Selected Challenges in Concurrent and Distributed Programming

Philipp Haller

KTH Royal Institute of Technology
Stockholm, Sweden

Joint work with Heather Miller, Normen Müller, Xin Zhao, Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Michael Eichberg, Guido Salvaneschi and Mira Mezini

Workshop on Programming Languages and Distributed Systems

March 5th & 6th, 2020
RISE Computer Science, Electrum Kista, Stockholm, Sweden

Goals

- Programming languages for distributed systems that provide high scalability, reliability, and availability
- Prevent bugs in distributed systems

Challenge 1: Ensuring Fault-Tolerance Properties

- Specific fault-tolerance mechanism:
  Lineage-based fault recovery
  - Lineage records dataset identifier plus transformations
  - Maintaining lineage information in available, replicated storage enables recovering from replica faults
- ***A widely-used fault-recovery mechanism*** (e.g., Apache Spark)

How to *statically* ensure fault-tolerance properties for languages based on lineage-based fault recovery?

Programming Model for Lineage-based Distributed Computation

- ***A programming model***
  - for ***functional processing of distributed data***,
  - which provides abstractions for building fault-tolerant distributed systems,
  - including ***first-class lineages*** and ***futures***.
- ***Complete formalization***
  - As an extension of typed lambda-calculus,
  - with futures and distributable closures ("spores"),
  - based on an ***asynchronous, distributed operational semantics***

Programming Model Illustrated

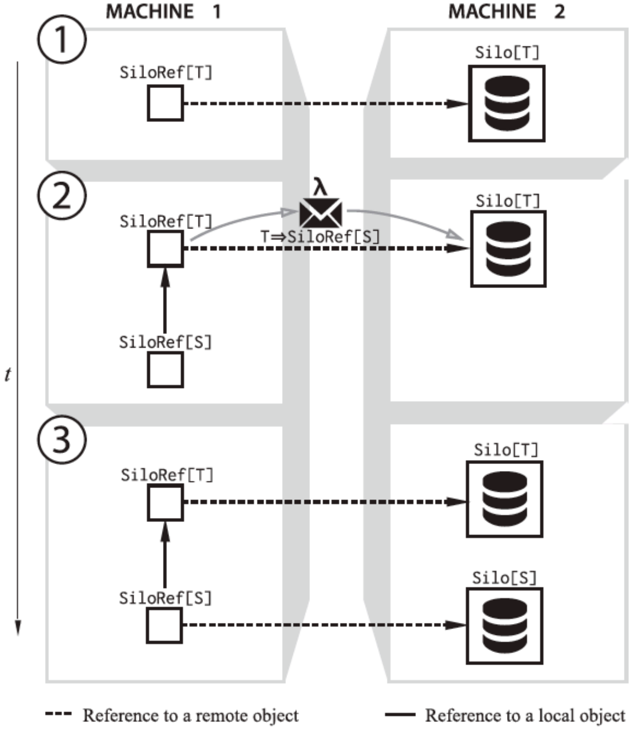


Fig. 1. Basic function passing model.

