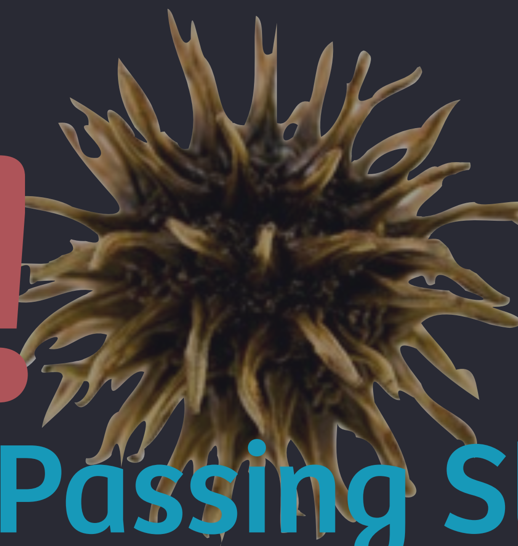
SPORES!



Towards Function-Passing Style in the Age of Concurrency & Distribution

Heather Miller

Scala Days 2014, Berlin, Germany

June 17th, 2014

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

The leading forum for research & development related to the Scala programming language.

AGENDA

SPORES

- WHY WE NEED THEM
- WHAT THEY ARE
- WHAT YOU CAN DO WITH THEM
- ¡DEMO!

AGENDA

SPORES

- WHY WE NEED THEM
- WHAT THEY ARE
- WHAT YOU CAN DO WITH THEM
- ¡DEMO!
  (spores in Scala & Javascript)

AGENDA

SPORES

- WHY WE NEED THEM
- WHAT THEY ARE
- WHAT YOU CAN DO WITH THEM
- ¡DEMO!
  (spores in Scala & Javascript)

SPOILER ALERT:
COOLEST PART OF THE TALK

TWO TRENDS

DATA-CENTRIC APPLICATIONS

CALLBACKS/REACTIVE.
CLOSURE-HEAVY.

FUNCTIONAL PROGRAMMING

APPLY FUNCTIONS TO
IMMUTABLE DATA.

OBSERVATION:

OBSERVATION:

THESE TRENDS ARE COMPLIMENTARY

OBSERVATION:

