JFP **28**, e7, 48 pages, 2018. © Cambridge University Press 2018
doi:10.1017/S0956796818000035

A programming model and foundation for lineage-based distributed computation

PHILIPP HALLER
School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden
(*e-mail:* phaller@kth.se)

HEATHER MILLER
School of Computer and Communication Sciences, EPFL, CH-1015 Lausanne, Switzerland
and
College of Computer and Information Science, Northeastern University, Boston, MA-02115, USA
(*e-mail:* heather.miller@epfl.ch)

NORMEN MÜLLER
Safeplace, DE-40667 Meerbusch, Germany

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, or middleware, upon which to build data-centric distributed systems like Spark. 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 performed on distributed data are stored in its nodes rather than the distributed data itself. One advantage of this model is that failure recovery is simplified by design – data can be 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. Moreover, we provide a complete formalization of the programming model in order to study the foundations of lineage-based distributed computation. In particular, we develop a theory of safe, mobile lineages based on a subject reduction theorem for a typed core language. Furthermore, we formalize a progress theorem that guarantees the finite materialization of remote, lineage-based data. Thus, the formal model may serve as a basis for further developments of the theory of data-centric distributed programming, including aspects such as fault tolerance. 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 this model.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

1 Introduction

Data-centric programming is growing in importance with the most successful systems for programming with "big data" all adopting ideas from functional programming; *i.e.,* programming with first-class functions. These functional ideas are often touted to be the key to the success of these frameworks. Functional, declarative interfaces to data, distributed over tens to thousands of nodes, provide a more natural way for end-users and data scientists to reason about data.

While leveraging functional programming *concepts*, popular implementations of Google's MapReduce (Dean & Ghemawat, 2008) model, such as Apache Hadoop's MapReduce Framework (Apache, 2015) for Java, have been developed without making use of functional language *features* such as closures. For nearly a decade, the Apache Hadoop open source interpretation of this model grew in popularity, remaining largely unchallenged–causing nearly all of industry to synchronize on this one implementation for nearly all 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 (Odersky *et al.*, 2010); such systems include Apache Spark (Zaharia *et al.*, 2010), 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. Further, the benefits provided by functional programming have also won over framework designers as well – some have noticed that immutability, and data transformation via higher-order functions, makes it much easier, by design, to tackle concerns central to distributed systems such as concurrency.

While widely adopted in practice, the aforementioned programming systems are not without important issues. On the one hand, their programming interfaces do not prevent common usage errors, such as unsafe closure serialization. As a result, the complexities of distribution may trickle even to end users, who are increasingly non-expert users. On the other hand, the foundations of their programming models remain largely unclear, in particular, foundations of core aspects such as fault tolerance, a critical aspect for distributed operation on a large scale.

This paper introduces a new programming model that embraces the principle of stationary data containers and mobile functions (move computation to the data). It can be viewed as a generalization of the MapReduce/Spark programming model. Our programming model, which we call *function passing* can be thought of as a programming model for a middleware, meant to underly systems like Spark. Function passing adopts the concept of *lineage* which is used by systems like Spark to handle fault tolerance. Importantly, lineage-based fault tolerance is facilitated by the core computational principle of *functional* transformations on immutable data.

The programming model is based on functional abstractions for lineage-based distributed computation. In order to prevent common usage errors, the model builds upon two previous veins of work – type-safe serialization based on functional pickler combinators (Kennedy, 2004; Rossberg *et al.*, 2004; Elsman, 2005; Miller *et al.*, 2013), and serializable closures (Epstein *et al.*, 2011; Miller *et al.*, 2014). We believe this unique combination of functional programming techniques

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

provides a more principled substrate upon which to build data-centric, distributed systems.

Importantly, we provide a complete formalization of the programming model. In particular, we develop a theory of safe, mobile lineages based on a subject reduction theorem for a typed core language. Furthermore, we formalize a progress theorem that guarantees the finite materialization of remote, lineage-based data. To our knowledge, these theorems constitute the first correctness results for a programming model for lineage-based distributed computation. Thus, our formal model may serve as a basis for further developments of the theory of data-centric distributed programming, including aspects such as fault tolerance.

1.1 Contributions

This paper makes the following contributions:

- A new data-centric programming model for functional processing of distributed data, which provides abstractions for building fault-tolerant distributed systems, including first-class *lineages*. The main computational principle is based on the idea of sending safe, guaranteed serializable functions to stationary data containers. Using standard monadic operations, our model enables creating directed acyclic graphs (DAG) of computations. Deferred evaluation enables optimizations such as operation fusion while keeping programs simple to reason about.
- A formalization of lineage-based distributed computation based on a small-step operational semantics. Our formalization extends previous theories of serializable closures to *serializable lineages*. The technical development enabling this extension combines (a) serializable types, (b) "static" closures, and (c) lineages.
- A proof of a subject reduction theorem for a typed, distributed core language based on lineages. To our knowledge, we present the first such proof for a lineage-based distributed programming model.
- A proof establishing the preservation of lineage mobility by reduction for a typed, distributed core language. This property provides a foundation for lineage-based fault tolerance.
- The formalization of a progress theorem, guaranteeing the finite materialization of remote, lineage-based data, including a detailed proof sketch.
- A distributed implementation of the programming model in and for Scala as a middleware.[1] In addition, we present prototype versions of programming abstractions provided by popular frameworks like Apache Spark and MBrace using the function passing model, and end-user applications we have built using these prototypes.

In the rest of the paper, our approach is as follows. First, we describe our model on a high level, elaborating upon key benefits and trade-offs, and then we zoom in to

[1] See the Git repository at `https://github.com/heathermiller/f-p/`, branch `jfp`.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

make each component part of our model more precise. We describe the basic model this way in Section 2. In Section 3, we go on to show how essential higher-order operations on distributed frameworks like Apache Spark can be implemented in terms of the primitives presented in Section 2. We formalize our programming model in Section 4, providing an operational semantics and a type system. In Section 5, we present the proof of a subject reduction theorem and formalize important progress properties. Finally, we discuss related work in Section 6, and conclude in Section 7.

2 Overview

2.1 Essence

The function passing model is intended to act as a middleware upon which to build up data-centric distributed systems like Spark.

In the broadest sense, it can be thought of as a sort of persistent functional data structure with monadic operations and structural sharing. However, ***rather than containing pure data, instead this data structure represents a graph***[2] ***of functional transformations, or operations, on distributed data.*** The root node contains immutable data read from stable storage (*e.g.,* Amazon S3); edges represent functional transformations. Said another way, the core of the function passing model can be thought of as a persistent functional data structure representing a history of the operations performed on some data, rather than the data itself.

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. Subgraphs of the DAG are called *lineages*; lineages enable restoring the data of failed nodes through re-applying their transformations. 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, local lineage. To make a network call and thus obtain the result of a computation, it is necessary to first "kick off" the computation in order to materialize the nodes of its lineage. Within our programming model, this force operation[3] 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). For these reasons, we believe that deferred evaluation should be viewed as an enabler in the design of distributed systems.

[2] A directed acyclic graph (DAG).

[3] Called `send()`, discussed in more depth in Section 2.4 along with the other primitive operations in the function passing model.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

2.2 Basic usage

We begin with a simple visual example to illustrate the intuition behind the function passing model. The function passing model consists of three main components:

- **Silos**, stationary, typed, immutable data containers.
- **Silo references** to local or remote silos.
- **Spores**, safe, and serializable functions.

The main handle users have to the framework is via `SiloRef`s. A `SiloRef[T]` can be thought of as an immutable handle to a remote value of type `T` contained within a corresponding silo. Users interact with this remote data by applying functions (as spores) to silo references. Those functions 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 remote data via a `SiloRef[T]`, a new `SiloRef[T']`, representing a new silo containing the transformed data `T'`, is returned.

We go into more detail about each of these components later in Section 2.3.

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]` that points to a piece of remote data contained within a `Silo[T]`. When the function, shown as λ, of type $T \Rightarrow$ `SiloRef[`S`]` is applied to `SiloRef[T]` and "forced" (sent over the network), a new silo reference of type `SiloRef[S]` is immediately returned. Note that `SiloRef[S]` contains a reference to its parent silo reference, `SiloRef[T]`. (This is how *lineages* are constructed.) Meanwhile, the function is asynchronously sent over the network and is then applied to `Silo[T]`, eventually producing a new `Silo[S]`[4] containing the data transformed by function λ. This new `SiloRef[S]` can be operated on 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 silo references and their semantics. We describe these in the following.

2.3 Programming model

With a basic intuition under our belt for how distributed computation is performed in the function passing model, we focus now on its three main components; **silos**, **silo references**, and **spores**.

Silos. A silo is a typed and immutable data container. The container 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 may either be loaded from stable storage,

[4] New silos are materialized on the same node as their parent.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

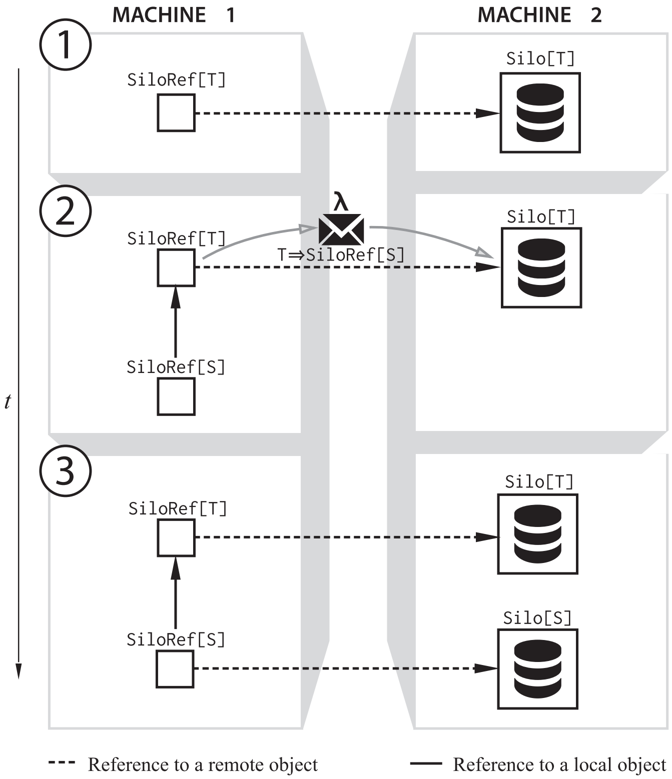
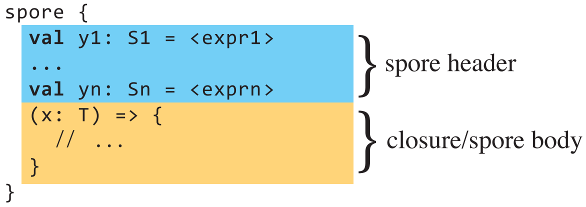


Fig. 1. Basic function passing model.

such as a distributed file system, or it may be the result of a computation. Thus, the *data* stored in a silo is often not stationary (in contrast to the containers, the silos, which are stationary), because it may need to be transferred to other machines to compute the contents of other silos. A program operating on data stored in a silo can only do so using a reference to the silo.

Silo references. Similar to a proxy object, a silo reference represents, and allows interacting with both local and remote silos. Silo references are immutable, storing identifiers to locate possibly remote silos. They are also typed (`SiloRef[T]`) corresponding to the type `T` of their silo's data, leading to well-typed network communication. A silo reference provides two principal operations: `apply` and `send`. The `apply` method makes use of deferred evaluation; it *eventually* applies a user-defined function to data pointed to by the `SiloRef[T]`, creating 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. This makes it possible to queue up transformations in order to optimize network communication. Note that the user-defined function passed to `apply` returns a `SiloRef[S]` whose contents is transferred to the new silo returned by `apply`. Essentially, `apply` enables accessing the contents of (local or remote) silos from within remote computations. We illustrate these primitives in more detail in Section 2.4.



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

Fig. 2. The shape of a spore.

Spores. Spores (Miller *et al.*, 2014) are safe closures that are guaranteed to be serializable and thus distributable. Essentially, a spore is a closure-like abstraction with associated type rules that 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 that 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, and the **spore body**, a regular closure; sometimes referred to as the "spore closure." This shape is illustrated in Figure 2.

The characteristic property of a spore is that the spore body is only allowed to access its parameter, the values in the spore header, as well as top-level singleton objects (Scala's form of modules). The spore closure is not allowed to capture variables other than those declared in the spore header (*i.e.*, a spore may not capture variables in the environment). By enforcing this shape, the environment of a spore is always declared explicitly in the spore header, which avoids accidentally capturing problematic references. Moreover, importantly for object-oriented languages like Scala, it is 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 may enable this type-checking by simply including information about excluded types (or other type-based properties) in the signature of a method. A concrete example would be to ensure that the `map` method on `RDD`s in Apache Spark (a distributed collection) accepts only spores that do not capture `SparkContext` (a non-serializable internal framework class).

2.4 Primitives

There are five basic primitive operations on silo references 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 introduce these primitives in the context of a running example. These primitives include `apply`, `send`, `persist`, `unpersist`, and `populate`;

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

```
trait SiloRef[T] {
  def apply[S](p: Spore[T, SiloRef[S]]): SiloRef[S]
  def send(): Future[T]
  def persist(): SiloRef[T]
  def unpersist(): SiloRef[T]
}
object SiloRef {
  def populate[T](host: Host, value: T): SiloRef[T]
}
```

Fig. 3. Type signatures of primitive operations.

their type signatures are shown in Figure 3.[5] Note that `populate` does not operate on a `SiloRef`, unlike the other primitives. Instead, `populate` is used to create new `SiloRefs`; therefore, it is defined as a (factory) method in the `SiloRef` *singleton object*.[6]

apply. `def apply[S](p: Spore[T, SiloRef[S]]): SiloRef[S]`
The `apply` method takes a spore that is to be applied to the data in the silo associated with the given `SiloRef` (*i.e.,* the receiver of the method call). Rather than immediately sending the spore across the network, and waiting for the operation to finish, the `apply` method's evaluation is *deferred*. Without involving any network communication, it immediately returns a `SiloRef` referring to a new, to-be-created silo. This new silo reference only contains lineage information, namely, a reference to the original `SiloRef` and a reference to the argument spore. As we explain below, another method, `send`, must be called explicitly to force the materialization of the result silo. Note that the result type of the spore parameter is a `SiloRef[S]`. Semantically, the new silo created by `apply` is defined to contain the data of the silo that the user-defined spore returns. This way, the `apply` combinator enables expressing computation DAGs.

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 invocation of `apply` defines a (not-yet-materialized) silo containing only the records of adults (graphically shown in Figure 5, part 1):

```
val persons: SiloRef[List[Person]] = ...
val adults: SiloRef[List[Person]] = persons.apply(spore { ps =>
  SiloRef.populate(currentHost, ps.filter(p => p.age >= 18))
})
```

[5] A trait in Scala can be thought of as an abstract class supporting mixin composition, essentially providing a safe form of multiple inheritance (Odersky & Zenger, 2005).

[6] Singleton objects are Scala's form of modules. The name of a singleton object refers to a value. Since in Scala the namespace for types is disjoint from the namespace for values, the trait and the singleton object can have the same name.

```
1  val vehicles: SiloRef[List[Vehicle]] = ...
2  val owners: SiloRef[List[(Person, Vehicle)]] =  // adults that own a vehicle
3    adults.apply(spore {
4      val localVehicles = vehicles  // spore header
5      (persons: List[Person]) =>
6        localVehicles.apply(spore {
7          val localPersons = persons  // spore header
8          (vs: List[Vehicle]) =>
9            SiloRef.populate(currentHost,
10             localPersons.flatMap(p =>
11               // list of (p, v) for a single person p
12               vs.flatMap(v =>
13                 if (v.owner.name == p.name) List((p, v))
14                 else Nil
15               )
16             )
17           )
18       })
19   })
```

Fig. 4. Matching persons and vehicle owners using the `apply` combinator.

