Serverless Cloud Computing Beyond FaaS: Programming Models and Abstractions

Philipp Haller

KTH Royal Institute of Technology
Stockholm, Sweden

2nd Vienna Software Seminar (VSS)
Vienna, Austria, Aug 29, 2019

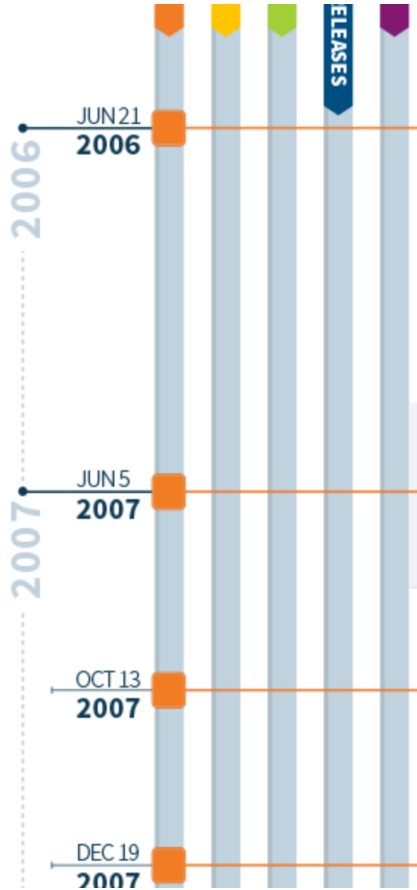
Background

- 2005-2014 Scala language team
  - 2012-2014 Typesafe, Inc.

- Co-author Scala language specification
- 2019: ACM SIGPLAN Programming Languages Software Award for Scala

Core contributors:
Martin Odersky, Adriaan Moors, Aleksandar Prokopec, Heather Miller, Iulian Dragos, Nada Amin, Philipp Haller, Sebastien Doeraene, Tiark Rompf

Scala Actors and Akka



Philipp Haller creates Scala Actors (the original standard library actors). His work becomes a major influence to Akka and the main reason for Jonas Bonér to choose Scala as the platform for Akka.

First commit: https://github.com/scala/scala/commit/0d8b14c6055e76c0bff3b65d0f428d711abe1f5a

Scala Actors used, e.g., in core message queue system of Twitter:

CONFERENCE PAPER

Actors that Unify Threads and Events

Haller, Philipp; Odersky, Martin
Editors: Vitek, Jan; Murphy, Amy L.

Presented at: International Conference on Coordination Models and Languages, Paphos, Cyprus, 5-8 June 2007
Published in: Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION), p. 171-190
Series: Lecture Notes in Computer Science (LNCS) 4467
: Springer, 2007

There is an impedance mismatch between message-passing concurrency and virtual machines, such as the JVM. VMs usually map their threads to heavyweight OS processes. Without a lightweight process abstraction, users are often forced to write parts of concurrent applications in an event-driven

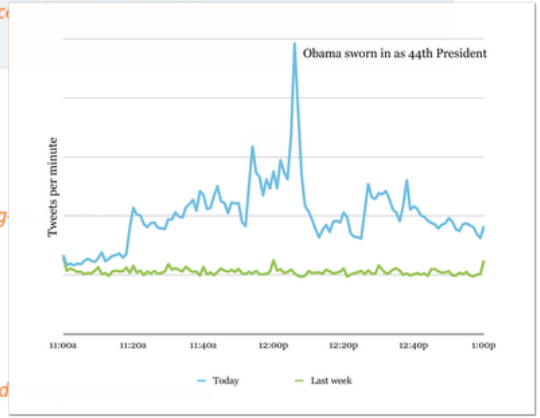
Philipp Haller publishes his influential paper on Scala's Actors; **'Actors that Unify Threads and Events'.**
http://infoscienc

ERLANG

Jonas gets seriously interested in **Erlang.**
Blog post:
http://jonasboner.com/2007/10/30/interview-with-joe-armstrong

Scala

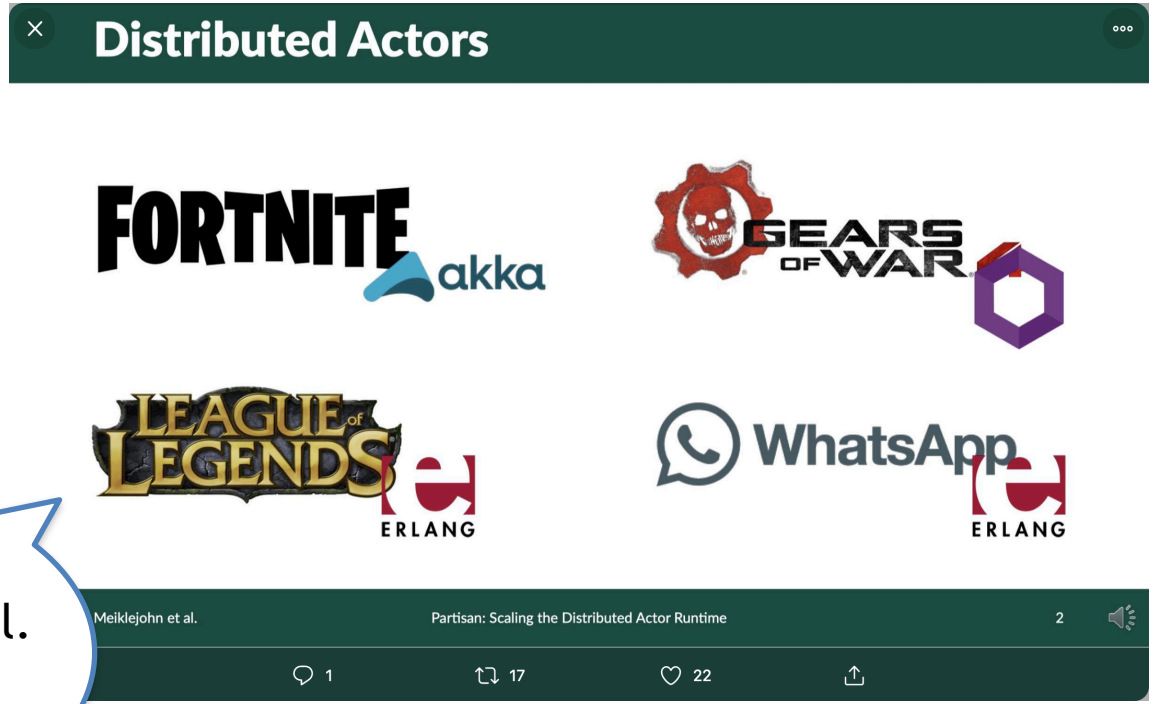
Jonas starts tinkering with **Scala Actors.**
Blog post:
'19/hotswap-cod