Philipp Haller

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]
```

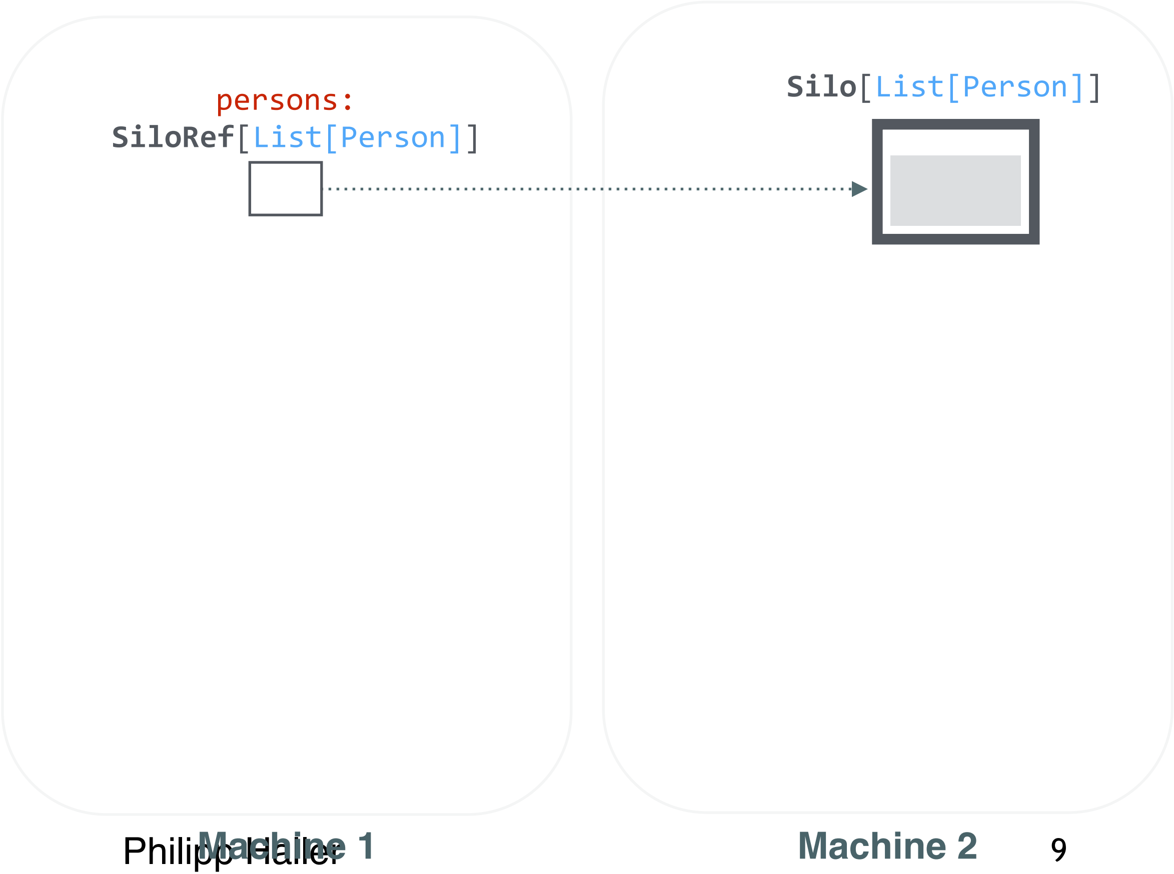
- Forces the built-up computation DAG to be sent to the associated node and applied.
- Future is completed with the result of the computation.

EAGER

More involved example

Let's make an interesting DAG!

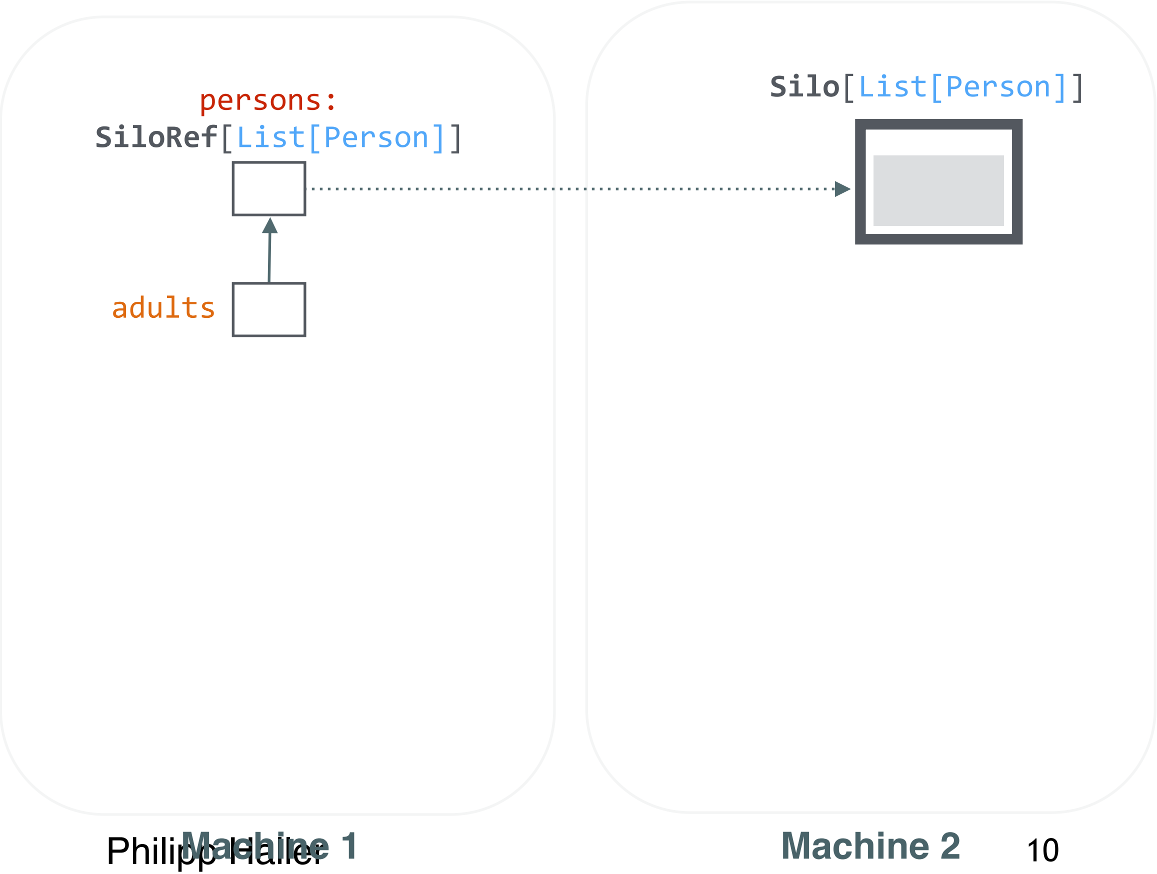
```
val persons: SiloRef[List[Person]] = ...
