Function Passing: A Model for Typed, Distributed Functional Programming

Heather Miller
EPFL, Switzerland
heather.miller@epfl.ch

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

Normen Müller
Trivadis, Germany
normen.mueller@trivadis.com

Jocelyn Boullier
EPFL, Switzerland
jocelyn.boullier@epfl.ch

Abstract

The most successful systems for "big data" processing have all adopted functional APIs. We present a new programming model we call *function passing* designed to provide a more principled substrate on which to build data-centric distributed systems. A key idea is to build up a persistent functional data structure representing transformations on distributed immutable data by passing well-typed serializable functions over the wire and applying them to this distributed data. Thus, the function passing model can be thought of as a persistent functional data structure that is *distributed*, where transformations to data are stored in its nodes rather than the distributed data itself. The model simplifies failure recovery by design–in the event of a failure, data is recovered by replaying function applications atop immutable data loaded from stable storage. Deferred evaluation is also central to our model; by incorporating deferred evaluation into our design only at the point of initiating network communication, the function passing model remains easy to reason about while remaining efficient in time and memory. We formalize our programming model using small-step operational semantics, and we provide an open-source implementation of our model in and for the Scala programming language, along with a case study of several example frameworks and end-user programs written atop of this model.

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

Data-centric programming is now commonplace. Meanwhile, 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, declarative interfaces to data distributed over tens to thousands of nodes, have become recognized for providing a more natural way for end-users and data scientists to reason about large-scale data.

Popular implementations of the MapReduce (Dean and Ghemawat 2008) model, such as Hadoop MapReduce (Apache 2015) for Java, have taken ideas from functional programming. But due to their impementation language, are unable to fully make use of functional language features such as closures, causing Hadoop developers to have to write significant amounts of boilerplate in order to emulate these functional patterns. Despite this, for nearly a decade, Hadoop has remained largely unchallenged, becoming the implementation of choice for many industrial large-scale data processing needs.

However, in recent years, a new generation of distributed systems for large-scale data processing have suddenly cropped up, built on top of emerging functional languages like Scala; such systems include Apache Spark (Zaharia et al. 2012), Twitter's Scalding (Twitter 2015), and Scoobi (NICTA 2015). These systems make use of functional language features in Scala in order to provide high-level, declarative APIs to end-users–requiring significantly less boilerplate. Moreover, these features have enabled computation to be shifted fully in memory, with systems like Spark achieving up to a 100x boost in performance over Hadoop (Apache 2016).

This sudden proliferation of new frameworks for distributed data-centric programming, all pushing the bar far beyond Hadoop, concurrent with the sudden growth in popularity of an emerging programming language begs the question–

has it been our programming languages that have limited us? Could it be that the primitives we build our systems upon are too low-level, causing us to struggle to reinvent the same tricky wheel over and over again? As these large-scale data processing applications continue to grow in importance, what can we as language designers do to make it easier for more of these frameworks to rise?

This paper presents a new programming model called the *function passing* model which has been designed to be a more principled substrate (or middleware) upon which to build data-centric distributed systems. It can be viewed as a generalization of the MapReduce/Spark programming model–though it is not limited to the MapReduce/Spark programming model alone as we will later show.

The key idea behind the function passing model is to keep distributed (immutable) data stationary, and to instead send functionality as function closures over the network. This enables two important benefits for distributed system builders; (a) since all computations are functional transformations on immutable data, fault-tolerance is made simple by design, and (b) communication is made well-typed by design, a common pain point for builders of distributed systems in Scala. Said another way, the function passing model attempts to more naturally model the paradigm of data-centric programming by extending monadic programming to the network. On this note, one might observe that the function passing model can actually be interpreted as somewhat of a dual to the actor model;[1] rather than keeping functionality stationary and sending data, in our model, we keep data stationary and send functionality to the data.

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

One important contribution of our model is a precise semantic specification of concepts central to fault recovery. The fault-recovery mechanisms of widespread systems such as Apache Spark, MapReduce (Dean and Ghemawat 2008) and Dryad (Isard et al. 2007) are based on the concept of *lineage* (Bose and Frew 2005; Cheney et al. 2009). 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 computation in a typed, functional setting–to our knowledge a novelty in the programming languages literature.

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 simpler 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 directed acyclic graphs of computations, supporting decentralized distributed computations. Deferred evaluation enables important optimizations while keeping programs simple to reason about.
- ***A formalization of our programming model*** based on small-step operational semantics. In particular, we present a formal model of the concept of lineage in the context of a typed functional language. Our treatment using standard techniques in programming languages aims to offer a fresh angle on a concept which so far has mainly been studied in the database and systems communities, using other methods. Furthermore, the meta-theoretic development has enabled us to uncover a deep connection between type soundness and serializability. Consequently, the presented type system leverages a notion of serializable types in order to ensure subject reduction.
- ***A distributed implementation of the programming model*** in and for Scala as a middleware.[2] In addition, we present prototype versions we have built of popular frameworks like Spark and MBrace using the function passing model, and end-user applications we have built using each of these prototype frameworks.

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. In Section 6, we show examples of different sorts of distributed frameworks built atop of the function passing model. Finally, we discuss related work in Section 7, and conclude in Section 8.

[1] There are many variations and interpretations of the actor model; in saying our model is somewhat of a dual, we simply mean to highlight that programmers need not focus on programming with typically stationary message handlers. Instead, our model focuses on a monadic interface for programming with data (and sending functions instead).

[2] `https://github.com/heathermiller/f-p`

2. Overview of Model

2.1 Essence

In the broadest sense, the function passing model can be thought of as a sort of persistent functional data structure with structural sharing. However, rather than containing pure data, instead the data structure represents a directed acyclic graph (DAG) of functional transformations on distributed data. The root node of is immutable data read from stable storage (*e.g.,* Amazon S3); edges represent functional transformations on immutable data represented as nodes of the DAG.

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. Subgraphs of the DAG are called *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 the function passing model is the careful use of deferred evaluation. Computations on distributed data are typically not executed eagerly; instead, applying a function to distributed data just creates an immutable lineage. To make a network call and thus 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 (called `send()`) makes network communication (and thus possibilities for latency) explicit, which is considered to be a strength when designing distributed systems (Waldo et al. 1996). 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 (Chambers et al. 2010).

2.2 The Model

The function passing model consists of three main components:

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

Silos A silo is a typed and immutable 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`.

