Distributed Programming via Safe Closure Passing

Philipp Haller[1] and Heather Miller[2]

[1] KTH Royal Institute of Technology, Sweden
phaller@kth.se
[2] EPFL, Switzerland
heather.miller@epfl.ch

Abstract

Programming systems incorporating aspects of functional programming, e.g., higher-order functions, are becoming increasingly popular for large-scale distributed programming. New frameworks such as Apache Spark leverage functional techniques to provide high-level, declarative APIs for in-memory data analytics, often outperforming traditional "big data" frameworks like Hadoop MapReduce. However, widely-used programming models remain rather ad-hoc; aspects such as implementation trade-offs, static typing, and semantics are not yet well-understood. We present a new asynchronous programming model that has at its core several principles facilitating functional processing of distributed data. The emphasis of our model is on simplicity, performance, and expressiveness. The primary means of communication is by passing functions (closures) to distributed, immutable data. To ensure safe and efficient distribution of closures, our model leverages both syntactic and type-based restrictions. We report on a prototype implementation in Scala. Finally, we present preliminary experimental results evaluating the performance impact of a static, type-based optimization of serialization.

1 Introduction

Programming systems for large-scala data processing are increasingly embracing functional programming, *i.e.,* programming with first-class functions and higher-order functions. Arguably, one of the first widely-used programming models for "big data" processing making use of concepts from functional programming is Google's MapReduce [3]. Indeed, [7] shows a precise executable semantics of MapReduce in Haskell. While leveraging functional programming *concepts*, popular implementations of the MapReduce model, such as Hadoop MapReduce[1] 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 [17], Twitter's Scalding[2], and Scoobi[3] build on functional language features in order to provide high-level, declarative APIs.

However, these programming systems suffer from several problems that negatively affect their usage, maintainance, and optimization:

- Their APIs cannot statically prevent *common usage errors*. As a result, users are often confronted with runtime errors that are hard to debug. A common example is unsafe closure serialization [12].
- Typically, only high-level user-facing abstractions are statically typed. The absence of static types in lower layers of the system makes *maintainance* tasks, such as code refactorings, more difficult.
- The absence of certain kinds of static type information precludes systems-centric *optimizations*. Importantly, type-based static meta-programming enables fast serialization [11], but this is only possible if also lower layers (namely those dealing with object serialization) are statically typed. Several studies [2, 8, 14, 16] report on the high overhead of serialization in widely-used runtime

[1] See http://hadoop.apache.org/
[2] See https://github.com/twitter/scalding/
[3] See http://nicta.github.io/scoobi/

environments such as the JVM. This overhead is so important in practice that popular systems, like Spark [17] and Akka [15], leverage alternative serialization frameworks such as Protocol Buffers (Google), Apache Avro [1], or Kryo [13].

Contributions This paper makes the following contributions:

- A new asynchronous programming model, called SCP ("safe closure passing"), for functional processing of distributed data (see Sec. 2). We propose to address the above problems through a novel combination of: (a) *safe closures;* to prevent common usage errors. Closures that are not guaranteed to be serializable are rejected at compile time; (b) *a statically-typed implementation of a generic distributed, persistent data structure.* Preserving static types through more system layers improves maintainability and enables type-based optimizations.
- An implementation of the SCP model in Scala (see Sec. 3). In addition, we report on preliminary experimental experience using SCP to evaluate the end-to-end performance impact of a type-based optimization of serialization.

An important goal of our model is to better understand programming systems such as Spark, Scalding, and Scoobi, by incorporating several of their core principles; namely, immutable distributed data and distributed closure passing. By focussing on simplicity, expressiveness, and performance (and ignoring many of the more ad-hoc refinements of the mentioned programming models) our programming model–together with its prototype implementation–enables exploring implementation trade-offs, and capturing the semantics of the core constructs more precisely.

To ensure safe and efficient distribution of closures, our model leverages both syntactic and type-based restrictions. For instance, closures sent to remote nodes are required to conform to the restrictions imposed by the so-called "spore" abstraction that the authors presented in previous work [12]. Among others, the syntax and static semantics of spores can guarantee the absence of runtime serialization errors due to closure environments that are not serializable.

The following sections 2 (programming model) and 3 (implementation) present our main contributions. Section 4 relates to prior work. Section 5 summarizes future work and concludes.

2 Programming Model

The programming model has a few basic abstractions at its center: first, the so-called *silo*. A silo is a typed data container. It is stationary in the sense that it does not move between machines. A silo 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 so-called *SiloRef*. Similar to a proxy object, a SiloRef represents, and allows interacting with, a silo possibly located on a remote node. Some programming patterns require combining data contained in silos located on different nodes (*e.g.,* joins). To support such patterns, our model includes a *pump* primitive for emitting data to silos on arbitrary nodes (explained further below).