https://www.lightbend.com/akka-five-year-anniversary

Side remark

The use of actors is common in industry



Slide from: Meiklejohn et al. “Partisan” at USENIX ATC ‘19

Ongoing work

- **Type systems**
  - *LaCasa:* lightweight affine types and object capabilities in Scala [Haller & Loiko 2016]
    - Static reasoning about capabilities and resources
  - *Types for safe distribution*
    - Closures [Miller et al. 2014], eventual consistency [Zhao & Haller 2019]
- **Concurrent and distributed programming**
  - Deterministic concurrency [Haller et al. 2016], function passing [Haller et al. 2018], asynchronous streams [Haller & Miller 2019]

Cloud computing

- Public cloud infrastructure **integral part** of numerous large-scale, commercial applications.
  - Amazon Web Services introduced > 12 years ago.
- Support for **enterprise services**: databases, queueing systems, object storage, etc.

So, cloud computing is now essentially a legacy enterprise service, right?

Unused potential?

So, cloud computing is now essentially a legacy enterprise service, right?

The cloud is…

"the biggest assemblage of data capacity and distributed computing power ever available to the general public, managed as a service." [1]

[1] Hellerstein et al. *Serverless Computing: One Step Forward, Two Steps Back.* CIDR 2019

Functions-as-a-Service (FaaS)

What is Serverless Computing?

Example event: "a commit was pushed to branch X of repository Y."

- Developers upload their code (functions) to the cloud.
- Cloud platform executes these functions in response to **events**.
- No need for operating or provisioning servers.
- Users only pay for compute resources used when their code is executed.
- Function execution is **autoscaling**: execution scales according to demand.

"Serverless"

Pay per use!

"Where is the catch?"

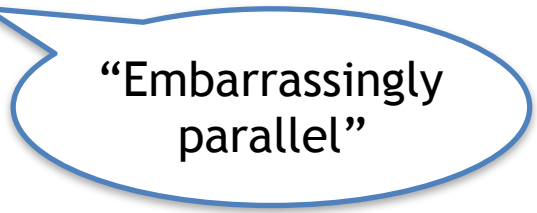
Important restrictions

- Functions are **stateless**.
  - Must use **external storage** for any data/state that needs to survive multiple function executions.
- Function **execution duration limited**.
  - AWS Lambda: all function executions must complete within 300 seconds.

Which use cases are well-supported?

Depending on the patterns of function invocation [1]:

- **Fully independent** function invocations.
  - Scale up or down on demand:
    “invocations never wait for each other”
- **Event-driven workflows** connected via queueing systems or object stores.
  - High latency due to task handling and state management.



Key limitations

- **Communication through slow storage**:
  Functions not directly network-addressable,
  all communication via external services
- Functions are **short-lived**
  - Cannot service repeated requests via internal caches.
  - Cannot implement general distributed systems.
- I/O bottlenecks

ication latency

invoking a
no-op Lambda function
on a 1KB argument

1KB network
message roundtrip

Latency of "communicating" 1KB:

	Func. Invoc. (1KB)	Lambda I/O (S3)	Lambda I/O (DynamoDB)	EC2 I/O (S3)	EC2 I/O (DynamoDB)	EC2 NW (0MQ)
Latency	303ms	108ms	11ms	106ms	11ms	290μs
Compared to best	1,045×	372×	37.9×	365×	37.9×	1×

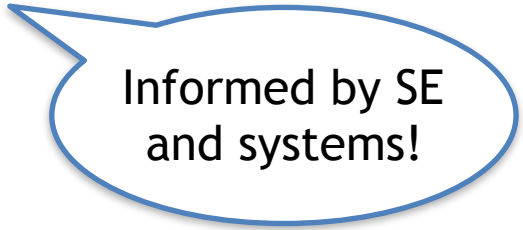
write+read from
"long-running" function

[1] Hellerstein et al. *Serverless Computing: One Step Forward, Two Steps Back.* CIDR 2019

Back to the roots

Re-thinking distributed systems building

- Re-think **fundamental building blocks**.
  - Improve distributed systems stack.
- Devise and study **programming models, languages, and systems**.



Challenge

Programming model

- From data-shipping to **function-shipping**
  - Enable entirely different classes of applications: big data, ML model training.
- **Principled fault-tolerance** based on lineages.
  - Guarantee properties related to fault tolerance.
  - *Example:* program execution should never "get stuck" if at most N-1 out of 2N replicas fail.
  - Requires **foundations for fault-tolerant programming**.