THESE TRENDS ARE COMPLIMENTARY

FUNCTIONAL PROGRAMMING A BOON TO DATA-CENTRIC PROGRAMMING

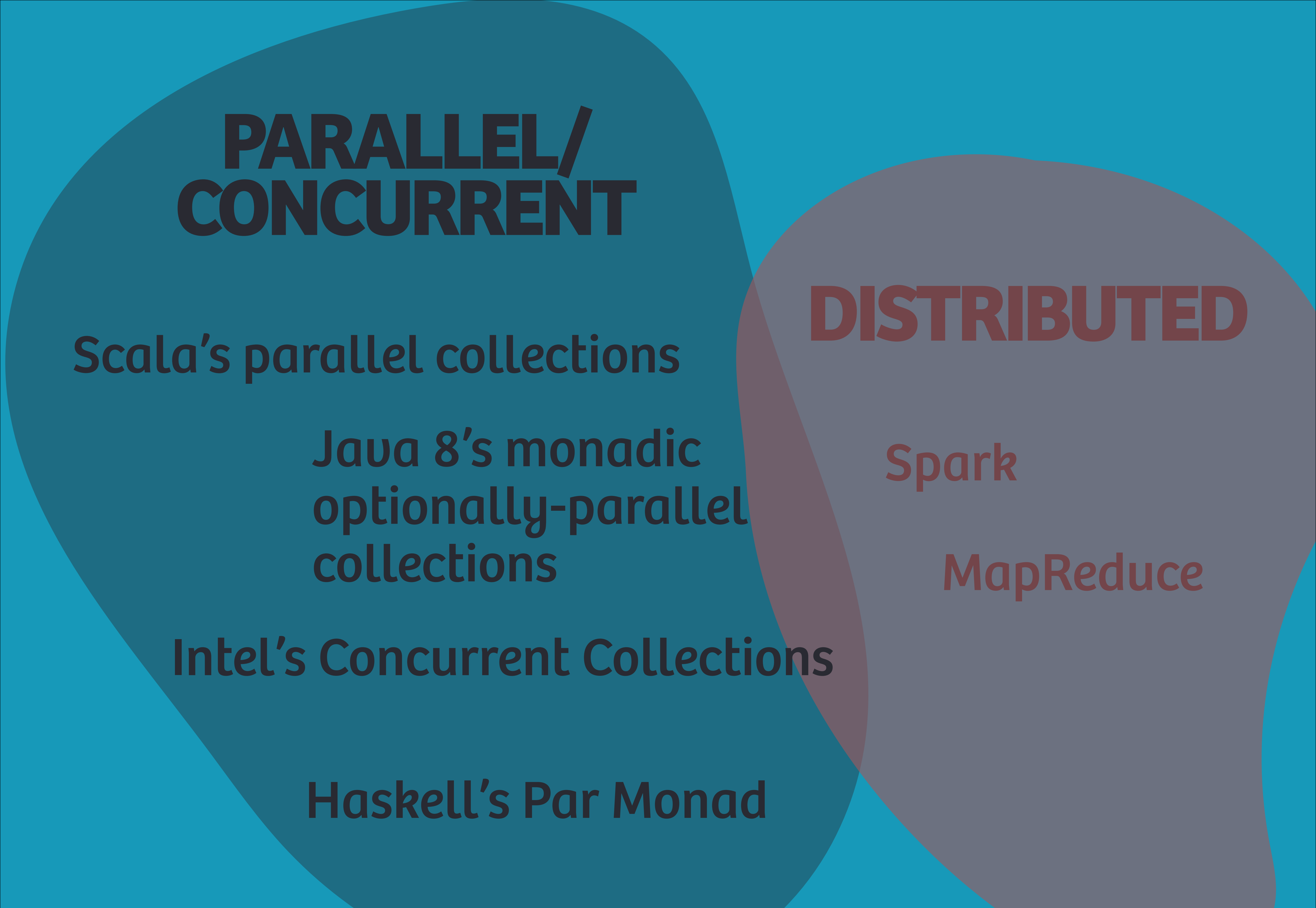
WHY?

1. The basic philosophy to transform immutable data by applying first-class functions.

2. The observation that this functional style simplifies reasoning about data in parallel, concurrent, and distributed code.

HENCE, THE POPULARITY OF DATA-PARALLEL FRAMEWORKS

DATA-PARALLEL FRAMEWORKS



WHY ARE SPORES NECESSARY?

Well,

CLOSURES ARE OFTEN A SOURCE OF HEADACHES

YOU CAN'T REALLY DISTRIBUTE THEM.

Well,

CLOSURES ARE OFTEN A SOURCE OF HEADACHES

YOU CAN'T REALLY DISTRIBUTE THEM.

NOT JUST IN SCALA OR JAVA. BUT CROSS-PARADIGM.

THE LAUNDRY LIST PROBLEMS W/ CLOSURES

1. Accidental capture of non-serializable variables (like `this`)

2. Language-specific compilation schemes that create implicit references to objects that are not serializable

3. transitive references that inadvertently hold on to excessively large object graphs creating memory leaks

THE LAUNDRY LIST PROBLEMS W/ CLOSURES CONT'D

4. Capturing references to mutable objects, leading to race conditions in a concurrent setting.

5. Unknowingly accessing object members that are not constant such as methods, which in a distributed setting can have logically different meanings on different machines.

motivating example: SPARK

```scala
class MyCoolRddApp {
 val param = 3.14
 val log = new Log(...)
 ...
 def work(rdd: RDD[Int]) {
   rdd.map(x => x + param)
      .reduce(...)
 }
}
```