When the spore is received on the machine hosting the silo corresponding to the `persons` silo reference, it is applied to the contents of the silo, a `List[Person]`. The body of the spore (a) filters this list (`ps`), creating a new list of adults, and (b) populates a new silo with the result of the `filter` invocation. Note that `populate` takes the host of the new silo as an argument; in the above example, this argument is `currentHost` that returns the host currently computing the silo's contents.

Section 2.4.1 below describes additional ways to create new silos. However, in each case, the host on which the silo should be created must be specified. The reason is that each instance of `SiloRef[T]` contains the host of the corresponding silo; this information is necessary for accessing the silo's data. Given the fact that the host of a `SiloRef` is fixed, `SiloRefs` are not suitable for direct use by applications requiring fault tolerance. Instead, `SiloRefs` should be regarded as a (low-level) building block for distributed systems, which implement fault-recovery mechanisms on top of the basic functionality provided by `SiloRefs`.

The `apply` combinator enables expressing also 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 example shown in Figure 4 `apply` is used to create a silo of `(Person, Vehicle)` pairs where the names of person and vehicle owner match. Here, it is necessary to read the data of the `vehicles` silo in addition to the `persons`, the list of `Person` records. This requires calling `apply` on `localVehicles` on line 6, whose argument spore captures the `persons` list and takes the `vs` list as a parameter; thereby, the two lists can be combined, and the result stored in a new silo (line 9). Note that with the use of `apply` on line 3, the call to `localVehicles.apply(..)` on

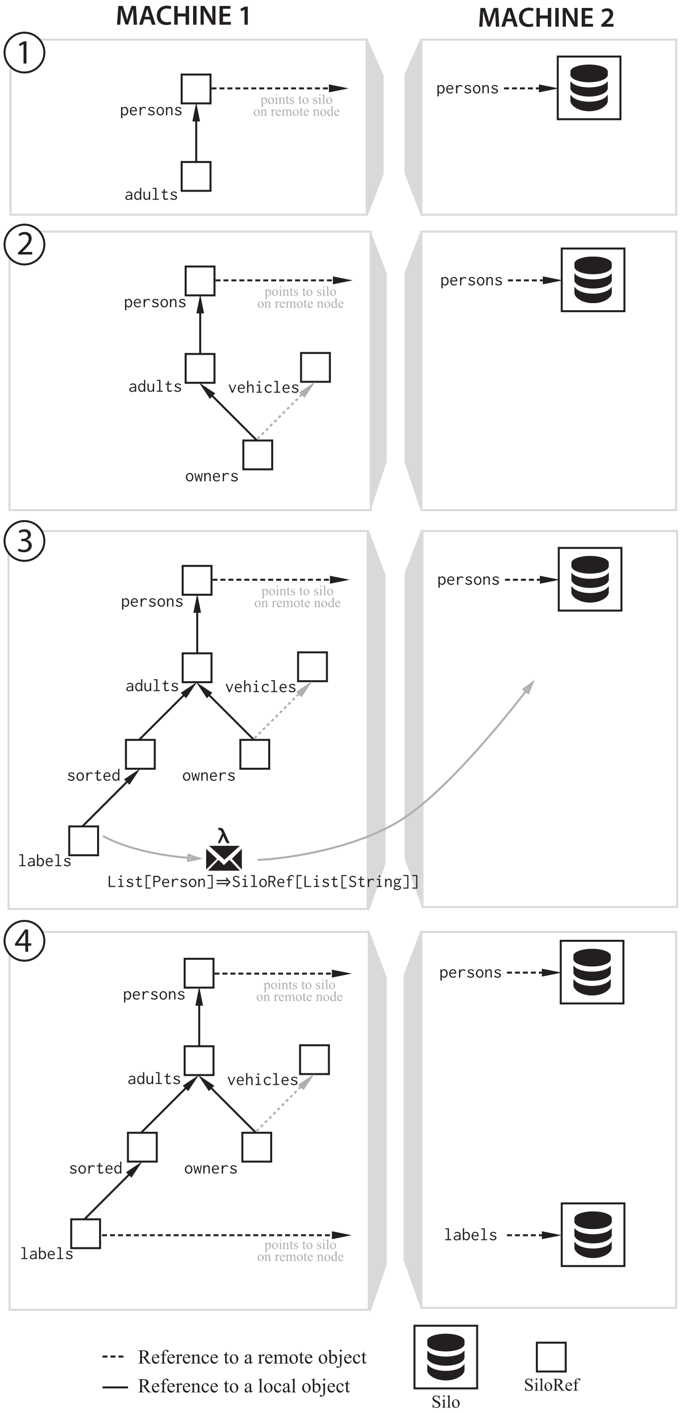

MACHINE 1
MACHINE 2
1
2
3
4
persons
adults
vehicles
owners
sorted
labels
points to silo on remote node
List[Person]⇒SiloRef[List[String]]
λ
Reference to a remote object
Reference to a local object
Silo
SiloRef

Fig. 5. A simple DAG in the function passing model.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

line 6 creates the final result silo, whose data is then also contained in the `owners` silo declared on line 2. (See Appendix A.1 for a diagram illustrating also the use of regular Scala collection combinators in the listing.)

To illustrate the data flow between hosts, let us kick off the materialization of the involved silos:

```
val adults: SiloRef[List[Person]] = ...
val vehicles: SiloRef[List[Vehicle]] = ...
val owners: Future[List[(Person,Vehicle)]] =
  adults.apply(...).send()
```

For illustration, we use the informal notation `@m` to denote the location of a value. We assume that the silo references are at machine `m1` but the actual data is distributed over `m2` and `m3` (note that these locations are different from Figure 5, part 2; however, they help make the required data transfers more precise):

```
adults   @ m1 --> Silo[List[Person]]   @ m2
vehicles @ m1 --> Silo[List[Vehicle]]  @ m3
```

To create `owners`, we must combine data hosted at `m2` with data hosted at `m3`. First, the invocation of `apply` on `adults` (line 3) transfers its spore, *i.e.,* the silo reference `vehicles` and the closure `(persons: List[Person]) => ...`, to `m2` hosting the referenced silo of grown-up `Person` records. Next, the invocation of `apply` on `localVehicles` (line 6) transfers its spore, *i.e.,* the collection of adults `persons` and the closure `(vs: List[Vehicle]) => ...`, to `m3` hosting the referenced silo of `Vehicle` records. Now, at `m3`, we have all required information: the adults `persons`, the vehicles `vehicles`, and the necessary computations to combine corresponding records, resulting in a new (anonymous) silo reference at `m2` referencing the new silo of `(Person,Vehicle)` records hosted at `m3`. This new silo reference at `m2` is used to transfer its referenced data from `m3` to `m2`, the origin of the `apply`, and, eventually, to `m1` where the materialization has been kicked off.

To reduce the amount of data that is transferred, the implementation in fact leverages *silo reference proxies*, delegating to the actual data:

```
adults   @ m1                --> Silo[List[Person]]            @ m2
vehicles @ m1                --> Silo[List[Vehicle]]           @ m3
owners   @ m1 --> proxy @ m2 --> Silo[List[(Person,Vehicle)]]  @ m3
```

The anonymous silo reference created at `m2` will not lead to a materialization of its contents at `m2`; instead, the silo reference functions as a proxy for the data hosted at `m3`. As a result, when `m1` requests the actual list of persons owning a vehicle, it is retrieved from `m3`. Avoiding a materialization also on `m2` alleviates network communication overhead.

Finally, note that the spore passed to `apply` on line 3 in Figure 4 declares the capturing of the `vehicles` silo reference in its 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.

send. `def send(): Future[T]`
As mentioned earlier, the execution of computations built using silo references is deferred. The `send` operation *forces* the deferred computation defined by the given `SiloRef[T]`. 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 on the same node as its parent, and will send its resulting 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).

persist. `def persist(): SiloRef[T]`
The performance of typical data analytics jobs can be increased dramatically by caching large datasets in memory (Zaharia *et al.*, 2010). To do this, silos containing computed datasets need to be materialized, and furthermore, re-materialization should be avoided when datasets are used multiple times.

The only way, that we have shown so far, to materialize a silo is using the `send` primitive. However, using `send` does not prevent the system from evicting the silo from memory at a later point in time, *e.g.,* when its host is running low on memory. In addition, `send` transfers the contents of the silo to the requesting host – too much if a large remote dataset should merely be cached in memory remotely.

Therefore, we provide an additional primitive called `persist` which immediately returns a `SiloRef` representing a silo which (a) has the same data when materialized, and (b) is guaranteed to remain in memory as long as at least one host has persisted the `SiloRef`.

Given the running example so far, we can add another lineage branching off of `adults` by sorting the list of `Person` records by age, and producing a greeting `String` for each record:

```
val sorted = adults.apply(spore { ps =>
  SiloRef.populate(currentHost, ps.sortWith(p => p.age))
})
val labels = sorted.apply(spore { ps =>
  SiloRef.populate(currentHost, ps.map(p => "Hi " + p.name))
})
val cachedLabels = labels.persist()
val done: Future[Boolean] = cachedLabels.map(x => true).send()
```

The `cachedLabels` silo contains the same (logical) data as the `labels` silo, however, the data of `cachedLabels` is cached in memory. Thus, `cachedLabels` needs to be materialized just once. In order to “kick off” remote computation, it is still necessary to call `send`; however, it is always possible to “prefetch” data into remote memory by mapping the data to a trivial value (`true` above) and invoking `send` to materialize the lineage.[7] The lineage for the above example looks as illustrated in Figure 5.

unpersist. `def unpersist(): SiloRef[T]`
The final primitive operation on silo references is `unpersist`. By invoking `unpersist` on a silo reference `r` the current host declares that it is no longer interested in the data of `r`. As a result, unless other hosts have persisted `r`, the memory occupied by the data of `r` may be reclaimed without negatively impacting performance. Just like `persist`, invoking `unpersist` immediately returns a `SiloRef` representing a silo which may not be cached in memory. However, if another host has persisted `r`, the silo's data remains cached in memory. In the context of future work, we plan to investigate techniques for inferring invocations of `unpersist` in order to automatically and efficiently manage silo memory (see Section 7).

2.4.1 Creating silos

As mentioned earlier, besides a type definition for `SiloRef[T]`, our framework also provides a companion singleton object. The singleton object provides a variety of factory methods for obtaining silo references referring to silos populated with some initial data:

```
object SiloRef {
  def populate[T](host: Host, value: T): SiloRef[T] = ...
  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[T]` 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 silo reference `s`. Note that only the silo reference is necessary for this operation to successfully complete; the silo is not required to be materialized.

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 (`Silo[T]`). Importantly, silos may not only store collections; silos are polymorphic 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

[7] Our system also provides a `cache` operation which abbreviates this pattern: semantically, `ref.cache()` is equivalent to `ref.persist().map(x => ()).send()`.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

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 Apache Spark, for example).

2.5 Limitations

Although fairly small and simple, the introduced programming model is quite flexible, and we have used it to implement a variety of examples and abstractions from the literature (see Section 3). However, as presented, the function passing model also has important limitations. Two principal limitations pertain to *distributed data* and *distributed behavior:*

- Data maintained in silos is *immutable*. As a result, computational patterns based on mutating distributed state cannot be expressed in the model. An example would be (potentially long-lived) distributed graph data that is asynchronously updated. Another example would be mutable, distributed data structures such as CRDTs (Shapiro *et al.*, 2011).
- A computation cannot spawn independent activities. While it is possible to create spore closures and send them to remote hosts for execution, a spore is applied only once, to materialize (the contents of) a silo. However, there is no way to create a behavior that remains active in some way across multiple interactions, like actors (Agha, 1986) or processes in the π-calculus (Milner *et al.*, 1992).

3 Examples

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.*, 2010). Section 3.1 demonstrates the implementation of some such operations in terms of silos.

Section 3.2 shows an extension of the higher-order operations of Section 3.1, providing a distributed collections abstraction reminiscent of Spark's RDDs. Finally, Section 3.3 shows an implementation of k-means clustering that demonstrates computational patterns supported by the MBrace framework (Dzik *et al.*, 2013), a programming system closely related to the function passing model.

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]]
```

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

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 `apply` invocations:

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

Then, we can use `apply` to create a new silo that contains the elements of both silo `s12` and silo `s22`:[8]

```
val combined = s12.apply(spore {
  val locals22 = s22
  (triples1: List[(K, Option[A], Option[B])]) =>
    locals22.apply(spore {
      val localTriples1 = triples1
      (triples2: List[(K, Option[A], Option[B])]) =>
        SiloRef.populate(currentHost, localTriples1 ++ triples2)
    })
})
```

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 a `SiloRef[List[(K, (A, B))]]` as required.[9]

Partitioning and `groupByKey`. A `groupByKey` operation on a group of silos containing collections needs to create multiple result silos, on each host, 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 multi-map). Then, `groupByKey` can be implemented as follows:

[8] The expression `localTriples1 ++ triples2` denotes the concatenation of lists `localTriples1` and `triples2`.

[9] Note that this way of merging triples would not be correct if multiple `A` or `B` values could have the same hash value. Using more sophisticated hash algorithms based on hash tables could be supported as well.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

- 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 `apply`, each pair of silos containing keys of the same range can be combined and materialized on the destination host.

3.2 Distributed collections

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 implemented a miniaturized example system that is inspired by Spark's Resilient Distributed Dataset (RDD).

RDDs provide an API for executing data-parallel operations on distributed data. Our simplified RDD implementation provides a collections abstraction distributed using a group of silos. 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.

Virtually all methods on RDDs are implemented using the `apply` method of `SiloRef`. RDD methods like `flatMap` or `filter` that do not require communication across silos are implemented using simple spores, which call the corresponding methods of the underlying Scala collections; each spore directly creates its result silo using the `populate` primitive. (Several examples shown earlier make use of this pattern.) In contrast, methods combining multiple silos, such as `join`, require nested invocations of the `apply` method, similar to the example shown in Figure 4.