Distributed programming with functional lineages a.k.a. function passing

- New data-centric programming model for functional processing of distributed data.
- **Key ideas:**
  - Utilize lineages for fault injection ***and*** recovery
  - Provide lineages by programming abstractions
  - Keep data stationary (if possible), send functions

Introducing

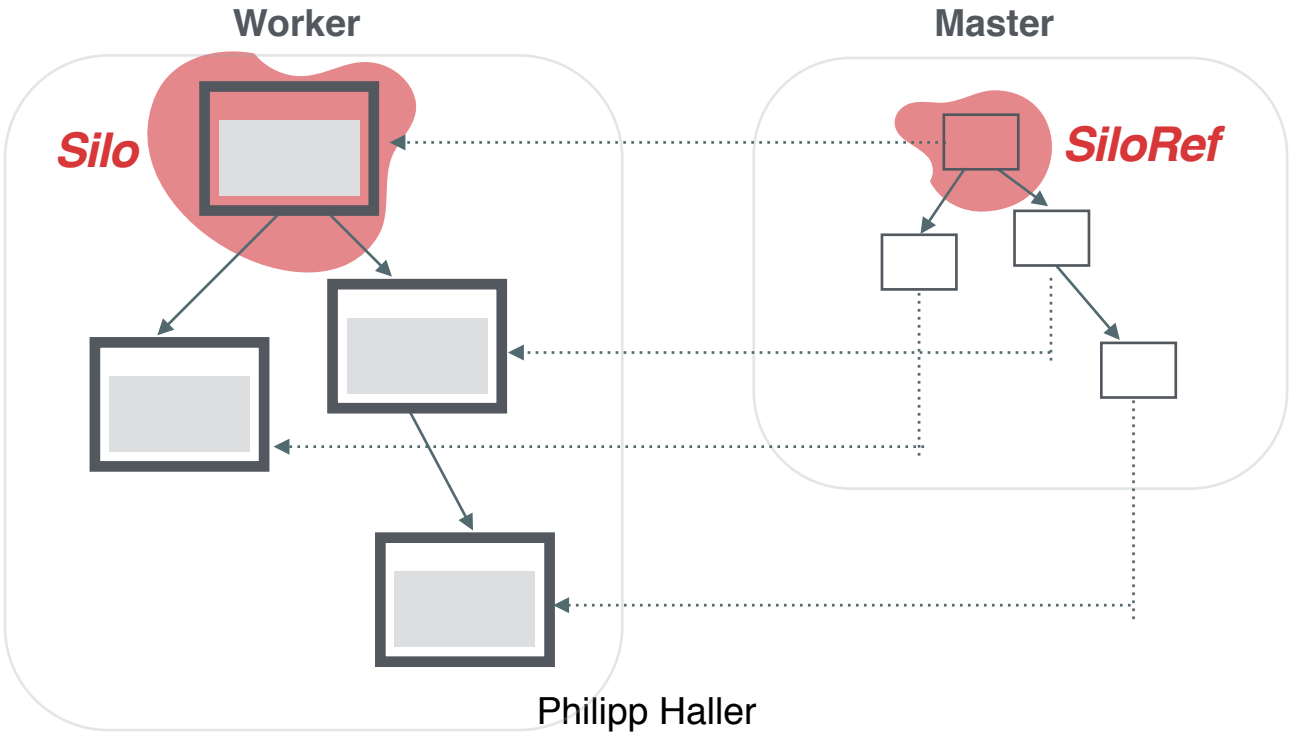
The function passing model

Consists of 3 parts:

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

Some visual intuition of

The function passing model



Silos

What are they?

Two parts.

- **SiloRef.** Handle to a Silo.
- **Silo.** Typed, stationary data container.

User interacts with SiloRef.

SiloRefs come with 4 primitive operations.

Silos

What are they?

Primitive: apply

```
def apply[S](fun: T => SiloRef[S]): SiloRef[S]
```

- Takes a function that is to be applied to the data in the silo associated with the SiloRef.
- Creates new silo to contain the data that the user-defined function returns; evaluation is *deferred*

Deferred

Enables interesting computation DAGs

Silos

What are they?

Primitive: send

```
def send(): Future[T]
```

- Forces the built-up computation DAG to be sent to the associated node and applied.
- Future is completed with the result of the computation.

EAGER

Silos

What are they?

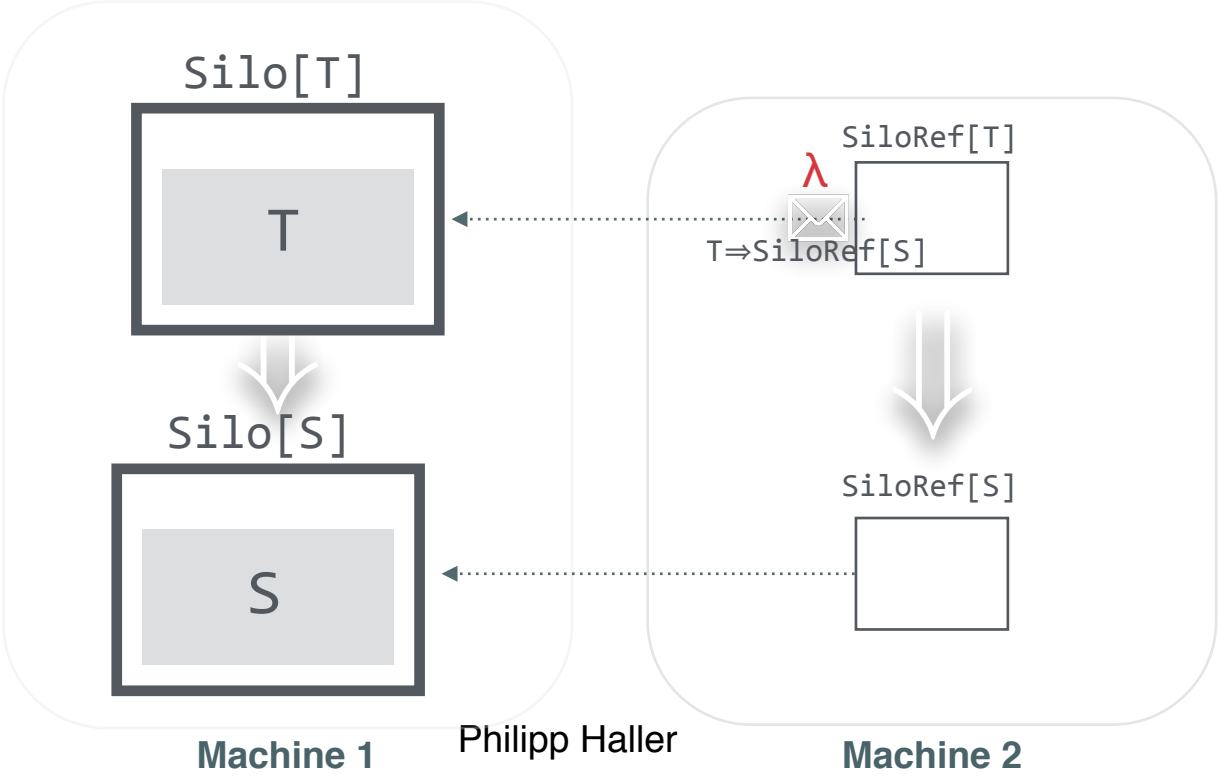
Silo factories:

```
object SiloRef {
  def populate[T](host: Host, value: T): SiloRef[T]
  def fromTextFile(host: Host, file: File): SiloRef[List[String]]
  ...
}
```