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: SPARK

```
class MyCoolRddApp {
 val param = 3.14
 val log = new Log(...)
 ...
 def work(rdd: RDD[Int]) {
   rdd.map(x => x + param)
      .reduce(...)
 }
}
```

PROBLEM:

`(x => x + param)` not serializable because it captures `this` of type `MyCoolRddApp` which is itself not serializable

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

motivating example:
AKKA/FUTURES

AKKA ACTOR SPAWNS A FUTURE TO CONCURRENTLY PROCESS INCOMING REQS

```
def receive = {
  case Request(data) =>
    future {
      val result = transform(data)
      sender ! Response(result)
    }
}
```

NOT A STABLE VALUE! IT'S A METHOD CALL!

PROBLEM: Akka actor spawns future to concurrently process incoming results

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

motivating example: AKKA/FUTURES

AKKA ACTOR SPAWNS A FUTURE TO CONCURRENTLY PROCESS INCOMING REQS

```
def receive = {
  case Request(data) =>
    future {
      val result = transform(data)
      sender ! Response(result)
    }
}
```

NOT A STABLE VALUE! IT'S A METHOD CALL!

PROBLEM: Akka actor spawns future to concurrently process incoming results

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

ENTER:

SPORES

two types:

1. **MAINLINE SPORES**
   proposed as Scala Improvement Proposal

2. **SPORES WITH TYPE CONSTRAINTS**
   new research published at ECOOP’14

THIS IS ALSO RESEARCH.

Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution

Heather Miller, Philipp Haller[1], and Martin Odersky

EPFL and Typesafe, Inc.[1]
{heather.miller, martin.odersky}@epfl.ch and philipp.haller@typesafe.com[1]

Abstract. Functional programming (FP) is regularly touted as the way forward for bringing parallel, concurrent, and distributed programming to the mainstream. The popularity of the rationale behind this viewpoint has even led to a number of object-oriented (OO) programming languages outside the Smalltalk tradition adopting functional features such as lambdas and thereby function closures. However,

FOR ALL THE GORY DETAILS...

see our paper accepted for publication at ECOOP'14
http://infoscience.epfl.ch/record/191239

ENTER:

SPORES

two types:

1. **MAINLINE SPORES**
proposed as Scala Improvement Proposal

2. **SPORES WITH TYPE CONSTRAINTS**
new research published at ECOOP'14

mainline SPORES

WHAT ARE THEY?

SMALL UNITS OF POSSIBLY MOBILE

FUNCTIONAL BEHAVIOR

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

mainline SPORES

WHAT ARE THEY?

A closure-like abstraction for use in distributed or concurrent environments.

GOAL:

Well-behaved closures with controlled environments that can avoid various hazards.

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

mainline SPORES

POTENTIAL HAZARDS WHEN USING CLOSURES INCORRECTLY:

- memory leaks
- race conditions due to capturing mutable references
- runtime serialization errors due to unintended capture of references

GOAL:

Well-behaved closures with controlled environments that can avoid various hazards.

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DO SPORES LOOK LIKE?

Basic usage:

```
val s = spore {
  val h = helper
  (x: Int) => {
    val result = x + " " + h.toString
    println("The result is: " + result)
  }
}
```

THE BODY OF A SPORE CONSISTS OF 2 PARTS

1. a sequence of local value (val) declarations only (the "*spore header*"), and
2. a closure

Scala Improvement Proposal
http://docs.scala-lang.org/sips/pending/spores.html

A SPORE Guarantees...
(vs CLOSURES)

1. All captured variables are declared in the spore header, or using `capture`

2. The initializers of captured variables are executed once, upon creation of the spore

3. References to captured variables do not change during the spore's execution

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

SPORES & CLOSURES

EVALUATION SEMANTICS:

Remove the `spore` marker, and the code behaves as before

SPORES & CLOSURES ARE RELATED:

You can write a full function literal and pass it to something that expects a spore.

(Of course, only if the function literal satisfies the spore rules.)

Scala Improvement Proposal:
http://docs.scala-lang.org/sips/pending/spores.html

Ok. So.
HOW CAN YOU USE A SPORE?

