A Programming Model and Foundation for Lineage-Based Distributed Computation

Heather Miller
EPFL & Northeastern University

WG 2.16 Language Design - Park City 2017

Heather Miller
EPFL & Northeastern University

#1 functions for the network

#2 implicits in Scala

Heather Miller
EPFL & Northeastern University

WG 2.16 Language Design - Park City 2017

Functions & Streaming

Recall the discussion surrounding Tom's example of XStream's transform method from yesterday.

Functions & Streaming

Recall the discussion surrounding Tom's example of XStream's transform method from yesterday.

poor reconstruction from memory:

```
function transform(f: (t:T)⇒U, ctx: {}):Stream<U> {
    …
}
```

Functions & Streaming

Recall the discussion surrounding Tom's example of XStream's transform method from yesterday.

poor reconstruction from memory:

```
function transform(f: (t:T)⇒U, ctx: {}):Stream<U> {
    …
}
```

free variables go here

Functions & Streaming

Recall the discussion surrounding Tom's example of XStream's transform method from yesterday.

observation:

- Embedding DSLs like XStream in a host language gets you all kinds of goodies like tooling, a type system you don't have to design yourself, etc.
- But you lose control in other areas; e.g., pure functions lost. Now we just have to deal with errors that may come from users not explicitly passing free variables as the ctx parameter.

Functions & Streaming

Recall the discussion surrounding Tom's example of XStream's transform method from yesterday.

question.

Manuel asked something along the lines of:

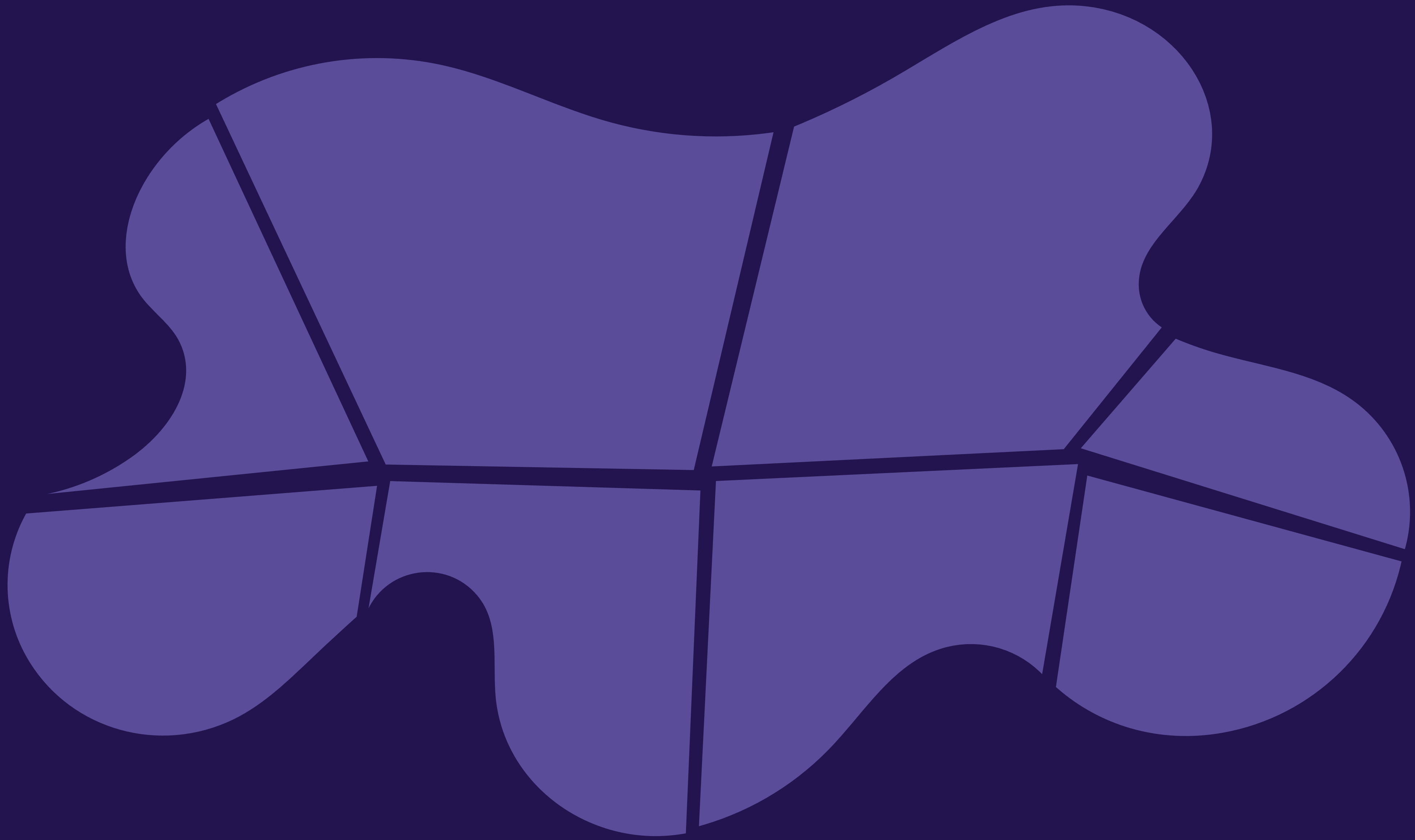
> "Would it be possible to introduce a new abstraction for functions that would ameliorate some of these issues?"

Yes.