Below, we show a simple example using our RDD abstraction. The example processes two documents, `content` and `lorem`, which are represented as RDDs containing lists of strings:

```
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 closure passed to RDD's `flatMap` method invoked on `contentWord` and `loremWord` splits each line into a list of words, and flattens everything as a single list. Each word is then mapped to a tuple containing its length and the word. Finally, we do an inner join, which in turn associates each length to the set of words of the same length, removing duplicate words in the process.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Finally, we collect the final result in a `Map` using the `collectMap` method on RDDs. Several other more detailed example programs using RDDs on top of the function passing model are available online.[10]

3.3 K-means clustering

In order to illustrate how the function passing model supports computation patterns provided by the closely related MBrace (Dzik *et al.*, 2013) framework, we ported an example implementation of k-means clustering.[11] Below, we show an excerpt of the implementation. (See Appendix A.2 for a diagram illustrating various Scala features used in the listing.) Our implementation of distributed k-means clustering using the function passing model is an almost identical port of the version using MBrace written in F♯. K-means 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] = {
  val clusterParts =
    partitionedPoints.map(silo => silo.apply(
      spore {
        val lCentroids = centroids  // spore header
        (points: Array[Point]) =>
          SiloRef.populate(currentHost, kmeansLocal(points, lCentroids))
      }
    ).send())

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

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

  if (diff < epsilon)  // check if converged, else iterate again
    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

[10] See the Git repository at `https://github.com/heathermiller/f-p/`, branch `jfp`.

[11] See the MBrace website, `http://mbrace.io/starterkit/HandsOnTutorial.FSharp/examples/200-kmeans-clustering-example.html`.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

$t ::=$	*terms:*
x	variable
$\mid v$	value
$\mid t\ t$	application
$\mid t \oplus t$	integer operator
$\mid \texttt{spore}\ \{\ \overline{x : T = t}\ ; (x : T) \Rightarrow t\ \}$	spore
$\mid \texttt{populate}(t, t)$	populate silo
$\mid \texttt{apply}(t, t)$	apply
$\mid \texttt{send}(t)$	send
$\mid \texttt{await}(t)$	await future
$\mid \texttt{respond}(h, t, t)$	respond
$v ::=$	*values:*
$(x : T) \Rightarrow t$	abstraction value
$\mid n$	integer literal
$\mid \texttt{unit}$	unit
$\mid p$	spore value
$\mid \iota$	decentralized identifier
$\mid r$	silo reference
$\mid h$	host
$p ::= \texttt{spore}\ \{\ \overline{x : T = v}\ ; (x : T) \Rightarrow t\ \}$	
$l ::=$	*lineages:*
$\mathrm{Mat}(\iota)$	materialized
$\mid \mathrm{Applied}(\iota, l, p)$	lineage with `apply`
$r ::= \mathrm{Ref}(l, h) \quad \text{where } h \in \mathscr{H}$	silo reference
$\iota ::= (h, i) \quad \text{where } h \in \mathscr{H} \text{ and } i \in \mathbb{N}$	decentralized identifier

Fig. 6. Abstract syntax of core language. Integer operators are represented using $\oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$.

all points assigned to the cluster.[12] 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 clustering, we start with a master node that partitions the points into silos. In each iteration, `apply` is called on the SiloRef that results in a spore function being applied to the data within the corresponding silo. The spore captures the centroids of the current iteration 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 verify the algorithm's convergence condition.

4 Formalization

We formalize our programming model in the context of a typed lambda calculus. Figure 6 shows the abstract syntax of our core language. Besides standard terms,

[12] Simply for the sake of illustration we assume no node failures in this computation.

$T ::=$	*types:*
$\quad T \Rightarrow T$	abstraction type
$\quad \mid \texttt{Int}$	integer type
$\quad \mid \texttt{Unit}$	unit type
$\quad \mid \mathscr{S}$	
$\quad \mid \texttt{Future}[T]$	future type
$\quad \mid \texttt{SiloRef}[T]$	silo reference type
$\quad \mid \texttt{Host}$	host type
$\mathscr{S} ::= T \Rightarrow T \;\{\; \texttt{type}\ \mathscr{C} = \overline{T} \;\}$	spore type
$\quad \mid T \Rightarrow T \;\{\; \texttt{type}\ \mathscr{C} \;\}$	abstract spore type

Fig. 7. Types of core language.

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 `populate` term initializes a new silo on a given host with a given value. The `apply` term creates a lineage of silo transformations represented as a silo reference (see below). The `send` term forces the materialization of the argument silo. The `send` expression 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. Decentralized identifiers ι are used to refer to futures and to identify silos via their lineages (each element in a lineage carries a decentralized identifier).

Values in our language are as expected: besides abstractions and integer literals they include spore values, decentralized identifiers, and silo references. Decentralized identifiers and silo references are not part of the "surface syntax" of our language; they are only introduced by evaluation (see Section 4.1). Silo reference values have the form $\mathrm{Ref}(l, h)$ where l is a lineage and h is a host. Lineages are values of a simple datatype with constructors *Mat* and *Applied*. The constructors include all information required for *materializing* a silo with the result of applying the described transformations. We defer a detailed explanation of the transformations described by a lineage to the following Section 4.1.

In addition to standard function types and `Int` and `Unit` types, the language has types for spores, futures, silo references, and hosts (see Figure 7). A spore type $T \Rightarrow T' \;\{\; \texttt{type}\ \mathscr{C} = \overline{T} \;\}$ includes the types $\overline{T}$ of the variables declared in the header of the spore.

4.1 Operational semantics

In the following, we present a small-step operational semantics of the introduced core language. The semantics is clearly stratified into a local (sequential) layer and a distributed (concurrent) layer. Importantly, this means our programming model can benefit from existing reasoning techniques for sequential programs. Program transformations that are correct for sequential programs are also correct for distributed programs. Our programming model shares this property with some existing approaches (Peyton Jones *et al.*, 1996).

$E ::=$	*evaluation contexts:*
$[\,]$	hole
$\mid E\ t$	application (fun)
$\mid v\ E$	application (arg)
$\mid \texttt{spore}\ \{\ \overline{x : T = v}; x_i : T_i = E; \overline{x' : T = t}; (x : T) \Rightarrow t\ \}$	spore
$\mid \texttt{populate}(E, t)$	populate (host)
$\mid \texttt{populate}(v, E)$	populate (val)
$\mid \texttt{apply}(E, t)$	apply (ref)
$\mid \texttt{apply}(v, E)$	apply (fun)
$\mid \texttt{send}(E)$	send
$\mid \texttt{await}(E)$	await
$\mid \texttt{respond}(h, E, t)$	respond (fut)
$\mid \texttt{respond}(h, v, E)$	respond (val)

Fig. 8. Evaluation contexts.

R-IntOp

$$\frac{v'' = v\ I(\oplus)\ v'}{E[v\ \oplus\ v'] \mid \sigma \rightarrow^h E[v''] \mid \sigma}$$

R-AppAbs

$$E[((x : T) \Rightarrow t)\ v] \mid \sigma \rightarrow^h E[[x \mapsto v]t] \mid \sigma$$

R-AppSpore

$$E[(\texttt{spore}\ \{\ \overline{x : T = v}\ ; (x : T) \Rightarrow t\ \})\ v] \mid \sigma \rightarrow^h E[\overline{[x \mapsto v]}[x \mapsto v]t] \mid \sigma$$

R-Await

$$\frac{\sigma(\iota) = v}{E[\texttt{await}(\iota)] \mid \sigma \rightarrow^h E[v] \mid \sigma}$$

R-Apply

$$\frac{r = \text{Ref}(l, h') \quad l' = \text{Applied}((h, i), l, p) \quad i \text{ fresh}}{E[\texttt{apply}(r, p)] \mid \sigma \rightarrow^h E[\text{Ref}(l', h')] \mid \sigma}$$

Fig. 9. Local reduction.

The semantics is based on two reduction relations for (a) local reduction of terms and (b) distributed reduction of sets of hosts. The reduction relations use the definition of evaluation contexts shown in Figure 8. Evaluation contexts capture the notion of the "next subterm to be evaluated." Following a standard approach (Pierce, 2002), we write $E[t]$ for the term obtained by replacing the hole in evaluation context E with term t.

Figure 9 shows the rules for local reduction. The local reduction relation has the form $E[t] \mid \sigma \rightarrow^h E[t'] \mid \sigma'$ with stores σ and σ'. Stores are required for the dynamic allocation of silos. A store σ is a partial function mapping decentralized identifiers ι to values v. The annotation with host h is used for creating decentralized identifiers $\iota = (h, i)$ for lineages. Rule R-IntOp reduces integer operator applications; function I interprets the operator symbol $\oplus$, mapping it to an actual operation on integer values. (The definition of I is trivial and standard, and thus omitted.) Rule R-AppAbs is completely standard. Analogous to rule R-AppAbs, rule R-AppSpore describes the application of a spore value to an argument value. Rule R-Await reduces $\texttt{await}(\iota)$ to v if future ι is already completed with v in store σ.

Rule R-Apply creates a lineage using the constructor *Applied*. The new lineage has a fresh identifier (h, i), which uniquely identifies the corresponding (logical) silo. The spore value p is stored in the new lineage; this enables a materialization of the silo identified by (h, i) using parent lineage l and spore p.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

$$\begin{array}{rl} m ::= & \textit{messages:} \\ \mathrm{Req}(h, r, \iota) & \text{request} \\ \mid \mathrm{Res}(\iota, v) & \text{response} \end{array}$$

Fig. 10. Messages.

Distributed reduction. The distributed reduction rules use helper functions *host*, *id*, and *parent*, which are defined as follows:

Definition 4.1 (Host identifier) *The host identifier of a silo reference.*
$host(Ref(l, h)) := h$

Definition 4.2 (Lineage identifier) *The identifier of a lineage.*

$$id(l) := \begin{cases} \iota & \textit{if } l = Mat(\iota) \\ \iota & \textit{if } l = Applied(\iota, \_, \_) \end{cases}$$

Definition 4.3 (Lineage parent) *The parent of a lineage.*

$$parent(l) := \big\{\, l' \quad \textit{if } l = Applied(\_, l', \_)$$

The distributed reduction relation has the form $H \mid M \twoheadrightarrow H' \mid M'$, where H, H' are sets of hosts and M, M' are multisets of messages. A host is a machine that executes a computation, and that can store silos in its local memory. In our formal model, a host is represented as a pair $(t, \sigma)^h$ consisting of a term t which is the executed computation, and a partial function σ which is the local store. Note that each host has a unique host identifier; for instance, the host identifier of host $(t, \sigma)^h$ is h. When it is clear from the context, we use the terms "host" and "host identifier" interchangeably. The multisets M, M' model *message sends* that are "in flight;" a message send $h \leftarrow m$ expresses the sending of message m to host h. If $h \leftarrow m \in M$ then message m has not yet been delivered to h (the message is still in transit).

As shown in Figure 10, there are two kinds of messages. A message of the form $\mathrm{Req}(h, r, \iota)$ requests the value of silo r to be sent to host h for materialization of identifier ι. A message of the form $\mathrm{Res}(\iota, v)$ represents the corresponding response, containing the identifier ι to be materialized using value v.

Figure 11 shows the distributed reduction rules. Rule R-LOCAL reduces a host h chosen non-deterministically from the set of hosts (the rule "schedules" host h for execution). Term t of host h is reduced according to the local reduction rules. Thus, no communication is taking place; this means that the multiset of in-flight messages M remains unchanged.

Rule R-SEND reduces a term `send`(r) on host h. The reduction initiates the materialization of silo r by sending a message $m = \mathrm{Req}(h, r, id(l))$ to the host of r, h'. Thus, the message send $h' \leftarrow m$ is added to the resulting multiset of in-flight messages M'. The `send` term itself is reduced to the identifier of r's lineage, $id(l)$.

Rule R-POPULATE asynchronously populates a new silo on host h' with value v by sending message $\mathrm{Res}(\iota, v)$ to h'. Note that the fresh decentralized identifier $\iota = (h, i)$ is already created on host h, which does not require any communication. The new lineage l is *materialized* under identifier ι, represented using the lineage value $\mathrm{Mat}(\iota)$.

Rule R-RESPOND reduces a term `respond`(h', ι, v) by sending a response message $\mathrm{Res}(\iota, v)$ to h'.

R-Local

$$\frac{t \mid \sigma \rightarrow^{h} t' \mid \sigma'}{\{(t,\sigma)^{h}\} \cup H \mid M \twoheadrightarrow \{(t',\sigma')^{h}\} \cup H \mid M}$$

R-Send

$$\frac{r = \mathrm{Ref}(l,h') \quad m = \mathrm{Req}(h,r,id(l)) \quad M' = M \uplus \{h' \leftarrow m\}}{\{(E[\mathtt{send}(r)],\sigma)^{h}\} \cup H \mid M \twoheadrightarrow \{(E[id(l)],\sigma)^{h}\} \cup H \mid M'}$$

R-Populate

$$\frac{\iota = (h,i) \quad i \text{ fresh} \quad l = \mathrm{Mat}(\iota) \quad M' = M \uplus \{h' \leftarrow \mathrm{Res}(\iota,v)\}}{\{(E[\mathtt{populate}(h',v)],\sigma)^{h}\} \cup H \mid M \twoheadrightarrow \{(E[\mathrm{Ref}(l,h')],\sigma)^{h}\} \cup H \mid M'}$$

R-Respond

$$\frac{M' = M \uplus \{h' \leftarrow \mathrm{Res}(\iota,v)\}}{\{(E[\mathtt{respond}(h',\iota,v)],\sigma)^{h}\} \cup H \mid M \twoheadrightarrow \{(E[\mathtt{unit}],\sigma)^{h}\} \cup H \mid M'}$$

R-Process

$$\frac{\iota \notin dom(\sigma) \quad M = M' \uplus \{h \leftarrow m\} \quad process(h,m,\sigma) = (t,M'',\sigma')}{\{(E[\mathtt{await}(\iota)],\sigma)^{h}\} \cup H \mid M \twoheadrightarrow \{(E[t\ ;\ \mathtt{await}(\iota)],\sigma')^{h}\} \cup H \mid M' \uplus M''}$$

R-Process-Val

$$\frac{M = M' \uplus \{h \leftarrow m\} \quad process(h,m,\sigma) = (t,M'',\sigma')}{\{(v,\sigma)^{h}\} \cup H \mid M \twoheadrightarrow \{(t,\sigma')^{h}\} \cup H \mid M' \uplus M''}$$

Fig. 11. Distributed reduction.

Rule R-Process models the processing of a message m taken from the multiset M of in-flight messages. Note that this rule is only enabled if h is *suspended*, waiting for a silo ι to be materialized in local store σ. A host is suspended if its current term has the form $E[\mathtt{await}(\iota)]$ and $\iota \notin dom(\sigma)$ (there is no mapping for ι in store σ). The premise $\iota \notin dom(\sigma)$ is important, since otherwise rule R-Local would also be enabled.

Rule R-Process-Val allows a host h to process a message when the host's term has been reduced to a value v. Note that host h may have silos, stored in σ, that are still required by other hosts. Therefore, hosts with terminated computations remain available to respond to messages. In rule R-Process-Val, the message to be processed is taken non-deterministically from the multiset M of in-flight messages (analogous to other rules, this reflects the non-deterministic order in which messages are received). Processing message m results in a term t to be evaluated next by host h, a set of produced messages M'', and the new state σ' of h. Since h is supposed to evaluate t next, t replaces v in the resulting host configuration. Note that term t may be of a different type than v. Importantly, this does not break soundness: in the resulting configuration $(t,\sigma')^{h}$ it is sufficient for term t to be well-typed *for some* type T. In contrast, local reduction of a term does not permit changing a term's type (see Theorem 5.2 in Section 5.1 as well as its proof in Appendix B.2.)

