Function-Passing Style
Typed, Distributed Functional Programming

Heather Miller
@heathercmiller
EPFL

Philipp Haller
@philippkhaller
KTH Royal Institute of Technology

Scala Days 2015, San Francisco, CA, USA. March 17, 2015
...been working on
Language support for distributed system builders.

Serialization

That happens mostly at compile time, so it’s performant.

- Type classes to allow users to serialize to different formats (binary, JSON, etc)
...been working on
Language support for distributed system builders.

2 Spores
- Portable (serializable) closures.
- Type constraints to restrict what they capture
In this talk...

A programming model.

- Builds on the basis of serializable functions
- to provide a substrate that distributed systems can be built upon
In this talk...

A programming model.

The result...

the model greatly simplifies the design and implementation of mechanisms for:

- **Fault-tolerance**
- **In-memory caching**
- **Debugging** (i.e., pushing types into more layers of the stack)

IN A CLEAN & FUNCTIONAL WAY. (STATELESS!)
Note:
Currently a research project. Thus, all aspects of it are under development + publication in the works.

(Thanks,.databricks)
FUNDAMENTAL IDEA:

Inversion of the actor model.
Can be thought of as a dual to actors.

A DUAL WHICH NICELY COMPLEMENTS ACTORS!
Um, HOW?
Actors...

- Encapsulate state and behavior.
- Are stationary.

Actors exchange data/commands through asynchronous messaging.
Function-passing...

- Stateless. Built on persistent data structures.
- Keep the data stationary.

Functions are exchanged through asynchronous messaging.

Of note:

This is a model for programming with data and not a new model of concurrent processes like actors.

Instead, we provide a new means of working with distributed data in a functional way.
1 Introduction

Much of the current work in distributed, object-oriented systems is based on the assumption that objects form a single ontological class. This class consists of all entities that can be fully described by the specification of the set of interfaces supported by the object and the semantics of the operations in those interfaces. The class includes objects that share a single address space, objects that are in separate address spaces on the same machine, and objects that are in separate address spaces on different machines (with, perhaps, different architectures). On the view that all objects are essentially the same kind of entity, these differences in relative location are merely an aspect of the implementation of the object. Indeed, the location of an object may change over time, as an object migrates from one machine to another or the implementation of the object changes.

It is the thesis of this note that this unified view of objects is mistaken. There are fundamental differences between the interactions of distributed objects and the interactions of non-distributed objects. Further, work in distributed object-oriented systems that is based on a model that ignores or denies these differences is doomed to failure, and could easily lead to an industry-wide rejection of the notion of distributed object-based systems.

1.1 Terminology

In what follows, we will talk about local and distributed computing. By local computing (local object invocation, etc.), we mean programs that are confined to a single address space. In contrast, we will use the term distributed computing (remote object invocation, etc.) to refer to programs that make calls to other address spaces, possibly on another machine. In the case of distributed computing, nothing is known about the recipient of the call (other than that it supports a particular interface). For example, the client of such a distributed object does not know the hardware architecture on which the recipient of the call is run.
A Note on Distributed Computing

Differences in latency, memory access, partial failure, and concurrency make merging of the computational models of local and distributed computing both unwise to attempt and unable to succeed.

A better approach is to accept that there are irreconcilable differences between local and distributed computing, and to be conscious of those differences at all stages of the design and implementation of distributed applications. Rather than trying to merge local and remote objects, engineers need to be constantly reminded of the differences between the two, and know when it is appropriate to use each kind of object.
So, WHAT DOES IT LOOK LIKE?
**Function-Passing Model**

Two concepts:

1. Stationary, immutable data.
2. Portable functions – move the functionality to the data.
Two concepts:

1. Stationary, immutable data.
2. Portable functions – move the functionality to the data.
the Function-Passing Model (illustrated)
Silos don’t have to all live on the same node.
**Silos.**

**WHAT ARE THEY?**

- **SiloRef[T]**
  - `def apply`
  - `def send`
  - `def flatMap`

The handle to a Silo.

*(The workhorse.)*

```python
def apply(s1: Spore, s2: Spore): SiloRef[T]
```

**Takes two spores:**
- framework logic (combinator), e.g. `map`
- user/application-provided argument function

**Defers application of fn to silo, returns SiloRef with info for later materialization of silo.**

**LAZY!**
**Function-Passing Model**

**Silo**

**WHAT ARE THEY?**

- **SiloRef[T]**
  - def apply
  - def send
  - def flatMap

The handle to a Silo.

- def apply(s1: Spore, s2: Spore): SiloRef[T]
- def send(): Future[T]

Sends info for function application and silo materialization to remote node

**EAGER!**

Asynchronous/nonblocking data transfer to local machine (via Future)
Silos. WHAT ARE THEY?

