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 innards of these systems are often built atop imperative and weakly-typed stacks, which complicates the design and implementation of distributed system essentials like fault-tolerance. We present a new programming model 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 immutable 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 distributed data itself. Thus, the model simplifies failure recovery by design--data is recovered by replaying function applications atop immutable data loaded 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 includes a precise specification of the semantics of functional fault recovery, and we provide an open-source implementation of our model in and for the Scala programming language.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications – Concurrent, distributed, and parallel languages; D.3.3 [Programming Languages]: Language Constructs and Features – Concurrent programming structures; Procedures, functions, and subroutines

Keywords Functional programming, distributed programming, functions, closures, serialization, concurrency, types, Scala

1. Introduction
It is difficult to deny that data-centric programming is growing in importance. At the same time, it is no secret that the most successful systems for programming with "big data" have all adopted ideas from functional programming; i.e., programming with first-class functions and higher-order functions. These functional ideas are often touted to be the key to the success of these frameworks. It is not hard to imagine why--a functional, declarative interface to data distributed over tens of thousands of nodes provides a more natural way for end-users and data scientists to reason about data.

While leveraging functional programming concepts, popular implementations of the MapReduce [10] model, such as Hadoop MapReduce [3] for Java, have been developed without making use of functional language features such as closures. In contrast, a new generation of programming systems for large-scale data processing, such as Apache Spark [39], Twitter’s Scalding [35], and Scoobi [28] build on functional language features in Scala in order to provide high-level, declarative APIs.

However, these frameworks are built atop of tall stacks of typically imperative and untyped code, losing most of the benefits enjoyed by the users of their high-level APIs. Consequently, concerns central to distributed systems such as concurrency and fault tolerance become more difficult to reason about and realize in practice. Some language features, like closures, are not able to be reliably distributed. Yet other problems manifest themselves in all layers of the stack, surprising users and impacting ease-of-use, complicating maintenance, and losing opportunities for optimization. Some such problems include:

- **Difficulty of Use** These systems’ APIs cannot statically prevent common usage errors resulting from some language features not being designed with distribution in mind, often confronting users with hard-to-debug runtime errors. A common example is unsafe closure serialization [25].
- **Complicated Maintenance** Typically, only high-level user-facing abstractions are statically typed. The absence of static types in lower layers of the system makes maintenance tasks, such as code refactorings, more difficult.
- **Lost Optimization Opportunities** The absence of certain kinds of static type information precludes systems-centric optimizations. Importantly, type-based static meta-programming enables fast serialization [24], but this is only possible if also lower layers (namely those dealing with object serialization) are statically typed. Several studies [7, 22, 30, 38] report on the high overhead of serialization in widely-used runtime environments such as the JVM. This overhead is so important in practice that popular systems, like Spark [39] and Akka [36], leverage alternative serialization frameworks such as Protocol Buffers [14], Apache Avro [2], or Kryo [34].

We present a new programming model we call function passing (F-P) designed to overcome most of these issues by providing a more principled substrate on which to build data-centric distributed systems. It builds upon two previous veins of work--an approach

* Work done while the author was affiliated with Databricks.
for generating type-safe and performant pickler combinators [24], and spores [25], closures that are guaranteed to be serializable. Our model attempts to fit the paradigm of data-centric programming more naturally by extending monadic programming to the network. Our model can be thought of as somewhat of a dual to the actor model; rather than keeping functionality stationary and sending data, in our model, we keep data stationary and send functionality to the data. This results in well-typed communication by design, a common pain point for builders of distributed systems in Scala.

Our model brings together immutable, persistent data structures, monadic higher-order functions, strong static typing, and lazy evaluation–pillars of functional programming–to provide a more type-safe, and easy to reason about foundation for data-centric distributed systems. Interestingly, we found that laziness was an enabler in our model, without complicating the ability to reason about programs. Without optimizations based on laziness, we found this model would be impractically inefficient in memory and time.

One important contribution of our model is a precise specification of the semantics of functional fault recovery. The fault-recovery mechanisms of widespread systems such as Apache Spark, MapReduce [10] and Dryad [21] are based on the concept of a lineage [6, 9]. Essentially, the lineage of a data set combines (a) an initial data set available on stable storage and (b) a sequence of transformations applied to initial and subsequent data sets. Maintaining such lineages enables fault recovery through recomputation. Practical implementations of lineage-based fault recovery suffer from complex code bases, typically eschewing strong static typing. This paper presents a principled approach to lineage-based fault recovery in a purely functional setting–to our knowledge a novelty in the PL literature.

Our programming model is also designed to enable adoption in real-world languages and systems. On the one hand, our model can be thought of as a generalization of the MapReduce/Spark computation model, which has been shown to be widely portable. On the other hand, the core primitives of the programming model can be implemented in any language that enables closures to be serialized.

This paper makes the following contributions:

• A new data-centric programming model for functional processing of distributed data which makes important concerns like fault tolerance simple by design. The main computational principle is based on the idea of sending safe, guaranteed serializable functions to stationary data. Using standard monadic operations our model enables creating immutable DAgS of computations, supporting decentralized distributed computations. Lazy evaluation enables important optimizations while keeping programs simple to reason about.

• A formalization of our programming model based on a small-step operational semantics. To our knowledge it is the first formal account of fault recovery based on lineage in a purely functional setting. Inspired by widespread systems like Spark, our formalization closely models real-world fault recovery mechanisms. The presented semantics is clearly stratified into a deterministic layer and a concurrent/distributed layer. Importantly, reasoning techniques for sequential programs are not invalidated by the distributed layer.