```



More involved example

Let's make an interesting DAG!

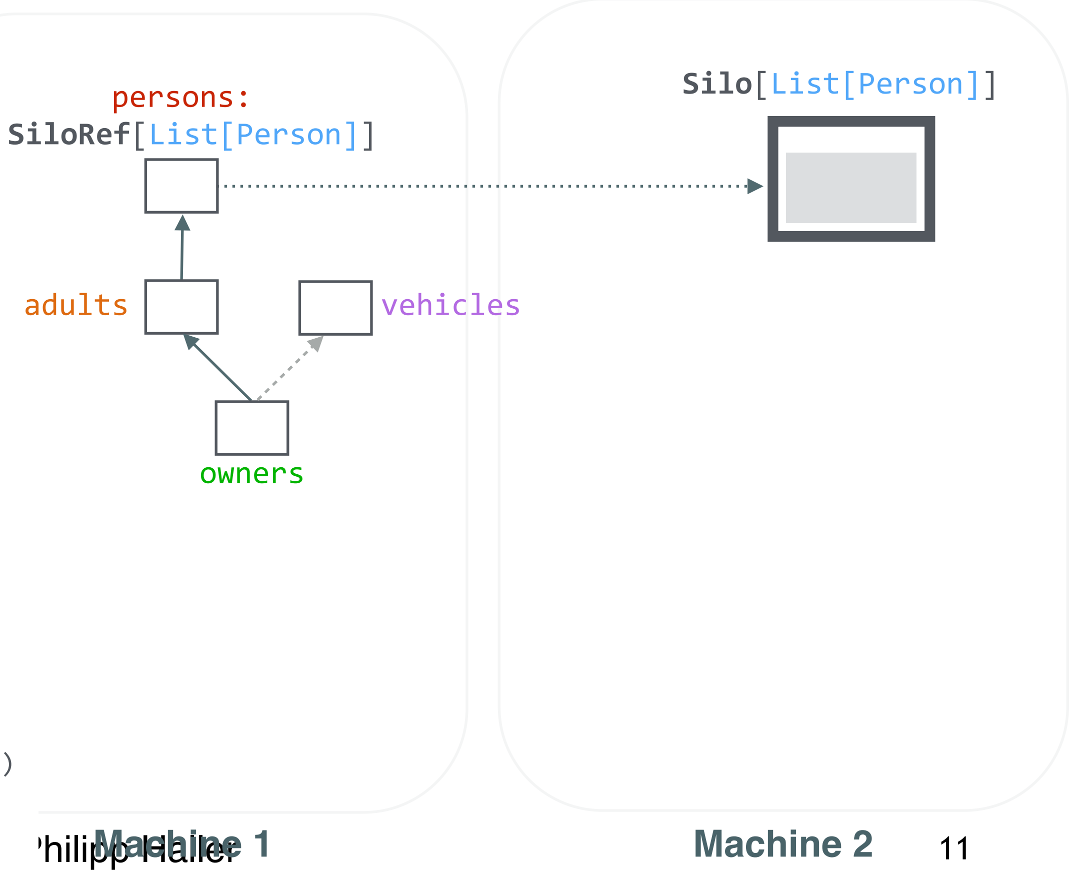
```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
```



More involved example

Let's make an interesting DAG!

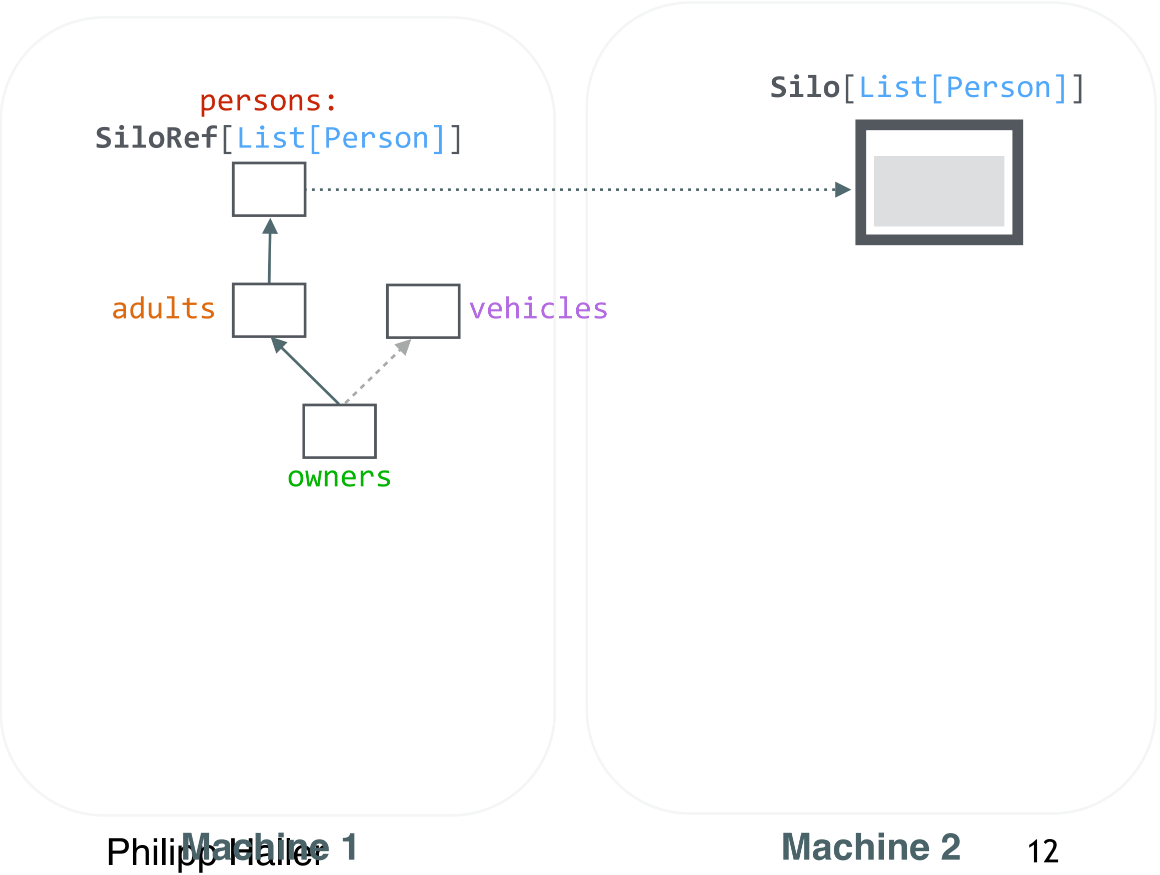
```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.apply(spore {
  val localVehicles =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