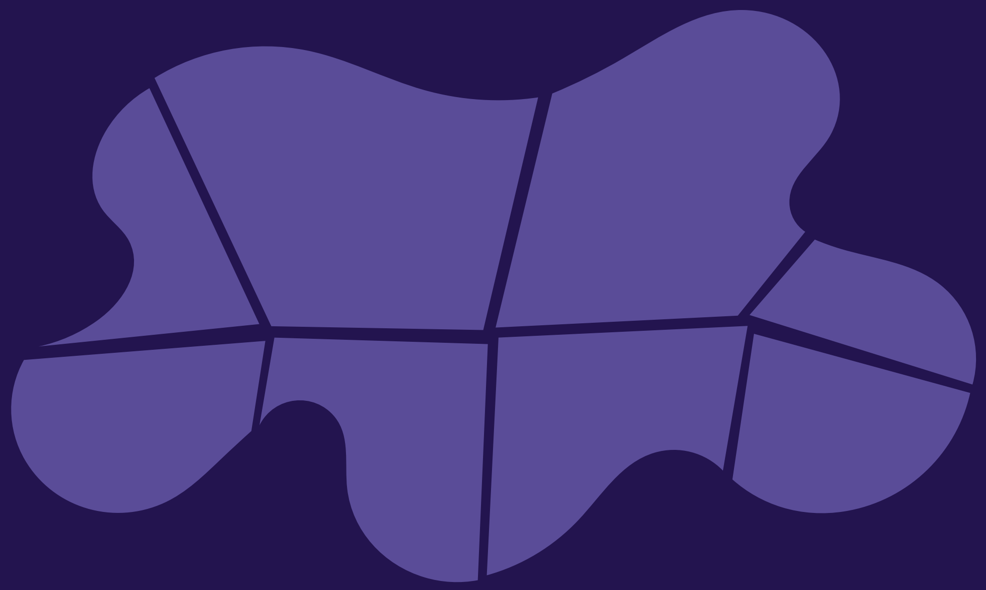
Or at least we made one attempt so far in the context of Scala/the JVM.

My work has been in the context of Spark.

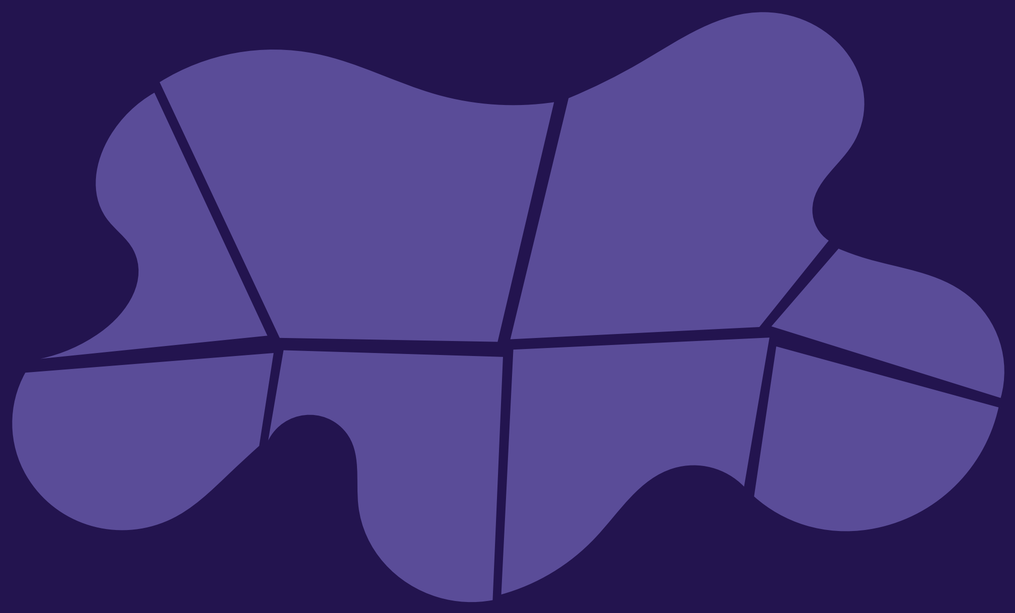
wikipedia
reduced, 48.4GB



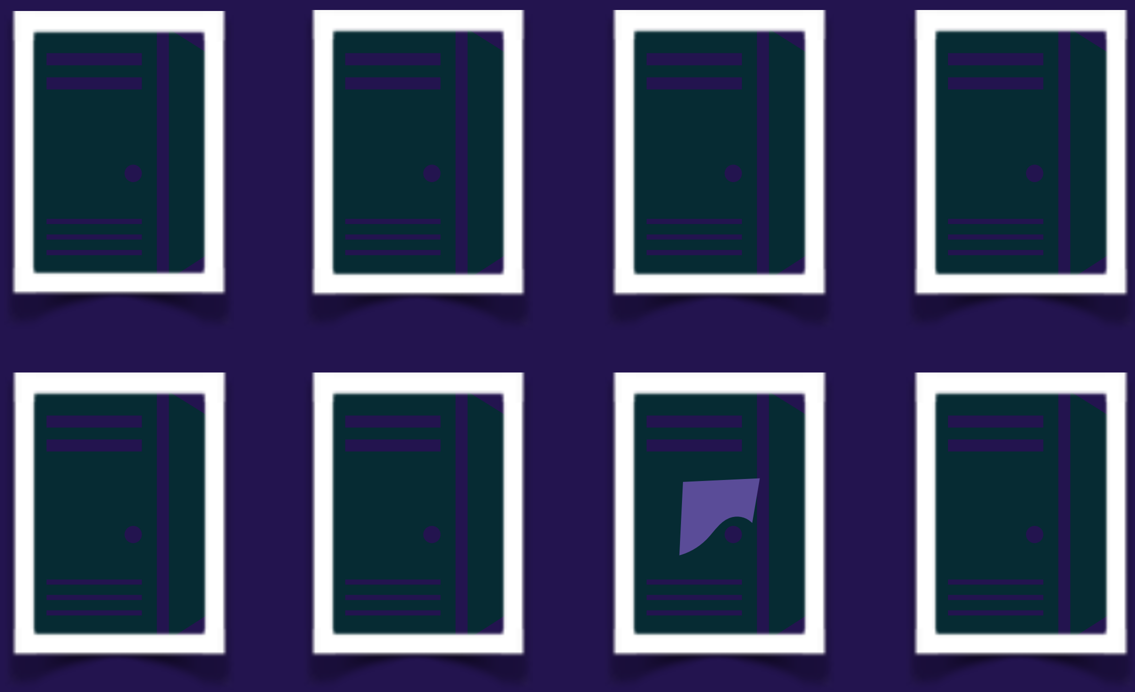
Chunk up the data...