• A distributed implementation of the programming model in and for Scala. We present experiments that show some of the benefits of the proposed design, and we report on a validation of spores in the context of distributed programming.

Our approach is to describe our model from a high-level, elaborating upon key benefits and trade-offs, and then to zoom in and make each component part of our model more precise. We describe the basic model this way in Section 2. We go on to show in Section 3 how essential higher-order operations on distributed frameworks like Spark can be implemented in terms of the primitives presented in Section 2. We present a formalization of our programming model in Section 4, and an overview of its prototypical implementation in Section 5. Finally, we discuss related work in Section 6, and conclude in Section 7.

2. Overview of Model

The best way to quickly visualize the F-P model is to think in terms of a persistent functional data structure with structural sharing. Then, rather than containing pure data, imagine instead that the structure represents a directed acyclic graph (DAG) of transformations on distributed data.

Importantly, since this DAG of computations is a persistent data structure itself, it is safe to exchange (copies of) subgraphs of a DAG between remote nodes. This enables a robust and easy-to-reason-about model of fault tolerance. We call subgraphs of a DAG lineages; lineages enable restoring the data of failed nodes through re-applying the transformations represented by their DAG. This sequence of applications must begin with data available from stable storage.

Central to our model is the careful use of laziness. Computations on distributed data are typically not executed eagerly; instead, applying a function to distributed data just creates an immutable lineage. To obtain the result of a computation, it is necessary to first “kick off” computation, or to force its lineage. Within our programming model, this force operation makes network communication (and thus possibilities for latency) explicit, which is considered to be a strength when designing distributed systems [37]. Deferred evaluation also enables optimizing distributed computations through operation fusion, which avoids the creation of unnecessary intermediate data structures–this is efficient in time as well as space. This kind of optimization is particularly important and effective in distributed systems [8].

For these reasons, we believe that laziness should be viewed as an enabler in the design of distributed systems.

The F-P model consists of three main components:

• Silos: stationary typed data containers.
• SiloRefs: references to local or remote Silos.
• Spores: safe, serializable functions.

Silos A silo is a typed data container. It is stationary in the sense that it does not move between machines – it remains on the machine where it was created. Data stored in a silo is typically loaded from stable storage, such as a distributed file system. A program operating on data stored in a silo can only do so using a reference to the silo, a SiloRef.

SiloRefs Similar to a proxy object, a SiloRef represents, and allows interacting with, both local and remote silos. SiloRefs are immutable, storing identifiers to locate possibly remote silos. SiloRefs are also typed (Silos are of the corresponding to the type of their silo’s data, leading to well-typed network communication. The SiloRef provides three primitive operations/combinators (some are lazy,
some are not): map, flatMap, and send. map lazily applies a user-defined function to data pointed to by the SiloRef, creating in a new silo containing the result of this function. Like map, flatMap lazily applies a user-defined function to data pointed to by the SiloRef. Unlike map, the user-defined function passed to flatMap returns a SiloRef whose contents is transferred to the new silo returned by flatMap. Essentially, flatMap enables accessing the contents of (local or remote) silos from within remote computations. We illustrate these primitives in more detail in Section 2.2.

Spores Spores [25] are safe closures that are guaranteed to be serializable and thus distributable. They are a closure-like abstraction and type system which gives authors of distributed frameworks a principled way of controlling the environment which a closure (provided by client code) can capture. This is achieved by (a) enforcing a specific syntactic shape which dictates how the environment of a spore is declared, and (b) providing additional type-checking to ensure that types being captured have certain properties.

A spore consists of two parts:

• the spore header, composed of a list of value definitions.
• the spore body (sometimes referred to as the “spore closure”), a regular closure.

This shape is illustrated below.

```scala
closure { val y1: S1 = <expr1> 
  val yn: Sn = <exprn>
  (x: T) => { 
    // ...
    }
  }
```

The characteristic property of a spore is that the spore body is only allowed to access its parameter, the values in the spore header, as well as top-level singleton objects (Scala’s form of modules). The spore closure is not allowed to capture variables other than those declared in the spore header (i.e., a spore may not capture variables in the environment). By enforcing this shape, the environment of a spore is always declared explicitly in the spore header, which avoids accidentally capturing problematic references. Moreover, importantly for object-oriented languages like Scala, it’s no longer possible to accidentally capture the this reference.

Spores also come with additional type-checking. Type information corresponding to captured variables is included in the type of a spore. This enables authors of distributed frameworks to customize type-checking of spores to, for example, exclude a certain type from being captured by user-provided spores. Authors of distributed frameworks may kick on this type-checking by simply including information about excluded types (or other type-based properties) in the signature of a method. A concrete example would be to ensure that the map method on Rdd’s in Spark (a distributed collection) accepts only spores which do not capture SparkContext (a non-serializable internal framework class).

For a deeper understanding of spores, see the corresponding publication [25].

2.1 Basic Usage

We begin with a simple visual example to provide a feeling for the basics of the F-P model.²

The only way to interact with distributed data stored in silos is through the use of SiloRefs. A SiloRef can be thought of as an immutable handle to the remote data contained within a corresponding silo. Users interact with this distributed data by applying functions to SiloRefs, which are transmitted over the wire and later applied to the data within the corresponding silo. As is the case for persistent data structures, when a function is applied to a piece of distributed data via a SiloRef, a SiloRef representing a new silo containing the transformed data is returned.