- Creates silo on given host populated with given value/text file/…

Deferred

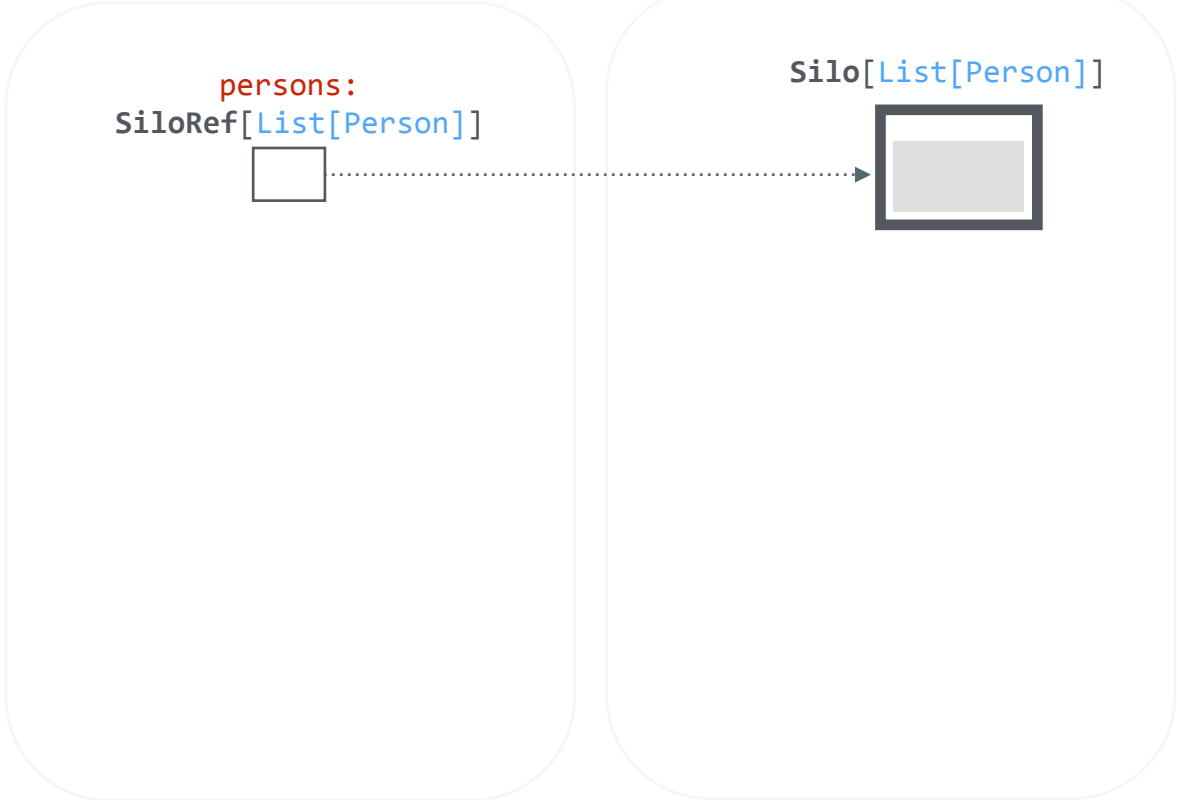
Basic idea: apply/send



More involved example

Let's make an interesting DAG!