Chunk up the data...



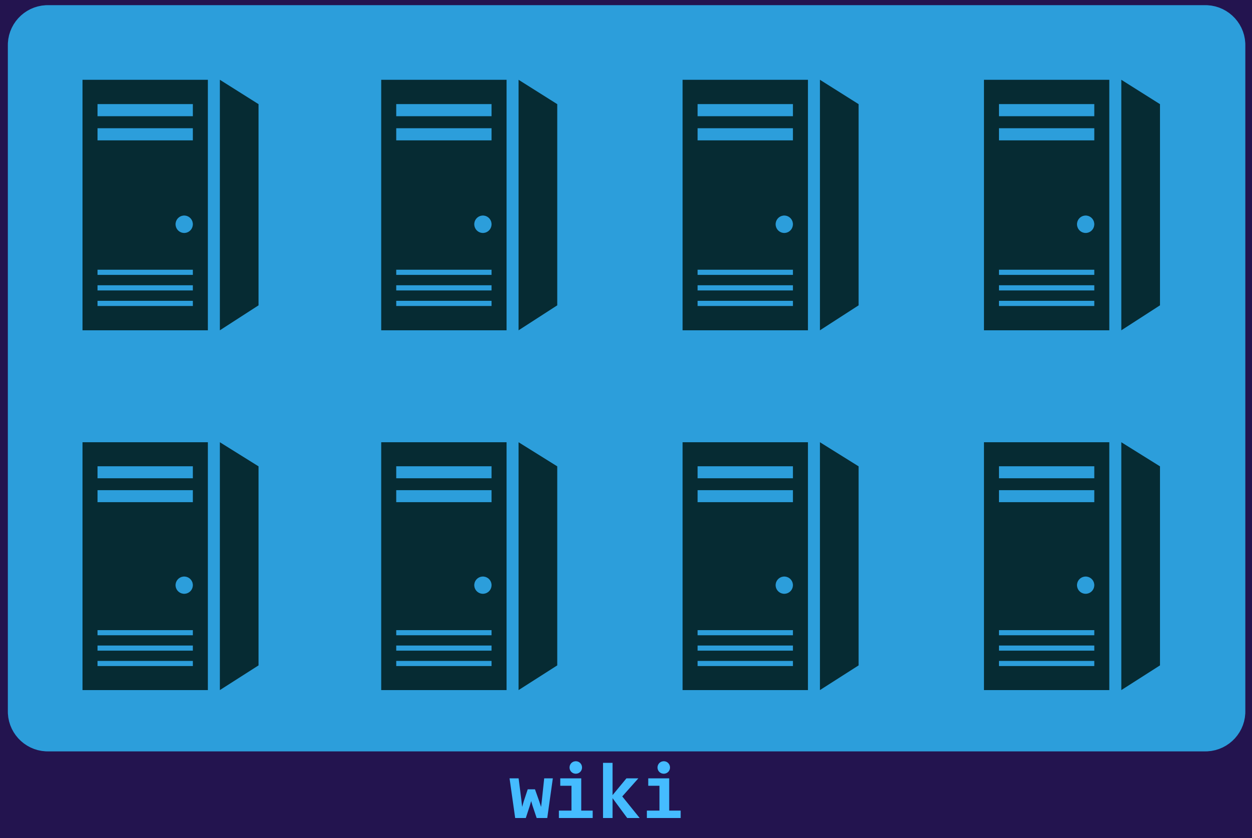
Distribute it over your cluster of machines.



Distribute it over your cluster of machines.

From there, think of your distributed data like a single collection...

```
val wiki: RDD[WikiArticle] = ...
```



From there, think of your distributed data like a single collection...

```
val wiki: RDD[WikiArticle] = ...
```

Example:

Transform the text (not titles) of all wiki articles to lowercase.

```
wiki.map { article =>
  article.text.toLowerCase
}
```

RECAP:

Some of the issues with sending closures over the network:

(in the context of Java/Scala)

RECAP:

Some of the issues with sending closures over the network:

(in the context of Java/Scala)

Transitive object graphs.

RECAP:

Some of the issues with sending closures over the network:

(in the context of Java/Scala)

Transitive object graphs.

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

RECAP:

Some of the issues with sending closures over the network:

(in the context of Java/Scala)

Transitive object graphs.

1. transitive references that inadvertently hold on to excessively large object graphs creating memory leaks
2. Capturing references to mutable objects, leading to race conditions in a concurrent setting.

RECAP:

Some of the issues with sending closures over the network:

(in the context of Java/Scala)

Transitive object graphs.

1. transitive references that inadvertently hold on to excessively large object graphs creating memory leaks
2. Capturing references to mutable objects, leading to race conditions in a concurrent setting.
3. Unknowingly accessing object member that are not constant such as methods, which in a distributed setting can have logically different meanings on different machines.