The actual message processing logic is factored out into function *process* that we discuss in the following. In general, processing a message m on host h in store σ results in (a) a term t to be evaluated by h, (b) a set of new in-flight messages M', and (c) an updated store σ'.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Proc-Res

$$\frac{\sigma' = [\iota \mapsto v]\sigma}{\mathit{process}(h, \mathrm{Res}(\iota, v), \sigma) = (\texttt{unit}, \emptyset, \sigma')}$$

Proc-ReqParent

$$\frac{r = \mathrm{Ref}(l, h) \quad l' = \mathit{parent}(l) \quad \mathit{id}(l') \notin \mathit{dom}(\sigma) \quad M = \{h \leftarrow \mathrm{Req}(h', r, \iota)\}}{\mathit{process}(h, \mathrm{Req}(h', r, \iota), \sigma) = (\texttt{send}(\mathrm{Ref}(l', h)), M, \sigma)}$$

Proc-ReqMat1

$$\frac{r = \mathrm{Ref}(l, h) \quad l = \mathrm{Mat}(\iota') \quad \sigma(\iota') = v \quad M = \{h' \leftarrow \mathrm{Res}(\iota, v)\}}{\mathit{process}(h, \mathrm{Req}(h', r, \iota), \sigma) = (\texttt{unit}, M, \sigma)}$$

Proc-ReqMat2

$$\frac{r = \mathrm{Ref}(l, h) \quad l = \mathrm{Mat}(\iota') \quad \iota' \notin \mathit{dom}(\sigma) \quad M = \{h \leftarrow \mathrm{Req}(h', r, \iota)\}}{\mathit{process}(h, \mathrm{Req}(h', r, \iota), \sigma) = (\texttt{unit}, M, \sigma)}$$

Proc-ReqApply

$$\frac{r = \mathrm{Ref}(l, h) \quad l = \mathrm{Applied}(\iota', l', p) \quad \sigma(\mathit{id}(l')) = v \quad t = \texttt{respond}(h', \iota, \texttt{await}(\texttt{send}(p\ v)))}{\mathit{process}(h, \mathrm{Req}(h', r, \iota), \sigma) = (t, \emptyset, \sigma)}$$

Fig. 12. Message processing.

Definition 4.4 (Message processing) *The function process* $\in \mathscr{H} \times m \times S \rightharpoonup t \times \mathscr{P}(\mathscr{H} \times m) \times S$ *handles the processing of a message resulting in a term to be evaluated, a set of messages, and an updated store; here,* $S = \mathscr{H} \times \mathbb{N} \rightharpoonup v$ *is the type of a store. The process function is defined in Figure 12.*

The *process* function must handle the two kinds of messages, $\mathrm{Req}(h', r, \iota)$ and $\mathrm{Res}(\iota, v)$. Processing a message of the form $\mathrm{Res}(\iota, v)$ means that a host is responding with the value v of a silo whose materialization has been requested by the current host h to complete its future ι. Thus, it suffices to create an updated store σ', which maps ι to v (rule Proc-Res).

Processing a message of the form $\mathrm{Req}(h', r, \iota)$ means that host h' requests the value of silo r in order to complete a future with identifier ι. In the absence of caching/persisting, this requires the receiver of the request to materialize the silo whose value is requested. Section 4.4 introduces a refinement of these rules to enable caching, thereby avoiding repeated materializations of the same silo.

The requested silo $r = \mathrm{Ref}(l, h)$ is materialized using its lineage l. Rule Proc-ReqParent handles the case where r's parent $\mathit{id}(l')$ is not yet materialized (l' is the lineage of r's parent). In this case, *process* returns the term $\texttt{send}(\mathrm{Ref}(l', h))$; this causes host h to request the materialization of the parent silo $\mathrm{Ref}(l', h)$ before any other message is processed. Note that a silo is always materialized on the same host as its parent silo; therefore, the hosts of r and its parent are guaranteed to be the same. The original request $\mathrm{Req}(h', r, \iota)$ is included in the new in-flight messages M, so that it is eventually handled, namely when r's parent is materialized.

Rules Proc-ReqMat1 and Proc-ReqMat2 handle the case where r does not have a parent. In the case of Proc-ReqMat1, the materialization of r is just the value v of its materialized lineage $\mathrm{Mat}(\iota')$; thus, a response $\mathrm{Res}(\iota, v)$ is sent to host h'. In the case of Proc-ReqMat2, lineage $\mathrm{Mat}(\iota')$ is not yet materialized (*i.e.*, host h is still

$$\textsc{T-Var}\ \frac{x : T \in \Gamma}{\Gamma;\Sigma \vdash x : T} \qquad \textsc{T-Int}\ \frac{n \text{ integer literal}}{\Gamma;\Sigma \vdash n : \texttt{Int}} \qquad \textsc{T-IntOp}\ \frac{\Gamma;\Sigma \vdash t : \texttt{Int} \quad \Gamma;\Sigma \vdash t' : \texttt{Int}}{\Gamma;\Sigma \vdash t \oplus t' : \texttt{Int}}$$

$$\textsc{T-Abs}\ \frac{\Gamma, x : T;\Sigma \vdash t : T'}{\Gamma;\Sigma \vdash ((x : T) \Rightarrow t) : T \Rightarrow T'} \qquad \textsc{T-App}\ \frac{\Gamma;\Sigma \vdash t : T \Rightarrow T' \quad \Gamma;\Sigma \vdash t' : T}{\Gamma;\Sigma \vdash (t\ t') : T'} \qquad \textsc{T-Unit}\ \Gamma;\Sigma \vdash \texttt{unit} : \texttt{Unit}$$

$$\textsc{T-Spore}\ \frac{\Gamma;\Sigma \vdash \overline{t} : \overline{T} \quad \overline{x : T}, x : T; \emptyset \vdash t : T' \quad \forall S \in \overline{T}, T'.\ \mathit{serializable}(S)}{\Gamma;\Sigma \vdash (\texttt{spore}\ \{\ \overline{x : T = t}\ ; (x : T) \Rightarrow t\ \}) : T \Rightarrow T'\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ \}}$$

$$\textsc{T-AppSpore}\ \frac{\Gamma;\Sigma \vdash t : T \Rightarrow T'\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ \} \quad \Gamma;\Sigma \vdash t' : T}{\Gamma;\Sigma \vdash (t\ t') : T'} \qquad \textsc{T-SiloRef}\ \frac{\Sigma(id(l)) = T \quad \Sigma \vdash \mathrm{Ref}(l,h)}{\Gamma;\Sigma \vdash \mathrm{Ref}(l,h) : \texttt{SiloRef}[T]}$$

$$\textsc{T-Apply}\ \frac{\Gamma;\Sigma \vdash t : \texttt{SiloRef}[T] \quad \Gamma;\Sigma \vdash t' : (T \Rightarrow \texttt{SiloRef}[T']\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ \})}{\Gamma;\Sigma \vdash \texttt{apply}(t,t') : \texttt{SiloRef}[T']} \qquad \textsc{T-Send}\ \frac{\Gamma;\Sigma \vdash t : \texttt{SiloRef}[T]}{\Gamma;\Sigma \vdash \texttt{send}(t) : \texttt{Future}[T]}$$

$$\textsc{T-Await}\ \frac{\Gamma;\Sigma \vdash t : \texttt{Future}[T]}{\Gamma;\Sigma \vdash \texttt{await}(t) : T} \qquad \textsc{T-Ident}\ \frac{\Sigma(\iota) = T}{\Gamma;\Sigma \vdash \iota : \texttt{Future}[T]} \qquad \textsc{T-Respond}\ \frac{h \in \mathscr{H} \quad \Gamma;\Sigma \vdash t : \texttt{Future}[T] \quad \Gamma;\Sigma \vdash t' : T}{\Gamma;\Sigma \vdash \texttt{respond}(h,t,t') : \texttt{Unit}}$$

$$\textsc{T-Populate}\ \frac{\Gamma;\Sigma \vdash t : \texttt{Host} \quad \Gamma;\Sigma \vdash t' : T \quad \mathit{serializable}(T)}{\Gamma;\Sigma \vdash \texttt{populate}(t,t') : \texttt{SiloRef}[T]}$$

Fig. 13. Type assignment.

waiting for a message $\mathrm{Res}(\iota', v')$); thus, the original request $\mathrm{Req}(h', r, \iota)$ sent to h is re-sent, to be processed again later.

Rule Proc-ReqApply is only enabled when the requested silo r can be materialized using its lineage l; in particular, r's parent must be materialized. In this case, the lineage of r begins with an *Applied* constructor. The materialization of r consists of the result of applying the spore p provided by the lineage to the value v of the parent silo. Evaluating the application $p\ v$ requires multiple reduction steps in general. Therefore, rule Proc-ReqApply returns a term containing the application $p\ v$ for evaluation by host h. However, note that $p\ v$ reduces to a *silo reference*. Therefore, the value of silo $p\ v$ is obtained by requesting its materialization (using `send`$(p\ v)$) and waiting for the completion of the returned future (using `await`). Finally, when the future is resolved, a response $\mathrm{Res}(\iota, v')$ (for some value v') is sent to host h' by evaluating the `respond` term.

4.2 Type assignment

Type assignment is based on a judgment of the form $\Gamma;\Sigma \vdash t : T$ that assigns term t type T. Γ is a type environment which maps variables x to types T; Σ is a store typing which maps identifiers ι to types T. Figure 13 shows type assignment rules. Rules T-Var, T-Abs, and T-App correspond to a standard typed lambda calculus (Pierce, 2002). Rules T-Int and T-IntOp assign types to integer literals and applications of integer operators, respectively. Rule T-Spore assigns a type to

$$\textsc{S-Int}\;\; \mathit{serializable}(\texttt{Int}) \qquad \textsc{S-Unit}\;\; \mathit{serializable}(\texttt{Unit}) \qquad \textsc{S-Host}\;\; \mathit{serializable}(\texttt{Host}) \qquad \textsc{S-SiloRef}\;\; \mathit{serializable}(\texttt{SiloRef}[T])$$

$$\textsc{S-Spore}\;\; \frac{\forall T_i \in \overline{T}.\ \mathit{serializable}(T_i)}{\mathit{serializable}(T \Rightarrow T'\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ \})}$$

Fig. 14. Serializable types.

spore literals. Importantly, the body of the spore's closure, t, must be well-typed in a type environment containing only the closure parameter x and the variables $\overline{x}$ in the spore's header, as well as an empty store typing. Furthermore, the types of captured variables as well as the result type T' must be serializable. The predicate *serializable* is defined in Figure 14. These constraints ensure that spore values are always independent of the environment of the creating host. This independence is expressed by the following theorem:

Theorem 4.1 (Serializable Values) *If* $\Gamma;\Sigma \vdash v : T$ *and* $\mathit{serializable}(T)$ *then* $\emptyset;\Sigma \vdash v : T$.

Proof
By induction on the derivation of $\Gamma;\Sigma \vdash v : T$. See Appendix B.1. □

Rule T-AppSpore is analogous to rule T-App. Rules T-Populate and T-Apply are straightforward; note that `apply` is polymorphic in the types of the captured variables of its spore argument type. Rules T-Send and T-Await are entirely unsurprising. Rules T-Ident and T-SiloRef are the only rules that use the store typing Σ. The type of an identifier ι has the form `Future`$[T]$ where type T is looked up in the store typing. Rule T-SiloRef is analogous; additionally, it requires the silo reference Ref(l,h) to be well-formed in Σ (see below).

4.3 Well-formed configurations

Figure 15 shows the rules for well-formed configurations. These rules are essential for establishing subject reduction (see Section 5.1). Rules WF-Store1-3 are straightforward. Rule WF-Lin1 requires Σ to be defined for the identifier of a materialized lineage Mat(ι). Rule WF-Lin2 requires the types of ι and $id(l)$ given by the store typing Σ to be consistent with the corresponding type of spore p. Parent lineage l must be well-formed in Σ. Rule WF-Ref extends well-formedness of lineages to silo references. Rules WF-Res and WF-Req specify well-formedness of messages in Σ. The remaining rules lift well-formedness to host configurations (WF-HostConfig), sets of hosts (WF-Host1-2), multisets of messages (WF-Messages-Emp, WF-Messages), and configurations (WF-Config), respectively.

4.4 Persist and unpersist

As explained in Section 2.4, silos may be cached in memory to avoid repeated materialization, which may be expensive for large data sets. The design is inspired by the popular Spark (Zaharia *et al.*, 2010) data processing system. Essentially, a silo r may be *persisted* using the `persist` primitive: the call `persist`(r) immediately

WF-Store1
$$\emptyset \vdash \emptyset$$

WF-Store2
$$\frac{\emptyset;\Sigma \vdash v : T \quad \Sigma \vdash \sigma}{[\iota \mapsto T]\Sigma \vdash [\iota \mapsto v]\sigma}$$

WF-Store3
$$\frac{\Sigma \vdash \sigma \quad \Sigma' \supseteq \Sigma}{\Sigma' \vdash \sigma}$$

WF-Lin1
$$\frac{\iota \in dom(\Sigma)}{\Sigma \vdash \mathrm{Mat}(\iota)}$$

WF-Lin2
$$\frac{\Sigma(\iota) = T \quad \Sigma(id(l)) = T' \quad \exists \Gamma.\ \Gamma;\Sigma \vdash p : T' \Rightarrow \texttt{SiloRef}[T]\ \{\ldots\} \quad \Sigma \vdash l}{\Sigma \vdash \mathrm{Applied}(\iota, l, p)}$$

WF-Ref
$$\frac{\Sigma \vdash l \quad h \in \mathcal{H}}{\Sigma \vdash \mathrm{Ref}(l, h)}$$

WF-Res
$$\frac{\Sigma(\iota) = T \quad \emptyset;\Sigma \vdash v : T}{\Sigma \vdash \mathrm{Res}(\iota, v)}$$

WF-Req
$$\frac{r = \mathrm{Ref}(l, h') \quad \Sigma(id(l)) = \Sigma(\iota) \quad \Sigma \vdash r}{\Sigma \vdash \mathrm{Req}(h, r, \iota)}$$

WF-HostConfig
$$\frac{\Sigma \vdash \sigma \quad \exists \Gamma.\ \Gamma;\Sigma \vdash t : T}{\Sigma \vdash (t, \sigma)^h}$$

WF-Host1
$$\Sigma \vdash \emptyset$$

WF-Host2
$$\frac{\Sigma \vdash (t, \sigma)^h \quad \Sigma \vdash H}{\Sigma \vdash \{(t, \sigma)^h\} \cup H}$$

WF-Messages-Emp
$$\Sigma \vdash \emptyset$$

WF-Messages
$$\frac{\Sigma \vdash m \quad h \in \mathcal{H} \quad \Sigma \vdash M}{\Sigma \vdash \{h \leftarrow m\} \uplus M}$$

WF-Config
$$\frac{\Sigma \vdash H \quad \Sigma \vdash M}{\Sigma \vdash H \mid M}$$

Fig. 15. Well-formedness.

returns a new silo reference r' whose value, when materialized, is equal to the value of r. The lineage of r' is equal to that of r except for one additional element: if l is the lineage of silo r, then the lineage of r' is of the form $l' = \mathrm{Persist}(\iota, l, f)$. As for the *Applied* constructor ι is the identifier of lineage l'. (As before, $\iota = (h, i)$ implies that host h has called `persist`.) The last argument is a binary operator $f \in \{\cdot \cup \cdot, \cdot \setminus \cdot\}$, which toggles between the behavior of `persist` and `unpersist`. If $f = \cdot \cup \cdot$ then the lineage $\mathrm{Persist}(\iota, l, f)$ implements the behavior of `persist`; otherwise, the behavior of `unpersist`. The extensions to syntax, lineages, and evaluation contexts are summarized in Figure 16.

4.4.1 *Operational semantics*

In order to distinguish between silos that have been persisted and those that have not, we extend stores σ to map identifiers ι not just to their associated value, but also to a so-called "persist set" P; if $\sigma(\iota) = (v, P)$ then P contains all hosts that have persisted silo ι. Stores have thus the following extended type:

Definition 4.5 (Store) $\sigma \in \mathcal{H} \times \mathbb{N} \rightharpoonup v \times \mathcal{P}(\mathcal{H})$.

As a next step, we extend the local reduction relation, as shown in Figure 17. First, we introduce the reduction rules R-Persist and R-Unpersist to enable the creation of Persist lineages. Second, we adjust reduction rule R-Await to use the extended store.

The rules for message processing are affected most. Intuitively, the reason is that the fact whether a silo is (re-)materialized is determined in the context of message processing. Figure 18 shows new or extended rules for message processing. Rule Proc-Res is simply adjusted to the new store definition. Rule Proc-ReqPersist is new; the rule enables processing requests $\mathrm{Req}(h', r, \iota)$ in cases where r has a Persist

$$
\begin{array}{lll}
t ::= & & \textit{terms:} \\
 & \ldots & \\
 & \mid \texttt{persist}(t) & \text{persist} \\
 & \mid \texttt{unpersist}(t) & \text{unpersist} \\
 & & \\
l ::= & & \textit{lineages:} \\
 & \ldots & \\
 & \mid \text{Persist}(\imath, l, f) & \text{lineage with } \texttt{persist} \\
 & & \\
E ::= & & \textit{evaluation contexts:} \\
 & \ldots & \\
 & \mid \texttt{persist}(E) & \text{persist} \\
 & \mid \texttt{unpersist}(E) & \text{unpersist}
\end{array}
$$

Fig. 16. Extensions to syntax, lineages, and evaluation contexts for `persist`/`unpersist`. f is a binary operator with $f \in \{\cdot \cup \cdot, \cdot \setminus \cdot\}$.

$$
\textsc{R-Persist}\quad \frac{r = \text{Ref}(l, h') \quad l' = \text{Persist}((h,i), l, \cdot \cup \cdot) \quad i \text{ fresh}}{E[\texttt{persist}(r)] \mid \sigma \rightarrow^{h} E[\text{Ref}(l', h')] \mid \sigma}
$$

$$
\textsc{R-Unpersist}\quad \frac{r = \text{Ref}(l, h') \quad l' = \text{Persist}((h,i), l, \cdot \setminus \cdot) \quad i \text{ fresh}}{E[\texttt{unpersist}(r)] \mid \sigma \rightarrow^{h} E[\text{Ref}(l', h')] \mid \sigma}
\qquad
\textsc{R-Await}\quad \frac{\sigma(\imath) = (v, P)}{E[\texttt{await}(\imath)] \mid \sigma \rightarrow^{h} E[v] \mid \sigma}
$$

Fig. 17. Extension to local reduction.