More involved example

Let's make an interesting DAG!

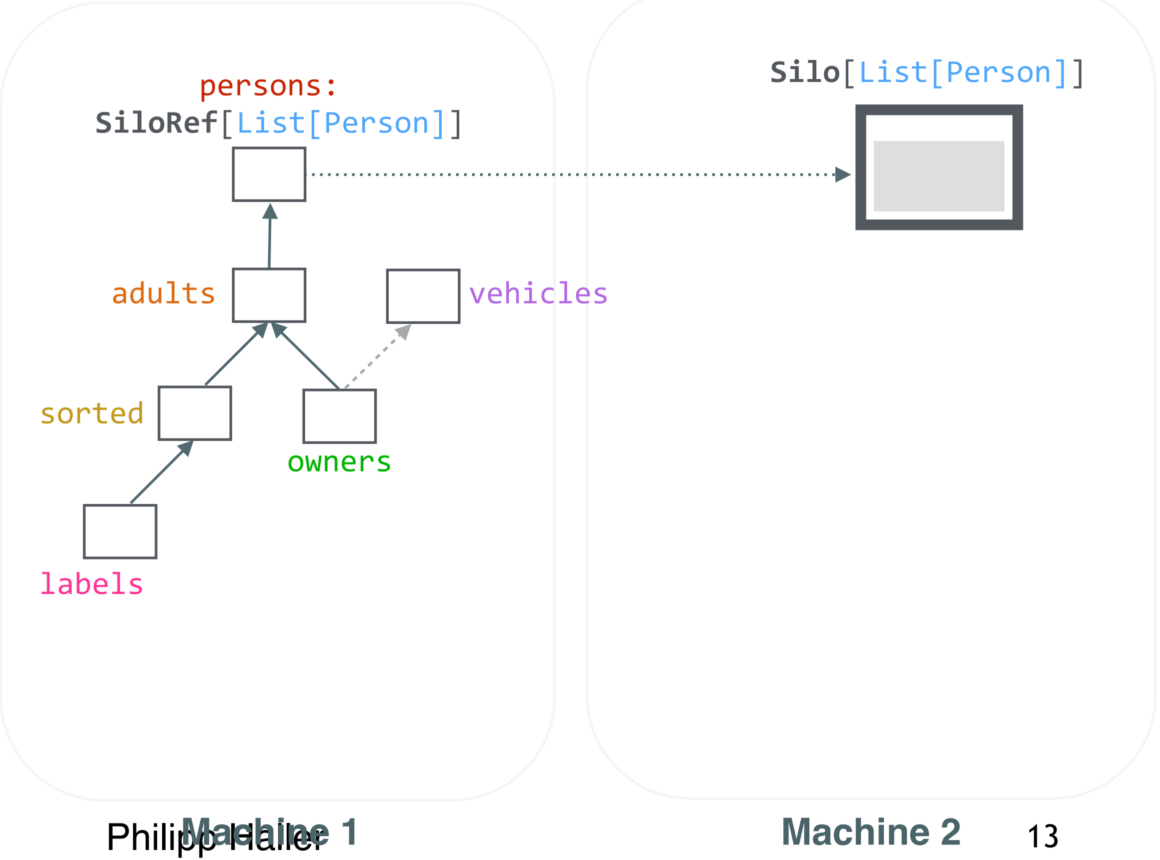
```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.apply(spore { ps =>
    val res = ps.filter(p => p.age >= 18)
    SiloRef.populate(currentHost, res)
  })
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.apply(...)
```