Going back to this example....

```
val wiki: RDD[WikiArticle] = ...
```

Going back to this example....

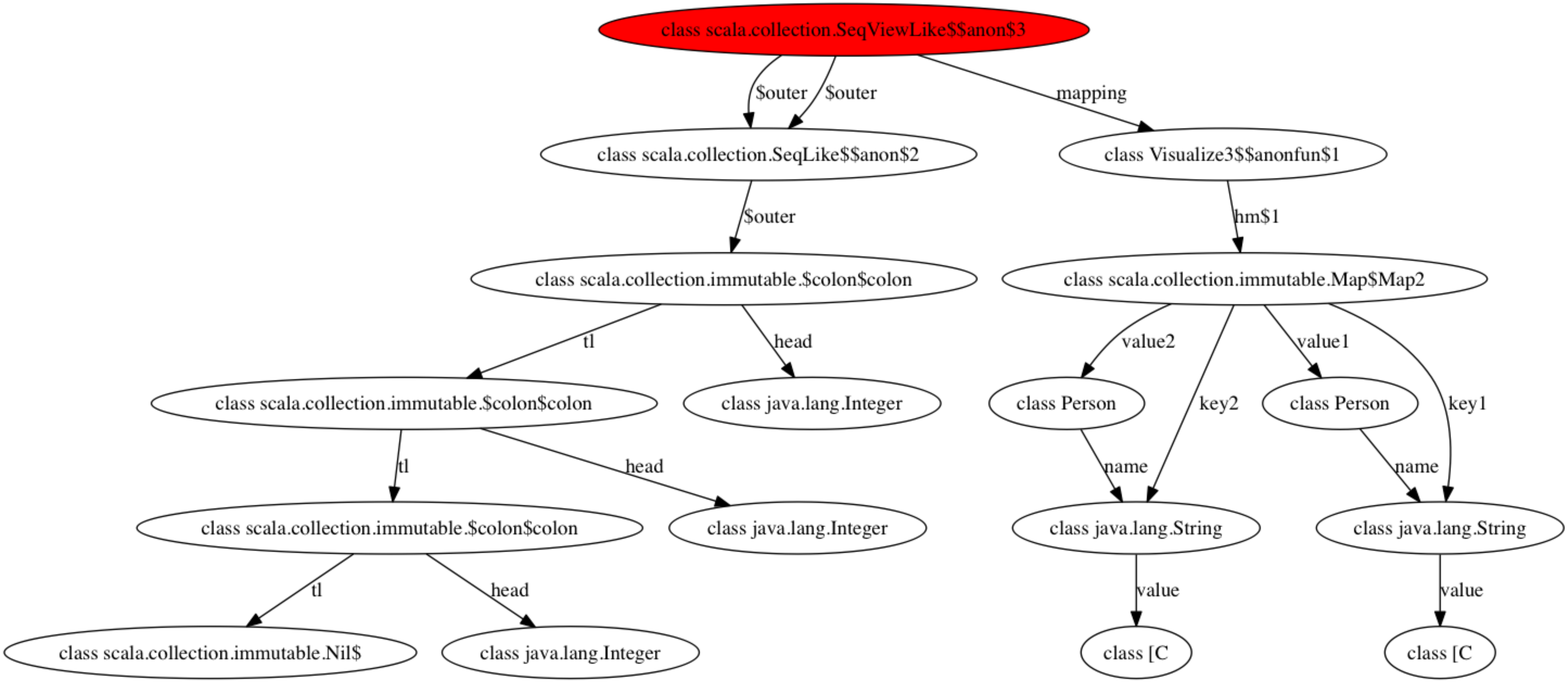
```scala
val wiki: RDD[WikiArticle] = ...
```

wiki

Example:

Transform the text (not titles) of all wiki articles to lowercase.

```scala
wiki.map { article =>
  article.text.toLowerCase
}
```


class scala.collection.SeqViewLike$$anon$3
$outer
$outer
mapping
class scala.collection.SeqLike$$anon$2
class Visualize3$$anonfun$1
$outer
hm$1
class scala.collection.immutable.$colon$colon
class scala.collection.immutable.Map$Map2
tl
head
value2
key2
value1
key1
class scala.collection.immutable.$colon$colon
class java.lang.Integer
class Person
class Person
tl
head
name
name
class scala.collection.immutable.$colon$colon
class java.lang.Integer
class java.lang.String
class java.lang.String
tl
head
value
value
class scala.collection.immutable.Nil$
class java.lang.Integer
class [C
class [C

RECAP:

Some of the issues with sending closures over the network:

(in the context of Java/Scala)

Serializability.

1. Accidental capture of non-serializable variables.
2. Compiler-specific translation schemes that create implicit references to objects that are not serializable

RECAP:

Some of the issues with sending closures over the network:

(in the context of FP)

```
sendFunc :: SendPort (Int → Int) → Int → ProcessM ()
sendFunc p x = sendChan p (\y → x + y + 1)
```

Serializing arbitrary lambdas not obvious even in FP languages.

How do we look up a pickler for x?

Spark example

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

Spark example

```
class MyCoolRddApp {
  val log = new Log( … )
  def shift(p: Int): Int = …
   …
  def work(rdd: RDD[Int]) {
    rdd.map(x ⇒ x + shift(x))
       .reduce( … )
  }
}
```

Fails with an exception at runtime!

Spark example

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

Fails with an exception at runtime!

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

Akka example

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

Akka example

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

No exception. But sometimes logically incorrect.

Akka example

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

No exception. But sometimes logically incorrect.

Akka actor spawns a future to concurrently process incoming reqs

Akka example

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

No exception. But sometimes logically incorrect.

Akka actor spawns a future to concurrently process incoming reqs

Not a stable value! It's a method call!