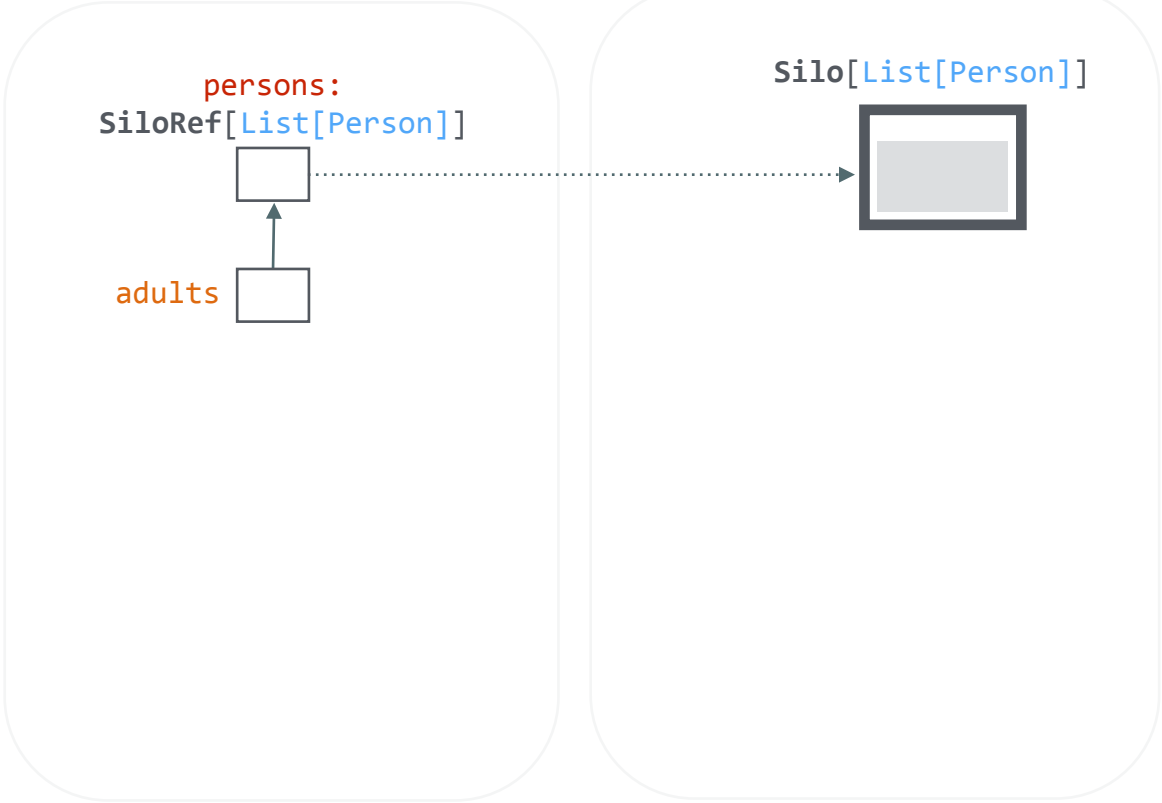
```
val persons: SiloRef[List[Person]] = ...
```



More involved example

Let's make an interesting DAG!

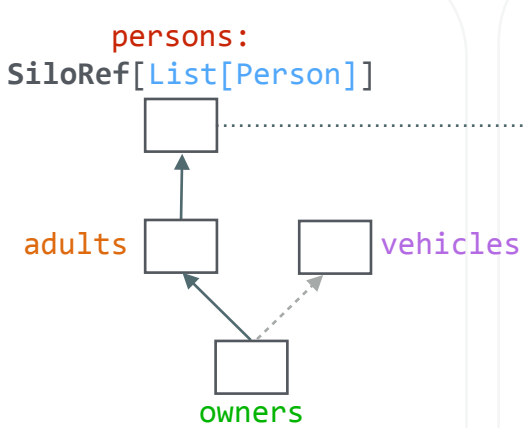
```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
```

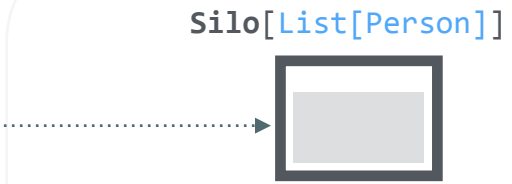


More involved example

Let's make an interesting DAG!

```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.apply(spore {
  val localVehicles = vehicles // spore header
  ps =>
    localVehicles.apply(spore {
      val localps = ps // spore header
      vs =>
        SiloRef.populate(currentHost,
          localps.flatMap(p =>
            // list of (p, v) for a single person p
            vs.flatMap {
              v =>
                if (v.owner.name == p.name) List((p, v))
                else Nil
            }
          )
```