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))
  })
```

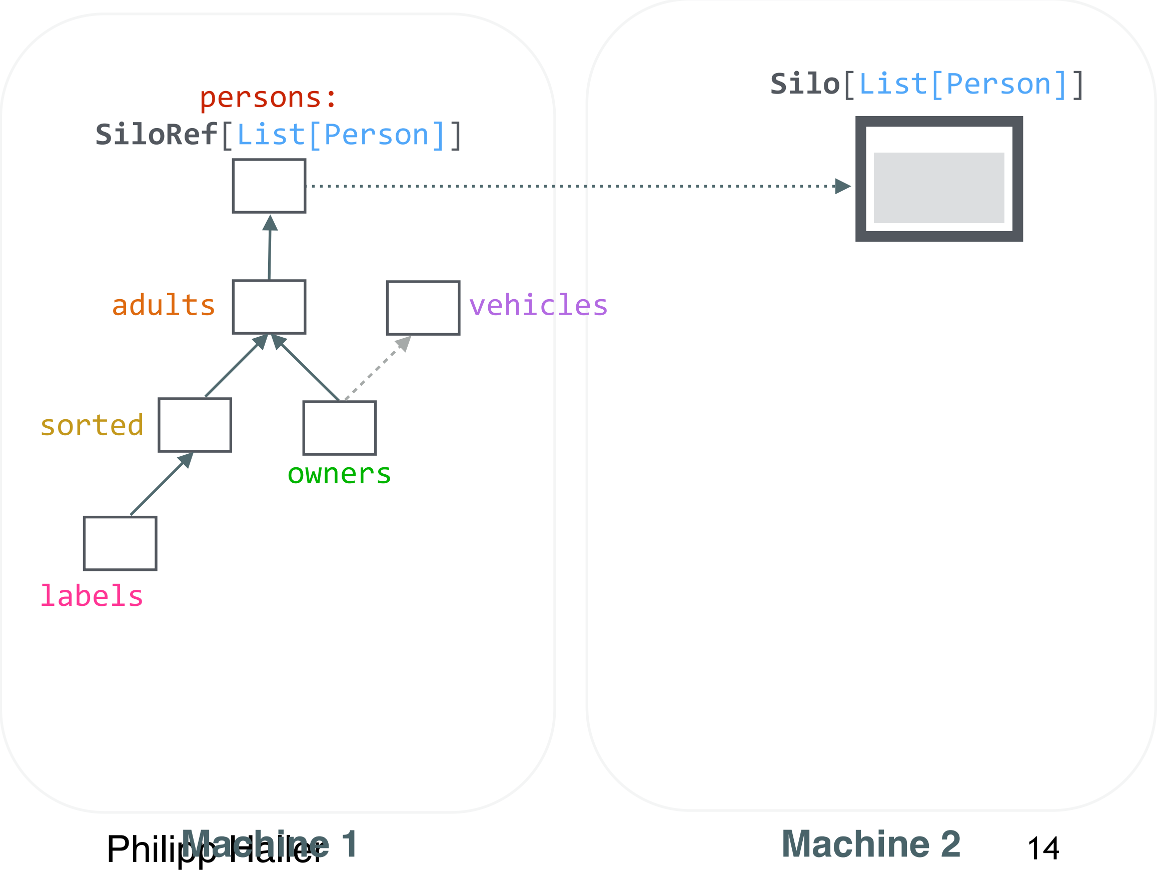


More involved example

Let's make an interesting DAG!