The handle to a Silo.

```python
def apply(s1: Spore, s2: Spore): SiloRef[T]
def send(): Future[T]
def flatMap[S](spore: Spore[T, SiloRef[S]]): SiloRef[S]
```

Sometimes you need to move data.

- send spore to silo of the receiver SiloRef (this)
- result of applying spore is a SiloRef whose contents should be made available in a new silo on the same node as this SiloRef
Sometimes you need to move data.

- send spore to silo of the receiver SiloRef (this)
- result of applying spore is a SiloRef whose contents should be made available in a new silo on the same node as this SiloRef
- SiloRef of the new silo is returned

Note: needs to transfer data if the SiloRef that's the result of the spore is not on the same node already
the Function-Passing Model (illustrated)
the Function-Passing Model (illustrated)
Two concepts:

1. Stationary, immutable data.

2. Portable functions – move the functionality to the data.
What do spores look like?

**Basic usage:**

```scala
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

http://docs.scala-lang.org/sips/pending/spores.html
A Spore Guarantees...

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

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

Heather Miller, Philipp Haller¹, and Martin Odersky

EPFL and Typesafe, Inc.¹
{heather.miller, martin.odersky}@epfl.ch and philipp.haller@typesafe.com¹

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, despite this established viewpoint of FP as an enabler, reliably distributing function closures over a network, or using them in concurrent environments nonetheless remains a challenge across FP and OO languages. This paper takes a step towards more principled distributed and concurrent programming by introducing a new closure-like abstraction and type system, called spores, that can guarantee closures to be serializable, thread-safe, or even have custom user-defined properties. Crucially, our system is based on the principle of encoding type information corresponding to captured variables in the type of a spore. We prove our type system supporting these guarantees through a case analysis of real-world distributed and concurrent frameworks that this safe foundation for closures facilitates.

Keywords: closures, concurrency, distributed programming, functional programming, type systems

http://docs.scala-lang.org/sips/pending/spores.html
Ok,
EXAMPLE,
PLEASE.
the Function-Passing Model (illustrated)

EXAMPLE:
Distributed List with operations map and reduce.
(This is what would be happening under the hood)

SiloRef[List[Int]] .apply
SiloRef[List[Int]] .apply
SiloRef[List[Int]] .apply
SiloRef[Int] .send()

map f
map (_*2)
reduce (_+_)
**EXAMPLE:**

Distributed List with operations **map** and **reduce**.
(This is what would be happening under the hood)
Ok, how does this help with fault tolerance?
Data in silos easily reconstructed:

- Silos and SiloRefs relate to each other by means of a persistent data structure.
- The persistent data structure is based on the chain of operations to derive the data of each silo.
- Thus, traversing the silo data structures yields the complete lineage of a silo.
- Since the lineage is composed of spores, it's serialized. This means it can be persisted or transferred to other machine.
Data (Silos) managed using persistent data structure.

All operations, including operations provided by system builders, are spores – so, serializable!

Taken together:
A lot simpler to build mechanisms for fault tolerance!
Function Passing: A Model for Typed, Distributed Functional Programming

Heather Miller
EPFL
heather.miller@epfl.ch

Philipp Haller
KTH Royal Institute of Technology
phaller@kth.se

Abstract
The most successful systems for “big data” processing have all adopted functional APIs. But the underpinnings of these systems are often built atop imperative and weakly-typed stacks, which complicates the design and implementation of distributed systems. We present a new programming model, which we call function passing designed to overcome many of these issues by providing a more principled substrate on which to build data-centric distributed systems. A key idea is to pass safe, well-typed serializable functions to immutably distributed data. The F-P model itself can be thought of as a distributed persistent functional data structure, which stores in its nodes transformations to data, rather than the data itself. Thus, the model simplifies failure recovery by design, data is recovered by replaying function applications, and immutable data logged from stable storage. Lazy evaluation is also central to our model; by carefully incorporating laziness into our design (only at the point of initiating network communication), our model remains easy to reason about while remaining efficient in time and memory. We formalize our programming model in the form of a small-step operational semantics which also specifies a precise implementation of functional finite recovery and provides an open-source implementation of our model in the Scala programming language.

http://infoscience.epfl.ch/record/205822
Looking for feedback, demanding use cases, contributors.

Get involved! function-passing@googlegroups.com

Thank you!
Can’t I just...
Send spores within messages between actors/processes?

- Granted, that’s already quite powerful. :-)
- However, spores + silos additionally provides a lot of benefits that actors + spores alone do not provide out-of-the-box:

Benefits:
- **deferred evaluation** (laziness) enables optimizations to reduce intermediate results.
- **statelessness + lineages** simplifies the implementation of mechanisms for fault tolerance for certain applications (think dist. collections)