IN APIS

If you want parameters to be spores, then you can write it this way

```
def sendOverWire(s: Spore[Int, Int]): Unit = ...
// ...
sendOverWire((x: Int) => x * x - 2)
```

Scala Improvement Proposal
http://docs.scala-lang.org/sips/pending/spores.html

Ok. So.

HOW CAN YOU USE A SPORE?

FOR-COMPREHENSIONS

```
def lookup(i: Int): DCollection[Int] = ...
val indices: DCollection[Int] = ...

for { i <- indices
      j <- lookup(i)
} yield j + capture(i)

trait DCollection[A] {
 def map[B](sp: Spore[A, B]): DCollection[B]
 def flatMap[B](sp: Spore[A, DCollection[B]]): DCollection[B]
}
```

Scala Improvement Proposal
http://docs.scala-lang.org/sips/pending/spores.html

Right,

WHAT DOES ALL OF THAT GET YOU?

WHAT DOES ALL OF THAT GET YOU?

SINCE...

→ Captured expressions are evaluated upon spore creation.

THAT MEANS...

→ Spores are like function values with an immutable environment.

→ Plus, environment is specified and checked, no accidental capturing.

Scala Improvement Proposal
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DOES ALL OF THAT GET YOU?

OR, GRAPHICALLY...

1
Right after creation

2
During execution

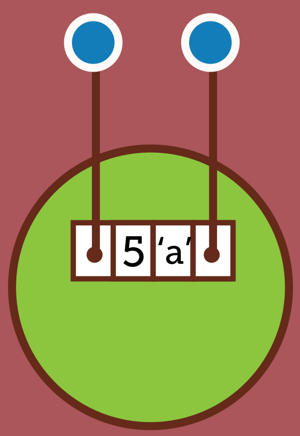
SPORES

CLOSURES

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DOES ALL OF THAT GET YOU?

OR, GRAPHICALLY...