A SiloRef has the following main operations:

```
trait SiloRef[T] {
  def apply(s: Spore[T, S]): SiloRef[S]
  def send(): Future[T]
}
```

Apply The `apply` method takes a spore, a kind of closure (see Appendix A for an overview), that is to be applied to the data in the silo of the receiver SiloRef. Rather than immediately sending the spore across the network, and waiting for the operation to finish, the apply method is *lazy*. It immediately returns a SiloRef that refers to the result silo.

To realize something like the `map` combinator of a (distributed) collection using `apply`, it is helpful to think of the spore argument to `apply` ("s") as the composition of a user-defined function passed to the `map` combinator with the actual implementation of `map`:

```
val ref: SiloRef[List[Int]] = ...
val userFun: Int => String = ...
val mapFun: (Int => String) => List[Int] => List[String] = ...
val ref2: SiloRef[List[String]] = ref.apply(mapFun(userFun))
```

In the above example, the higher-order `mapFun` function is expressed in curried style where the user's function argument is passed as the first argument. Applying `mapFun` to a function of type `Int => String` returns a function of type `List[Int] => List[String]`.

The result of invoking `apply` is another SiloRef, which has a reference to the spore and the SiloRef that it was derived from. Note that this is semantically the same as programming with normal functional data structures, where a new data structure is defined by a transformation of an original data structure.

Send The `send` method takes no argument, and returns a future. Unlike `apply`, `send` is *eager* (readers familiar with the concept of *views* might recognize a similarity to forcing a view). That is, it sends whatever operations are queued up (by invocations of `apply`) on a given SiloRef to the node that contains the corresponding silo, and kicks off the materialization of the result silo. Once the materialization is done, the future returned by `send` is completed. Example:

```
val ref2 = ref1.apply(s)  // lazy
val fut  = ref2.send()    // eager
```

The invocation of `send` kicks off the following sequence of actions:

1. A "send" control message is sent to the node where `ref2`'s silo is located.
2. Since `ref2` is derived from `ref1`, `ref1`'s silo is located on the same node. Thus, the runtime demands `ref1` to be materialized; once this is done, spore `s` is applied, populating `ref2`'s silo (on the same node).
3. Once `ref2`'s silo is materialized, its data is sent to the master node, completing the `fut` future.

Note that since a `send` operation sends the data of a silo to the master node, it should only be invoked on silos containing small bits of data.

pumpTo The SiloRef singleton object provides an additional method for combining silos storing collections:

```
def pumpTo[T <: Traversable[U], V, R](
      p: Place,
      silo1: SiloRef[T],
      silo2: SiloRef[T],
      fun: Spore[(U, Emitter[V]), Unit],
      bf: BuilderFactory[V, R]): SiloRef[R]
```

The `pumpTo` method requires the silos `silo1` and `silo2` to contain collections of element type `U` (`Traversable[U]`). Using `pumpTo`, the elements (of type `U`) of the two silos are passed one-by-one to the user-provided spore `fun`. This spore takes a pair as argument containing two components: first, a single element of one of the silo's collections, and second, an *emitter* to which the spore can output values of type `V`. By emitting such elements, a new silo at the destination (`Place p`) is filled, yielding a collection of type `R`. A `BuilderFactory[V, R]` provides the functionality for building a collection of type `R` based on elements of type `V`. Finally, `pumpTo` returns the SiloRef of the silo that was created at `Place p`.

Although conceptually related, the `Emitter` and the `BuilderFactory` address two separate issues: the `Emitter` provides a way to output elements from the source silo; the `BuilderFactory` provides a way to input data into a newly created silo at the destination. Both abstractions are explained in more detail below.

An `Emitter` is a simple trait which allows the spore parameter `fun` of `pumpTo` to emit zero, one, or multiple values per element of a silo's collection, using an `emit` function:

```
def emit(v: T)(implicit p: Pickler[T]): Unit
```

Note that `Emitter` differs from the well-known *observable* abstraction [10] in one important way: the emit method requires an implicit type-specific *pickler*. In Scala, type-specialized picklers enable fast serialization through compile-time meta-programming [11]. Thus, `Emitter` is an abstraction specially designed for distributed programming. The main reason why the pickler is required already at this point is that we would like to enable picklers to be *specialized* to the type of the pickled values. However, this means a pickler has to be constructed at the point when the static type of the emitted value is still available (essentially, before the value loses its type when treated as generic data to be sent across the network).

2.1 Combining Multiple Silos

Operations on distributed collections such as *union*, *groupByKey*, or *join*, involve multiple data sets, possibly located on different nodes. In the following we explain how such operations can be expressed using the introduced primitives.

union The union of two unordered collections stored in two different silos can be expressed directly using the above `pumpTo` primitive.