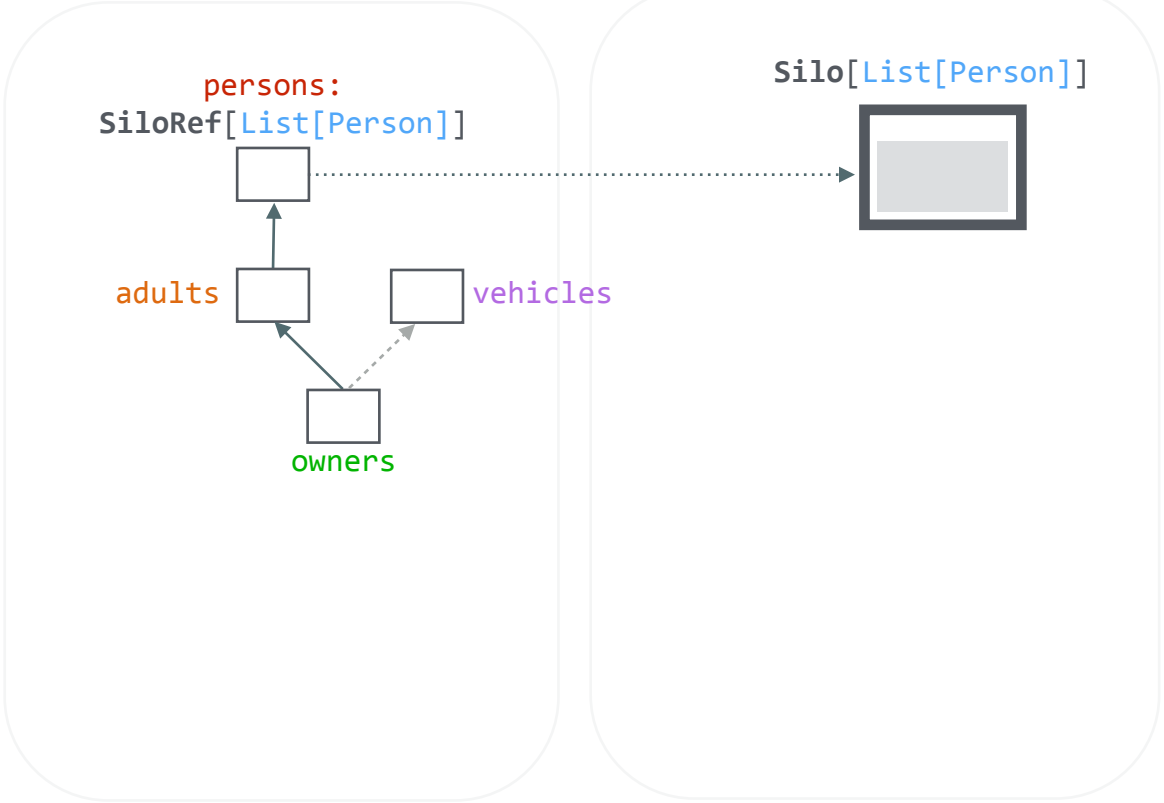
Machine 1

Machine 2

More involved example

Let's make an interesting DAG!

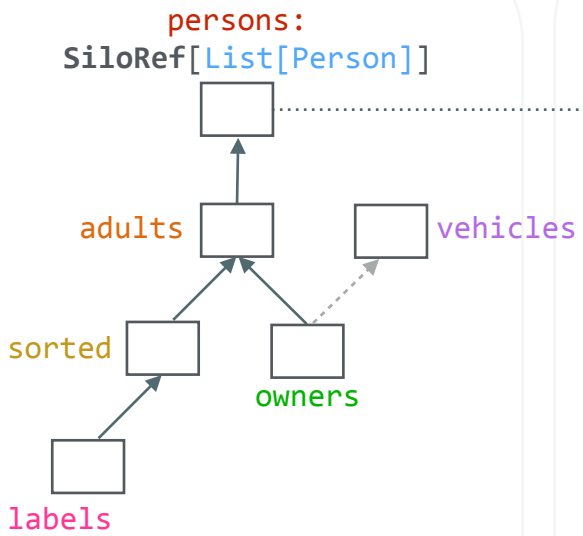
```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.apply(...)
```



More involved example

Let's make an interesting DAG!

```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.apply(...)
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))
  })
```



Machine 1

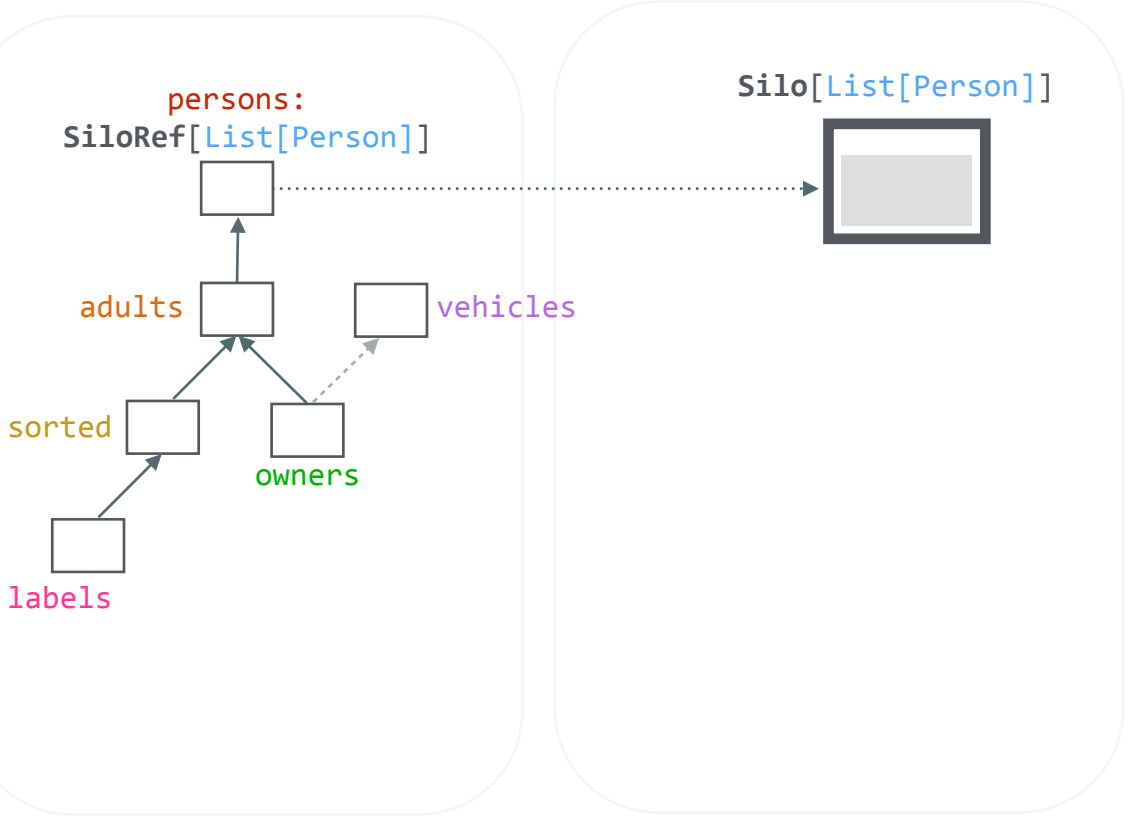
Machine 2

More involved example

Let's make an interesting DAG!

```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.apply(...)
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))
  })
```