1
Right after creation

2
During execution

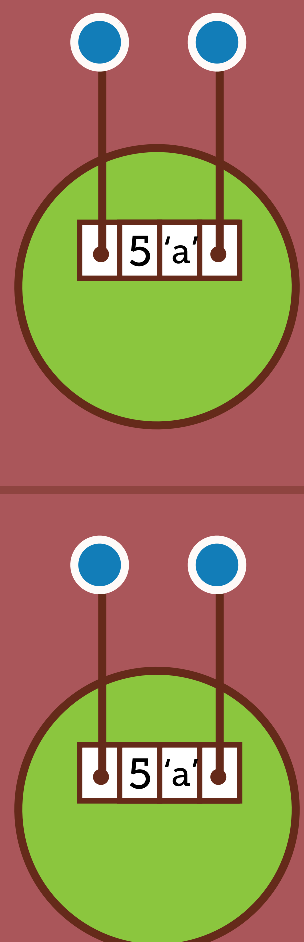
SPORES

CLOSURES

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

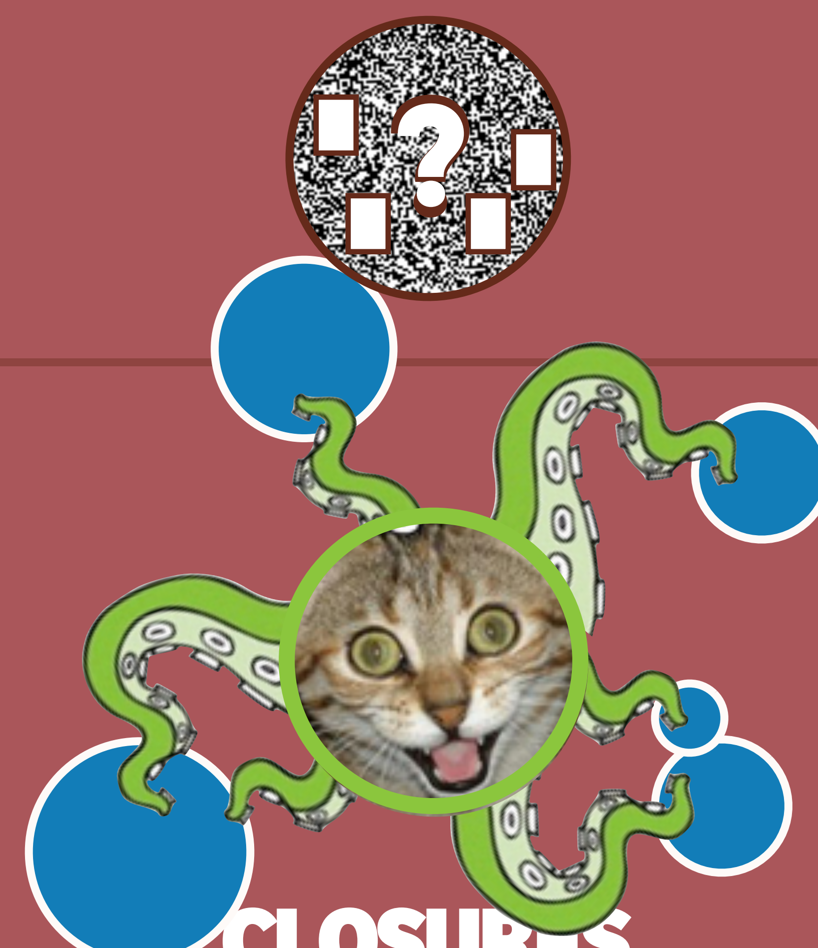
WHAT DOES ALL OF THAT GET YOU?

OR, GRAPHICALLY...



1
Right after creation

2
During execution



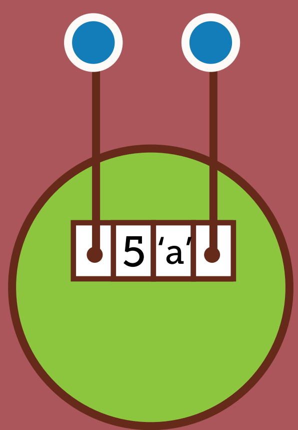
SPORES

CLOSURES

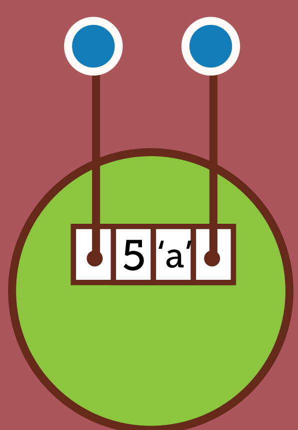
Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

WHAT DOES ALL OF THAT GET YOU?

OR, GRAPHICALLY...



1
Right after creation



2
During execution

SPORES

CLOSURES

Proposed for inclusion in Scala 2.11
http://docs.scala-lang.org/sips/pending/spores.html

Cool.

Cool. WHAT IF I CAPTURE A

SOCKET?

Cool.

WHAT IF I CAPTURE A SOCKET?

REALM OF RESEARCH

ENTER:

SPORES

two types:

1 **MAINLINE SPORES**
proposed as Scala Improvement Proposal

2 **SPORES WITH TYPE CONSTRAINTS**
new research published at ECOOP'14

Cool.

WHAT IF I CAPTURE A SOCKET?

WOULDN'T IT BE NICE IF WE COULD ADD THESE CONSTRAINTS, IN A FRIENDLY, AND COMPOSABLE WAY?

Creating SPORES w/ constraints

Idea: KEEP TRACK OF CAPTURED TYPES
...at compile-time

```
spore { val x: Int = list.size; val a: ActorRef = this.sender
  (y: Int) => ...
} exclude[Actor]
```

The spore macro can synthesize precise types automatically for newly created spores:

(a whitebox macro)

SYNTHESIZED TYPE:

```
Spore[Int, ...] {
  type Excluded = NoCapture[Actor]
  type Facts = Captured[Int] with Captured[ActorRef]
}
```

Composing SPORES *w/ constraints*

BASIC COMPOSITION OPERATORS

- `andThen`
- `compose`

(same as for regular functions)

How do we synthesize the result type of `s1 andThen s2`?