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 for elements of type `A` and `B`, respectively:

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

The goal is to compute the hash-join of `silo1` and `silo2`:

```
val hashJoin: SiloRef[List[(K, (A, B))]] = ???
```

To be able to use `pumpTo`, the types of the two silos first have to be made equal, through initial `apply` invocations:

```
val silo12: SiloRef[List[(K, Option[A], Option[B])] =
      silo1.apply { x => (hashA(x), Some(x), None) }
val silo22: SiloRef[List[(K, Option[A], Option[B])] =
      silo2.apply { x => (hashB(x), None, Some(x)) }
```

Then, we can use `pumpTo` to create a new silo (at some destination place), which contains the elements of both `silo12` and `silo22`:

```
val combined = SiloRef.pumpTo(destPlace, silo12, silo22,
                              (elem, emitter) => emitter.emit(elem),
                              listBuilderFactory[...])
```

The combined silo contains triples of type `(K, Option[A], Option[B])`. Using an additional `apply`, the collection can be sorted by key, and adjacent triples be combined, yielding finally 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 nodes. These destination nodes 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 nodes, with nodes N_1, N_2, etc. We assume each silo contains an unordered collection of key-value pairs (a multi-map). Then, *groupByKey* can be implemented as follows:

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

3 Implementation and Preliminary Experimental Results

We have developed a prototype[4] of the SCP model in Scala, which builds on our earlier work on Scala Pickling [11] and Spores [12]. The implementation does not require extensions to the Scala language or compiler; it is developed using the current stable Scala release 2.11.

We have used our implementation to measure the impact of compile-time-generated serializers [11] on end-to-end application performance. In our benchmark application, a group of 4 silos is distributed across 4 different nodes/JVMs. The silos are first transformed using *map*, and then using *groupBy*. For an input size of 100'000 "person" records, the use of compile-time-generated serializers resulted in an overall speedup of about 48% (experiment run 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).

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

4 Related Work

Cloud Haskell [4] 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. Closures and continuations in Termite Scheme [5] 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, SCP is statically typed, enabling advanced type-based optimizations. In non-process-oriented models, parallel closures [9] and RiverTrail [6] address important safety issues. SCP integrates a distributed, persistent data structure. Other prior work related to spores is discussed in [12].

5 Conclusion and Future Work

We have presented a new asynchronous distributed programming model. A novel combination of (a) closures with syntactic and semantic restrictions and (b) abstractions for distributed data "silos" prevents usage errors common in widely-used "big data" frameworks. We have implemented our model in Scala; preliminary experimental results evaluate the performance impact of a static type-based optimization. In future work, we intend to expand the practical experiments and explore the impact of implementation trade-offs. Furthermore, we would like to exploit the simplicity of the programming model for a formal treatment of its properties.

References

[1] Apache. Avro®. `http://avro.apache.org`. Accessed: 2013-08-11.

[2] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *Java Grande*, pages 66–71, 1999.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the cloud. In *Proc. Haskell Symposium*, pages 118–129. ACM, 2011.

[5] Guillaume Germain. Concurrency oriented programming in Termite Scheme. In *Erlang Workshop*, 2006.

[6] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. River trail: a path to parallelism in JavaScript. In *OOPSLA*, pages 729–744, 2013.

[7] Ralf Lämmel. Google's mapreduce programming model - revisited. *Sci. Comput. Program*, 70(1):1–30, 2008.

[8] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java's remote method invocation. In *PPOPP*, pages 173–182, August 1999.

[9] Nicholas D. Matsakis. Parallel closures: a new twist on an old idea. In *HotPar*. USENIX, 2012.

[10] Erik Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[11] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *OOPSLA*. ACM, 2013.

[12] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP*, pages 308–333. Springer, 2014.

[13] Nathan Sweet et al. Kryo. `https://code.google.com/p/kryo/`. Accessed: 2013-08-11.

[14] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency - Practice and Experience*, 12(7):495–518, 2000.

[15] Typesafe. Akka. `http://akka.io/`, 2009. Accessed: 2013-08-11.

[16] Matt Welsh and David E. Culler. Jaguar: enabling efficient communication and I/O in Java. *Concurrency - Practice and Experience*, 12(7), 2000.

[17] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2012.

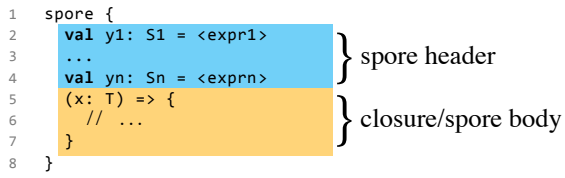

Figure 1: The syntactic shape of a spore.

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

(a) A closure block.