The simplest illustration of the model is shown in Figure 1 (time flows vertically from top to bottom). Here, we start with a SiloRef[T] which points to a piece of remote data contained within a Silo[T]. When the function shown as \( \lambda \) of type \( T \Rightarrow S \) is applied to SiloRef[T] and “forced” (sent over the wire), a new SiloRef of type SiloRef[S] is immediately returned. Note that SiloRef[S] contains a reference to its parent SiloRef, SiloRef[T]. (This is how lineages are constructed.) Meanwhile, the function is asynchronously sent over the wire and is applied to Silo[T], eventually producing a new Silo[S] containing the data transformed by function \( \lambda \). This new SiloRef[S] can be used even before its corresponding silo is materialized (i.e., before the data in Silo[S] is computed) – the F-P framework queues up operations applied to SiloRef[S] and applies them when Silo[S] is fully materialized.

Different sorts of complex DAGs can be asynchronously built up in this way. Though first, to see how this is possible, we need to develop a clearer idea of the primitive operations available on SiloRefs and their semantics. We describe these in the following section.

2.2 Primitives

There are four basic primitive operations on SiloRefs that together can be used to build the higher-order operations common to popular data-centric distributed systems (how to build some of these higher-order operations is described in Section 3). In this section we’ll introduce these primitives in the context of a running example. These primitives include:

• map
• flatMap
• send
• cache

```scala
def map[S](s: SiloRef[T, S]): SiloRef[S]
```

The map method takes a spore that is to be applied to the data in
As mentioned earlier, the execution of computations built using SiloRefs is deferred. The `send` operation forces the lazy computation defined by the given SiloRef. Forcing is explicit in our model, because it requires sending the lineage to the remote node on which the result silo should be created. Given that network communication because it requires sending the lineage to the remote node on which the result silo should be created, we will develop a running example throughout this section. Given a silo containing a list of `Person` records, the following application of `map` defines a (not-yet-materialized) silo containing only the records of adults (graphically shown in Figure 2, part 1):

```
val persons: SiloRef[List[Person]] = ...
val adults = persons.map(spore { p => p.age >= 18 })
```

The `map` method takes a spore that is to be applied to the data in the silo of the given SiloRef. The crucial difference is in the type of the spore argument whose result type is a SiloRef in this case. Semantically, the new silo created by `map` is defined to contain the data of the silo that the user-defined spore returns. The `map` combinator adds expressiveness to our model that is essential to express more interesting computation DAGs. For example, consider the problem of combining the information contained in two different silos (potentially located on different hosts). Suppose the information of a silo containing `Vehicle` records should be enriched with other details only found in the adults silo. In the following, `flatMap` is used to create a silo of `Pair[Person, Vehicle]` where the names of person and vehicle owner match (graphically shown in Figure 2, part 2):

```
val vehicles: SiloRef[List[Vehicle]] = ...
val owners = adults.flatMap(spore { v => v.owner.name == p.name })
```

Note that the spore passed to `flatMap` declares the capturing of the `Vehicle` silo by its so-called "spore header." The spore header spans all variable definitions between the spore marker and the parameter list of the spore’s closure. The spore header defines the variables that the spore’s closure is allowed to access. Essentially, spores limit the free variables of their closure’s body to the closure’s parameters and the variables declared in the spore’s header. Within the spore’s closure, it is necessary to read the data of the `Vehicle` silo in addition to the `Person` records. This requires calling `map` on `localVehicles`. However, `map` returns a SiloRef; thus, invoking `map` on `localVehicles` would be impossible, since there would be no way to get the data out of the silo returned by `localVehicles.map(...)`. With the use of `flatMap`, however, the call to `localVehicles.map(...)` creates the final result silo, whose data is then also contained in the silo returned by `flatMap`.

Although the expressiveness of the `flatMap` combinator subsumes that of the `map` combinator (see Section 2.2.2), keeping `map` as a (lightweight) primitive enables more opportunities for optimizing computation DAGs (e.g., operation fusion [8]).
has a latency several orders of magnitude greater than accessing a word in main memory, providing an explicit send operation is a judicious choice [37].

To enable materialization of remote silos to proceed concurrently, the send operation immediately returns a future [16]. This future is then asynchronously completed with the data of the given silo. Since calling send will materialize a silo and send its data to the current node, send should only be called on silos with reasonably small data (for example, in the implementation of an aggregate operation such as reduce on a distributed collection).

cache def cache(): Future[Unit]
The performance of typical data analytics jobs can be increased dramatically by caching large data sets in memory [39]. To do this, the silo containing the computed data set needs to be materialized. So far, the only way to materialize a silo that we have shown is using the send primitive. However, send additionally transfers the contents of a silo to the requesting node—too much if a large remote data set should merely be cached in memory remotely. Therefore, an additional primitive called cache is provided, which forces the materialization of the given SiloRef, returning a future to be completed as soon as the SiloRef has been materialized.

Given the running example so far, we can add another subgraph branching off of adults, which sorts each Person by age, produces a String greeter, and then “kicks-off” remote computation by calling cache and caching the result in remote memory (graphically shown in Figure 2, part 3 and 4):

val sorted = adults.map(spore { ps => ps.sortWith(p => p.age })
val labels = sorted.map(spore { ps => ps.map(p => "Welcome, " + p.name })
labels.cache()

Assuming we would also cache the owners SiloRef from the previous example, the resulting lineage graph would look as illustrated in Figure ???. Note that vehicles is not a regular parent in the lineage of owners; it is an indirect input used to compute owners by virtue of being captured by the spore used to compute owners.