lineage and r's parent is materialized (otherwise, the above Proc-ReqParent rule would be enabled). Importantly, a silo with lineage $\text{Persist}(\imath', l', \star)$ has the same value as the parent silo which has identifier $id(l')$. Therefore, rule Proc-ReqPersist updates the store to map the identifier of the persisted silo, $\imath'$, to the value of $id(l')$ in σ. In addition, a *persist set* P' is computed based on (1) the persist set P of the parent silo, (2) the host h'' that created the Persist lineage, and (3) the operator $\star$ provided in the Persist lineage. The latter may either be set union or set difference; accordingly, the persisting host h'' is either added or removed from the persist set P of the parent of the silo to-be-persisted. In case h'' is added to the persist set (because host h'' called `persist` rather than `unpersist`), it means that the persist set P' is non-empty. As a result, the mapping for $\imath'$ remains resident in the store, avoiding re-materialization, as the following rules show.

Rule Proc-Req is also new. The rule is enabled when the requested silo r has already been materialized, *i.e.*, $\sigma(id(l)) = (v, P)$. In this case, a response $\text{Res}(\imath, v)$ can directly be sent back to the requesting host h'. Importantly, the rule also checks whether the silo has been persisted, by examining its persist set P in store σ. Using the *consume* function, the mapping for identifier $id(l)$ is *removed* from the updated store σ' if the persist set P is empty. An empty persist set indicates that the corresponding silo has not been persisted. Concretely, the *consume* function is defined as follows:

Definition 4.6 (Consume silo) *Consume silo referenced by $\imath$ with persist set P in store σ* $consume(\imath, P, \sigma) := \begin{cases} \sigma - \imath & \textit{if } P = \emptyset \\ \sigma & \textit{otherwise} \end{cases}$

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Proc-Res

$$\frac{\sigma' = [\iota \mapsto (v, \emptyset)]\sigma}{process(h, \mathrm{Res}(\iota, v), \sigma) = (\mathtt{unit}, \emptyset, \sigma')}$$

Proc-ReqPersist

$$\frac{\begin{array}{c} r = \mathrm{Ref}(l, h) \quad l = \mathrm{Persist}(\iota', l', \star) \quad \iota' = (h'', i) \\ \sigma(id(l')) = (v, P) \quad P' = P \star \{h''\} \quad \sigma' = [\iota' \mapsto (v, P')]\sigma \end{array}}{process(h, \mathrm{Req}(h', r, \iota), \sigma) = (\mathtt{unit}, \{h' \leftarrow \mathrm{Res}(\iota, v)\}, \sigma')}$$

Proc-Req

$$\frac{r = \mathrm{Ref}(l, h) \quad \sigma(id(l)) = (v, P) \quad \sigma' = \mathrm{consume}(id(l), P, \sigma)}{process(h, \mathrm{Req}(h', r, \iota), \sigma) = (\mathtt{unit}, \{h' \leftarrow \mathrm{Res}(\iota, v)\}, \sigma')}$$

Proc-ReqApply

$$\frac{\begin{array}{c} r = \mathrm{Ref}(l, h) \quad l = \mathrm{Applied}(\iota', l', p) \quad \sigma(id(l')) = (v, P) \\ t = \mathtt{respond}(h', \iota, \mathtt{await}(\mathtt{send}(p\ v))) \quad \sigma' = \mathrm{consume}(id(l'), P, \sigma) \end{array}}{process(h, \mathrm{Req}(h', r, \iota), \sigma) = (t, \emptyset, \sigma')}$$

Fig. 18. Extensions to message processing.

T-Persist

$$\frac{\Gamma; \Sigma \vdash t : \mathtt{SiloRef}[T]}{\Gamma; \Sigma \vdash \mathtt{persist}(t) : \mathtt{SiloRef}[T]}$$

T-Unpersist

$$\frac{\Gamma; \Sigma \vdash t : \mathtt{SiloRef}[T]}{\Gamma; \Sigma \vdash \mathtt{unpersist}(t) : \mathtt{SiloRef}[T]}$$

WF-Store2

$$\frac{\emptyset; \Sigma \vdash v : T \quad \Sigma \vdash \sigma}{[\iota \mapsto T]\Sigma \vdash [\iota \mapsto (v, P)]\sigma}$$

WF-Lin3

$$\frac{\Sigma(\iota) = \Sigma(id(l)) \quad \Sigma \vdash l}{\Sigma \vdash \mathrm{Persist}(\iota, l, \star)}$$

Fig. 19. Extension to type assignment and well-formedness.

Rule Proc-ReqApply is extended to consume the parent silo in case it has not been persisted.

Figure 19 shows the required extensions to type assignment and well-formedness. The type rules T-Persist and T-Unpersist are straightforward. Rule WF-Store2 is simply adjusted to the extended store definition. Rule WF-Lin3 defines well-formedness for Persist lineages: the store typings for ι and $id(l)$ must be equal and parent lineage l must be well-formed.

5 Correctness properties

5.1 *Subject reduction*

This section establishes a subject reduction theorem for the presented core language. The complete proof is provided in the appendix; here, we restrict ourselves to summarizing the main results.

Lemma 5.1 (Substitution) *If* $\Gamma, x : T'; \Sigma \vdash t : T$ *and* $\Gamma; \Sigma \vdash v : T'$ *then* $\Gamma; \Sigma \vdash [x \mapsto v]t : T$.

Proof

By induction on the derivation of $\Gamma, x : T'; \Sigma \vdash t : T$. □

Theorem 5.2 (Subject Reduction) 1. *If* $\Gamma;\Sigma \vdash t : T$, $\Sigma \vdash \sigma$, *and* $t \mid \sigma \rightarrow^h t' \mid \sigma'$ *then* $\Gamma;\Sigma' \vdash t' : T$ *and* $\Sigma' \vdash \sigma'$ *for some* $\Sigma' \supseteq \Sigma$.

2. *If* $\Sigma \vdash H \mid M$ *and* $H \mid M \twoheadrightarrow H' \mid M'$ *then* $\Sigma' \vdash H' \mid M'$ *for some* $\Sigma' \supseteq \Sigma$.

Proof
Part 1: by induction on the derivation of $t \mid \sigma \rightarrow^h t' \mid \sigma'$. Part 2: by induction on the derivation of $H \mid M \twoheadrightarrow H' \mid M'$. See Appendix B.2 for the complete proof. □

5.2 *Progress*

This section formulates progress properties. The main Theorem 5.5 states that materialization requests are satisfied after a finite number of reduction steps in so-called "responsive configurations."

In the following, we assume a *fair scheduling* property that ensures that in a well-formed configuration $H \mid M$, each message $h \leftarrow m \in M$ is eventually received by host h. Fair scheduling is also assumed in other models of distributed computing like actors (Agha, 1986; Agha *et al.*, 1997). Formally, fair scheduling is defined as follows:

Definition 5.1 (Fair Scheduling) *Let* $\Sigma \vdash H \mid M$ *and* $h \leftarrow m \in M$ *where* $\Sigma \vdash m$.

Then $H \mid M \twoheadrightarrow^* H' \mid M' \twoheadrightarrow H'' \mid M''$ *after a finite number of reduction steps, and* $H' \mid M' \twoheadrightarrow H'' \mid M''$ *by rule* R-Process *or* R-Process-Val *such that* $M' = M_{old} \uplus \{h \leftarrow m\}$, $(t,\sigma)^h \in H'$, $process(h,m,\sigma) = (t', M_{new}, \sigma')$, *and* $M'' = M_{old} \uplus M_{new}$.

Although our focus is the establishment of desirable progress properties for distributed reduction, it is necessary to consider the following *strong normalization* property of single-host reductions. For this, we consider the reduction relation $\rightsquigarrow$ defined as the subset of the reduction relation $\twoheadrightarrow$ excluding reduction rules R-Process and R-Process-Val.

Lemma 5.3 (Single-Host Strong Normalization) *Let* $\Sigma \vdash H \mid M$ *where* $H = \{(t,\sigma)^h\} \cup H'$. *Then* $H \mid M \rightsquigarrow^* \{(t',\sigma')^h\} \cup H' \mid M'$ *after a finite number of reduction steps and either* t' *is a value or* $t' = E[\mathtt{await}(\iota)]$.

Note that in the above reduction, only a single host h is reduced. Furthermore, the set of in-flight messages may change, *e.g.*, by applying rule R-Send. A proof of Lemma 5.3 is outside the scope of this paper. However, our core language is, fundamentally, not more expressive than the simply typed lambda calculus, for which strong normalization holds.

As a prerequisite for the establishment of our main progress theorem, we introduce a small amount of bookkeeping into the reduction relations. The aim is to keep track of silo references created during reduction. Importantly, this additional bookkeeping information does not introduce any change in the semantic behavior–the information can be erased without affecting the run-time semantics in any way.

The augmented local reduction relation has the form $E[t] \mid \sigma \rightarrow^h E[t'] \mid \sigma' \mid R$ where R is either the empty set or a singleton set containing the created silo reference. Rule R-Apply is the only rule resulting in a non-empty set of created references:

R-Apply
$$\frac{r = \mathrm{Ref}(l,h') \quad l' = \mathrm{Applied}((h,i),l,p) \quad i \text{ fresh} \quad r' = \mathrm{Ref}(l',h')}{E[\mathtt{apply}(r,p)] \mid \sigma \rightarrow^h E[r'] \mid \sigma \mid \{r'\}}$$

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

The augmented distributed reduction relation has the form $H \mid M \mid R \twoheadrightarrow H' \mid M' \mid R'$ where R is the set of references already existing before performing the reduction step and R' is the set of references existing after performing the reduction step. Rules R-Local and R-Populate are the only rules resulting in a set of references R' where $R' \neq R$ is possible:

R-Local

$$\frac{t \mid \sigma \rightarrow^{h} t' \mid \sigma' \mid R'}{\{(t,\sigma)^{h}\} \cup H \mid M \mid R \twoheadrightarrow \{(t',\sigma')^{h}\} \cup H \mid M \mid R \cup R'}$$

R-Populate

$$\frac{\iota = (h,i) \quad i \text{ fresh} \quad l = \mathrm{Mat}(\iota) \quad M' = M \uplus \{h' \leftarrow \mathrm{Res}(\iota,v)\} \quad r' = \mathrm{Ref}(l,h')}{\{(E[\texttt{populate}(h',v)],\sigma)^{h}\} \cup H \mid M \mid R \twoheadrightarrow \{(E[r'],\sigma)^{h}\} \cup H \mid M' \mid R \cup \{r'\}}$$

Finally, the extension of well-formed configurations is straightforward:

WF-Config

$$\frac{\Sigma \vdash H \quad \Sigma \vdash M \quad \Sigma \vdash R}{\Sigma \vdash H \mid M \mid R}$$

WF-Refs

$$\frac{\Sigma \vdash r \quad \Sigma \vdash R}{\Sigma \vdash \{r\} \cup R}$$

Using the augmented reduction rules, we introduce a responsiveness property, *responsive configurations*. Informally, in a responsive configuration requesting (the materialization of) any previously created silo reference results in a corresponding response after a finite number of reduction steps. The property is defined as follows:

Definition 5.2 (Responsive Configuration) *A configuration $\Sigma \vdash H \mid M \mid R$ is responsive, written Responsive(H,M,R), iff*

$\forall r = Ref(l,h) \in R.\ (m = Req(h',r,\iota) \wedge \Sigma \vdash m) \implies H \mid M \uplus \{h \leftarrow m\} \mid R \twoheadrightarrow^{*} H' \mid M' \uplus \{h' \leftarrow Res(\iota,v)\} \mid R'$ after a finite number of reduction steps.

The following lemma ensures that the ability to materialize a silo after a finite number of reduction steps is preserved under reduction.

Lemma 5.4 (Responsiveness) *Let $\Sigma \vdash H \mid M \mid R \cup \hat{R}$ and Responsive$(H,M,\hat{R})$.*

If $H \mid M \mid R \cup \hat{R} \twoheadrightarrow H' \mid M' \mid R' \cup \hat{R}$ then Responsive$(H',M',\hat{R})$.

Proof Sketch

By induction on the derivation of $H \mid M \mid R \cup \hat{R} \twoheadrightarrow H' \mid M' \mid R' \cup \hat{R}$ with case analysis of the last applied rule, using Def. 5.1 and Lemma 5.3. □

We are now ready to introduce the main progress theorem. Theorem 5.5 states that finite materialization of silos is a universal property of our core language.

Theorem 5.5 (Finite Materialization) *Let $\Sigma \vdash H \mid M \mid R$ such that Responsive(H, M,R).*

If $H \mid M \mid R \twoheadrightarrow H' \mid M' \mid R'$ then Responsive(H',M',R').

Proof

See Appendix B.3. □

6 Related work

Alice ML (Rossberg *et al.*, 2004) is an extension of Standard ML that 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 our 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, 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 `apply` 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.*, 2005) is a dialect of ML that proposes numerous primitives for distributed programming, such as type-safe serialization, dynamic linking and rebinding, and versioning. The function passing model, in contrast, 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.

MapReduce (Dean & Ghemawat, 2008), Dryad (Isard *et al*., 2007), and Apache Spark (Zaharia *et al*., 2010) are distributed systems for large-scale data processing, building on concepts from functional programming, such as higher-order functions. Ciel (Murray *et al*., 2011) is an execution engine for distributed data-flow programs which, like the function passing model, supports dynamic task dependencies and data-dependent control flow, thereby going beyond the capabilities of MapReduce and Dryad. Like Spark, MapReduce, and Dryad, Ciel supports transparent scaling, and fault tolerance is transparent. Our system shares several aspects of its design with Spark, such as lazy materialization of datasets, and serialization of computations, including closures, for remote shipping. In contrast to Spark's RDDs, silos in the function passing model are lower-level abstractions without transparent fault tolerance or transparent scaling. Instead, the goal of our programming model is to provide a foundation on top of which higher-level abstractions for distributed programming, like RDDs, can be built. An important focus of our work is the precise and detailed formalization of lineages and silos as a first step toward formal models of distributed systems like Spark. DryadLINQ (Yu *et al*., 2008) extends Dryad with a functional language, LINQ, for expressing transformations on distributed datasets that enables sophisticated optimizations, including those typically employed by databases. Like the function passing model, DryadLINQ enables the use of function closures within distributed computations. DryadLINQ uses dynamic code generation to ensure the serializability of these closures: captured variables are either eliminated by partial evaluation or serialized as resources shipped to machines in the cluster at runtime. In our system, serializability of closures (spores) is ensured at compile time using macros instead of using dynamic code generation. Like Spark, but unlike DryadLINQ and MapReduce, silos may be persisted in memory across multiple usages in our system. Nectar (Gunda *et al*., 2010) provides a caching service to improve the resource utilization of Dryad/DryadLINQ clusters. For this, Nectar identifies derived datasets by the computations that generate them, similar to lineages in the function passing model. Nectar has been shown to significantly improve space utilization as well as provide speed-ups via incremental computation and shared sub-computations. It would be interesting to investigate potential usages of the lineages of the functional passing model for similar purposes. In contrast to these above 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 finer-grained control over details such as fault handling. Rather than system building and experimental evaluation, our focus is on a precise formalization of the programming model, as well as the proof of preservation and progress properties.

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.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

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

7 Future work and conclusion

7.1 Ongoing and future work

Our ongoing efforts are three-fold; (a) we are working on a semantics and implementation of fault handling on top of the function passing model, (b) we are exploring approaches for memory reclamation, and (c) we are working to better understand the concerns of separate compilation.

7.1.1 Fault handling

The current implementation of the function passing model includes overloaded variants of function passing's primitive operations to enable 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 silo references.

What follows are illustrations of this ongoing work based on the running example introduced in Section 2.3.

Defining fault handlers. Fault handlers may be specified whenever the lineage of a silo reference is extended. For this purpose, the introduced `apply` primitive is overloaded. For example, consider the running example illustrated in Figure 5, but extended with a fault handler:

```
val persons: SiloRef[List[Person]] = ...
val vehicles: SiloRef[List[Vehicle]] = ...
// copy of ‘vehicles‘ on different host ‘h‘, see Section 2.4.1
val vehicles2 = SiloRef.fromLineage(h, vehicles)
val adults = persons.apply(spore { ps =>
  SiloRef.populate(currentHost, 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.apply(...)
}

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

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Importantly, in the apply call on the last line, in addition to computeOwners (vehicles), the regular spore argument of apply, 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 silo reference 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.apply(..). In this example, we specified computeOwners(vehicles2) as the failure handler; thus, in case vehicles has failed, the computation is retried using vehicles2 instead.

A limitation of this basic failure handling model is the fact that in the above example, the fall-back silo vehicles2 is defined up front using a specific host h. However, note that the computation DAG defined by a SiloRef can easily be materialized on *any* host using the SiloRef.fromLineage function shown in Section 2.4.1. Thus, assuming the existence of a function that randomly returns one of the healthy hosts in the cluster, say, getHealthyHost(), the above fault handler could be made more dynamic as follows:

```
val owners: SiloRef[List[(Person, Vehicle)]] =
  adults.apply(computeOwners(vehicles),
    spore {
      val localVehicles = vehicles
      (ps: List[Person]) =>
        val recoveredVehicles =
          SiloRef.fromLineage(getHealthyHost(), localVehicles)
        recoveredVehicles.apply(...)
    }
  )
```

In order to implement more flexible fault handling mechanisms, including strategies for straggler mitigation, additional information pertaining to the execution of (parts of) DAGs would need to be provided. For example, to mitigate stragglers, materializations could be initiated on alternative machines after a timeout. The specification (on the API level) and implementation of more flexible execution policies is left for future work.

7.1.2 Memory reclamation

We are exploring multiple approaches for memory reclamation. The first approach uses Java's *weak references* 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 & Odersky, 2010; Haller & Loiko, 2016). Here, SiloRefs are locally unique, and the programmer can explicitly declare a SiloRef as unused; the

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

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 between microservices.[13]

7.2 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, the model directly provides lineages which can form the basis for fault recovery, 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, are closely related to persistent data structures in functional programming. Therefore, we believe that fault tolerance based on lineages may benefit from further study by the functional programming community. We have also presented a formalization of the function passing model, providing an operational semantics and a type system for lineage-based distributed computation. While our formal model does not yet provide an approach to fault tolerance, our hope is that aspects of the model including lineages, silo references, and silo materialization can eventually form the basis of a complete formal treatment of lineage-based fault tolerance in future work. Finally, we have implemented our approach in and for Scala, and have shown that it is possible to support different popular patterns of distributed processing, such as batch processing with Apache Spark’s RDDs and MBrace’s cloud-based asynchronous tasks.

References

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

Agha, G. A., Mason, I. A., Smith, S. F. & Talcott, C. L. (1997) A foundation for actor computation. *J. Funct. Prog.* **7**(1), 1–72.

Apache. (2015) *Hadoop*. Available at: `http://hadoop.apache.org/`, accessed January 30, 2018.

Billings, J., Sewell, P., Shinwell, M. & Strniša, R. (2006) Type-safe distributed programming for OCaml. In Proceedings of the 2006 Workshop on ML. New York, NY, USA: ACM, pp. 20–31.

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

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Chambers, C., Raniwala, A., Perry, F., Adams, S, Henry, R. R., Bradshaw, R. & Weizenbaum, N. (2010) FlumeJava: Easy, efficient data-parallel pipelines. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York NY, USA: ACM. pp. 363–375.

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

Dzik, J., Palladinos, N., Rontogiannis, K., Tsarpalis, E. & Vathis, N. (2013) MBrace: Cloud computing with monads. In *PLOS@SOSP*, Harris, T. & Madhavapeddy, A. (eds). New York, NY, USA: ACM.

Elsman, M. (2005) Type-specialized serialization with sharing. In Proceedings of the Symposium on Trends in Functional Programming, pp. 47–62.

Epstein, J., Black, A. P. & Jones, S. L. P. (2011) Towards Haskell in the cloud. In Proceedings of the Haskell Symposium, pp. 118–129.

Germain, G. (2006) Concurrency oriented programming in Termite Scheme. In Proceedings of the 2006 ACM SIGPLAN workshop on Erlang, p. 20.

Gunda, P. K., Ravindranath, L., Thekkath, C. A., Yu, Y. & Zhuang, L. (2010) Nectar: Automatic management of data and computation in datacenters. In *OSDI*, Arpaci-Dusseau, R. H. & Chen, B. (eds). Berkeley, CA, USA: USENIX Association, pp. 75–88.

Haller, P. & Loiko, A. (2016) LaCasa: Lightweight affinity and object capabilities in Scala. In *OOPSLA*, Visser, E. & Smaragdakis, Y. (eds). New York, NY, USA: ACM, pp. 272–291.

Haller, P. & Odersky, M. (2009) Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.* **410**(2), 202–220.

Haller, P. & Odersky, M. (2010) Capabilities for uniqueness and borrowing. In Proceedings of the European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21–25, 2010, pp. 354–378.

Haller, P., Prokopec, A., Miller, H., Klang, V., Kuhn, R. & Jovanovic, V. (2012) Futures and promises. Available at: `http://docs.scala-lang.org/overviews/core/futures.html`, accessed January 30, 2018.

He, J., Wadler, P. & Trinder, P. (2014) Typecasting actors: From Akka to TAkka. In Proceedings of the 5th Scala Workshop. New York, NY, USA: ACM, pp. 23–33.

Herhut, S., Hudson, R. L., Shpeisman, T. & Sreeram, J. (2013) River Trail: A path to parallelism in JavaScript. In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA: ACM, pp. 729–744.

Hickey, R. (2008) The Clojure programming language. In Proceedings of the Dynamic Languages Symposium. New York, NY, USA: ACM, p. 1.

Isard, M., Budiu, M., Yu, Y., Birrell, A. & Fetterly, D. (2007) Dryad: Distributed data-parallel programs from sequential building blocks. In Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems. New York, NY, USA: ACM, pp. 59–72.

Kennedy, A. (2004) Pickler combinators. *J. Funct. Program.* **14**(6), 727–739.

Matsakis, N. D. (2012) Parallel closures: A new twist on an old idea. In Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism, Boehm, H.-J. & Ceze, L. (eds), *HotPar*. Berkeley, CA, USA: USENIX Association, p. 5.

Miller, H., Haller, P., Burmako, E. & Odersky, M. (2013) Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA: ACM, pp. 183–202.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Miller, H., Haller, P. & Odersky, M. (2014) Spores: A type-based foundation for closures in the age of concurrency and distribution. In Proceedings of the European Conference on Object-Oriented Programming. Berlin, Heidelberg, Germany: Springer-Verlag, pp. 308–333.

Milner, R., Parrow, J. & Walker, D. (1992) A calculus of mobile processes. *Inf. Comput*. **100**(1), 1–77.

Murphy, T. VII, Crary, K. & Harper, R. (2007) Type-safe distributed programming with ML5. In Proceedings of the International Symposium on Trustworthy Global Computing. Berlin, Heidelberg, Germany: Springer-Verlag, pp. 108–123.

Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A. & Hand, S. (2011) CIEL: A universal execution engine for distributed data-flow computing. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, Andersen, D. G. & Ratnasamy, S. (eds). Berkeley, CA, USA: USENIX Association.

NICTA. (2015) Scoobi. Available at: `https://github.com/nicta/scoobi`, accessed January 30, 2018.

Odersky, M. & Zenger, M. (2005) Scalable component abstractions. In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Johnson, R. E. & Gabriel, R. P. (eds). New York, NY, USA: ACM, pp. 41–57.

Odersky, M., Spoon, L. & Venners, B. (2010) *Programming in Scala*, 2nd edn. Walnut Creek, CA, USA: Artima.

Peyton Jones, S., Gordon, A. & Finne, S. (1996) Concurrent Haskell. In Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 295–308.

Pierce, B. C. (2002) *Types and Programming Languages*. Cambridge, MA, USA: MIT Press.

Rossberg, A., Le Botlan, D., Tack, G., Brunklaus, T. & Smolka, G. (2004) Alice through the looking glass. *Trends Funct. Program.* **5**, 79–96.

Sewell, P., Leifer, J. J., Wansbrough, K., Nardelli, F. Z., Allen-Williams, M., Habouzit, P. & Vafeiadis, V. (2005) Acute: High-level programming language design for distributed computation. In Proceedings of the ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA: ACM, pp. 15–26.

Shapiro, M., Preguiça, N. M., Baquero, C. & Zawirski, M. (2011) Conflict-free replicated data types. In *SSS*, Défago, X., Petit, F. & Villain, V. (eds), Lecture Notes in Computer Science, vol. 6976. Berlin, Heidelberg, Germany: Springer, pp. 386–400.

Twitter. (2015) Scalding. Available at: `https://github.com/twitter/scalding`, accessed January 30, 2018.

Typesafe. (2015) Akka. Available at: `http://akka.io/`, accessed January 30, 2018.

Waldo, J., Wyant, G., Wollrath, A. & Kendall, S. C. (1996) A note on distributed computing. In Proceedings of the International Workshop on Mobile Object Systems, Vitek, J., & Tschudin, C. (eds). Berlin, Heidelberg, Germany: Springer-Verlag, pp. 49–64.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K. & Currey, J. (2008) DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, Draves, Richard, & van Renesse, Robbert (eds). Berkeley, CA, USA: USENIX Association, pp. 1–14.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. & Stoica, I. (2010) Spark: Cluster computing with working sets. In Proceedings of the USENIX Workshop on Hot

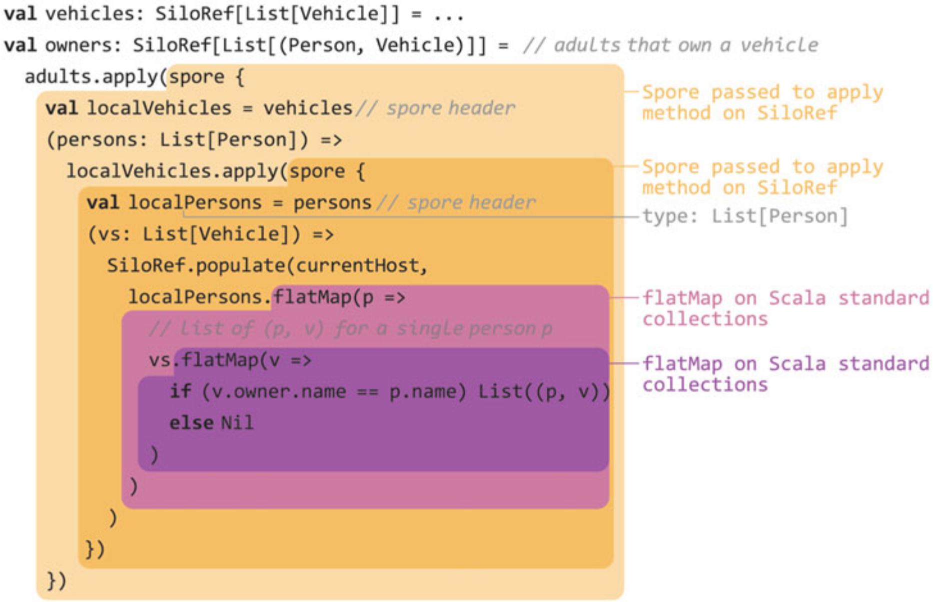
https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Topics in Cloud Computing. HotCloud'10. Berkeley, CA, USA: USENIX Association, pp. 10–10.

Appendix A. Illustrated listings

A.1 Matching vehicles and owners

Figure A1 shows an illustrated version of the listing in Figure 4.



```
val vehicles: SiloRef[List[Vehicle]] = ...
val owners: SiloRef[List[(Person, Vehicle)]] = // adults that own a vehicle
 adults.apply(spore {
  val localVehicles = vehicles// spore header
  (persons: List[Person]) =>
   localVehicles.apply(spore {
    val localPersons = persons // spore header
    (vs: List[Vehicle]) =>
     SiloRef.populate(currentHost,
      localPersons.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
       )
      )
     )
    })
  })
```

Spore passed to apply method on SiloRef
Spore passed to apply method on SiloRef
type: List[Person]
flatMap on Scala standard collections
flatMap on Scala standard collections

Fig. A1. Matching persons and vehicle owners using the `apply` combinator.

A.2 K-means clustering

Figure A2 shows an illustrated version of the listing of the k-means clustering example in Section 3.3.