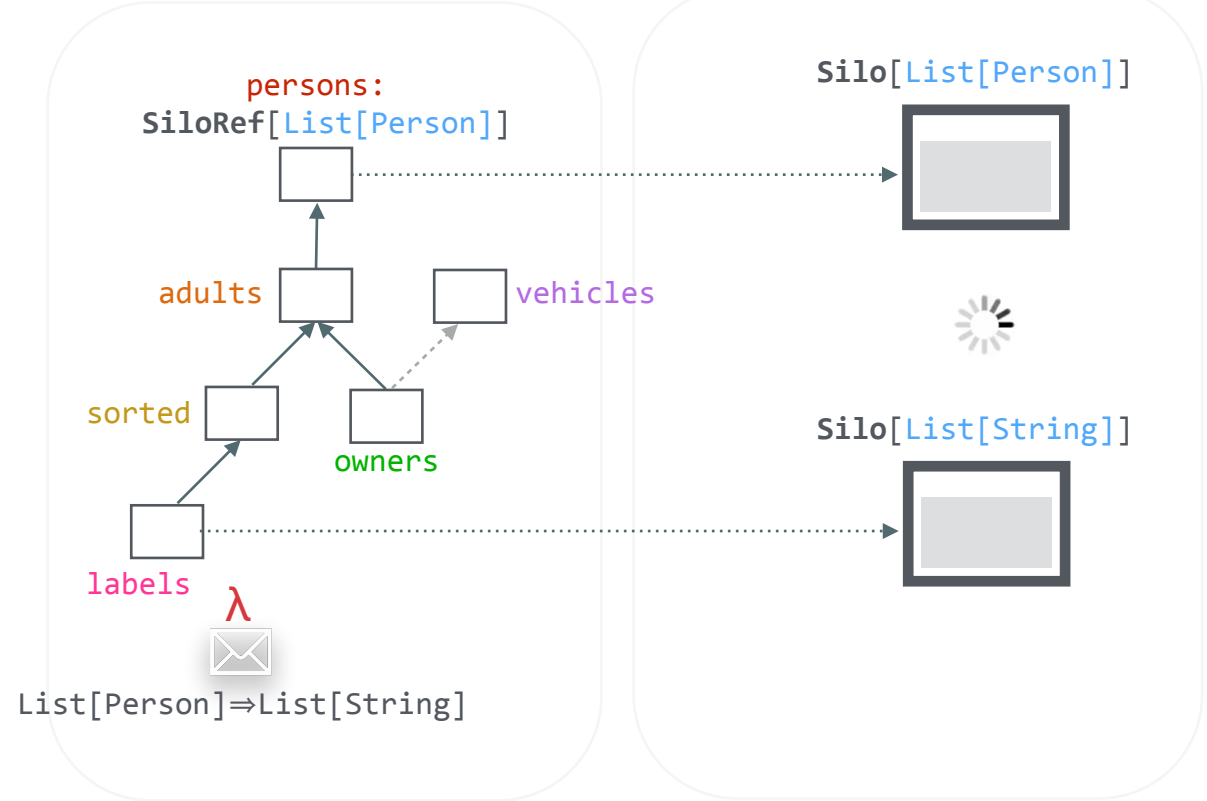
so far we just staged computation, we haven't yet "kicked it off".



More involved example

Let's make an interesting DAG!

```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.apply(...)
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))
  })
labels.persist().send()
```



A functional design for fault-tolerance

- A SiloRef is a **lineage**, a persistent (in the sense of functional programming) data structure.
- The lineage is the DAG of operations used to derive the data of a silo.
- Since the lineage is composed of spores [2], it is serializable. This means it can be **persisted** or **transferred** to other machines.

[2] Miller, Haller, and Odersky. *Spores: a type-based foundation for closures in the age of concurrency and distribution.* ECOOP '14

Putting lineages to work

A functional design for fault-tolerance

Next: we formalize **lineages**, a concept from the database + systems communities, in the context of PL. Natural fit in context of functional programming!

Formalization: typed, distributed core language with spores, silos, and futures.

Abstract syntax

$t ::=$		*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

Local reduction and lineages

R-IntOp

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

R-AppAbs

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

R-AppSpore

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

R-Await

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

R-Apply

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

$p ::= \texttt{spore}\ \{\ \overline{x:T=v}\ ;(x:T) \Rightarrow t\ \}$

$l ::=$		*lineages:*
	$\mathrm{Mat}(\iota)$	materialized
	$\mid \mathrm{Applied}(\iota,l,p)$	lineage with `apply`
$r ::= \mathrm{Ref}(l,h)$	where $h \in \mathscr{H}$	silo reference
$\iota ::= (h,i)$	where $h \in \mathscr{H}$ and $i \in \mathbb{N}$	decentralized identifier