Akka example

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

No exception. But sometimes logically incorrect.

Result: Response is sometimes sent where it's not expected.

Akka actor spawns a future to concurrently process incoming reqs

Not a stable value! It's a method call!

Our version of safe functions:
Spores

Spores

What are they?

- A closure-like abstraction
- A type system

Spores

What are they?

- A closure-like abstraction
- A type system

Goal:

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

Spores

This is achieved by:

(a) enforcing a specific **syntactic shape** which dictates how the environment of a spore is declared.

(b) providing **additional type checking** to ensure that types being captured have certain properties.

Spores

This is achieved by:

(a) enforcing a specific **syntactic shape** which dictates how th… …nt of a…

(b) p… …**g** to e… …ave c…

Crucially,
spores encode extra type information corresponding to the captured environment in their type.

Spores

Basic usage (long-form):

```
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

Spores

Basic usage (long-form):

```
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

Spores

Basic usage (long-form):

```
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

Spores

Basic usage (long-form):

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

The body of a spore

#1 a sequence of loca
(the "*spore header*

#2 a closure

Syntactic rule: free variables of the closure body must be either:
(a) parameters of the closure or
(b) defined in the spore header

Spores and Closures

Evaluation semantics

The semantics of a spore is equivalent to the closure that is the result of removing the spore marker

Spores and Closures

Evaluation semantics

The s
closu e
mark

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

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

Spores and Closures

Evaluation semantics

The semantics of a spore is equivalent to the closure that is the result of removing the spore marker

Coercions: from closure to spore

Function literals can be implicitly turned into a spore if the function literal satisfies the spore rules

A spore guarantees...
(vs a closure)

1. All captured variables are declared in the spore header
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

That gets 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.

That gets you... (graphically)

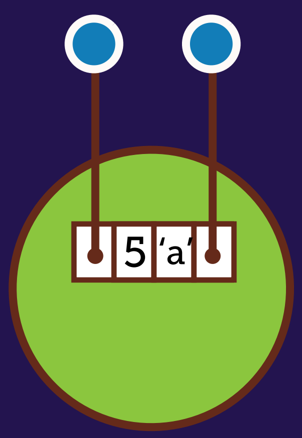
1
Right after creation

2
During execution

Spores

closures

That gets you... (graphically)



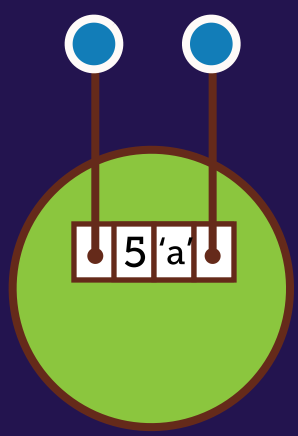
1
Right after
creation

2
During
execution

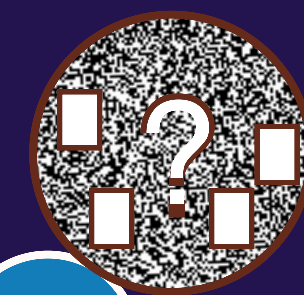
Spores

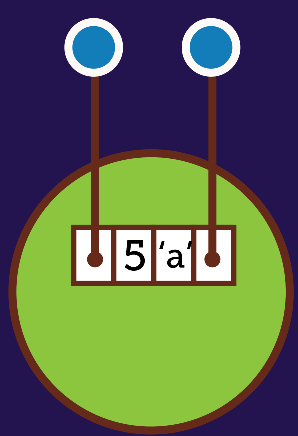
closures

That gets you... (graphically)

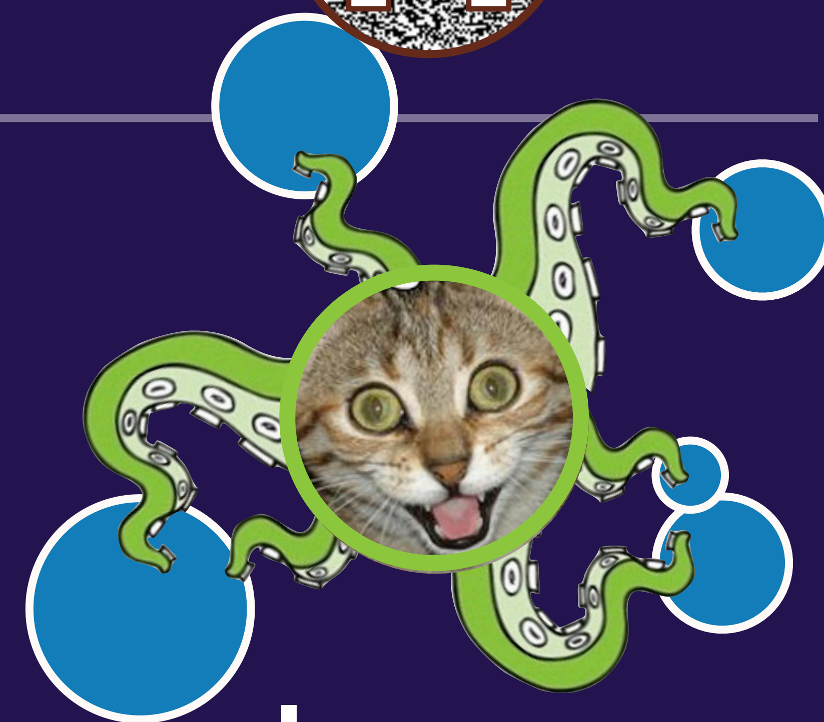


1
Right after creation





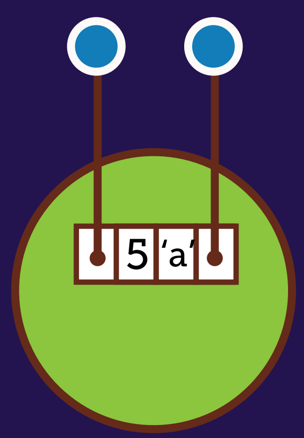
2
During execution



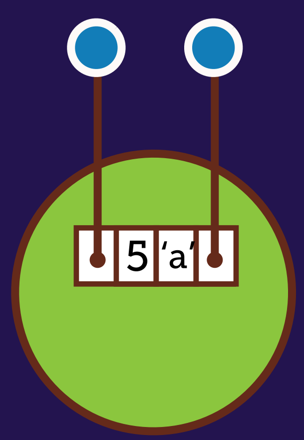
Spores

closures

That gets you... (graphically)



1
Right after creation



2
During execution

Spores **closures**

Formalization

Central idea:
Spore types are refinements of function types.

$T \Rightarrow T$	function type
$\mathcal{S} ::= T \Rightarrow T \; \{ \; \mathsf{type} \; \mathcal{C} = \overline{T} \; ; \; \overline{pn} \; \}$	spore type
$\quad \mid T \Rightarrow T \; \{ \; \mathsf{type} \; \mathcal{C} \; ; \; \overline{pn} \; \}$	abstract spore type

Spore types include more information than function types:

Formalization

Central idea:

Spore types are refinements of function types.

$T \Rightarrow T$	function type
$\mathcal{S} ::= T \Rightarrow T \; \{ \; \mathsf{type} \; \mathcal{C} = \overline{T} \; ; \; \overline{pn} \; \}$	spore type
$\quad \mid T \Rightarrow T \; \{ \; \mathsf{type} \; \mathcal{C} \; ; \; \overline{pn} \; \}$	abstract spore type

Spore types include more information than function types:

captured types

Formalization

Central idea:

Spore types are refinements of function types.

$T \Rightarrow T$	function type
$\mathcal{S} ::= T \Rightarrow T \ \{ \ \mathsf{type} \ \mathcal{C} = \overline{T} \ ; \ \overline{pn} \ \}$	spore type
$\quad \mid T \Rightarrow T \ \{ \ \mathsf{type} \ \mathcal{C} \ ; \ \overline{pn} \ \}$	abstract spore type

Spore types include more information than function types:

captured types and **properties**

Formalization

Central idea:
Spore types are r

Properties: a property could express: “each captured type must have a Pickler type class instance.”

$$T \Rightarrow T$$

$$\mathcal{S} ::= T \Rightarrow T \; \{ \; \mathsf{type} \; \mathcal{C} = \overline{T} \; ; \; \overline{pn} \; \} \quad \text{spore type}$$
$$\mid T \Rightarrow T \; \{ \; \mathsf{type} \; \mathcal{C} \; ; \; \overline{pn} \; \} \quad \text{abstract spore type}$$

Spore types include more information than function types:

captured types and **properties**

Spores with Constraints

Idea:

- Keep track of types of captured variables
- Allow restricting captured types

Spores with Constraints

Idea:

- Keep track of types of captured variables
- Allow restricting captured types

Example:

An API may prevent argument spores from capturing variables of type Socket:

Spores with Constraints

Idea:

- Keep track of types of captured variables
- Allow restricting captured types

Example:

An API may prevent argument spores from capturing variables of type Socket:

```
type SafeSpore[-T, +R] = Spore[T, R] {
  type Excluded <: No[Socket]
}

def sendOverWire(s: SafeSpore[Int, Int]): Unit = ...
```

Formalization

To express excluded types, we use an additional type member

$T \Rightarrow T$	function type

$$\mathcal{S} ::= T \Rightarrow T \, \{ \, \texttt{type} \, \mathcal{C} = \overline{T} \, ; \, \texttt{type} \, \mathcal{E} = \overline{T} \, ; \, \overline{pn} \, \} \quad \text{spore type}$$
$$\mid T \Rightarrow T \, \{ \, \texttt{type} \, \mathcal{C} \, ; \, \texttt{type} \, \mathcal{E} = \overline{T} \, ; \, \overline{pn} \, \} \quad \text{abstract spore type}$$

Formalization

To express excluded types, we use an additional type member

$T \Rightarrow T$	function type
$\mathcal{S} ::= T \Rightarrow T \{ \mathsf{type}\ \mathcal{C} = \overline{T} ; \mathsf{type}\ \mathcal{E} = \overline{T} ; \overline{pn} \}$	spore type
$\mid T \Rightarrow T \{ \mathsf{type}\ \mathcal{C} ; \mathsf{type}\ \mathcal{E} = \overline{T} ; \overline{pn} \}$	abstract spore type

Formalization

SPORE COMPOSITION PRESERVES TYPE CONSTRAINTS

Sound composition of constraints: avoid calculating constraints that are not guaranteed to hold at runtime

…In the paper:

Soundness proof based on a small-step operational semantics and progress+preservation.

Correspondence to the Scala implementation

Using Spores in APIs

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)
```

How are they serializable? They’re not. Yet.

Properties

IDEA:

allow expressing a **type-based property** that must be satisfied by all captured variables upon creation of a spore.