```scala
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))
  })
```

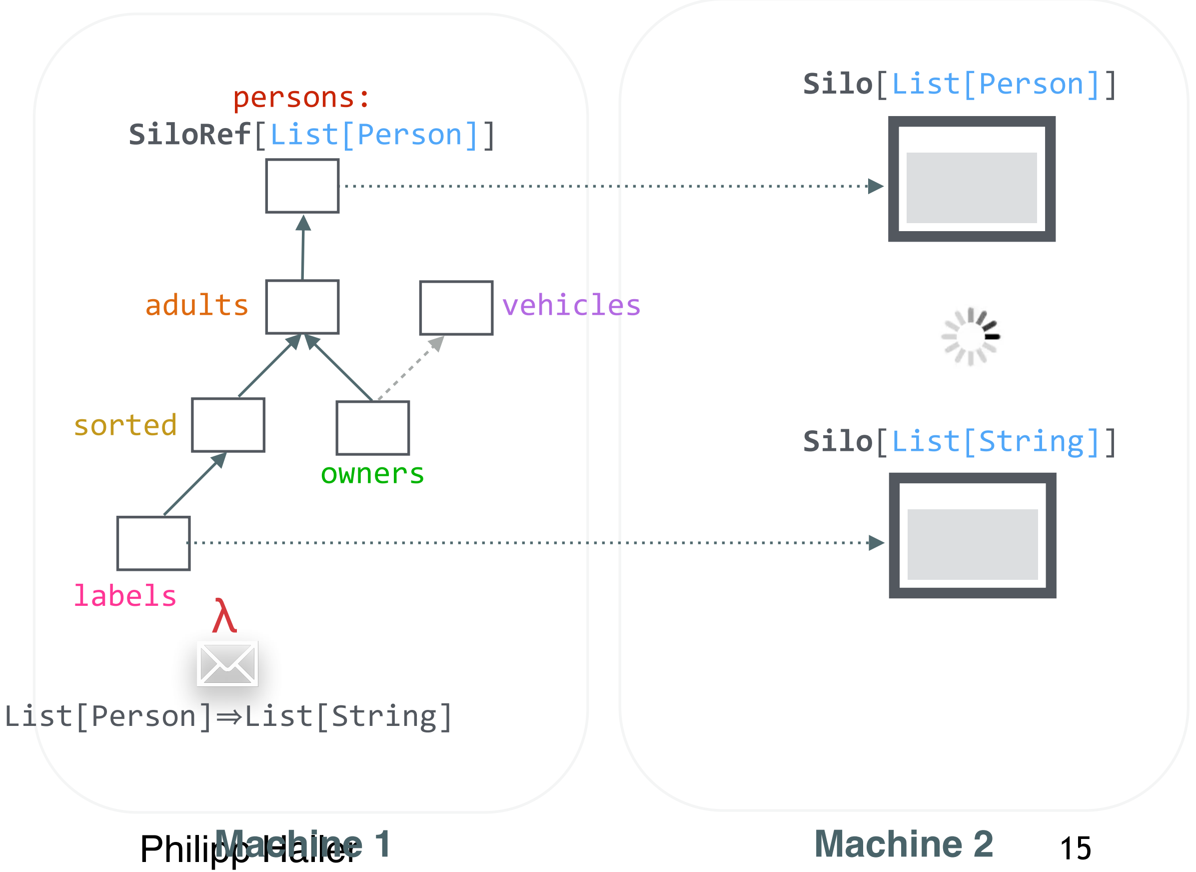
so far we just staged computation, we haven't yet "kicked it off".



More involved example

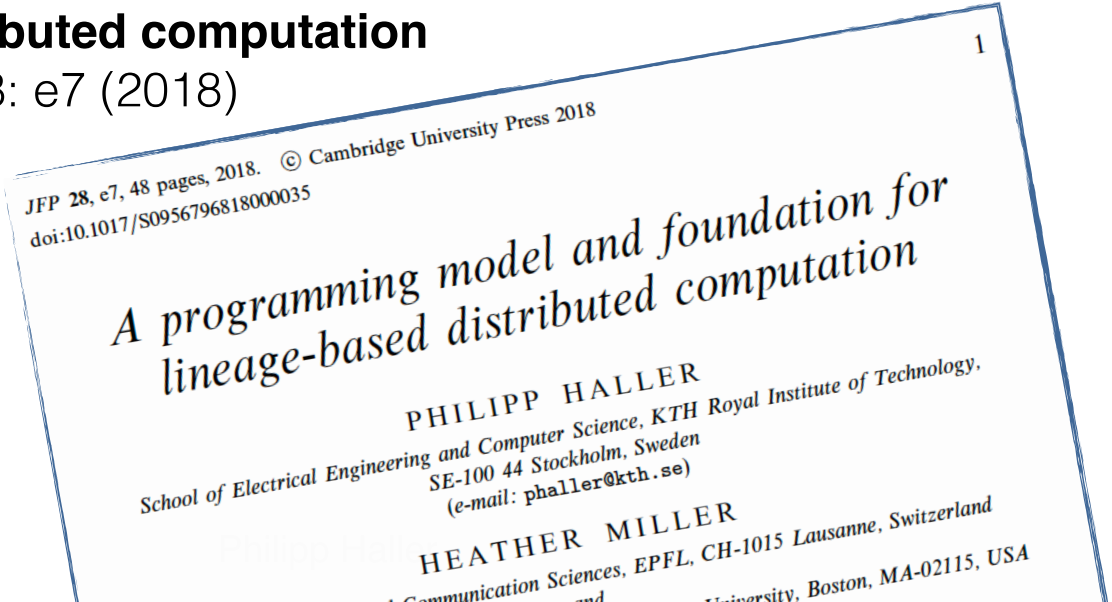
Let's make an interesting DAG!