RESULT TYPE SYNTHESIZED BY `andThen` MACRO

- type member `Facts` takes "union" of the facts of `s1` and `s2`
- type member `Excluded`: conjunction of excluded types, needs to check `Facts` to see if possible

Example: Composing SPORES w/ constraints

```
val s1: Spore[Int, String] {
  type Excluded = NoCapture[Actor]
  type Facts = Captured[Int] with Captured[ActorRef]
} = ...
val s2: Spore[String, String] {
  type Excluded = NoCapture[RDD[Int]]
  type Facts = Captured[Actor]
}
s1 andThen s2  // does not compile
```

Example: Composing SPORES w/ constraints

```
val s1: Spore[Int, String] {
  type Excluded = NoCapture[Actor]
  type Facts = Captured[Int] with Captured[ActorRef]
} = ...
val s2: Spore[String, String] {
  type Excluded = NoCapture[RDD[Int]]
}
s1 andThen s2: Spore[Int, String] {
  type Excluded = NoCapture[Actor] with
NoCapture[RDD[Int]]
  type Facts = Captured[Int] with Captured[ActorRef]
}
```

WHAT DO TYPE CONSTRAINTS BUY US?

- Stronger constraints checked at compile time (not "just" basic spore rules)
- Frameworks can make stronger assumptions about spores created by users.
- Confidence in consuming, creating, and composing spores:
  - Constraints accumulate monotonically
  - Constraints are never lost when composing spores
- Less brittleness.

WHEN WOULD I WANT TO SHIP A SPORE?

HERE ARE 4 EXAMPLES WHICH COULD BE ADVANTAGEOUS (& lots probably more)

1. Move functionality to distributed (in-memory) data e.g., Spark
2. Shippable stream pipelines e.g., reconfigurable streams
3. Hot-swapping actor behavior
4. Portable closures e.g., JVM to Javascript

2 SHIPPABLE STREAM PIPELINES

```
Flow(text.split("\\s").toVector).
      // transform
      map(line => line.toUpperCase).
      // print to console (can also use ``foreach(println)``)
      foreach(transformedLine => println(transformedLine)).
      onComplete(FlowMaterializer(MaterializerSettings())) {
        case Success(_) => system.shutdown()
        case Failure(e) =>
          println("Failure: " + e.getMessage)
          system.shutdown()
      }
```

2 SHIPPABLE STREAM PIPELINES

- Each stage has a closure that deals with incoming streaming data
- However, one could imagine that it could be advantageous that each stage is on the machine that's closest to the data
- Yet we still want the code of the entire pipeline to be assembled on one machine
- That means we have to send pipeline stages together with their closures to different machines after the pipeline has been assembled

HOW DO SPORES HELP?

2 SHIPPABLE STREAM PIPELINES

- ✓ Spores ensure that serialization doesn't fail at runtime.
- ✓ Spores enable different serialization frameworks (e.g., Scala Pickling)
- ✓ Spores enable restricting types that are captured by each closure.

  e.g., if the pipeline is built by an actor, we want to ensure that the enclosing actor is never captured.

2 SHIPPABLE STREAM PIPELINES

```
Flow(text.split("\\s").toVector).
      // transform
      map(line => line.toUpperCase).
      // print to console (can also use ``foreach(println)``)
      foreach(transformedLine => println(transformedLine)).
      onComplete(FlowMaterializer(MaterializerSettings())) {
        case Success(_) => system.shutdown()
        case Failure(e) =>
          println("Failure: " + e.getMessage)
          system.shutdown()
      }
```