2.2.1 Creating Silos
Besides a type definition for SiloRef, our framework also provides a companion singleton object (Scala’s form of modules). The singleton object provides factory methods for obtaining SiloRefs referring to silos populated with some initial data.3

```
object SiloRef {
  def fromTextFile(host: Host)(file: File): SiloRef[List[String]] = ...
  def fromFun[T](host: Host)(s: Spore[Unit, T]): SiloRef[T] = ...
  def fromLineage[T](host: Host)(s: SiloRef[T]): SiloRef[T] = ...
}
```

Each of the factory methods has a host parameter that specifies the target host (address/port) on which to create the silo. Note that the fromFun method takes a spore closure as an argument to make sure it can be serialized and sent to host. In each case, the returned SiloRef contains its host as well as a host-unique identifier. The fromLineage method is particularly interesting as it creates a copy of a previously existing silo based on the lineage of a SiloRef’s. Note that only the SiloRef is necessary for this operation to successfully complete; the silo originally hosting s might already have failed.

2.2.2 Expressiveness

Expressing map Leveraging the above-mentioned methods for creating silos, it is possible to express map in terms of flatMap:

```
def map[S,T](s: Spore[T, S]): SiloRef[S] = this.flatMap(spore {
  val localSpore = s
  (x: T) => localSpore(x)
  SiloRef.fromFun(currentHost)(spore {
    val localRes = res
    () => localRes
  })
})
```

This should come as no surprise, given that flatMap is the monadic bind operation on SiloRefs, and SiloRef.fromFun is the monadic return operation. The reason why map is provided as one of the main operations of SiloRefs is that direct uses of map enable an important optimization based on operation fusion.

Expressing cache The cache operation can be expressed using flatMap and send:

```
def cache(): Future[Unit] = this.flatMap(spore {
  val localDoneSiloRef = DoneSiloRef(res => localDoneSiloRef)
  ).send()
```

Here, we first use flatMap to create a new silo that will be completed with the trivial value of the DoneSiloRef singleton object (e.g., unit). Essentially, invoking send on this trivial SiloRef causes the resulting future to be completed as soon as this SiloRef has been materialized in memory.

2.3 Fault Handling
F-P includes overloaded variants of the primitives discussed so far which enable the definition of flexible fault handling semantics. The main idea is to specify fault handlers for subgraphs of computation DAGs. Our guiding principle is to make the definition of the failure-free path through a computation DAG as simple as possible, while still enabling the handling of faults at the fine-granular level of individual SiloRefs.

Defining fault handlers Fault handlers may be specified whenever the lineage of a SiloRef is extended. For this purpose, the introduced map and flatMap primitives are overloaded. For example, consider our previous example, but extended with a fault handler:

```
object SiloRef {
  def fromTextFile(host: Host)(file: File): SiloRef[List[String]] = ...
  def fromFun[T](host: Host)(s: Spore[Unit, T]): SiloRef[T] = ...
  def fromLineage[T](host: Host)(s: SiloRef[T]): SiloRef[T] = ...
}
```

```
val adults = persons.map(spore { ps => ps.filter(p => p.age >= 18 })
val localDoneSiloRef = DoneSiloRef(res => localDoneSiloRef)
val adults2 = SiloRef.fromFun(h)(spore {
  val localVehicles = vehicles
  () => localVehicles
})
```

```
val adults = persons.map(spore { ps => ps.filter(p => p.age >= 18 })
```

```
// adults that own a vehicle
def computeOwners(v: SiloRef[List[Vehicle]]) = spore {
  val localVehicles = v
  (ps: List[Person]) => localVehicles.map(...)
}
```

```
val owners: SiloRef[List[Person, Vehicle]] = adults.flatMap(computeOwners)(vehicles2)
```

Importantly, in the flatMap call on the last line, in addition to computeOwners(vehicles2), the regular spore argument of flatMap,
The combined silo contains triples of type \((K, \text{Option}[A], \text{Option}[B])\). Using an additional map, the collection can be sorted by key, and adjacent triples be combined, yielding a \(\text{SiloRef}[:]\text{List}[(K, (A, B))]\) as required.

**Partitioning and groupByKey** A `groupByKey` operation on a group of silos containing collections needs to create multiple result silos, on each node, with ranges of keys supposed to be shipped to destination hosts. These destination hosts are determined using a partitioning function. Our goal, concretely:

\[
\text{val groupedSilos} = \text{groupByKey}(\text{silos})
\]

Furthermore, we assume that \(\text{silos.size} = N\) where \(N\) is the number of hosts, with hosts \(h_1, h_2, \text{etc.}\) We assume each silo contains an unordered collection of key-value pairs (a multi-map). Then, `groupByKey` can be implemented as follows:

- Each host \(h_i\) applies a **partitioning function** (example: `hash(key)` mod \(N\)) to the key-value pairs in its silo, yielding \(N\) (local) silos.
- Using `flatMap`, each pair of silos containing keys of the same range can be combined and materialized on the right destination host.

Using just the primitives introduced earlier, applying the partitioning function in this way would require \(N\) map invocations per silo. Thus, the performance of `groupByKey` could be increased significantly using a specialized combinator, say, “mapPartition” that would apply a given partitioning function to each key-value pair, simultaneously populating \(N\) silos (where \(N\) is the number of “buckets” of the partitioning function).