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

(b) A spore.

Figure 2: The evaluation semantics of a spore is equivalent to that of a closure, obtained by simply leaving out the spore marker.

A Spores

Spores are a closure-like abstraction and type system which aims to give users a principled way of controlling the environment which a closure 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 crucial insight of spores is that, by including type information of captured variables in the type of a spore, type-based constraints for captured variables can be composed and checked, making spores safer to use in a concurrent, distributed, or in arbitrary settings where closures must be controlled.

A.1 Spore Syntax

A spore is a closure with a specific shape that dictates how the environment of a spore is declared. The shape of a spore is shown in Figure 1. 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.

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 (public, global state). In particular, the spore closure is not allowed to capture variables in the environment. Only an expression on the right-hand side of a value definition in the spore header is allowed to capture variables.

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, it's no longer possible to accidentally capture the `this` reference.

```
1  trait Function1[-A, +B] {
2    def apply(x: A):  B
3  }
```

(a) Scala's arity-1 function type.

```
1  trait Spore[-A, +B]
2  extends Function1[A, B] {
3    type Captured
4    type Excluded
5  }
```

(b) The arity-1 `Spore` type.

Figure 3: The `Spore` type.

A.1.1 Evaluation Semantics

The evaluation semantics of a spore is equivalent to a closure obtained by leaving out the `spore` marker, as shown in Figure 2. In Scala, the block shown in Figure 2a first initializes all value definitions in order and then evaluates to a closure that captures the introduced local variables `y1, ..., yn`. The corresponding spore, shown in Figure 2b has the exact same evaluation semantics. Interestingly, this closure shape is already used in production systems such as Spark in an effort to avoid problems with accidentally captured references, such as `this`. However, in systems like Spark, the above shape is merely a convention that is not enforced.

A.2 The `Spore` Type

Figure 3 shows Scala's arity-1 function type and the arity-1 spore type.[5] Functions are contravariant in their argument type `A` (indicated using −) and covariant in their result type `B` (indicated using +). The `apply` method of `Function1` is abstract; a concrete implementation applies the body of the function that is being defined to the parameter `x`.

Individual spores have *refinement types* of the base `Spore` type, which, to be compatible with normal Scala functions, is itself a subtype of `Function1`. Like functions, spores are contravariant in their argument type `A`, and covariant in their result type `B`. Unlike a normal function, however, the `Spore` type additionally contains information about *captured* and *excluded* types. This information is represented as (potentially abstract) `Captured` and `Excluded` type members. In a concrete spore, the `Captured` type is defined to be a tuple with the types of all captured variables. [12] discusses the `Excluded` type member in detail.

A.3 Basic Usage

A.3.1 Definition

A spore can be defined as shown in Figure 4a, with its corresponding type shown in Figure 4b. As can be seen, the types of the environment listed in the spore header are represented by the `Captured` type member in the spore's type.

A.3.2 Using Spores in APIs

Consider the following method definition:

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

[5] For simplicity, we omit `Function1`'s definitions of the `andThen` and `compose` methods.

```
1  val s = spore {
2    val y1: String = expr1;
3    val y2: Int = expr2;
4    (x: Int) => y1 + y2 + x
5  }
```

(a) A spore `s` which captures a `String` and an `Int` in its spore header.

```
1  Spore[Int, String] {
2    type Captured = (String, Int)
3  }
```

(b) `s`'s corresponding type.

Figure 4: An example of the `Captured` type member.
Note: we omit the `Excluded` type member for simplicity; we discuss it in detail in [12].

In this example, the `Captured` (and `Excluded`) type member is not specified, meaning it is left abstract. In this case, so long as the spore's parameter and result types match, a spore type is always compatible, regardless of which types are captured.

Using spores in this way enables libraries to enforce the use of spores instead of plain closures, thereby reducing the risk for common programming errors, even in this very simple form.

A.3.3 Composition

Like normal functions, spores can be composed. By representing the environment of spores using refinement types, it is possible to preserve the captured type information (and later, constraints) of spores when they are composed.

For example, assume we are given two spores `s1` and `s2` with types:

```
s1: Spore[Int, String] { type Captured = (String, Int) }
s2: Spore[String, Int] { type Captured = Nothing }
```

The fact that the `Captured` type in `s2` is defined to be `Nothing` means that the spore does not capture anything (`Nothing` is Scala's bottom type). The composition of `s1` and `s2`, written `s1 compose s2`, would therefore have the following refinement type:

```
Spore[String, String] { type Captured = (String, Int) }
```

Note that the `Captured` type member of the result spore is equal to the `Captured` type of `s1`, since it is guaranteed that the result spore does not capture more than what `s1` already captures. Thus, not only are spores composable, but so are their (refinement) types.