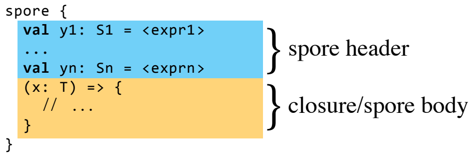
SiloRefs Similar to a proxy object, a SiloRef represents, and allows interacting with, both local and remote silos. SiloRefs are immutable, storing identifiers to locate possibly remote silos. SiloRefs are also typed (`SiloRef[T]`) corresponding to the type of their silo's data, leading to well-typed network communication. The SiloRef provides three primitive operations/combinators (some are lazy, some are not): `map`, `flatMap`, `send`, and `cache`. The `map` method makes use of deferred evaluation; it applies a user-defined function to data pointed to by the SiloRef, creating in a new silo containing the result of this application, though this application is *deferred*. That is, this computation is only kicked off when the `send` method is invoked. Calling `send` triggers queued up operations, like those scheduled by `map` invocations, to be possibly sent over the network and applied to their corresponding silo (whether local or remote), and the result to be returned to the caller, completed as a future. This makes it possible to queue up or stage transformations in order to optimize network communication. Like `map`, the application of `flatMap` is deferred. `flatMap` applies a user-defined function to data pointed to by the SiloRef. Unlike `map`, however, the user-defined function passed to `flatMap` returns a SiloRef whose contents are transferred to the new silo returned by `flatMap`. Essentially, `flatMap` enables accessing the contents of (local or remote) silos from within remote computations. `cache` kicks off queued up, deferred evaluation, and caches the result in memory (possibly remotely). We illustrate these primitives in more detail in Section 2.4.

Spores Spores (Miller et al. 2014) 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.



```
spore {
  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

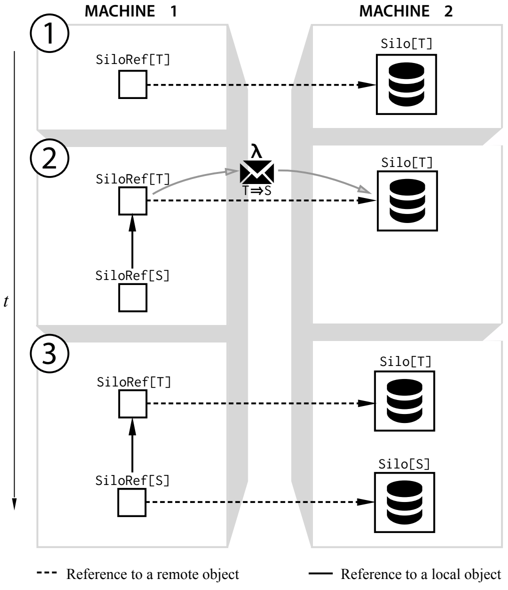


Figure 1: Basic function passing model.

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 are 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 opt into 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 `RDDs` 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 (Miller et al. 2014).

2.3 Basic Usage

We begin with a simple visual example to illustrate the basics of the function passing model.

The main handle users have to the framework is via SiloRefs. A SiloRef can be thought of as an immutable handle to distributed data contained within a corresponding silo. Users interact with this distributed data by applying functions (as spores) 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 λ 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 λ. 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 function passing 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.

2.4 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`

map `def map[S](s: Spore[T, S]): SiloRef[S]`
The `map` method takes a spore that is to be applied to the data in the silo associated with the given SiloRef. Rather than immediately sending the spore across the network, and waiting for the operation to finish, the `map` method's evaluation is *deferred*. Without involving any network communication, it immediately returns a SiloRef referring to a new, soon-to-be-created silo. This new SiloRef only contains lineage information, namely, a reference to the original SiloRef, a reference to the argument spore, and the information that it is the result of a `map` invocation. As we explain below, another method, `send` or `cache`, must be called explicitly to force the materialization of the result silo.

To better understand how DAGs are created and how remote silos are materialized, 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 { ps => ps.filter(p => p.age >= 18) })
```

flatMap

`def flatMap[S](s: Spore[T, SiloRef[S]]): SiloRef[S]`
Like `map`, the `flatMap` method takes a spore that is to be applied to the data in the silo of the given SiloRef. However, 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 `flatMap` is defined to contain the data of the silo that the user-defined spore returns. The `flatMap` 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 `(Person, Vehicle)` pairs where the names of person and vehicle owner match (graphically shown in Figure 2, part 2):

```
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.flatMap(spore {
  val localVehicles = vehicles // spore header
  ps =>
    localVehicles.map(spore {
      val localps = ps // spore header
      vs =>
        localps.flatMap(p =>
          // list of (p, v) for a single person p
          vs.flatMap {
            v =>
              if (v.owner.name == p.name) List((p, v))
              else Nil
          }
        )
    })
})
```

Note that the spore passed to `flatMap` declares the capturing of the `vehicles` SiloRef in 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 `vehicles` silo in addition to the `ps` list of `Person` records. This requires calling `map` on `localVehicles`. However, `map` returns a SiloRef; thus, invoking `map` on `adults` instead of `flatMap` 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`.