Appendix B. Proofs

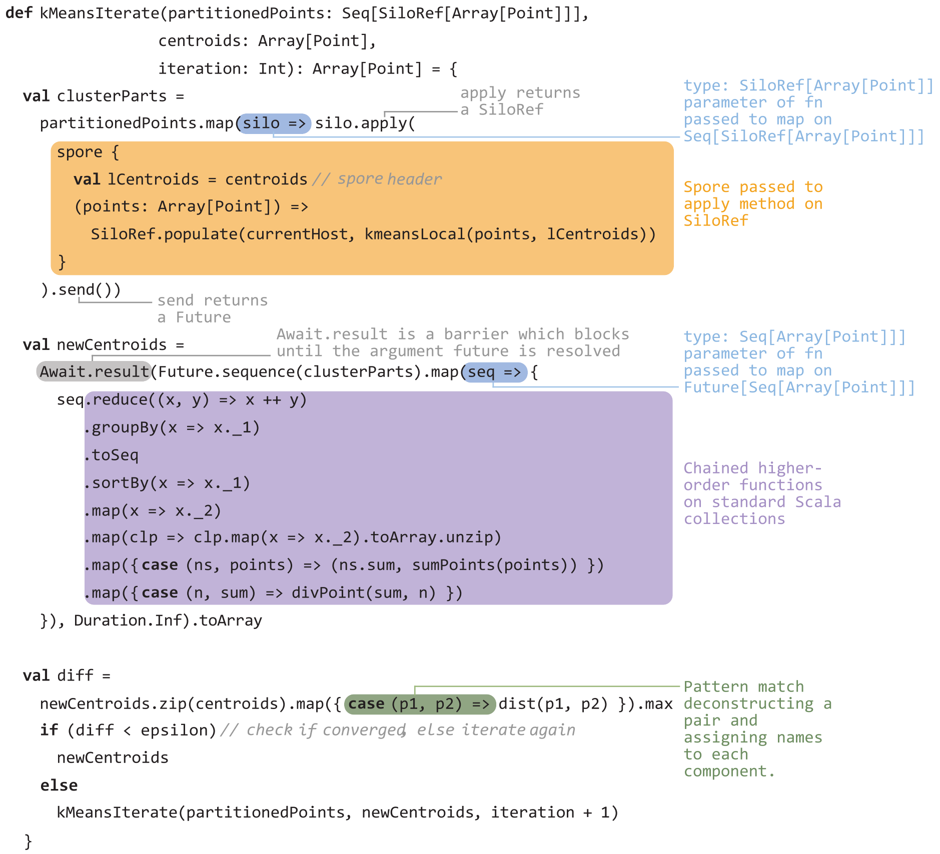
B.1 Proof of Theorem 4.1

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

Proof

By induction on the derivation of $\Gamma;\Sigma \vdash v : T$ with a case analysis of the last applied rule.

- Cases T-Int, T-Unit, and T-Host are trivial.
- Case T-SiloRef.



```
def kMeansIterate(partitionedPoints: Seq[SiloRef[Array[Point]]],
                  centroids: Array[Point],
                  iteration: Int): Array[Point] = {
  val clusterParts =
    partitionedPoints.map(silo => silo.apply(
      spore {
        val lCentroids = centroids // spore header
        (points: Array[Point]) =>
          SiloRef.populate(currentHost, kmeansLocal(points, lCentroids))
      }
    ).send())
  val newCentroids =
    Await.result(Future.sequence(clusterParts).map(seq => {
      seq.reduce((x, y) => x ++ y)
         .groupBy(x => x._1)
         .toSeq
         .sortBy(x => x._1)
         .map(x => x._2)
         .map(clp => clp.map(x => x._2).toArray.unzip)
         .map({ case (ns, points) => (ns.sum, sumPoints(points)) })
         .map({ case (n, sum) => divPoint(sum, n) })
    }), Duration.Inf).toArray

  val diff =
    newCentroids.zip(centroids).map({ case (p1, p2) => dist(p1, p2) }).max
    if (diff < epsilon) // check if converged, else iterate again
      newCentroids
    else
      kMeansIterate(partitionedPoints, newCentroids, iteration + 1)
}
```

apply returns a SiloRef

type: SiloRef[Array[Point]] parameter of fn passed to map on Seq[SiloRef[Array[Point]]]

Spore passed to apply method on SiloRef

send returns a Future

Await.result is a barrier which blocks until the argument future is resolved

type: Seq[Array[Point]] parameter of fn passed to map on Future[Seq[Array[Point]]]

Chained higher-order functions on standard Scala collections

Pattern match deconstructing a pair and assigning names to each component.

Fig. A2. Excerpt of an implementation of k-means clustering.

1. By the assumptions
   a. $\Gamma; \Sigma \vdash v : T$
   b. $serializable(T)$
2. By 1. (a) and T-SiloRef
   a. $v = \text{Ref}(l, h)$
   b. $T = \texttt{SiloRef}[T']$
   c. $\Sigma(id(l)) = T'$
   d. $\Sigma \vdash \text{Ref}(l, h)$
3. By 2. (a–d), and T-SiloRef, $\emptyset; \Sigma \vdash v : T$.

- Case T-Spore follows by S-Spore and the IH.

□

B.2 Proof of Theorem 5.2

Lemma B.1 (Weakening) *If* $\Gamma; \Sigma \vdash t : T$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : T'; \Sigma \vdash t : T$.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Proof

By induction on the derivation of $\Gamma; \Sigma \vdash t : T$. □

Lemma B.2 (Weakening of Store Typing)

1. *If* $\Gamma; \Sigma \vdash t : T$ *and* $\iota \notin dom(\Sigma)$ *then* $\Gamma; \Sigma' \vdash t : T$ *where* $\Sigma' = [\iota \mapsto T']\Sigma$.
2. *If* $\Sigma \vdash (t, \sigma)^h$ *and* $\iota \notin dom(\Sigma)$ *then* $\Sigma' \vdash (t, \sigma)^h$ *where* $\Sigma' = [\iota \mapsto T]\Sigma$.
3. *If* $\Sigma \vdash H$ *and* $\iota \notin dom(\Sigma)$ *then* $\Sigma' \vdash H$ *where* $\Sigma' = [\iota \mapsto T]\Sigma$.

Proof

Part 1: By induction on the derivation of $\Gamma; \Sigma \vdash t : T$. Part 2: By induction on the derivation of $\Sigma \vdash (t, \sigma)^h$. Part 3: By induction on the derivation of $\Sigma \vdash H$. □

Lemma B.3 (Process) *If* $\Sigma \vdash \sigma$, $\Sigma \vdash m$, *and* $process(h, m, \sigma) = (t, M, \sigma')$ *then* $\emptyset; \Sigma' \vdash t : T$ *for some* T, $\Sigma' \vdash M$, *and* $\Sigma' \vdash \sigma'$ *for some* $\Sigma' \supseteq \Sigma$.

Proof

- Case Proc-Req.
  1. By the assumptions
     a. $\Sigma \vdash \sigma$
     b. $\Sigma \vdash m$
     c. $process(h, m, \sigma) = (t, M, \sigma')$
  2. By Proc-Req
     a. $m = \mathrm{Req}(h', r, \iota)$
     b. $r = \mathrm{Ref}(l, h)$
     c. $\sigma(id(l)) = (v, P)$
     d. $M = \{h' \leftarrow \mathrm{Res}(\iota, v)\}$
     e. $\sigma' = \mathrm{consume}(id(l), P, \sigma)$
     f. $t = \texttt{unit}$
  3. Define $\Sigma' := \Sigma$.
  4. By 2. (e) and Definition 4.6, $dom(\sigma') \subseteq dom(\sigma)$.
  5. By 1. (a), 3., 4., and WF-Store1-3, $\Sigma' \vdash \sigma'$.
  6. By 1. (b), 2. (a,b), and WF-Req
     a. $\Sigma(id(l)) = \Sigma(\iota)$
     b. $\Sigma \vdash r$
  7. Define $T := \Sigma(id(l))$.
  8. By 1. (a), 2. (c), 7., and WF-Store2, $\emptyset; \Sigma \vdash v : T$.
  9. By 6. (a), 7., 8., and WF-Res, $\Sigma \vdash \mathrm{Res}(\iota, v)$.
  10. By 2. (d), 9., and WF-Messages, $\Sigma \vdash M$.
  11. By 2. (f), T-Unit, and Lemma B.2, $\emptyset; \Sigma \vdash t : T'$ for some T'.
  12. 3., 5., 10., and 11. close this case.
- Cases Proc-ReqMat1, Proc-ReqMat2, and Proc-ReqParent follow analogously.
- Case Proc-ReqApply.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

1. By the assumptions
   a. $\Sigma \vdash \sigma$
   b. $\Sigma \vdash m$
   c. $process(h, m, \sigma) = (t, M, \sigma')$
2. By Proc-ReqApply
   a. $m = \text{Req}(h', r, \iota)$
   b. $r = \text{Ref}(l, h)$
   c. $l = \text{Applied}(\iota', l', p)$
   d. $\sigma(id(l')) = (v, P)$
   e. $t = \texttt{respond}(h', \iota, \texttt{await}(\texttt{send}(p\ v)))$
   f. $\sigma' = \text{consume}(id(l'), P, \sigma)$
   g. $M = \emptyset$
3. By 1. (b), 2. (a,b), and WF-Req
   a. $\Sigma(id(l)) = \Sigma(\iota)$
   b. $\Sigma \vdash r$
4. By 2. (b,c), 3. (b), WF-Ref, and WF-Lin2
   a. $\Sigma(\iota') = T$
   b. $\Sigma(id(l')) = T'$
   c. $\exists \Gamma.\ \Gamma; \Sigma \vdash p : T' \Rightarrow \texttt{SiloRef}[T]\ \{\ldots\}$
   d. $\Sigma \vdash l'$
5. By 1. (a), 2. (d), 4. (b), and WF-Store2, $\emptyset; \Sigma \vdash v : T'$.
6. By 4. (c), T-Spore, Def. *serializable*, and Lemma 4.1, $\emptyset; \Sigma \vdash p : T' \Rightarrow \texttt{SiloRef}[T]\ \{\ldots\}$.
7. By 5., 6., and T-AppSpore, $\emptyset, \Sigma \vdash p\ v : \texttt{SiloRef}[T]$.
8. By 7. and T-Send, $\emptyset, \Sigma \vdash \texttt{send}(p\ v) : \texttt{Future}[T]$.
9. By 8. and T-Await, $\emptyset, \Sigma \vdash \texttt{await}(\texttt{send}(p\ v)) : T$.
10. By 3. (a), 4. (a), and Def. 4.2, $\Sigma(\iota) = T$ and thus by T-Ident, $\emptyset; \Sigma \vdash \iota : \texttt{Future}[T]$.
11. By 2. (e), 9., 10., and T-Respond, $\emptyset, \Sigma \vdash t : \texttt{Unit}$.
12. By 2. (g) and WF-Messages-Emp, $\Sigma \vdash M$.
13. By 2. (f) and Def. 4.6, $dom(\sigma') \subseteq dom(\sigma)$.
14. By 1. (a), 13., and WF-Store1-3, $\Sigma \vdash \sigma'$.
15. 11., 12., and 14. close this case.

- Cases Proc-ReqPersist and Proc-Res follow analogously.

□

Theorem (Subject Reduction)

1. *If* $\Gamma; \Sigma \vdash t : T$, $\Sigma \vdash \sigma$, *and* $t \mid \sigma \to^h t' \mid \sigma'$ *then* $\Gamma; \Sigma' \vdash t' : T$ *and* $\Sigma' \vdash \sigma'$ *for some* $\Sigma' \supseteq \Sigma$.
2. *If* $\Sigma \vdash H \mid M$ *and* $H \mid M \twoheadrightarrow H' \mid M'$ *then* $\Sigma' \vdash H' \mid M'$ *for some* $\Sigma' \supseteq \Sigma$.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Proof
Part 1: by induction on the derivation of $t \mid \sigma \rightarrow^h t' \mid \sigma'$ with case analysis of the last applied rule.

- Case R-AppAbs.
  1. By the assumptions
     a. $\Gamma;\Sigma \vdash t : T$
     b. $\Sigma \vdash \sigma$
     c. $t \mid \sigma \rightarrow^h t' \mid \sigma'$
  2. By R-AppAbs
     a. $t = E[((x : T') \Rightarrow t'')\ v']$
     b. $t' = E[[x \mapsto v']t'']$
     c. $\sigma' = \sigma$
  3. By 1. (a) and 2. (a), $\Gamma;\Sigma \vdash ((x : T') \Rightarrow t'')\ v' : T''$.
  4. By 3. and T-App,
     a. $\Gamma;\Sigma \vdash ((x : T') \Rightarrow t'') : T' \Rightarrow T''$
     b. $\Gamma;\Sigma \vdash v' : T'$
  5. By 4. (a) and T-Abs, $\Gamma, x : T';\Sigma \vdash t'' : T''$.
  6. By 4. (b), 5., and Lemma 5.1, $\Gamma;\Sigma \vdash [x \mapsto v']t'' : T''$.
  7. By 1. (a), 2. (a–b), 3., and 6., $\Gamma;\Sigma \vdash t' : T$.
  8. 2. (c) and 7. close this case.
- Cases R-IntOp, R-AppSpore, and R-Await follow analogously.
- Case R-Apply.
  1. By the assumptions
     a. $\Gamma;\Sigma \vdash t : T$
     b. $\Sigma \vdash \sigma$
     c. $t \mid \sigma \rightarrow^h t' \mid \sigma'$
  2. By R-Apply
     a. $t = E[\texttt{apply}(r, p)]$
     b. $r = \text{Ref}(l, h')$
     c. $t' = E[r']$
     d. $r' = \text{Ref}(l', h')$
     e. $l' = \text{Applied}((h, i), l, p)$ where i fresh
     f. $\sigma' = \sigma$
  3. By 1. (a) and 2. (a), $\Gamma;\Sigma \vdash \texttt{apply}(r, p) : \hat{T}$.
  4. By 3. and T-Apply,
     a. $\hat{T} = \texttt{SiloRef}[T']$
     b. $\Gamma;\Sigma \vdash r : \texttt{SiloRef}[T'']$
     c. $\Gamma;\Sigma \vdash p : T'' \Rightarrow \texttt{SiloRef}[T']\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ \}$

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

5. By 2. (b), 4. (b), T-SiloRef, and WF-Ref
   a. $\Sigma(id(l)) = T''$
   b. $\Sigma \vdash r$
   c. $\Sigma \vdash l$
   d. $h' \in \mathscr{H}$
6. Define $\Sigma' := [(h,i) \mapsto T']\Sigma$.
7. By 2. (d–e), 4. (c), 5. (a–d), 6., WF-Lin2, and WF-Ref, $\Sigma' \vdash r'$.
8. By 2. (d–e), 6., 7., and T-SiloRef, $\Gamma;\Sigma' \vdash r' : \texttt{SiloRef}[T']$.
9. By 2. (e), 3., 4. (a), 6., and part 1 of Lemma B.2, $\Gamma;\Sigma' \vdash \texttt{apply}(r,p) : \texttt{SiloRef}[T']$.
10. By 1. (a), 2. (e), 6., and part 1 of Lemma B.2, $\Gamma;\Sigma' \vdash t : T$.
11. By 2. (a,c), 8., 9., and 10., $\Gamma;\Sigma' \vdash t' : T$.
12. By 1. (b) and 2.f), $\Sigma \vdash \sigma'$.
13. By 6., $\Sigma' \supseteq \Sigma$.
14. By 12., 13., and WF-Store3, $\Sigma' \vdash \sigma'$.
15. 11., 13., and 14. close this case.

- Cases R-Persist and R-Unpersist follow analogously.

Part 2: by induction on the derivation of $H \mid M \twoheadrightarrow H' \mid M'$ with case analysis of the last applied rule.

- Case R-Local.

1. By the assumptions
   a. $\Sigma \vdash H \mid M$
   b. $H \mid M \twoheadrightarrow H' \mid M'$
2. By R-Local
   a. $H = \{(t,\sigma)^h\} \cup H''$
   b. $H' = \{(t',\sigma')^h\} \cup H''$
   c. $t \mid \sigma \rightarrow^h t' \mid \sigma'$
   d. $M' = M$
3. By 1. (a) and WF-Config, $\Sigma \vdash H$.
4. By 2. (a), 3., and WF-Host2
   a. $\Sigma \vdash (t,\sigma)^h$
   b. $\Sigma \vdash H''$
5. By 4. (a) and WF-HostConfig
   a. $\Sigma \vdash \sigma$
   b. $\Gamma;\Sigma \vdash t : T$ for some Γ
6. By 2. (c), 5. (a,b), and part 1
   a. $\Gamma;\Sigma' \vdash t' : T$
   b. $\Sigma' \vdash \sigma'$ for some $\Sigma' \supseteq \Sigma$
7. By 6. (a,b) and WF-HostConfig, $\Sigma' \vdash (t',\sigma')^h$.