EXAMPLE: ENSURE THE FOLLOWING SPORE IS SERIALIZABLE.

```
case class Person(name: String, age: Int)

val fun = spore {
  val p: Person = ...
  val luckyNum: Int = ...
  () => s"${p.name}'s lucky number is: $luckyNum"
}
```

Properties

IDEA:
allow expressing a **type-based property** that must be satisfied by all captured variables upon

To serialize a spore, it's necessary that **for all captured variables of type T, there is an implicit pickler of type Pickler[T] in scope.**

```
case class Person(name: String, age: Int)

val fun = spore {
  val p: Person = ...
  val luckyNum: Int = ...
  () => s"${p.name}'s lucky number is: $luckyNum"
}
```

Properties

IDEA:
allow expressing a **type-based property** that
must be satisfied by all captured variables upon

To serialize a spore, it's necessary that **for all captured variables of type T, there is an implicit pickler of type Pickler[T] in scope.**

Pickler[T] CAN BE A SUCH A PROPERTY

```
  val luckyNum: Int = ...
  () => s"${p.name}'s lucky number is: $luckyNum"
}
```

Properties

WHAT ARE THEY?

- Properties are defined using a type Property[T]
- Use a property by **importing** it.
- When importing a property of type Property[Prop] all spores in scope are guaranteed to satisfy the property or not compile

(= all captured types have property Prop)

Properties

WHAT ARE THEY?

- Properties are defined using a type Property[T]
- Use a property by **importing** it.
- When importing a property of type Property[Prop] all

so, in the case of pickling, just:

```
import picklable
```

(has to have type Property[Pickler])

and the framework ensures that a Pickler[T] exists for every captured type T.

Properties

Not limited to pickling.

CAN HAVE ARBITRARY PROPERTIES.

…plugs in nicely with other pluggable type systems.

EXAMPLE: DEEP IMMUTABILITY

Integrate with a deep immutability checker like OIGJ (Zibin et al. 2010)

Properties

Not limited to pickling.

IDEA:

Automatically generate type class instances for all types that satisfy a transitive predicate, using macros.

```
implicit def isImmutable[T: TypeTag]: Immutable[T]
```

which returns a type class instance for all types of the shape C[O, Immut] that's deeply immutable (analyzing the TypeTag).

Properties

Not limited to pickling.

IDEA:

To enforce transitive immutability for a spore, it's then sufficient to define an implicit of type Property[Immutable].

```
implicit def isImmutable[T: TypeTag]: Immutable[T]
```

which returns a type class instance for all types of the shape C[O, Immut] that's deeply immutable (analyzing the TypeTag).

Mini Empirical Study #1

How much effort is required to convert existing programs that crucially rely on closures to spores?

Program	LOC	#closures	#converted	LOC changed	#captured vars	
funsets	99	8	8	7	9	MOOC
forcomp	201	6	4	4	0	
mandelbrot	325	1	1	9	6	Parallel Collections
barneshut	722	7	7	8	1	
spark pagerank	64	5	5	8	0	Spark
spark kmeans	92	5	4	9	2	
Total	1503	32	29	45	18	

For each closure, we had to change 1.4LOC on average, or only 45/1503 LOC

Mini Empirical Study #2

How widespread are patterns that can be statically enforced by spores?

Project	average LOC per closure	average # of captured vars	% closures that don't capture
sameeragarwal/blinkdb ★268 👥33 LOC 22,022	1.39	1	93.5%
freeman-lab/thunder ★89 👥2 LOC 2,813	1.03	1.30	23.3%
bigdatagenomics/adam ★86 👥16 LOC 19,055	1.90	1.44	80.2%
ooyala/spark-jobserver ★79 👥6 LOC 5,578	1.60	1	80.0%
Sotera/correlation-approximation ★12 👥2 LOC 775	4.55	1.25	63.6%
aecc/stream-tree-learning ★1 👥2 LOC 1,199	5.73	2	54.5%
lagerspetz/TimeSeriesSpark ★5 👥1 LOC 14,882	2.85	1.77	75.0%
Total LOC 66,324	2.25	1.39	67.2%

Mini Empirical Study #2

How widespread are patterns that can be statically enforced by spores?

67.2% of all closures can be automatically converted. The remaining 32.8% capture only 1.39 variables on average.

Thus, unchecked patterns are widespread in real applications, and require only little overhead to enable spore guarantees.

sotera/correlation-approximation ★12 👥2 LOC 775	4.55	1.25	63.6%
aecc/stream-tree-learning ★1 👥2 LOC 1,199	5.73	2	54.5%
lagerspetz/TimeSeriesSpark ★5 👥1 LOC 14,882	2.85	1.77	75.0%
Total LOC 66,324	2.25	1.39	67.2%

Implicit, what?

Question:

How widespread are implicits in the Scala ecosystem?

Question:

How widespread are implicits in the Scala ecosystem?

Different kinds:

- Implicit parameters/val **(configuration)**
- Typeclasses
- Coercions

Implicits are popular.

...more than we thought.

Implicits are popular.

...more than we thought.