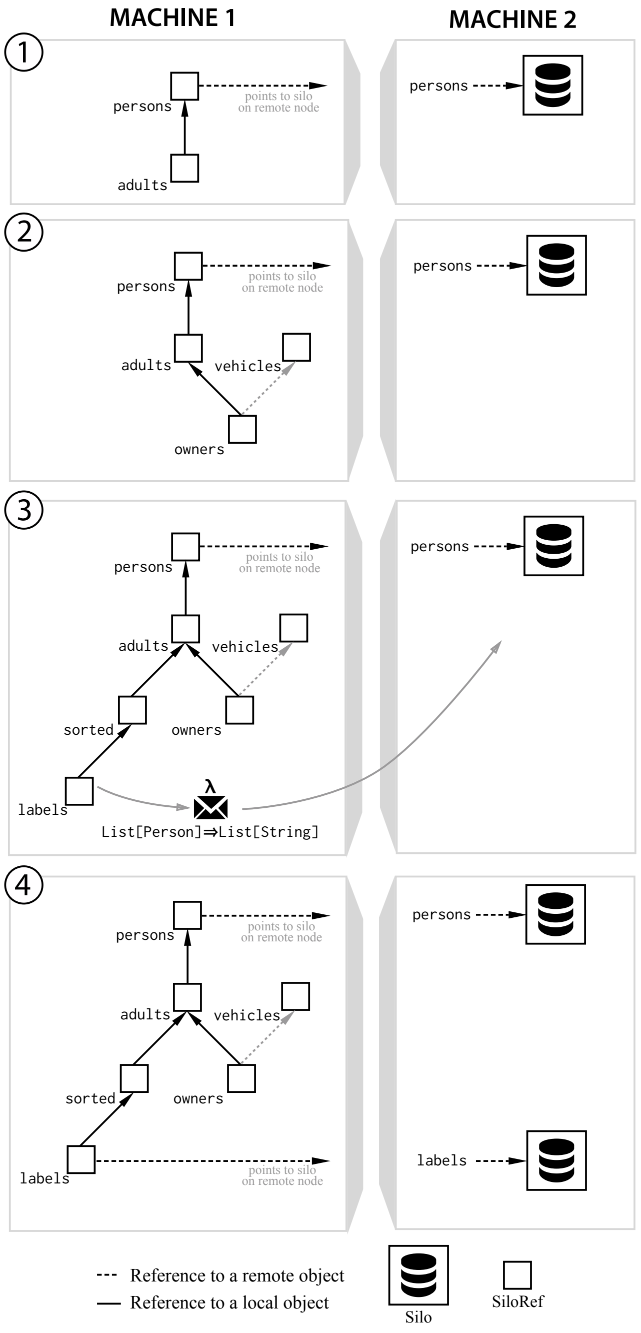

MACHINE 1
MACHINE 2
1
persons
points to silo on remote node
adults
persons
2
persons
points to silo on remote node
adults
vehicles
owners
persons
3
persons
points to silo on remote node
adults
vehicles
sorted
owners
labels
λ
List[Person]⇒List[String]
persons
4
persons
points to silo on remote node
adults
vehicles
sorted
owners
labels
points to silo on remote node
persons
labels
Reference to a remote object
Reference to a local object
Silo
SiloRef

Figure 2: A simple DAG in the function passing model.

Although the expressiveness of the `flatMap` combinator subsumes that of the `map` combinator (see Section 2.4.3), keeping `map` as a (lightweight) primitive enables more opportunities for optimizing computation DAGs (*e.g.,* operation fusion (Chambers et al. 2010)).

send `def send(): Future[T]`
As mentioned earlier, the execution of computations built using SiloRefs is deferred. The `send` operation *forces* the deferred 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 has a latency several orders of magnitude greater than accessing a word in main memory, providing an explicit send operation is a judicious choice (Waldo et al. 1996).

To enable materialization of remote silos to proceed concurrently, the `send` operation immediately returns a future (Haller et al. 2012). 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 (Zaharia et al. 2012). 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.

Given the running example so far, we can add another subgraph branching off of `adults`, which sorts each `Person` by age, produces a `String` greeting, 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 2. 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`.

2.4.1 Creating Silos

Besides a type definition for SiloRef, our framework provides a companion singleton object (a Scala module). 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.

2.4.2 Type Polymorphism and Silos/SiloRefs

An important property of silos is that they are polymorphic in the type of data that they hold. Importantly, silos are not just polymorphic in the *element type* of their data, but in the *type of their entire dataset*. For example, a silo might contain a Red-Black tree with elements of type `Person` for some ADT `Person`, ordered by one of the fields of the `Person` type. Another silo might contain a completely different collection type, say, a linked list. This type polymorphism enables optimizing silos according to their data access patterns. Given that different data types may have specialized operations (e.g., a tree map could provide a range projection), the key to enabling this type polymorphism is the fact that a spore sent to a silo may apply arbitrary functions to the silo's data. Thus, the SiloRef API itself is not limited to providing just a fixed set of built-in operations (in contrast to RDDs in Spark, for example).

2.4.3 Expressiveness

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

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

[3] For clarity, only method signatures are shown.

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.

2.5 Fault Handling

The function passing model 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:

```
val persons: SiloRef[List[Person]] = ...
val vehicles: SiloRef[List[Vehicle]] = ...
// copy of `vehicles` on different host `h`
val vehicles2 = 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(vehicles),
                 computeOwners(vehicles2))
```

Importantly, in the flatMap call on the last line, in addition to computeOwners(vehicles), the regular spore argument of flatMap, computeOwners(vehicles2) is passed as an additional argument. The second argument registers a *failure handler* for the subgraph of the computation DAG starting at adults. This means that if during the execution of computeOwners(vehicles) it is detected that the vehicles SiloRef has failed, it is checked whether the SiloRef that the higher-order combinator was invoked on (in this case, adults) has a failure handler registered. In that case, the failure handler is used as an alternative spore to compute the result of adults.flatMap(..). In this example, we specified computeOwners(vehicles2) as the failure handler; thus, in case vehicles has failed, the computation is retried using vehicles2 instead.

3. Higher-Order Operations

The introduced primitives enable expressing surprisingly intricate computational patterns.

Higher-order operations such as variants of map, reduce, and join, operating on collections of data partitions, distributed across a set of hosts, are required when implementing abstractions like Spark's distributed collections (Zaharia et al. 2012). Section 3.1 demonstrates the implementation of some such operations in terms of silos.

In addition, even more patterns are possible thanks to the decentralized nature of our programming model, which removes the limitations of master/worker host configurations. Section 3.2 shows examples of peer-to-peer patterns that are still fault-tolerant.

3.1 Higher-Order Operations

join Suppose we are given two silos with the following types:

```
val silo1: SiloRef[List[A]]
val silo2: SiloRef[List[B]]
```

as well as two hash functions computing hashes (of type K) for elements of type A and type B, respectively:

```
val hashA: A => K = ...
val hashB: B => K = ...
```

The goal is to compute the hash-join of silo1 and silo2 using a higher-order operation hashJoin:

```
def hashJoin[A, B, K](s1: SiloRef[List[A]],
                      s2: SiloRef[List[B]],
                      f: A => K,
                      g: B => K)
  : SiloRef[List[(K, (A, B))]] = ???
```

To implement hashJoin in terms of silos, the types of the two silos first have to be made equal, through initial map invocations:

```
val s12: SiloRef[List[(K, Option[A], Option[B])]] =
  s1.map(spore { l1 => l1.map(x => (f(x), Some(x), None)) })
val s22: SiloRef[List[(K, Option[A], Option[B])]] =
  s2.map(spore { l2 => l2.map(x => (g(x), None, Some(x))) })
```

Then, we can use `flatMap` to create a new silo which contains the elements of both silo `s12` and silo `s22`:

```
val combined = s12.flatMap(spore {
  val localS22 = s22
  (triples1: List[(K, Option[A], Option[B])]) =>
    s22.map(spore {
      val localTriples1 = triples1
      (triples2: List[(K, Option[A], Option[B])]) =>
        localTriples1 ++ triples2
    })
})
```

The combined silo contains triples of type `(K, Option[A], Option[B])`. Using an additional `map`, the collection can be sorted by key, and adjacent triples be combined, yielding a `SiloRef[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:

```
val groupedSilos = groupByKey(silos)
```

Furthermore, we assume that `silos.size` $= N$ where N is the number of hosts, with hosts h_1, h_2, etc. We assume each silo contains an unordered collection of key-value pairs (a multimap). 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

3.2.1 Essence

So far, our examples have focused on master-worker topologies that underly models like Spark–*i.e.,* a master node specifies identical DAGs of computation for all worker nodes to follow.