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

8. By 4. (b), 6. (b), and part 3 of Lemma B.2, $\Sigma' \vdash H''$.
9. By 2. (b), 7., 8., and WF-Host2, $\Sigma' \vdash H'$.
10. By 1. (a), 2. (d), 9., WF-Config, and WF-Messages, $\Sigma' \vdash H' \mid M'$.

- Case R-Send.
   1. By the assumptions
      a. $\Sigma \vdash H \mid M$
      b. $H \mid M \twoheadrightarrow H' \mid M'$
   2. By R-Send
      a. $H = \{(E[\mathtt{send}(r)], \sigma)^h\} \cup H''$
      b. $H' = \{(E[id(l)], \sigma)^h\} \cup H''$
      c. $r = \mathrm{Ref}(l, h')$
      d. $m = \mathrm{Req}(h, r, id(l))$
      e. $M' = M \uplus \{h' \leftarrow m\}$
   3. By 1. (a) and WF-Config
      a. $\Sigma \vdash H$
      b. $\Sigma \vdash M$
   4. By 2. (a), 3. (a), and WF-Host2
      a. $\Sigma \vdash (E[\mathtt{send}(r)], \sigma)^h$
      b. $\Sigma \vdash H''$
   5. By 4. (a) and WF-HostConfig
      a. $\Sigma \vdash \sigma$
      b. $\Gamma; \Sigma \vdash E[\mathtt{send}(r)] : T$ for some Γ
   6. By 5. (b), $\Gamma; \Sigma \vdash \mathtt{send}(r) : \hat{T}$.
   7. By 6. and T-Send
      a. $\hat{T} = \mathtt{Future}[T'']$
      b. $\Gamma; \Sigma \vdash r : \mathtt{SiloRef}[T'']$
   8. By 2. (c), 7. (b), and T-SiloRef
      a. $\Sigma(id(l)) = T''$
      b. $\Sigma \vdash r$
   9. By 8. (a) and T-Ident, $\Gamma; \Sigma \vdash id(l) : \mathtt{Future}[T'']$.
   10. By 5. (b), 6., 7. (a), and 9., $\Gamma; \Sigma \vdash E[id(l)] : T$.
   11. By 5. (a), 10., and WF-HostConfig, $\Sigma \vdash (E[id(l)], \sigma)^h$.
   12. By 4. (b), 11., and WF-Host2, $\Sigma \vdash H'$.
   13. By 2. (c,d), 8. (b), and WF-Req, $\Sigma \vdash m$.
   14. By 2. (c,e), 3. (b), 8. (b), 13., WF-Ref, and WF-Messages, $\Sigma \vdash M'$.
   15. By 12., 14., and WF-Config, $\Sigma \vdash H' \mid M'$.
- Cases R-Populate and R-Respond follow analogously.
- Case R-Process.
   1. By the assumptions

   a. $\Sigma \vdash H \mid M$

   b. $H \mid M \twoheadrightarrow H' \mid M'$

2. By R-Process

   a. $H = \{(E[\mathtt{await}(\imath)], \sigma)^h\} \cup H''$

   b. $H' = \{(E[t \ ; \ \mathtt{await}(\imath)], \sigma')^h\} \cup H''$

   c. $M' = \hat{M} \uplus M''$

   d. $\mathit{process}(h, m, \sigma) = (t, M'', \sigma')$

   e. $M = \hat{M} \uplus \{h \leftarrow m\}$

3. By 1. (a) and WF-Config

   a. $\Sigma \vdash H$

   b. $\Sigma \vdash M$

4. By 2. (a), 3. (a), and WF-Host2

   a. $\Sigma \vdash (E[\mathtt{await}(\imath)], \sigma)^h$

   b. $\Sigma \vdash H''$

5. By 4. (a) and WF-HostConfig

   a. $\Sigma \vdash \sigma$

   b. $\Gamma; \Sigma \vdash E[\mathtt{await}(\imath)] : T$ for some Γ

6. By 2. (e), 3. (b), and WF-Messages, $\Sigma \vdash m$.
7. By 2. (d), 5. (a), 6., and Lemma B.3 (Process), $\exists \Sigma', T'$ such that

   a. $\emptyset; \Sigma' \vdash t : T'$

   b. $\Sigma' \vdash M''$

   c. $\Sigma' \vdash \sigma'$

   d. $\Sigma' \supseteq \Sigma$

8. By 5. (b), 7. (d), and part 1 of Lemma B.2, $\Gamma; \Sigma' \vdash E[\mathtt{await}(\imath)] : T$.
9. By 7. (a) and 8., $\Gamma; \Sigma' \vdash E[t \ ; \ \mathtt{await}(\imath)] : T$.
10. By 7. (c), 9., and WF-HostConfig, $\Sigma' \vdash (E[t \ ; \ \mathtt{await}(\imath)], \sigma')^h$.
11. By 4. (b), 7. (d), and part 3 of Lemma B.2, $\Sigma' \vdash H''$.
12. By 2. (b), 10., 11., and WF-Host2, $\Sigma' \vdash H'$.
13. By 3. (b), 7. (d), WF-Res, WF-Req, WF-Ref, and WF-Messages, $\Sigma' \vdash M$.
14. By 2. (c), 2. (e), 7. (b), 13., and WF-Messages, $\Sigma' \vdash M'$.
15. By 12., 14., and WF-Config, $\Sigma' \vdash H' \mid M'$.

- Case R-Process-Val.

1. By the assumptions

   a. $\Sigma \vdash H \mid M$

   b. $H \mid M \twoheadrightarrow H' \mid M'$

2. By R-Process-Val

   a. $H = \{(v, \sigma)^h\} \cup H''$

   b. $H' = \{(t, \sigma')^h\} \cup H''$

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

c. $M' = \hat{M} \uplus M''$

d. $process(h, m, \sigma) = (t, M'', \sigma')$

e. $M = \hat{M} \uplus \{h \leftarrow m\}$

3. By 1. (a) and WF-Config

a. $\Sigma \vdash H$

b. $\Sigma \vdash M$

4. By 2. (a), 3. (a), and WF-Host2

a. $\Sigma \vdash (v, \sigma)^h$

b. $\Sigma \vdash H''$

5. By 4. (a) and WF-HostConfig

a. $\Sigma \vdash \sigma$

b. $\Gamma; \Sigma \vdash v : T$ for some Γ

6. By 2. (e), 3. (b), and WF-Messages, $\Sigma \vdash m$.
7. By 2. (d), 5. (a), 6., and Lemma B.3 (Process), $\exists \Sigma', T'$ such that

a. $\emptyset; \Sigma' \vdash t : T'$

b. $\Sigma' \vdash M''$

c. $\Sigma' \vdash \sigma'$

d. $\Sigma' \supseteq \Sigma$

8. By 7. (a), 7. (c), and WF-HostConfig, $\Sigma' \vdash (t, \sigma')^h$.
9. By 4. (b), 7. (d), and part 3 of Lemma B.2, $\Sigma' \vdash H''$.
10. By 2. (b), 8., 9., and WF-Host2, $\Sigma' \vdash H'$.
11. By 3. (b), 7. (d), WF-Res, WF-Req, WF-Ref, and WF-Messages, $\Sigma' \vdash M$.
12. By 2. (c), 2. (e), 7. (b), 11., and WF-Messages, $\Sigma' \vdash M'$.
13. By 10., 12., and WF-Config, $\Sigma' \vdash H' \mid M'$.

□

B.3 Proof of Theorem 5.5

Lemma B.4 (Responsive Population) *Let $\Sigma \vdash H \mid M \mid R$ and $H = \{(E[t], \sigma)^h\} \uplus \hat{H}$. Then $\forall h' \in hosts(H)$: $\{(E[\mathtt{send}(r)], \sigma)^h\} \uplus \hat{H} \mid M \uplus \{h' \leftarrow m\} \twoheadrightarrow^* H' \mid M' \uplus \{h \leftarrow m\}$ after a finite number of reduction steps where $r = Ref(l, h')$, $l = Mat(\imath)$ for some $\imath$, and $m = Res(\imath, v)$.*

Proof Sketch
By R-Send, $\{(E[\mathtt{send}(r)], \sigma)^h\} \uplus \hat{H} \mid M \uplus \{h' \leftarrow m\} \twoheadrightarrow \{(E[\imath], \sigma)^h\} \uplus \hat{H} \mid M \uplus \{h' \leftarrow m\} \uplus \{h' \leftarrow m'\}$ where $m' = \mathrm{Req}(h, r, \imath)$. By Def. 5.1 message m is processed by h' after a finite number of reduction steps. As a result, the store of h' is updated with the mapping $\imath \mapsto v$. After another finite number of reduction steps, h' processes message m'. By R-Process, R-Process-Val, and Proc-ReqMat1, the resulting multiset of messages includes $\{h \leftarrow \mathrm{Res}(\imath, v)\}$ as required. □

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press

Lemma B.5 (Responsive Apply) *Let* $\Sigma \vdash H \mid M \mid R$ *such that* $Responsive(H, M, R)$.

If $H = \{(E[\mathit{apply}(r, p)], \sigma)^h\} \uplus \hat{H}$ *then* $H \mid M \mid R \twoheadrightarrow H' \mid M' \mid R'$ *and* $Responsive(H', M', R')$.

Proof

1. By R-Local and R-Apply
   a. $E[\texttt{apply}(r, p)] \mid \sigma \rightarrow^h E[r'] \mid \sigma \mid \{r'\}$
   b. $r' = \mathrm{Ref}(l', h')$
   c. $r = \mathrm{Ref}(l, h')$
   d. $l' = \mathrm{Applied}((h, i), l, p)$ where i fresh
   e. $H' = \{(E[r'], \sigma)^h\} \uplus \hat{H}$
   f. $M' = M$
   g. $R' = R \uplus \{r'\}$
2. By Lemma 5.4, $Responsive(H', M', R)$.
3. Consider $H' \mid M' \uplus \{h' \leftarrow m\} \mid R'$ where $m = \mathrm{Req}(h'', r', \imath)$.
4. By Def. 5.1 $H' \mid M' \uplus \{h' \leftarrow m\} \mid R' \twoheadrightarrow^* H_p \mid M_p \mid R_p$ such that $(E_p[\texttt{await}(\imath_p)], \sigma_p)^{h'} \in H_p \vee (E_p[v_p], \sigma_p)^{h'} \in H_p$ and $\{h' \leftarrow m\} \in M_p$.
5. There are two cases. *Case 1:* $id(l) \notin dom(\sigma')$. In this case, $H_p \mid M_p \mid R_p$ can be reduced according to R-Process (or R-Process-Val) and Proc-ReqParent. As a result, h' reduces $\texttt{send}(\mathrm{Ref}(l, h'))$. By R-Send, $\mathrm{Ref}(l, h') \in R$, and Lemma 5.4, $H_p \mid M_p \mid R_p \twoheadrightarrow^* H_r \mid M_r \uplus \{h' \leftarrow \mathrm{Res}(id(l), v)\} \mid R_r$ after a finite number of reduction steps. By Def. 5.1 and Proc-Res, $H_r \mid M_r \uplus \{h' \leftarrow \mathrm{Res}(id(l), v)\} \mid R_r \twoheadrightarrow^* H'' \mid M'' \mid R''$ after a finite number of reduction steps, such that
   a. $H'' = \{(E'[t'], \sigma')^{h'}\} \cup H_3$ where $t' = v'$ or $t' = \texttt{await}(\imath')$
   b. $M'' = \{h' \leftarrow m\} \uplus M_3$
   c. $\sigma'(id(l)) = v$
   d. $\Sigma'' \vdash H'' \mid M'' \mid R''$

   Case 2: $\sigma'(id(l)) = v$. In this case, $H'' = H_p$, $M'' = M_p$, and $R'' = R_p$.
6. By 5.a–c, R-Process, R-Process-Val, and Proc-ReqApply
   a. $process(h', m, \sigma') = (t'', \emptyset, \sigma')$
   b. $t'' = \texttt{respond}(h'', \imath, \texttt{await}(\texttt{send}(p\ v)))$
   c. $H'' \mid M'' \mid R'' \twoheadrightarrow H_3 \mid M_3 \mid R''$
   d. $H_3 = \{(E'[t''\ ;\ t'], \sigma')^{h'}\} \cup H_4$
7. By 5 . (d), 6. (c), and Theorem 5.2 (Subject Reduction), $\Sigma_3 \vdash \{(E'[t''\ ;\ t'], \sigma')^{h'}\} \cup H_4 \mid M_3 \mid R''$ for some $\Sigma_3 \supseteq \Sigma''$.
8. By 7., WF-Config, WF-Host2, and WF-HostConfig
   a. $\Sigma_3 \vdash \sigma'$
   b. $\Gamma_3; \Sigma_3 \vdash E'[t''\ ;\ t'] : T_3$ for some Γ_3, T_3
9. By 6. (b), 8. (b), and the type rules
   a. $\Gamma_4; \Sigma_3 \vdash p : T' \Rightarrow \texttt{SiloRef}[T]\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ \}$ for some Γ_4
   b. $p = \texttt{spore}\ \{\ \overline{x : T = v}\ ; (x : T') \Rightarrow t\ \}$
10. By 9. (a,b), and T-Spore
    a. $\overline{x : T}, x : T'; \emptyset \vdash t : \texttt{SiloRef}[T]$

b. $\forall S \in \overline{T}, \texttt{SiloRef}[T].\ \mathit{serializable}(S)$

11. By the type rules, derivation 10. (a) does not contain applications of T-SiloRef or T-Ident. By 10. (b) spore p does not capture futures. Thus, any occurrence of $\texttt{await}(\hat{\imath})$ within $p\ v$ is preceded by a reduction of a corresponding `send` resulting in future $\hat{\imath}$. By Lemma 5.4 (Responsiveness), $\mathit{Responsive}(H_3, M_3, R)$. There are two cases.
Case 1: $d = 0$. Then p does not contain a nested `apply` invocation. Therefore, by Lemma B.4, $p\ v$ reduces to a value r'' after a finite number of reduction steps. Since either $r'' \in R$ or r'' is newly populated, $\texttt{send}(r'')$ results in a response $h' \leftarrow \mathrm{Res}(\mathit{id}(r''), v'')$ after a finite number of reduction steps. This enables h' to reduce $\texttt{respond}(h'', \imath, v'')$ which concludes this case.
Case 2: $d > 0$. The depth of nested `apply`s of term $p\ v$ is less than the depth of the term $\texttt{apply}(r, p)$. Therefore, by the induction hypothesis, reductions of nested `apply` invocations within $p\ v$ result in responsive configurations. By Lemma B.4, $p\ v$ reduces to a value r'' after a finite number of reduction steps. Since either $r'' \in R$, or r'' is newly populated, or r'' is the result of a nested `apply` invocation, $\texttt{send}(r'')$ results in a response $h' \leftarrow \mathrm{Res}(\mathit{id}(r''), v'')$ after a finite number of reduction steps. This enables h' to reduce $\texttt{respond}(h'', \imath, v'')$ which concludes this case.

□

Theorem (Finite Materialization) *Let* $\Sigma \vdash H \mid M \mid R$ *such that* $\mathit{Responsive}(H, M, R)$. *If* $H \mid M \mid R \twoheadrightarrow H' \mid M' \mid R'$ *then* $\mathit{Responsive}(H', M', R')$.

Proof
Corollary of Lemmas 5.4, B.4, and B.5. □

https://doi.org/10.1017/S0956796818000035 Published online by Cambridge University Press