### 3.2 Peer-to-Peer Patterns

To illustrate the decentralized nature of our model, consider the following example: the local host aggregates some data as soon as two silos `vehicles` and `persons` have been materialized. The aggregation result is then combined with a silo `info` on local host. The final result is written to a distributed file system:

```scala
def aggregate(vs: \text{SiloRef}[:\text{List}[\text{Vehicle}]]),
               ps: \text{SiloRef}[:\text{List}[\text{Person}]],
               info: \text{SiloRef}[\text{Info}]) = ...

def write(result: String, fileName: String): Unit = ...
```

\[
\text{val vehicles} = \text{SiloRef}[:\text{List}[\text{Vehicle}]] = ...
\]

\[
\text{val persons} = \text{SiloRef}[:\text{List}[\text{Person}]] = ...
\]

\[
\text{val info} = \text{SiloRef}[\text{Info}] = ...
\]

\[
\text{val fileName: String} = "hdfs://"
\]

\[
\text{val done} = \text{info.flatMap}(\text{spore} \{ ...
\]

\[
\text{val localVehicles} = \text{vehicles}
\]

\[
\text{val localPersons} = \text{persons}
\]

\[
\text{val localInfo: Info} = ...
\]

\[
\text{aggregate(localVehicles, localPersons).map}(\text{spore} \{ ...
\]

\[
\text{val in = localInfo}
\]

\[
\text{res} => \text{combine}(\text{res}, \text{in})
\]

\[
\})\.map(\text{spore} \{
\]

\[
\text{val captured = fileName}
\]

\[
\text{combined} => \text{Utils.write(combined, captured)}
\]

\[
\text{done.cache()} // force computation
\]

This program does not tolerate failures of the local host: if it fails before the computation is complete, the result is never written to the file. Using fault handlers, though, it is easy to introduce a backup host that takes over in case the local host fails at any point:

\[
\text{val doCombine} = \text{spore}
\]

```
val localVehicles = vehicles
val localPersons = persons
(localInfo: Info) =>
aggregated = localVehicles.map((spore {
  val in = localInfo
  res => combine(res, in)
})

val done = info.flatMap(doCombine).map(doWrite)
val spore = spore {
  val
  val

new silo created on the backup host
after the materialization of
retry the original computation. In case the original host failed only
the captured of the
case. The spore for the non-failure case simply returns the
passing (a) spore for the non-failure case (b) a spore for the failure
has been completed asynchronously. The
transformations that are correct for sequential programs are also
existing reasoning techniques for sequential programs. Program
primitives of our language. The semantics is clearly stratified into
4.1 Operational semantics
In the following we give a small-step operational semantics of the
primitives of our language. The semantics is clearly stratified into
deterministic layer and a non-deterministic (concurrent) layer.
Importantly, this means our programming model can benefit from
existing reasoning techniques for sequential programs. Program
transformations that are correct for sequential programs are also
correct for distributed programs. Our programming model shares
this property with some existing approaches such as [29].

Notation and conventions. We write \( S' = S + (t \mapsto v) \) to express
the fact that \( S' \) maps \( t \) to \( v \) and otherwise agrees with \( S \). We write
\( S(t) = \text{Some}(v) \) to express the fact that \( S \) maps \( t \) to \( v \). We write
\( S(t) = \text{None} \) if \( S \) does not have a mapping for \( t \). Reduction is
defined using reduction contexts [31]. We omit the definition of
reduction contexts, since they are completely standard.

Configurations. The reduction rules of the deterministic layer
define transitions of host configurations \((t, E, S)^h\) of host \( h \) where
\( t \) is a term, \( E \) is a message queue, and \( S \) is a silo store. The
reduction rules of the non-deterministic layer define transitions of
sets \( H \) of host configurations. The reduced host configurations
are chosen non-deterministically in order to express concurrency
between hosts.
component of the Mapped and FMapped objects, in Figure 4 there are two types of messages: requests (Req) and
in both cases, the new SiloRef is deferred, and an object representing this derivation is returned.

4.1.1 Decentralized identification

(Section 4.2). In the second step we explain how these simplified rules have to be re-
tion rules in two steps. In the first step we explain simplified rules for subsequent updates of their information; decentralized identi-
ters are immutable upon construction. The key principles of the fault handling mechanism are:

4.1.2 Deterministic layer

A important property of our programming model is the fact that silos are uniquely identified using decentralized identifiers. A de-
centralized identifier i has two components: (a) the identifier of the host h that created i, and (b) a name i created fresh on h (e.g., an integer value): i = (h, i). Decentralized identifiers are important, since they reconcile two conflicting properties central to our model. The first property is building computation DAGs locally, without remote communication. This is possible using decentralized iden-
tifiers, since each host can generate new identifiers independently of other remote hosts. The second property is allowing SiloRefs to be freely copied between remote hosts. This is possible, since decentralized identifiers uniquely identify silos without the need for subsequent updates of their information; decentralized identi-
ters are immutable. This latter property is essential to enable com-
putational DAGs that are immutable upon construction. In our prog-
mograming model, computational DAGs are created using the standard monadic operations of SiloRefs. In particular, the flatten operation (monadic bind) in general requires that its argument spore captures SiloRefs that are subsequently copied to a remote host. Hence it is essential that SiloRefs and the decentralized identifiers they contain be freely copyable between remote hosts.

4.1.3 Nondeterministic layer

All reduction rules in the nondeterministic layer, shown in Figure 6, involve communication between two hosts.

4.2 Fault handling

The key principles of the fault handling mechanism are:

Fault handling. In the interest of clarity we present the reduction rules in two steps. In the first step we explain simplified rules
without fault handling semantics (Sections 4.1.2 and 4.1.3). In the second step we explain how these simplified rules have to be re-

Figure 5: Deterministic reduction.

at location i has value v. A request Req(h, r, i) is sent on behalf of host h to request the value of silo r at location i. The reception of a response Res(i, v) is handled by adding a mapping (i → v) to the store (rule R-Res). The reception of a request Req(h', r', i') is handled locally if materialization of the requested silo r is deferred and the parent silo r' in r's lineage has not been materialized either. In this case, the host sends a request to materialize r' to itself.

4.2.1 Deterministic layer

We first consider the reduction rules of the deterministic layer shown in Figure 5. The reduction rules for map (R-Map) and flatten (R-FMap) do not involve communication with other hosts. In each case, a new SiloRef r' is created that is derived from SiloRef r. The execution of the actual operation (map or flatten, respectively) is deferred, and an object representing this derivation is returned. In both cases, the new SiloRef r' refers to a silo created on host h by applying the spore value p to the value of silo r. The first component of the Mapped and FMapped objects, (h, i), is a fresh location created by host h to uniquely identify the result silo.

Most reduction rules are enabled when the current redex is an await term. The reduction of a term await(i) only continues when store S maps location i to value v. In all other cases, the current host removes the next message from its message queue E. As shown in Figure 4 there are two types of messages: requests (Req) and responses (Res). A response Res(i, v) tells its receiver that the silo

at the current host intends to send a message to a host h' where h' ≠ h, it is checked whether failed(h'). If it is the case that failed(h'), either the corresponding location (silo or future) is declared as failed (and fault handling deferred), or a suitable fault handler is located and a recovery step is attempted. In the following we explain the extended reduction rules shown in Figure 7.

In rule RF-Send, the host of the requested silo r is detected to have failed. However, the parent silos of r are all located on the