The function passing model, however, is not limited to these sorts of topologies. It is indeed possible to develop decentralized, peer-to-peer topologies on top of the function passing model. For example, a single compute node may host silos that are remotely referenced by remote SiloRefs, as well as SiloRefs remotely referencing silos on other compute nodes.

Further, as we show in the following example, it's also possible for multiple clients to build completely different DAGs of computation off of some source silo. In effect, this enables datasets to be shared–they exist once in memory on some node, but can be used and transformed in different ways by different clients.

Consider the following example. We start by populating an initial silo representing a dataset of `Vehicle` objects on `Host("lmpsrv1.scala-lang.org", 9999)`.

```
val lmpsrv1 = Host("lmpsrv1.scala-lang.org", 9999)

// client #1
// populate initial silo
val vehicles: SiloRef[List[Vehicle]] =
  Silo.fromTextFile(lmpsrv1)("hdfs://...")

val silo2 = vehicles.map(spore {
  (vs: List[Vehicle]) =>
    // extract US state from license plate string,
    // e.g, "FL329098"
    vs.map(v => (v.licensePlate.take(2), v)).toMap
})
val vehiclesPerState = silo2.send()

// client #2
// get siloref for silo that is being materialized
// due to client #1
val vehicles: SiloRef[List[Vehicle]] =
  Silo.fromTextFile(lmpsrv1)("hdfs://...")

val silo2 = vehicles.map(spore {
  // list all vehicles manufactured since 2013
  (vs: List[Vehicle]) =>
    vs.filter(v => v.yearManufactured >= 2013)
})
val vehiclesSince2013 = silo2.send()
```

Here, client #1 would like to perform some sort of computation based on the states that vehicles are registered in. Another client, client #2 would also like to access this dataset. To do so, one must simply once again invoke `fromTextFile` on the same host, `Host("lmpsrv1.scala-lang.org",9999)` to obtain a SiloRef that points to a corresponding silo that is already or soon to be materialized. From here, client #2 is able to build an entirely different DAG of computations, for instance in this example, filtering the original `vehicle` dataset to obtain only vehicles manufactured since 2013.

3.2.2 Decentralized Fault-Handling

Another peer-to-peer pattern possible in the function passing model is decentralized fault handling. One may specify strategies to transfer computation to other nodes in the event of failure.

Consider the following example: an aggregation should be performed as soon as two silos `vehicles` and `persons` have been materialized. The aggregation result is then combined with a silo `info` on some host different from the local host. The final result is written to a distributed file system:

```
object Utils {
  def aggregate(vs: SiloRef[List[Vehicle]],
                ps: SiloRef[List[Person]]): SiloRef[String] = ...
  def write(result: String, fileName: String): Unit = ...
}
val vehicles: SiloRef[List[Vehicle]] = ...
val persons:  SiloRef[List[Person]]  = ...
val info:     SiloRef[Info]          = ...
val fileName: String                 = "hdfs://..."
val done    = info.flatMap(spore {
  val localVehicles = vehicles
  val localPersons  = persons
  (localInfo: Info) =>
    aggregate(localVehicles, localPersons).map(spore {
      val in = localInfo
      res => combine(res, in)
    })
}).map(spore {
  val captured = fileName
  combined => Utils.write(combined, captured)
})
done.cache() // force computation
```

This program does not tolerate failures of the host of `info`: if it fails before the computation is complete, the result is never written to the file.

We can overcome this using fault handlers. It is possible to introduce another backup host which takes over in case the host of `info` (which is the same as the host of `done`) fails at any point. Let's try the above computation again, this time using fault handlers to transfer the computation to a backup node in the event of a failure:

```
val doCombine = spore {
  val localVehicles = vehicles
  val localPersons  = persons
  (localInfo: Info) =>
    aggregate(localVehicles, localPersons).map(spore {
      val in = localInfo
      res => combine(res, in)
    })
}
val doWrite = spore {
  val captured = fileName
  combined => Utils.write(combined, captured)
}
val done      = info.flatMap(doCombine).map(doWrite)
val backup    = SiloRef.fromFun(hostb)(spore { () => true })
val recovered = backup.flatMap(
  spore {
    val localDone = done
    x => localDone
  },
  spore { // fault handler
    val localInfo      = info
    val localDoCombine = doCombine
    val localDoWrite   = doWrite
    val localHostb     = hostb
    x =>
      // fromLineage makes sure, we re-run on hostb,
      // rather than the host of info. That is, we
      // just duplicate the lineage.
      val restoredInfo =
        SiloRef.fromLineage(localHostb)(localInfo)

      restoredInfo.flatMap(localDoCombine)
                  .map(localDoWrite)
  }
)
done.cache()      // force computation on host of local
recovered.cache() // force computation on backup host
```

First, the local variables `doCombine` and `doWrite` refer to the verbatim spores passed to `flatMap` and `map` above. Second, `backup` is a dummy silo on a backup host `hostb`. It is used to send a spore to the backup host in a way that allows it to detect whether the host of `done`/`info` has failed. The fault handling is done by calling `flatMap` on `backup`, passing (a) a spore for the non-failure case and (b) a spore for the failure case. The spore for the non-failure case simply returns the `done` SiloRef. Importantly, this enables `hostb` to detect failures of the host of `done`. Upon detecting such a failure, `backup.flatMap` applies the spore for the failure case. In this case, the lineage of the captured `info` SiloRef is used to restore its original contents in a new silo created on the backup host `hostb`. Its SiloRef is then used to retry the original computation.

4. Formalization

While so far we have focused on a high-level description of our model, visualizing, and building intuition via examples, we now shift gears in an effort to make the primitives of our model more precise. Rather than presenting a full formalization of our model, we will zoom in on a few interesting highlights that fell out of our formalization. In particular:

- A new treatment of the concept of lineage using PL techniques as opposed to the way lineage has been studied in the database and systems communities.
- A newly-uncovered connection between type soundness and serializability.

This is based on a formalization of our programming model in the context of a typed lambda calculus with records. The full formalization and the proof of a subject reduction theorem can be found in the companion technical report (Haller et al. 2016).

