```

- Type projection(#) is **unsound**
- With it in the type-system, you can't trust the types!

CLOSING THE BOXES: PERFORMANCE & MEMORY LEAKS - STRUCTURAL TYPES

```
val s = new Object { val name = "foo" }
s.name
```

- Are very easy to write
- Compile into reflective calls

```
anon s = new Object() {
    private final String name;

    public String name() {
      return this.name;
    }
};
try {
(String)Test$.reflMethod$Method1(qual1.getClass())
  .invoke(qual1, new Object[0]);
}
catch (InvocationTargetException var4_4) {
 throw var4_4.getCause();
}
```

```
class Test$ {
  public static final Test$ MODULE$;
  private static Class[] reflParams$Cache1;
  private static volatile SoftReference reflPoly$Cache1;

  public static Method reflMethod$Method1(Class x$1) {
    Method method1;
    MethodCache methodCache1 =
    (MethodCache) reflPoly$Cache1.get();
    if (methodCache1 == null) {
      methodCache1 = new EmptyMethodCache();
      reflPoly$Cache1 =
        new SoftReference<MethodCache>(methodCache1);
    }
    if ((method1 = methodCache1.find(x$1)) != null) {
      return method1;
    }
    method1 = ScalaRunTime$.MODULE$.ensureAccessible(
```

CLOSING THE BOXES: INITIALIZATION ORDER

```
trait A {
  val b = 1
  println(b)
}

class A1 extends A {
} // prints 1

class A2 extends A {
  override val b = 2
} // prints 0

class A3 extends A {
  override final val b = 3
} // prints 3
```

CLOSING THE BOXES: INITIALIZATION ORDER

```
class A2 extends A {
  override val b = 2
} // prints 0
```

- People rely on this behaviour without knowing it
- Unfeasible to change at this moment

CLOSING THE BOXES: INITIALIZATION ORDER

```
class A3 extends A {
  override final val b = 3
} // prints 3
```

- We are mixing two questions: "should the member be overriden", and "should the value be inlined"
- Now we have `@inline`, which ignores initialization order

CLOSING THE BOXES: AUTO-TUPLING

- Tried to get rid of auto-tupling
- Found out that the standard library relies on it
- Spray introduced the "Magnet" design pattern that relies on it
- Need to keep it
- Can opt out using a language import

CLOSING THE BOXES: DELAYEDINIT

```scala
object InitMain{
  def main(args: Array[String]): Unit = {
    val s = p
    val p = 1
    println(s)
  }
}

object InitApp extends App {
  val s = p
  val p = 1
  println(s)
}
```

CLOSING THE BOXES: DELAYEDINIT

Classes and objects (but note, *not* traits) directly or indirectly inheriting the `DelayedInit` marker trait will have their initialization code rewritten as follows:

```
class C extends DelayedInit {
  <code>
}
```

becomes

```
class C extends DelayedInit {
  delayedInit(<code>)
}
```

CLOSING THE BOXES: DELAYEDINIT

```
class C extends DelayedInit {
  def delayedInit(body: => Unit) = {}
}
class Injector {
  def test = {
    val name = ""
    class crash extends C {
      println(name) // kills scalac
    }
  }
}
```

CLOSING THE BOXES: DELAYEDINIT

- This is now deprecated
- In Dotty, the same functionality can be achieved with trait arguments:

```
trait NewDelayedInit(body: => Unit) {
  def delayedInit(body: => Unit) = {}

  delayedInit(body)
}
```

OK, SO....WHAT CAN I DO TODAY?

- Style guide
- Linter
- Compiler plugins
- Compiler options

-Xglobal-warming

These are all "opt-in" choices...

...and typically you only opt-in after it goes wrong

for the n-th time

LEARN FROM OTHER PUZZLERS

scalapuzzlers.com

TRY DOTTY!

dotty.epfl.ch

QUESTIONS?