We analyzed the most popular Scala projects on GitHub

120 Scala projects

Average lines of code per project: **31,135**

Lines of code analyzed: </> **3.7 million**

Average # of stars per project: **1,977** ★

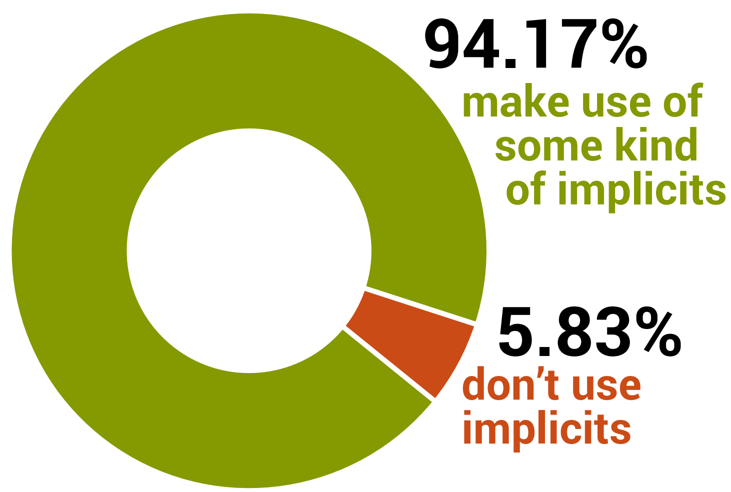
Project	Stars				
apache/spark	13365	laurilehmijoki/s3_website	1800	filodb/FiloDB	876
apache/incubator-predictionio	10267	lampepfl/dotty	1759	pathikrit/better-files	874
playframework/playframework	9495	spark-jobserver/spark-jobserver	1653	japgolly/scalajs-react	874
scala/scala	8478	apache/incubator-openwhisk	1641	tpolecat/doobie	868
shadowsocks/shadowsocks-android	7969	twitter/finatra	1605	kamon-io/Kamon	859
akka/akka	7305	twitter/algebird	1527	vkostyukov/scalacaster	850
gitbucket/gitbucket	6424	mesos/spark	1462	sbt/sbt-native-packager	844
twitter/finagle	5799	GravityLabs/goose	1427	functional-streams-for-scala/fs2	839
lhartikk/ArnoldC	4992	lagom/lagom	1414	lihaoyi/Metascala	829
airbnb/aerosolve	3961	Netflix/atlas	1406	scala/pickling	816
yahoo/kafka-manager	3816	lihaoyi/Ammonite	1319	sryza/aas	814
mesos/chronos	3750	PkmX/lcamera	1284	eligosource/eventsourced	811
twitter/snowflake	3513	twitter/iago	1243	monix/monix	809
snowplow/snowplow	3432	rickynils/scalacheck	1231	akka/reactive-kafka	800
mesosphere/marathon	3333	datastax/spark-cassandra-connector	1222	sryza/spark-timeseries	799
ornicar/lila	3250	jaliss/securesocial	1211	scala/async	796
rtyley/bfg-repo-cleaner	3235	guardian/grid	1197	lihaoyi/scala.rx	791
fpinscala/fpinscala	3189	ensime/ensime-server	1193	julien-truffaut/Monocle	789
scalaz/scalaz	3139	non/spire	1192	http4s/http4s	787
sbt/sbt	3115	lw-lin/CoolplaySpark	1186	twitter/ostrich	782
twitter-archive/flockdb	3112	foundweekends/giter8	1158	sangria-graphql/sangria	778
gatling/gatling	3049	lift/framework	1090	jrudolph/sbt-dependency-graph	768
scala-js/scala-js	3012	mpeltonen/sbt-idea	1085	scalikejdbc/scalikejdbc	765
scala-native/scala-native	2885	finagle/finch	1065	databricks/spark-csv	764
twitter/diffy	2858	scala-exercises/scala-exercises	1051	twitter/twitter-server	734
twitter/scalding	2839	quantifind/KafkaOffsetMonitor	1048	ReactiveMongo/ReactiveMongo	718
twitter-archive/kestrel	2780	mauricio/postgresql-async	1041	adamw/macwire	711
spray/spray	2523	killrweather/killrweather	1018	playframework/play-slick	706
linkerd/linkerd	2315	ThoughtWorksInc/Binding.scala	994	jdegoes/blueeyes	702
scalatra/scalatra	2188	tumblr/colossus	989	nscala-time/nscala-time	696

Table 1. Top 120 open source Scala projects on GitHub, by star count.

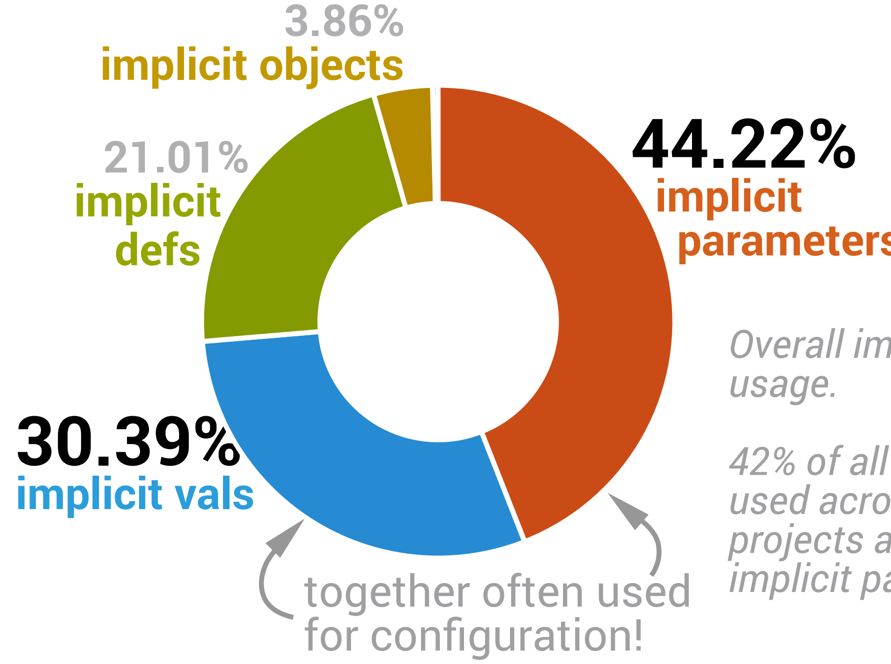
Implicits are popular.
...more than we thought.

How many projects make use of implicits?

of the 120 most popular Scala projects on GitHub?



What sorts of implicits do projects use?



Overall implicits usage.

42% of all implicits used across all projects analyzed are implicit parameters.

How much of each code base uses implicits?

24% *Average percentage of project source files using implicits.*

Questions?