4.1 Formalizing Lineages

In the following we formally model the concept of lineage which has existed in other communities (in particular, the database and systems communities), but which has not been treated in the context of standard PL techniques such as small-step operational semantics. Formalizing lineages is interesting for two reasons: first, our formal model shows that lineages nicely map to and integrate with concepts from functional programming. Second, the reduction relation of the computation model is significantly different from standard reduction relations for higher-order functional languages.

In the following we first summarize the abstract syntax of a core language with essential constructs of our programming model. Then, we provide an overview of reduction semantics, highlighting select reduction rules for lineages.

4.1.1 Syntax

Figure 3 shows the abstract syntax of our core language. Besides standard terms, the language includes terms related to (a) spores, (b) silos, and (c) futures. The `spore` term creates a new spore. It contains a list of variable definitions, the spore header, and a closure which may only refer to its parameter and variables in the spore header. The `spawn` term creates a new host capable of hosting silos. The `populate` term initializes a new silo on a given host with a given data value. The `map`, `flatMap`, and `persist` terms create lineages of silo transformations represented as silo references. The `send` term forces the materialization of the silo corresponding to its argument silo reference; `send` returns a future which is asynchronously completed with the silo's value. The `await` term waits for the completion of its argument future and returns the future's value. Locations ι are used to refer to both futures and hosts.

Values in our language are as expected: besides abstractions and record values they include spore values, locations, and lineages. Locations and lineages are not part of the "surface syntax" of our language; they are only introduced by reduction (see Section 4.1.2). A lineage is a value of a simple datatype with constructors *Mat, Mapped, FMapped*, and *Persist*. The constructors include all information required for *materializing* a silo with the result of applying the described transformations.

In addition to standard function and record types, the language has types for spores, hosts, lineages, and futures. A spore type $T \Rightarrow T' \ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \}$ includes the types $\overline{T}$ of the variables declared in the header of the spore. (The name $\mathcal{C}$ stands for "capture types".)

$t ::=$	*terms:*
x	variable
$\mid (x : T) \Rightarrow t$	abstraction
$\mid t\ t$	application
$\mid \{\overline{l = t}\}$	record
$\mid t.l$	selection
$\mid \texttt{spore}\ \{\ \overline{x : T = t}\ ; (x : T) \Rightarrow t\ \}$	spore
$\mid \texttt{spawn}(t)$	spawn host
$\mid \texttt{populate}(t, t)$	populate silo
$\mid \texttt{map}(t, t)$	map
$\mid \texttt{flatMap}(t, t)$	flatMap
$\mid \texttt{persist}(t)$	persist
$\mid \texttt{send}(t)$	send
$\mid \texttt{await}(t)$	await future
$\mid \iota$	location
$\mid r$	lineage
$v ::=$	*values:*
$(x : T) \Rightarrow t$	abstraction
$\mid \{\overline{l = v}\}$	record value
$\mid p$	spore value
$\mid \iota$	location
$\mid r$	lineage
$p ::= \texttt{spore}\ \{\ \overline{x : T = v}\ ; (x : T) \Rightarrow t\ \}$	
$r ::=$	*lineage:*
$\mathrm{Mat}(\omega)$	materialized
$\mid \mathrm{Mapped}(\omega, r, p)$	`map` lineage
$\mid \mathrm{FMapped}(\omega, r, p)$	`flatMap` lineage
$\mid \mathrm{Persist}(\omega, r, v)$	`persist` lineage
$\omega ::= (h, i) \quad \text{where } i \in \mathbb{N}$	identifier
$T ::=$	*types:*
$T \Rightarrow T$	abstraction type
$\mid \{\overline{l : T}\}$	record type
$\mid T \Rightarrow T\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \}$	spore type
$\mid \texttt{Host}$	host type
$\mid \texttt{SiloRef}[T]$	lineage type
$\mid \texttt{Future}[T]$	future type

Figure 3: Abstract syntax of core language.

4.1.2 Reduction Semantics for Lineages

One of our goals is a formal model of lineages using tools familiar to the programming languages community. Thus, in the following we present a small subset of reduction rules related to the construction of lineages (complete reduction semantics can be found in (Haller et al. 2016)).

Every lineage originates from a materialized silo, represented as a value $Mat(\omega)$ in our formal model (see Figure 3). Here, ω is the silo's identifier such that $\omega = (h, i)$ where h is a host identifier and i is an integer that is unique on host h. Assume that a materialized silo $Mat(\omega)$ is bound to a variable x; then the following expression (in core language syntax) creates an extended lineage based on the `map` combinator:

$$
\begin{array}{lll}
E ::= & & \textit{eval. contexts:} \\
& [\,] & \text{hole} \\
& \mid E\ t & \text{application (fun)} \\
& \mid v\ E & \text{application (arg)} \\
& \mid \{\overline{l = v}; l_i = E; \overline{l' = t}\} & \text{record} \\
& \mid E.l & \text{selection} \\
& \mid \mathsf{spore}\ \{\ \overline{x : T = v}; x_i : T_i = E; & \\
& \quad \overline{x' : T = t}; (x : T) \Rightarrow t\ \} & \text{spore} \\
& \mid \mathsf{spawn}(E) & \text{spawn} \\
& \mid \mathsf{populate}(E, t) & \text{populate (host)} \\
& \mid \mathsf{populate}(v, E) & \text{populate (spore)} \\
& \mid \mathsf{map}(E, t) & \text{map (ref)} \\
& \mid \mathsf{map}(v, E) & \text{map (fun)} \\
& \mid \mathsf{flatMap}(E, t) & \text{flatMap (ref)} \\
& \mid \mathsf{flatMap}(v, E) & \text{flatMap (fun)} \\
& \mid \mathsf{persist}(E) & \text{persist} \\
& \mid \mathsf{send}(E) & \text{send} \\
& \mid \mathsf{await}(E) & \text{await}
\end{array}
$$

Figure 4: Evaluation context.

R-Map

$$\frac{r' = \text{Mapped}((h, i), r, p) \quad i \text{ fresh}}{E[\mathsf{map}(r, p)] \mid \mu \rightarrow^h E[r'] \mid \mu'}$$

R-FMap

$$\frac{r' = \text{FMapped}((h, i), r, p) \quad i \text{ fresh}}{E[\mathsf{flatMap}(r, p)] \mid \mu \rightarrow^h E[r'] \mid \mu'}$$

R-Persist

$$\frac{r' = \text{Persist}((h, i), r, \cdot \cup \cdot) \quad i \text{ fresh}}{E[\mathsf{persist}(r)] \mid \mu \rightarrow^h E[r'] \mid \mu'}$$

Figure 5: Select rules for sequential reduction.

$$\mathsf{map}(x, \mathsf{spore}\ \{\ (y : T) \Rightarrow t\ \})$$

The result of reducing this expression is the lineage $Mapped(\omega', Mat(\omega), \mathsf{spore}\ \{\ (y : T) \Rightarrow t\ \})$ where $\omega' = (h, i')$, i.e., the resulting silo ω' is materialized on the same host h as silo ω.