Distributed reduction

R-Local

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

R-Send

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

R-Populate

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

R-Respond

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

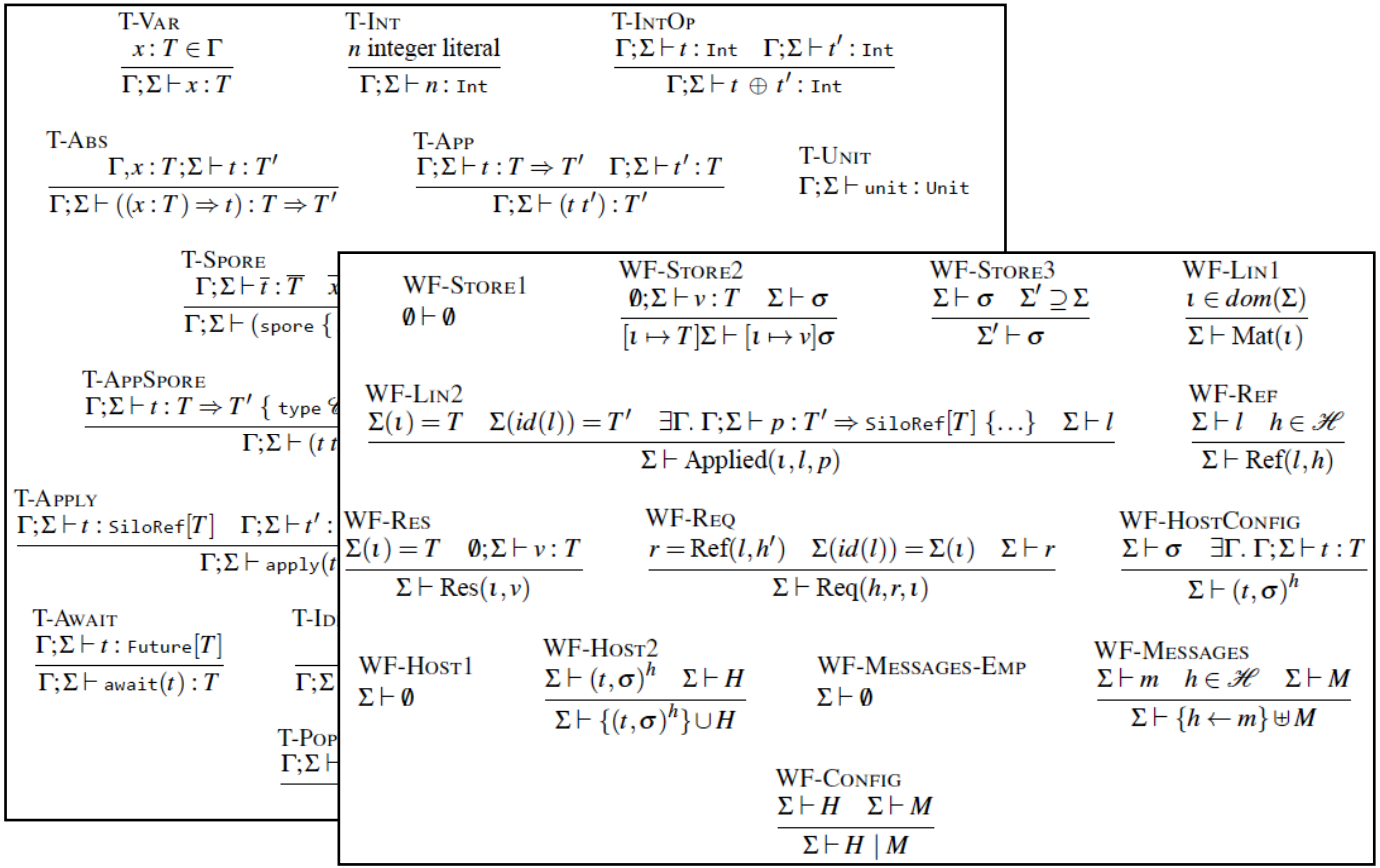
R-Process

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

R-Process-Val

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

Type assignment



Properties of function passing model

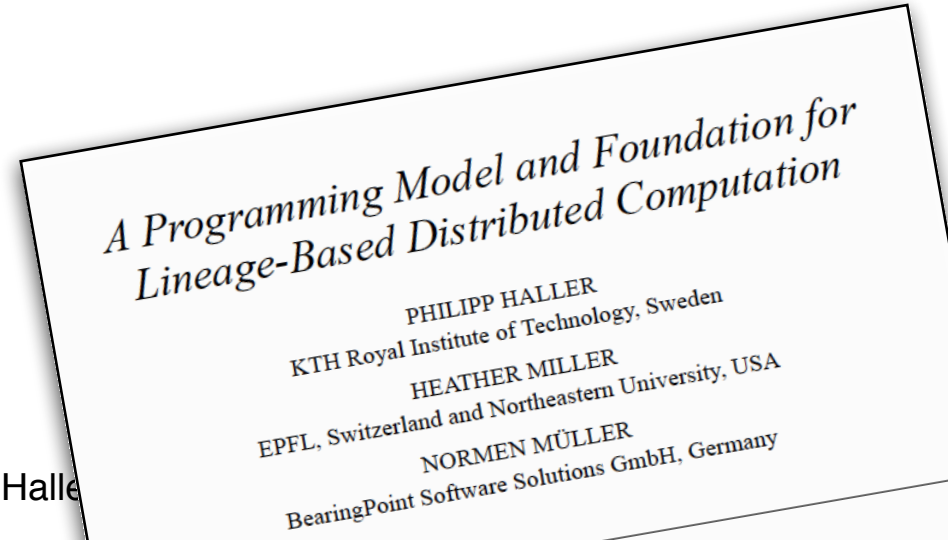
- **Subject reduction theorem** guarantees preservation of types under reduction, as well as preservation of **lineage mobility**.
- **Progress theorem** guarantees the **finite materialization** of remote, lineage-based data.

First correctness results for a programming model for lineage-based distributed computation.

Details, proofs, etc.

Paper

Haller, Miller, and Müller. *A Programming Model and Foundation for Lineage-Based Distributed Computation.* Journal of Functional Programming 28 (2018)
https://infoscience.epfl.ch/record/230304



Onward

Ongoing and future work

Distributed Shared State	Security & Privacy	Chaos Engineering
Consistency, availability, partition tolerance	**Privacy-aware distribution**	**Testing hypotheses about resilience in production systems**
Determinism	**Information-flow security**	

Conclusion

- ***Serverless computing***
  - Promising direction, intriguing properties
  - Important limitations
- ***Foundations for function-shipping***
  - Lineage-based distributed computation
  - First correctness results for a programming model based on lineages
- ***Goal: principles and foundations for a new distributed systems stack***