```
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()
```



Lineage-based Distributed Computation: Results

- Proof establishing the ***preservation of lineage mobility***
- Proof of ***finite materialization of remote, lineage-based data***
- P. Haller, H. Miller, N. Müller: **A programming model and foundation for lineage-based distributed computation**
  J. Funct. Program. 28: e7 (2018)


1

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
Communication Sciences, EPFL, CH-1015 Lausanne, Switzerland
University, Boston, MA-02115, USA

Challenge 2: Geo-Distribution

- Operating a service in multiple datacenters can ***improve latency and availability*** for geographically distributed clients
- Geo-distribution directly supported by today's cloud platforms
- *Challenge:* ***round-trip latency***
  - < 2ms between servers within the same datacenter
  - up to ***two orders of magnitude higher*** between distant datacenters

Naive reuse of single-datacenter application architectures and protocols leads to poor performance!

Data Consistency

- In order to satisfy latency, availability, and performance requirements of distributed systems, developers use ***variety of data consistency models***
  - Theoretical limit given by CAP theorem[1]
- There is no one-size-fits-all consistency model

How to *safely* use both consistent and available (but inconsistent) data within the same application?

[1] Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51-59 (2002)

Consistency Types: Idea

To satisfy a range of performance, scalability, and consistency requirements, provide two different kinds of replicated data types

1. ***Consistent data types***:
   - Serialize updates in a global total order: ***sequential consistency***
   - ***Do not provide availability*** (in favor of partition tolerance)
2. ***Available data types***:
   - Guarantee ***availability and performance*** (and partition tolerance)
   - ***Weaken consistency***: strong eventual consistency

Consistency Types in LCD

First-class functions

Replicated data types

LCD:

- A higher-order language with distributed references and ***consistency types***
- Values and types annotated with ***labels indicating their consistency***

$$
\begin{array}{lll}
\ell & ::= & \cdot \mid \mathsf{con} \mid \mathsf{ava} \\
t & ::= & v \mid t \bigoplus t \mid t \; \mathsf{op} \; t \mid t \; t \mid \mathtt{if} \; x \; \mathtt{then} \\
 & & \mid \mathsf{ref}_\ell \; t \mid \,!t \mid t := t \\
r & ::= & \mathsf{d} \mid \mathsf{true} \mid \mathsf{false} \mid (\lambda^\ell x : \tau. \; t) \mid \mathsf{unit} \\
v & ::= & r_\ell \mid x \\
\tau & ::= & \mathsf{Bool}_\ell \mid \mathsf{Unit}_\ell \mid \mathsf{Lat}_\ell \mid \mathsf{Ref}_\ell \; \tau \mid \tau \xrightarrow{\ell}_\ell \tau \\
\bigoplus & ::= & \vee \mid \wedge \\
\mathsf{op} & ::= & \preceq \mid \prec
\end{array}
$$

- Typed lambda-calculus
- ML-style references
- Labeled values and types

Consistency Types: Results

LCD: a higher-order language with replicated types and consistency labels

- Consistency types enable ***safe use*** of both strongly consistent and available (weakly consistent) data within the same application
- Proofs of ***type soundness*** and ***noninterference***
- Noninterference:
  Cannot observe mutations of available data via consistent data
- X. Zhao and P. Haller: **Foundations of consistency types for a higher-order distributed language**
  32nd Workshop on Languages and Compilers for Parallel Computing (LCPC 2019)
  Companion technical report with proofs:
  https://arxiv.org/abs/1907.00822

Challenge 3: Parallel Programming

- Increasing importance of ***static analysis*** (program analysis)
  - Bug finding, security analysis, taint tracking, etc.
- Precise and powerful analyses have ***long running times***
  - Infeasible to integrate into nightly builds, CI, IDE, …
  - ***Parallelization difficult:*** advanced static analyses not data-parallel
- Scaling static analyses to ever-growing software systems requires ***maximizing utilization of multi-core CPUs***

Our Approach

Ongoing work on checking correctness

- Novel ***concurrent programming model***
  - Generalization of futures/promises
  - Guarantees deterministic outcomes *(if used correctly)*
- Implemented in Scala
  - Statically-typed, integrates functional and object-oriented programming
  - Supported backends: JVM, JavaScript (+ experimental native backend)
- Integrated with OPAL, a state-of-the-art ***JVM bytecode analysis framework***

Example

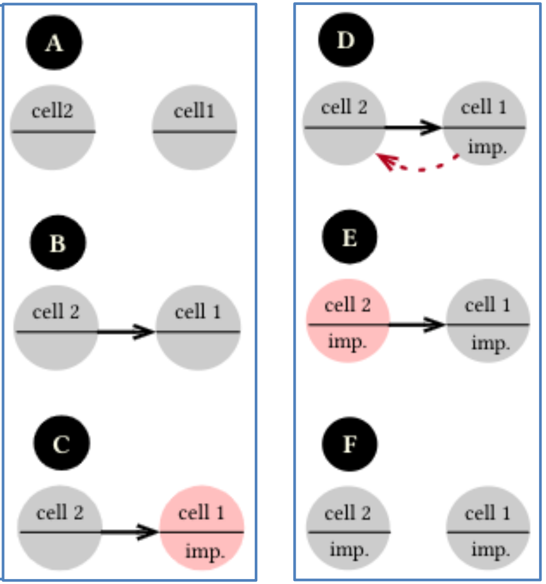
- Two key concepts: ***cells*** and ***handlers***
- Cell completers permit ***writing***, cells only ***reading*** (concurrently)

```
val completer1 = CellCompleter[...]
val completer2 = CellCompleter[...]
val cell1 = completer1.cell
val cell2 = completer2.cell

cell2.when(cell1) { update =>
  if (update.value == Impure) FinalOutcome(Impure)
  else NoOutcome
}
completer1.putFinal(Impure)
```

Example

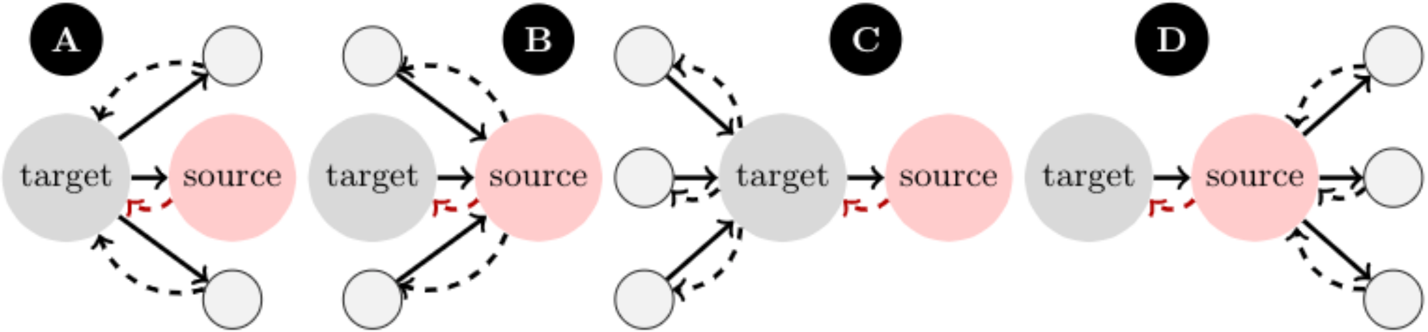
- Two key concepts: ***cells*** and ***handlers***
- Cell completers permit ***writing***, cells only ***reading*** (concurrently)