Since new lineage identifiers can be created locally, creating lineages does not require network communication. Therefore, the reduction rules for creating lineages are part of the local reduction relation.

Figure 5 shows three example rules. The reduction rules use the definition of evaluation contexts shown in Figure 4. Evaluation contexts capture the notion of the "next subterm to be evaluated." We write $E[t]$ for the term obtained by replacing the hole in evaluation context E with term t.

The sequential reduction relation has the form $E[t] \mid \mu \rightarrow^h E[t'] \mid \mu'$ with stores μ and μ'. Stores are required for the dynamic allocation of futures and hosts. A store μ is a partial function mapping locations ι to values v. The annotation with host h is used for creating *identifiers* $\omega = (h, i)$ for lineages.

Rules R-Map, R-FMap, and R-Persist describe the creation of lineages. Rules R-Map and R-FMap create lineages using the constructors *Mapped* and *FMapped*, respectively. The new lineage has a fresh identifier (h, i) which uniquely identifies the corresponding (logical) silo. In each case, the spore value p is stored in the new lineage; this enables a materialization of the silo identified by (h, i) using parent lineage r and spore p. Rule R-Persist creates a lineage using the *Persist* constructor. *Persist* contains a function $(\cdot \cup \cdot)$ enabling host h to persist silo r; essentially, the set-union function is used to add the silo identifier (h, i) to a set of persisted silos. The result of materializing a persisted silo is cached. (See our complete formalization (Haller et al. 2016) for details, including a corresponding rule for unpersisting a silo.)

As we can see, the above rules do not execute their corresponding higher-order functions; they merely create lineages. A lineage contains everything necessary to materialize the contents of its corresponding silo. Moreover, lineages are serializable: as a result, they can be used to initiate the materialization of a silo located on a remote node.

The immutability of silos and lineages enables equational reasoning about programs, a cornerstone of functional programming. Lineages are also persistent data structures, where multiple lineages are able to share common ancestors, thereby avoiding duplicate materializations. Taken together, we believe that lineages in general, and the function passing model in particular are a natural fit for functional programming, and lend themselves for embedding in a functional programming language (see Section 5).

4.2 Type Soundness and Serializability

As part of the meta theory of our core language, we have proved a subject reduction theorem, an important property for establishing type soundness (see the companion technical report (Haller et al. 2016) for the complete proof). The meta-theoretic development enabled us to uncover a connection between type soundness and serializability. Essentially, without a static serializability guarantee for certain values, subject reduction could be violated. Consequently, we have devised a type system, both as part of our formal development and as part of our implementation, which provides the required guarantee, ensuring subject reduction.

In a language without concurrency or distribution, subject reduction (or type preservation) means that the result of reducing a well-typed term of some type T is again a well-typed term of type T. In our case, reduction is more complex due to distributed computation across multiple hosts. Thus, our formalization extends subject reduction from well-typed terms to *well-formed configurations* of hosts.

A configuration H is a set of hosts $(t, \mu, Q, S)^h \in H$ where h is a host identifier, t is a term reduced by h, μ is a store (see Section 4.1.2), Q is a queue of incoming messages, and S is a silo store. The message queue Q buffers incoming messages which are processed asynchronously by h. The silo store S maps identifiers ω to silos. Essentially, S contains all silos hosted on h.

We write $\Delta \vdash H$ to express that configuration H is well-formed in *silo store typing* Δ. Analogous to the standard notion of a store typing (Pierce 2002), a silo store typing maps silo identifiers ω to types T. The silo store typing is used to assign types to lineages (see rule T-SiloRef in the technical report (Haller et al. 2016)).

Using the above definitions the subject reduction theorem is formulated as follows:

Theorem 1. (Subject Reduction) *If* $\Delta \vdash H$ *and* $H \twoheadrightarrow H'$ *then* $\Delta' \vdash H'$ *for some* $\Delta' \supseteq \Delta$.

Essentially, for the subject reduction theorem to hold, reduction of a well-formed configuration must result in another well-formed configuration. Importantly, the well-formedness of a configuration H implies the well-formedness of each host in H. In turn, the well-formedness of a host implies the well-formedness of its message queue. This means that the message queue of a host that received a message as a result of a reduction step must be well-formed. This is only possible if values contained in newly-received messages are well-typed in an empty environment. For example, rule WF-Q3 in (Haller et al. 2016) requires the value of the current message to be well-typed in an empty environment. Being able to type-check received values in an empty environment is essential, since the type environments of the sender and the receiver are disjoint.

The following theorem establishes that the *serializability* of types implies this property (we use the typing judgement $\Gamma; \Sigma; \Delta \vdash t : T$ as defined in (Haller et al. 2016)):

Theorem 2. (Serializable Values) *If* $\Gamma; \Sigma; \Delta \vdash v : T$ *and* $serializable(T)$ *then* $\emptyset; \emptyset; \Delta \vdash v : T$.

The predicate $serializable(T)$ is defined inductively on the structure of type T. (It corresponds precisely to the use of the `Pickler[T]` type class in our implementation.) Essentially, the type system of our core language ensures that the types of values sent within messages are serializable. By Theorem 2 these values are well-typed in an empty environment, which, as discussed, is essential for subject reduction to hold.

5. Implementation

The presented programming model has been fully implemented in Scala, a functional programming language that runs on both JVMs and JavaScript runtimes. The function passing model is compiled and run using Scala 2.11.8, and considers only the JVM backend for now. Our prototype, which has been published as an open-source project,[4] builds on two main Scala extensions:

- First, Pickling (Miller et al. 2013),[5] 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 function passing implementation benefits from the maturity of Pickling, which supports pickling/unpickling a wide range of Scala type constructors.
- Second, the programming model makes extensive use of spores, closure-like objects with explicit, typed environments. While (Miller et al. 2014) has reported an 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, a simple version of the MBrace (Dzik et al. 2013) framework, and example data analytics applications, such as word count and group-by-join pipelines. We give an introduction and step through some of these example applications in Section 6.

5.1 Static Serialization and Generics

In a function passing program, code using `SiloRef`s may access multiple silos hosted on different remote nodes. Therefore, applying a combinator such as `map` to a `SiloRef` generally requires sending a message to a remote node instructing it to apply `map` to the contents of the corresponding silo. For example, consider the following invocation of `map` on a SiloRef `in`:

```
val out =
  in.map(spore { (l: List[String]) => l.map(_.toUpperCase) })
```

The above snippet takes a list of strings in an input silo `in` and produces an output silo `out` containing the list with each string in upper case.