same (failed) host. Thus, in this case silo $s$ requesting silo $h$ hosted on $r'$ f

\[
\text{RF-ReqF} \quad E = \text{Req}(h', r', v') \cdot E' \Rightarrow R\text{-Send} \quad host(r') = h' \quad h' \neq h \quad i \text{ fresh} \quad i = (h, i) \quad m = E\text{Req}(h, r, i) \\
\{(R\text{send}(r), E, S)\} \cdot H \Rightarrow \{(R\text{send}(r), E, S, (t, E' \cdot m, S')h) \} \cup H
\]

\[
\text{R-Req1} \quad E = \text{Req}(h', r, v) \cdot E' \Rightarrow \text{Mat}(\iota) \quad S(\iota) = \text{Some}(v) \quad m = \text{Req}(v, \iota) \\
\{(R\text{await}(\iota)), E, S\} \cdot H \Rightarrow \{(R\text{await}(\iota)), E', S, (t, E' \cdot m, S')h) \} \cup H
\]

\[
\text{R-Req2} \quad E = \text{Req}(h', r, v) \cdot E' \Rightarrow \text{Mapped}(\iota, h, r', p, \text{None}) \quad r' = \text{Mat}(\iota) \quad S(\iota) = \text{None} \\
\{(R\text{await}(\iota)), E, S\} \cdot H \Rightarrow \{(R\text{await}(\iota)), E', S, (t, E' \cdot m, S')h) \} \cup H
\]

\[
\text{R-Req3} \quad E = \text{Req}(h'', r, v) \cdot E'' \Rightarrow \text{FMapped}(\iota, h, r'', p, \text{Some}(p)) \quad S(\iota) = \text{None} \quad m = E\text{Req}(h', r', v) \\
\{(R\text{await}(\iota)), E, S\} \cdot H \Rightarrow \{(R\text{await}(\iota)), E', S, (t, E' \cdot m, S')h) \} \cup H
\]

\[
\text{R-Req4} \quad E = \text{Req}(h''', r, v) \cdot E''' \Rightarrow \text{FMapped}(\iota, h, r''', p, \text{None}) \quad S(\iota) = \text{None} \\
\{(R\text{await}(\iota)), E, S\} \cdot H \Rightarrow \{(R\text{await}(\iota)), E', S, (t, E' \cdot m, S')h) \} \cup H
\]

\[
\text{RF-Req5} \quad E = \text{Req}(h''', r, v) \cdot E'''' \Rightarrow \text{FMapped}(\iota, h, r''', p, \text{None}) \quad S(\iota) = \text{None} \\
\{(R\text{await}(\iota)), E, S\} \cdot H \Rightarrow \{(R\text{await}(\iota)), E', S, (t, E' \cdot m, S')h) \} \cup H
\]

6: Nondeterministic reduction.

\[
\text{R-ReqF} \quad E = \text{Req}(h', r', v) \cdot E' \Rightarrow \text{Mapped}(\iota, h, r''', p, \text{Some}(p)) \quad alloc(v) = S(\iota) = \text{Some}(v) \\
\{(R\text{await}(\iota)), E, S\} \Rightarrow \{(R\text{await}(\iota)), E', S, (t, E' \cdot m, S')h) \} \cup H
\]

Figure 7: Fault handling.

same (failed) host. Thus, in this case silo $r$ is simply declared as failed, and fault handling is delegated to other parts of the computation DAG that require the value of $r$ (if any). Since $\text{send}$ is essentially a “sink” of a DAG, no suitable fault handler can be located at this point.

This is different in rule R-Req4. Here, host $h$ processes a message requesting silo $r$ which is the result of a $\text{flatMap}$ call. Materializing $r$ requires obtaining the value of silo $r'$, the result of applying $p$ to the value $v$ of the materialized parent $r''$. Importantly, if the host of $r'$ is failed, it means the computation of the DAG defined by $p$ did not result in a silo on an available host. Consequently, if the $\text{flatMap}$ call deriving $r$ specified a fault handler $p_f$, $p_f$ is applied to $v$ in order to recover from the failure. If the host of the resulting silo $r_f$ is not failed, the original request for $r$ is “modified” to request $r_f$ instead. This is done by removing message $\text{Req}(h', r, v')$ from the message queue and prepending message $\text{Req}(h'', r_f, v')$. Moreover, host $h$ sends a message to itself, requesting the value of silo $r_f$. 

9 2015/2/28
Rule RF-Req5 shows fault recovery in the case where the lineage of a requested silo does not specify a fault handler itself. In this case, host \( h \) creates two fresh locations \( l_p, l_a \). \( l_p \) is supposed to be eventually mapped to the result value of executing the fault handler of parent silo \( r' \). Host \( h \) requests this value from itself using a special message \( \text{ReqF}(h, r', l_p) \). Finally, the original request for silo \( r \) in message queue \( E \) is replaced with a request for silo \( r_a \). The silo \( r_a \) is created analogous to \( r \), but using silo \( \text{Mat}(l_p) \) as parent (eventually, location \( l_p \) is mapped to the result of applying the parent’s fault handler). As demonstrated by rule RF-ReqF, ReqF messages used to request the application of the fault handler are handled in a way that is completely analogous to the way regular Req messages are handled, except that fault handlers \( p_f \) are applied as opposed to regular spores \( p \).

5. Implementation

The presented programming model has been fully implemented in Scala, a functional programming language that runs on both JVMs and JavaScript runtimes. F-P is compiled and run using Scala 2.11.5, and considers only the JVM backend for now. Our implementation, which has been published as an open-source project, builds on two main Scala extensions:

- First, Pickling, a type-safe and performant serialization library with an accompanying, optional macro extension that is focused on distributed programming. It is used for all serialization tasks. Our F-P implementation benefits from the maturity of Pickling, which supports pickling/unpickling a wide range of Scala type constructors. Pickling has evolved from a research prototype to a production-ready serialization framework that is now in widespread commercial use.

- Second, the programming model makes extensive use of spores, closure-like objects with explicit, typed environments. While previous work has reported on an empirical evaluation of spores, our presented programming model and implementation turned out to be an extensive validation of spores in the context of distributed programming. In addition, our implementation required a thorough refinement of the way spores are pickled.

So far, we have used our implementation to build a small Spark-like distributed collections abstraction, and example data analytics applications, such as word count and group-by-join pipelines. Our prototype has also served as an experimentation platform for type-based optimizations, which we present in more detail below.

Further, we maintain a growing collection of these toy distributed frameworks (built on top of F-P) and applications on top of them (such as data analytics) at http://lampwww.epfl.ch/~hmiller/f-p/.

5.1 Serialization in the presence of existential quantification

Initially, to serialize most message types exchanged by the network communication layer, runtime-based unpicklers had to be used (meaning unpickling code discovering the structure of a type through introspection at runtime). A major disadvantage of runtime-based unpickling is its significant impact on performance. The reason for its initial necessity was that message types are typically generic, but the generic type arguments are existentially-quantified type variables on the receiver’s side. For example, the lineage of a SiloRef may contain instances of a type Mapped. This generic type has four type parameters. The receiver of a freshly unpicked Mapped instance typically uses a pattern match:

```scala
case mapped: Mapped[u, t, v, s] =>
```

The type arguments \( u, t, v, \) and \( s \) are type variables. While unknown, the static type of \( mapped \) is still useful for type-safety:

```scala
val newSilo = new LocalSilo[v, s](mapped.fun(value))
```

However, it is impossible to generate type-specific code to unpickle a type like \( \text{Mapped}[u, t, v, s] \). As a solution to this problem we propose what we call “self-describing” pickles. Basically, the idea is to augment the serialized representation with additional information about how to unpickle. The key is to capture the type-specific pickler and unpickler when the fully-concrete type of a Mapped instance is known:

```scala
def doPickle[T](msg: T) {
  implicit pickler: Pickler[T],
  unpickler: Unpickler[T] = Array[Byte] = ... 
}
```

Essentially, this means when \( \text{doPickle} \) is called with a concrete type \( T \), say:

```scala
doPickle[Mapped[Int, List[Int], String, List[String]]](mapped)
```

not only a type-specific implicit pickler (a type class instance) is looked up, but also a type-specific implicit unpickler. The \( \text{doPickle} \) method can then build a self-describing pickle as follows. First, the actual message is pickled using the pickler, yielding a byte array. Then, an instance of the following simple record-like class is created:

```scala
case class SelfDescribing(blob: Array[Byte],
  unpicklerClassName: String)
```

Besides the just produced byte array, it contains the class name of the type-specific unpickler. This enables, using this fully type-specific unpickler, even when the message type to be unpickled is only partially known. All that is required is an unpickler for type \( \text{SelfDescribing} \). First, it reads the byte array and class name from the pickle. Second, it instantiates the type-specific unpickler reflectively using the class name. (Note that this is possible on both the JVM as well as on JavaScript runtimes using Scala’s current JavaScript backend.) Finally, the unpickler is used to unpickle the byte array. In conclusion, this approach ensures (a) that a type that is pickleable using a type-specific pickler is guaranteed to be unpickleable by the receiver of the pickled \( \text{SelfDescribing} \) instance, and (b) that unpickling is as efficient as pickling, thanks to using type-specific unpicklers.

5.2 Type-based optimization of serialization

We have used our implementation to measure the impact of type-specific, compile-time-generated serializers (see above) on end-to-end application performance. In our benchmark application, a group of 4 silos is distributed across 4 different nodes/JVMs. Each silo is populated with a collection of “person” records. The application first transforms each silo using \( \text{map} \), and then using \( \text{groupBy} \) and \( \text{join} \). For the benchmark we measure the running time for a varying number of records.

We ran our experiments on a 2.3 GHz Intel Core i7 with 16 GB RAM under Mac OS X 10.9.5 using Java HotSpot Server 1.8.0-b132. For each input size we report the median of 7 runs. Figure 8 shows the results. Interestingly, for an input size of 100,000 records, the use of type-specific serializers resulted in an overall speedup of about 48% with respect to the same system using runtime-based serializers.

\(^7\) Note that the type arguments are inferred by the Scala compiler; they are only shown for clarity.

\(^8\) Review note: this site is publicly accessible, has no tracking capabilities, and is linked to in talks and from other various webpages meaning it already gets public traffic. One may visit it without identity concerns.

\(^4\) https://github.com/heathermiller/f-p
\(^5\) https://github.com/scala/pickling
\(^6\) Note that the type arguments are inferred by the Scala compiler; they are only shown for clarity.
6. Related Work

Alice ML [32] is an extension of Standard ML which adds a number of important features for distributed programming such as futures and proxies. The design leading up to F-P has incorporated many similar ideas, such as type-safe, generic and platform-independent pickling. In Alice, functions are stationary, but it is possible to send proxies, mobile wrappers for functions. Sending a proxy will not transfer the wrapped function; instead, when a proxy function is applied, the call is forwarded by the system to the original site as a remote invocation (pickling arguments and result appropriately). In F-P, however, functions are not wrapped in proxies but sent directly. Thus, calling a received function will not lead to remote invocations.

Cloud Haskell [12] leverages guaranteed-serializable, static closures for a message-passing communication model inspired by Erlang. In contrast, in our model spores are sent between passive, persistent silos. In contrast, in our model spores are sent between passive, persistent silos. Moreover, the coordination of concurrent activity is based on futures, instead of message passing. Closures and continuations in Termite Scheme [13] are always serializable; references to non-serializable objects (like open files) are automatically wrapped in processes that are serialized as their process ID. Similar to Cloud Haskell, Termite is inspired by Erlang. In contrast to Termite, F-P is statically typed, enabling advanced type-based optimizations. In non-process-oriented models, parallel closures [23] and RiverTrail [18] address important safety issues of closures in a concurrent setting. However, RiverTrail currently does not support capturing variables in closures, which is critical for the flatMap combinator in F-P. In contrast to parallel closures, spores do not require a type system extension in Scala.

Acute ML [33] is a dialect of ML which proposes numerous primitives for distributed programming, such as type-safe serialization, dynamic linking and rebinding, and versioning. F-P, in contrast, is based on spores, which ship with their serialized environment or they fail to compile, obviating the need for dynamic re binding. HashCaml [4] is a practical evolution of Acute ML’s ideas in the form of an extension to the OCaml bytecode compiler, which focuses on type-safe serialization and providing globally meaningful type names. In contrast, F-P merely a programming model, which does not require extensions to the Scala compiler.

ML5 [27] provides mobile closures verified not to use resources not present on machines where they are applied. This property is enforced transitively (for all values reachable from captured values), which is stronger than what plain spores provide. However, type constraints allow spores to require properties not limited to mobility. Transitive properties are supported either using type constraints based on type classes which enforce a transitive property or by integrating with type systems that enforce transitive properties. Unlike ML5, spores do not require a type system extension. Further, the F-P model sits on top of these primitives to provide a full programming model for distribution, which also integrates spores and type-safe pickling.

Systems like Spark [39], MapReduce [10], and Dryad [21] are just that—distributed systems. F-P is meant to act as more of a middleware to facilitate the design and implementation of such systems, and as a result provides much finer-grained control over details such as fault handling and network topology (i.e., peer-to-peer vs master/worker).

The Clojure programming language proposes agents [19]—stationary mutable data containers that users apply functions to in order to update an agent’s state. F-P, in contrast, proposes that data in stationary containers be immutable, and that transformations by function application form a persistent data structure. Further, Clojure’s agents are designed to manage state in a shared memory scenario, whereas F-P is designed with remote references for a distributed scenario.

The F-P model is also related to the actor model of concurrency [1], which features multiple implementations in Scala [15, 17, 36]. Actors can serve as in-memory data containers in a distributed system, like our silos. Unlike silos, actors encapsulate behavior in addition to immutable or mutable values. While only some actor implementations support mobile actors (none in Scala), mobile behavior in the form of serializable closures is central to the F-P model.

7. Conclusion and Future Work

We have presented F-P, a new programming model and principled substrate for building data-centric distributed systems. Built atop a foundation consisting of performant and type-safe serialization, and safe, serializable closures, we have shown that it’s possible to build elegant fault-tolerant functional systems. One insight of our model is that lineage-based fault recovery mechanisms, used in widespread frameworks for distribution, can be modeled elegantly in a functional way using persistent data structures. Our operational semantics shows that this approach makes it even amenable to formal treatment. We have also shown that F-P is able to express patterns of computation richer than those supported by common “big data” frameworks while maintaining fault-tolerance—such as decentralized peer-to-peer patterns of communication. Finally, we have implemented our approach in and for Scala, as well as numerous applications on top, and have discovered new ways to reconcile type-specific serializers with patterns of static typing common in distributed systems.

A great deal of future work remains. In the short-term, we aim to continue to build different sorts of distributed frameworks and applications atop F-P in an effort to work towards a production-ready implementation of our model for consumption by the Scala community at large.

In the long-term, we plan to better understand concerns of separate compilation in order to evaluate whether our model could be of help in coordinating between microservices.8

References


8 Microservices are small, independent (separately-compiled) services running on different machines which communicate with each other to together make up a single and complex application. They are a predominant trend in industry amongst rich and complicated web-based services.