```
val completer1 = CellCompleter[...]
val completer2 = CellCompleter[...]
val cell1 = completer1.cell
val cell2 = completer2.cell

cell2.when(cell1) { update =>
  if (update.value == Impure) FinalOutcome
  else NoOutcome
}
completer1.putFinal(Impure)
```

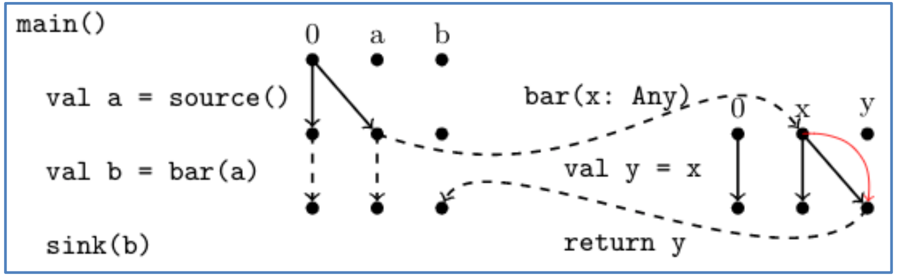


Scheduling Strategies

- ***Priorities for message propagations*** depending on number of dependencies of source/target nodes and dependees/dependers



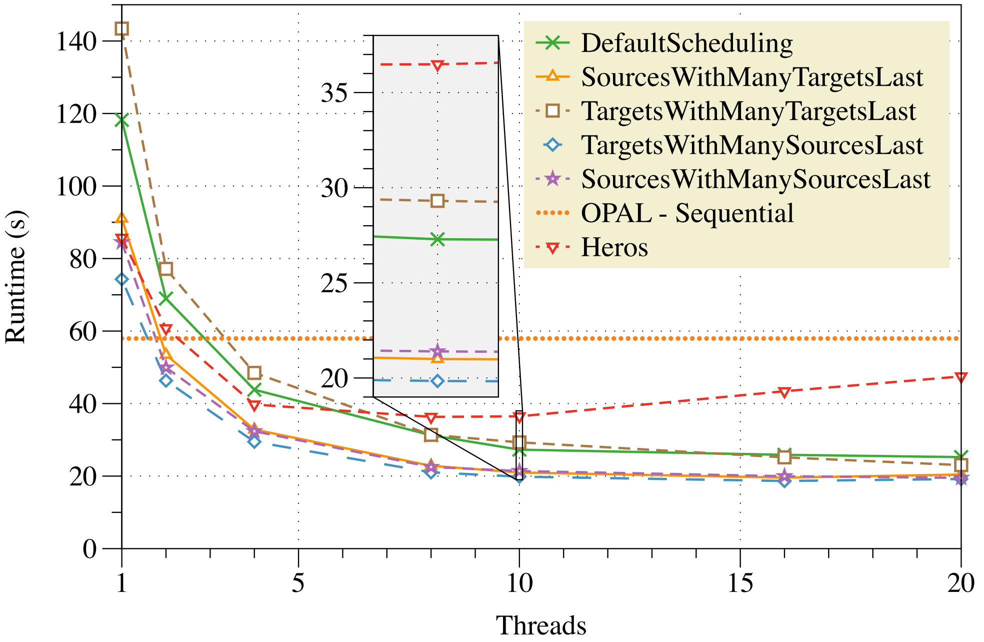
Experimental Evaluation



- Implementation of IFDS[1] analysis framework
- Use IFDS framework to implement ***taint analysis***
  - search for methods with String parameter that is later used in an invocation of `Class.forName` (i.e., reflective, dynamic class loading)

[1] Interprocedural Finite Distributive Subset

Parallel Static Analysis: Results



- Heros: best speed-up 2.36x @ 8 threads
- RANG (us): speed-up 3.53x @ 8 threads, 3.98x @ 16 threads

Analysis executed on Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz (10 cores) using 16 GB RAM running Ubuntu 18.04.3 and OpenJDK 1.8_212

Conclusion

- ***Challenge:***
  Building distributed systems providing high scalability, reliability, and availability
  - System builders use various ***unsafe techniques*** to achieve these properties
  - How can we support system builders and prevent bugs?
- ***Thesis:***
  Programming language techniques can help!
  - ***Language constructs, abstractions***
    - for composing systems modularly
    - for exploiting parallelism, replication, etc.
  - ***Type systems and static analysis*** for preventing hard-to-reproduce bugs