Assuming that the silo that `in` refers to is hosted on some remote node RN, materializing `out` (on the same node) requires sending a message representing this `map` invocation to RN. In the above example, the materialization of `out` would entail sending an instance of a class `Mapped[T, S]` where `T = S = List[String]` to RN.

While serializing the instance of `Mapped` with the concrete type `Mapped[List[String], List[String]]` is straightforward, the key question is: how to deserialize this on RN?

Importantly, the combinator code on the server-side is generic; it is not specific to any client code where types are made concrete.

[4] `https://github.com/heathermiller/f-p`

[5] `https://github.com/scala/pickling`

In the example, this means that the concrete type argument `List[String]` is only known at run time on RN. As a result, the remote node must interpret the run-time type on the fly in order to deserialize the instance of `Mapped`.

Such interpretation-based deserializers are know to be significantly slower than type-specific, statically-generated deserializers (Miller et al. 2013). In fact, a recent study has identified serialization as a major bottleneck in data-intensive applications (Ousterhout et al. 2015), one of our main target applications.

In order to minimize the cost of deserialization, we have devised a serialization scheme that enables utilizing statically-generated deserializers, thereby avoiding the performance penalty of interpretation-based deserializers.

The approach which we call "self-describing pickles" is as follows. Basically, the idea is to augment the serialized representation with additional information about how to deserialize ("unpickle") it. The key is to capture the type-specific pickler and unpickler when the fully-concrete type of a `Mapped` instance is known:

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

Essentially, this means when `doPickle` is called with a concrete type `T`, say:[6]

```
doPickle[Mapped[List[Int], 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 `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:

```
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 `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 `SelfDescribing` instance, and (b) that unpickling is as efficient as pickling, thanks to using type-specific unpicklers.

6. Examples

To show that the function passing model is able to serve as a substrate upon which to build different sorts of data-centric distributed frameworks, we have built two miniaturized example systems inspired by popular big data frameworks: Spark's RDD (Resilient Distributed Dataset) (Zaharia et al. 2012) and MBrace (Dzik et al. 2013). Using each example system, we have implemented several example applications, excerpts of some of which are discussed below.

Spark's RDDs provide a set of operations to execute parallel operations on distributed data. MBrace extends F#'s asynchronous workflows (Syme et al. 2011), a way to declare asynchronous tasks, to distribute computation in the cloud.

Our simplified RDD implementation enables the use of data structures distributed using silos (see Section 2.2). We have implemented some of the operations of Spark's RDD such as `map`, `reduce`, `groupBy`, and `join` in terms of the primitives of the function passing model. Methods on RDDs like `flatMap` or `filter` that do not need to communicate across silos are implemented using the SiloRef `map` combinator. On the other hand, `join` is implemented using the SiloRef `flatMap` combinator. Below, we use it to show a simple application associating the length of all words in two documents in a `Map` to a set of words with the corresponding length.

```
val content: RDD[String, List[String]] = ...
val lorem: RDD[String, List[String]] = ...

val contentWord = content.flatMap(line => {
  line.split(' ').toList
}).map(word => (word.length, word))

val loremWord = lorem.flatMap(line => {
  line.split(' ').toList
}).map(word => (word.length, word))

val res: Map[Int, Set[String]] =
  contentWord.join[Set, Map](loremWord).collectMap()
```

In this example, the RDD's `flatMap` method is called on the RDDs `content` and `lorem`, respectively. The closures passed as arguments in these calls split each line into a list of words; the `flatMap` calls then flatten the created collections of lists into single, flat RDDs. The flat RDDs (containing just words) are mapped to RDDs containing pairs of the form `(word.length, word)` where `word` is a string (a word) taken from a flat RDD. Then, we do an inner join, which associates each length to the set of words of the same length, removing duplicate words in the process. Finally, we collect the final result in a `Map` using the `collectMap` method of RDD. Several other more detailed example programs using Spark on the function passing model are available on GitHub.[7]

In the context of MBrace, we have also implemented k-means clustering (based on an example available on the

[6] Note that the type arguments are inferred by the Scala compiler; they are only shown for clarity.

[7] `https://github.com/heathermiller/f-p` (branch `onward2016`)

MBrace website[8]), an excerpt of which is shown below. Our implementation of distributed k-means clustering using the function passing model is an almost identical port of the MBrace version written in F#. K-means clustering is an algorithm to categorize data points across k different clusters. It starts with the centroids of the k clusters.

```
def kMeansIterate(
  partitionedPoints: Seq[SiloRef[Array[Point]]],
  centroids: Array[Point],
  iteration: Int): Array[Point] = {

  // Running iteration...
  val clusterParts = partitionedPoints.map(silo => {
    silo.map(spore {
      val lCentroids = centroids
      points => kmeansLocal(points, lCentroids)
    }).send()
  })

  val newCentroids =
    Await.result(Future.sequence(clusterParts).map(seq =>
      seq.reduce(_ ++ _)
         .groupBy(_._1)
         .toSeq
         .sortBy(_._1)
         .map(_._2)
         .map(clp => clp.map(_._2).toArray.unzip)
         .map({ case (ns, points) =>
           ns.sum -> sumPoints(points)
         })
         .map({ case (n, sum) => divPoint(sum, n) })
    ), 10.seconds).toArray

  val diff = newCentroids
    .zip(centroids)
    .map({ case (p1, p2) => dist(p1, p2) })
    .max

  // check if converged, else iterate again
  if (diff < epsilon) {
    newCentroids
  } else {
    kMeansIterate(partitionedPoints,
                  newCentroids,
                  iteration + 1)
  }
}
```

The algorithm proceeds in two steps. It first assigns data points to the closest cluster. Then, it assigns to each cluster a new centroid by computing the mean of the points assigned to the clusters. It stops when the centroids stop changing; if this convergence condition has not been met, the algorithm is called recursively with the updated set of centroids. In the distributed version of k-means, we start with a master node that partitions the points into silos. In each iteration, `map` is called on the SiloRef which results in a spore (function) being applied to the data within the corresponding silo. The spore captures the current iterations' centroids and uses them to compute the new cluster for its local set of points (using the `kmeansLocal` function). The results are then sent back to the master node to compute the new centroids, and to check the algorithm's convergence condition.