BEFORE

2 SHIPPABLE STREAM PIPELINES

```
Flow(text.split("\\s").toVector).
      // transform
      map(spore { line => line.toUpperCase }).
      // print to console (can also use ``foreach(println)``)
      foreach(spore { transformedLine => println(transformedLine) }).
      onComplete(FlowMaterializer(MaterializerSettings())) {
        case Success(_) => system.shutdown()
        case Failure(e) =>
          println("Failure: " + e.getMessage)
          system.shutdown()
      }
```

AFTER

3 HOT-SWAPPING ACTOR BEHAVIOR

```
class HotSwapActor extends Actor {
  import context._

  def receive = {
    case HotSwap(spore) =>
      val newBehavior: Receive = { case msg => spore(msg) }
      become(newBehavior)
    case ..
  }
}
```

4 PORTABLE CLOSURES (E.G., BETWEEN JVM & JS)

Imagine you have a rich UI on a browser-based client interacting with a server.

If fine-grained information has to be exchanged between client and server, then sending spores can simplify the problem.

1. Compose functions based on UI selections.
2. Send the composed function, and the server is very simple because it just applies the function.
3. No manual translation between low-level message fields to functions applied on the server.

TOTALLY COMPOSABLE. EASILY EXTENDABLE.

4 PORTABLE (E.G., BETWEEN JVM & JS) CLOSURES EXAMPLE

SEARCH TOOL FOR USED CAR OFFERS.

WEBSITE LETS USERS DEFINE A NUMBER OF PREFERENCES SUCH AS PRICE RANGE.

WHEN ALL PREFERENCES HAVE BEEN SELECTED, SENT TO SERVER, WHICH FILTERS CARS.

4 PORTABLE CLOSURES (E.G., BETWEEN JVM & JS) EXAMPLE

SEARCH TOOL FOR USED CAR OFFERS.

ISSUES:

Message containing all user prefs complex.

Extending website with new feature to filter for is complicated, code has to be changed in multiple locations:

- UI code
- encoding pref setting into message to send
- decoding pref setting from message received
- adapting server-side logic for new pref

4 PORTABLE (E.G., BETWEEN JVM & JS) CLOSURES EXAMPLE

CAN DO WITH SPORES IN A WAY WHERE ONLY UI NEEDS TO BE CHANGED!

1. Define spores that filter in code that's shared between client & server.

   Each filter spore has type:
   `Spore[(Car, Boolean), (Car, Boolean)]`

   This allows composing two filters using `andThen`
   ```
   val filter = filter1 andThen filter2
   ```

   A car matches in the case where
   ```
   filter((car, true))._2
   ```

4 PORTABLE (E.G., BETWEEN JVM & JS) CLOSURES EXAMPLE

Example filter spore:

```
def priceRange(from: Int, to: Int) =
  spore {
    val localFrom = from
    val localTo = to
    (pair: (Car, Boolean)) => {
      val (car, valid) = pair
      (car,
       valid && (car.price >= localFrom && car.price < localTo))
    }
  }
```

4 PORTABLE CLOSURES (E.G., BETWEEN JVM & JS) EXAMPLE

2. Compose filters on client side

   Suppose there's a collection "selections" which contains filter spores. Then, we simply fold it to get the composed filter:

```
val userPrefs =
  selections.foldLeft(idFilter)(
    (f1, f2) => f1 andThen f2
  )
```

3. Pickle and send the composed spore to the server

4 PORTABLE (E.G., BETWEEN JVM & JS) CLOSURES EXAMPLE

3. On the server side:

   Unpickle and use the filter spore to churn through a potentially larget dataset. (Can even use frameworks like Spark for that!)

```
val userPrefs =
  received.unpickle[Spore[(Car, Boolean), (Car, Boolean)]]

val eligible = carsRdd.filter {
  car => userPrefs((car, true))._2
}

// send eligible back to user
```

WHAT'S IN THE RELEASE

Spores implementation as described in SIP-21.

Pickling integration module
(see github.com/scala/pickling)

Support for a subset of type constraints
described in the ECOOP'14 paper.

Alpha version hits sonatype in the next day
or two.

THANK YOU.