7. Related Work

Alice ML (Rossberg et al. 2004) 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 the function passing model has incorporated many similar ideas, such as type-safe, generic and platform-independent pickling. In Alice, functions intend to be mobile. Only those functions which capture (either directly or indirectly) local resources remain stationary. In the case of functions that must remain stationary, 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 the function passing model, however, functions are not wrapped in proxies but sent directly. Thus, calling a received function will not lead to remote invocations.

Cloud Haskell (Epstein et al. 2011) 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. Moreover, the coordination of concurrent activity is based on futures, instead of message passing. Closures and continuations in Termite Scheme (Germain et al. 2006) 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, the function passing model is statically typed, enabling advanced type-based optimizations. In non-process-oriented models, parallel closures (Matsakis 2012) and RiverTrail (Herhut et al. 2013) 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 the function passing model. In contrast to parallel closures, spores do not require a type system extension in Scala.

Acute ML (Sewell et al. 2007) is a dialect of ML which proposes numerous primitives for distributed programming, such as type-safe serialization, dynamic linking and rebinding, and versioning. The function passing model, in contrast,

[8] http://mbrace.io/starterkit/HandsOnTutorial.FSharp/examples/200-kmeans-clustering-example.html

is based on spores, which ship with their serialized environment or they fail to compile, obviating the need for dynamic rebinding. HashCaml (Billings et al. 2006) 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, function passing is merely a programming model, which does not require extensions to the Scala compiler.

ML5 (Murphy VII et al. 2007) 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 function passing 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 (Zaharia et al. 2012), MapReduce (Dean and Ghemawat 2008), and Dryad (Isard et al. 2007) are just that–distributed systems. The function passing model 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 (Hickey 2008)–stationary mutable data containers that users apply functions to in order to update an agent's state. The function passing model, 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 the function passing model is designed with remote references for a distributed scenario.

The function passing model is also related to the actor model of concurrency (Agha 1986), which features multiple implementations in Scala (Haller and Odersky 2009; Typesafe 2015; He et al. 2014). 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 function passing model.

8. Future Work and Conclusion

8.1 Ongoing and Future Work

In ongoing work we are exploring approaches for memory reclamation. The first approach uses Java's WeakReferences to detect when a SiloRef is no longer reachable from local GC roots. Upon detection the host of the corresponding silo is notified to decrease the silo's reference count; the host's reference(s) to the silo are nulled out when the reference count reaches zero. It is important to note that this strategy requires notifying a silo's host whenever a SiloRef to the silo reaches a new machine, to increase the silo's reference count. The second approach leverages uniqueness types in Scala (Haller and Odersky 2010; Haller and Loiko 2016). Here, SiloRefs are locally unique, and the programmer can explicitly declare a SiloRef as unused (or "consumed"); the type system ensures that such an "unused" SiloRef is not used again subsequently. As in the first approach, upon marking a SiloRef as unused, the corresponding silo's host is notified to decrease the silo's reference count.

Other future work includes better understanding concerns of separate compilation in order to evaluate whether our model could be of help in coordinating microservices.[9]

8.2 Conclusion

We have presented the function passing model, a new programming model and new substrate or middleware upon which to build data-centric distributed systems. This enables two important benefits for distributed system builders; since (a) all computations are functional transformations on immutable data, fault-tolerance is made simple by design, and (b) communication is made well-typed by design, the function passing model attempts to more naturally model the paradigm of data-centric programming by extending monadic programming to the network. 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 the function passing model 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, and have shown that it's possible to support different popular patterns of distributed processing, such as batch processing with Spark's RDDs and MBrace's cloud-based asynchronous tasks.

Acknowledgments

We would like to thank the anonymous reviewers for their thorough reviews and helpful suggestions which greatly improved the quality of the paper. We would like to thank Jorge Vicente-Cantero for improvements to spores and pickling which were not yet used in the presented prototype and case studies, but could be helpful to one day add. We would like

[9] 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 complex web-based services.

to thank Matei Zaharia for several helpful conversations in the development of this work, and Databricks for hosting Heather while part of this work was undertaken.

References

G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

Apache. Hadoop. http://hadoop.apache.org/, 2015.

Apache. *Spark*, 2016. URL http://http://spark.apache.org/.

J. Billings, P. Sewell, M. Shinwell, and R. Strniša. Type-safe distributed programming for OCaml. In *Proceedings of the 2006 workshop on ML*, pages 20–31, 2006.

R. Bose and J. Frew. Lineage retrieval for scientific data processing: A survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1 (4):379–474, 2009.

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

J. Dzik, N. Palladinos, K. Rontogiannis, E. Tsarpalis, and N. Vathis. MBrace: Cloud computing with monads. In *PLOS*, pages 7:1–7:6, 2013.

J. Epstein, A. P. Black, and S. L. P. Jones. Towards Haskell in the cloud. In *Haskell Symposium*, pages 118–129, 2011.

G. Germain, M. Feeley, and S. Monnier. Concurrency oriented programming in Termite Scheme. In *workshop on Scheme and Functional Programming*, 2006.

P. Haller and A. Loiko. LaCasa: Lightweight affinity and object capabilities in Scala. In *OOPSLA*, 2016. to appear.

P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410 (2):202–220, 2009.

P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, pages 354–378, 2010.

P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Futures and promises. *http://docs.scala-lang.org/overviews/core/futures.html*, 2012.

P. Haller, N. Müller, and H. Miller. The function passing model: Types, proofs, and semantics. Technical Report EPFL-REPORT-221395, EPFL, May 2016.

J. He, P. Wadler, and P. W. Trinder. Typecasting actors: From Akka to TAkka. In *SCALA@ECOOP*, pages 23–33, 2014.

S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: A path to parallelism in JavaScript. In *OOPSLA*, pages 729–744, 2013.

R. Hickey. The Clojure programming language. In *DLS*, 2008.

M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

N. D. Matsakis. Parallel closures: A new twist on an old idea. In *HotPar*, 2012.

H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *OOPSLA*, pages 183–202, 2013.

H. Miller, P. Haller, and M. Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP*, pages 308–333, 2014.

T. Murphy VII, K. Crary, and R. Harper. Type-safe distributed programming with ML5. In *TGC*, pages 108–123, 2007.

NICTA. Scoobi. https://github.com/nicta/scoobi, 2015.

K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307, 2015.

B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. Alice through the looking glass. *Trends in Functional Programming*, 5:79–96, 2004.

P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *J. Funct. Program*, 17(4-5):547–612, 2007.

D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *PADL*, pages 175–189, 2011.

Twitter. Scalding. https://github.com/twitter/scalding, 2015.

Typesafe. Akka. http://akka.io/, 2015.

J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. In *Mobile Object Systems*, pages 49–64, 1996.

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.