Towards Safe, Large-Scale Concurrent and Distributed Programming

Philipp Haller

Assistant Professor
School of Computer Science and Communication
KTH Royal Institute of Technology, Sweden
Plot: https://blog.twitter.com/2009/inauguration-day-twitter
How to construct programming systems that support such extreme situations?

Plot: https://blog.twitter.com/2009/inauguration-day-twitter
How to construct programming systems that support such extreme situations?

Plot: https://blog.twitter.com/2009/inauguration-day-twitter

Hint: you’ll see a system used during this event!
Context
Context

• Concurrent and distributed programming
Context

- Concurrent and distributed programming
- Research foci:
  - Scalable concurrency
  - Static safety
  - Efficient distribution
Research Topics

Concurrent programming

Types + generic programming
Concurrent programming

Types + generic programming

Systems
Research Topics

Concurrent programming

Event-driven actors [JMLC’06]

Integrating threads and events [TCS’09, Artima’12]

Systems

Scala Actors

Types + generic programming
## Research Topics

<table>
<thead>
<tr>
<th>Concurrent programming</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven actors</td>
<td>Systems</td>
</tr>
<tr>
<td>[JMLC’06]</td>
<td>Scala Actors</td>
</tr>
<tr>
<td>Integrating threads and events</td>
<td>Scala Async</td>
</tr>
<tr>
<td>[TCS’09, Artima’12]</td>
<td></td>
</tr>
<tr>
<td>Event-driven concurrency</td>
<td>[SIP’13]</td>
</tr>
</tbody>
</table>

Types + generic programming
# Research Topics

<table>
<thead>
<tr>
<th>Concurrent programming</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven actors [JMLC’06]</td>
<td>Scala Actors</td>
</tr>
<tr>
<td>Integrating threads and events [TCS’09, Artima’12]</td>
<td></td>
</tr>
<tr>
<td>Event-driven concurrency [SIP’13]</td>
<td>Scala Async</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types + generic programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero-copy messaging [ECOOP’10]</td>
</tr>
<tr>
<td>Fast serialization [OOPSLA’13]</td>
</tr>
<tr>
<td>Safe closure passing [ECOOP’14]</td>
</tr>
<tr>
<td>Spores</td>
</tr>
</tbody>
</table>
How Can Programming Languages Help?
How Can Programming Languages Help?

Programming language technology...
How Can Programming Languages Help?

Programming language technology...

• ... can provide important static guarantees using (lightweight) formal methods
How Can Programming Languages Help?

Programming language technology...

• ... can provide *important static guarantees* using (lightweight) formal methods

"Formal methods will never have any impact until they can be used by people that don’t understand them."
— (attributed to) Tom Melham
How Can Programming Languages Help?

Programming language technology...

• ... can provide **important static guarantees** using (lightweight) formal methods
How Can Programming Languages Help?

Programming language technology...

• ... can provide *important static guarantees* using (lightweight) formal methods

• ... can *improve developer productivity*
How Can Programming Languages Help?

Programming language technology...

• ... can provide **important static guarantees** using (lightweight) formal methods

• ... can **improve developer productivity**
Motivating Example
Motivating Example

Chat proxy
Motivating Example

Chat proxy

- Many long-lived connections
- Usually idle, with short bursts of traffic
Chat Proxy: First Try

- Thread per session

- **Does not scale**
Chat Proxy: Second Try
Chat Proxy: Second Try

• Asynchronous I/O and thread pool
Chat Proxy: Second Try

• Asynchronous I/O and thread pool

• Session state maintained in regular objects
Chat Proxy: Second Try

• Asynchronous I/O and thread pool

• Session state maintained in regular objects

• Much more scalable
Chat Proxy: Second Try

- Asynchronous I/O and thread pool
- Session state maintained in regular objects
- **Much more scalable**
- Problems:
Chat Proxy: Second Try

- Asynchronous I/O and thread pool
- Session state maintained in regular objects
- Much more scalable
- Problems:
  - Code difficult to maintain
Chat Proxy: Second Try

- Asynchronous I/O and thread pool
- Session state maintained in regular objects
- **Much more scalable**
- Problems:
  - **Code difficult to maintain**
  - **Blocking calls fatal**
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction

- High scalability on mainstream VMs
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction

- High scalability on mainstream VMs
- Integration with native VM threads
Actor Model
Actor Model

- Model of concurrent computation whose universal primitive is the “actor” [IJCAI’73]
Actor Model

• Model of concurrent computation whose universal primitive is the “actor” [IJCAI’73]

• Actors = concurrent processes communicating using asynchronous messages
Actor Model

• **Model of concurrent computation** whose universal primitive is the “actor” [IJCAI’73]

• Actors = concurrent processes communicating using **asynchronous messages**

• Upon reception of a message, an actor may
Actor Model

- **Model of concurrent computation** whose universal primitive is the “actor” [IJCAI’73]

- Actors = concurrent processes communicating using **asynchronous messages**

- Upon reception of a message, an actor may
  - change its behavior/state
Actor Model

- **Model of concurrent computation** whose universal primitive is the “actor” [IJCAI’73]

- Actors = concurrent processes communicating using **asynchronous messages**

- Upon reception of a message, an actor may
  - change its behavior/state
  - send messages to actors (including itself)
Actor Model

- **Model of concurrent computation** whose universal primitive is the “actor” [IJCAI’73]

- Actors = concurrent processes communicating using **asynchronous messages**

- Upon reception of a message, an actor may
  - change its behavior/state
  - send messages to actors (including itself)
  - create new actors
Implementing Actors
Implementing Actors

Thread-based implementation:
Implementing Actors

Thread-based implementation:

• One thread per actor
Implementing Actors

Thread-based implementation:

• One thread per actor

• Receive blocks thread while waiting for message
Implementing Actors

Thread-based implementation:

- One thread per actor
- Receive blocks thread while waiting for message

**PROS**
- Multicore utilization
- Interoperability with threads
- No inversion of control
Implementing Actors

Thread-based implementation:

- One thread per actor
- Receive blocks thread while waiting for message

<table>
<thead>
<tr>
<th><strong>Pros</strong></th>
<th><strong>Cons</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore utilization</td>
<td>High memory consumption</td>
</tr>
<tr>
<td>Interoperability with threads</td>
<td>Context switching overhead</td>
</tr>
<tr>
<td>No inversion of control</td>
<td></td>
</tr>
</tbody>
</table>
Implementing Actors

Thread-based implementation:

- One thread per actor
- Receive blocks thread while waiting for message

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore utilization</td>
<td>High memory consumption</td>
</tr>
<tr>
<td>Interoperability with threads</td>
<td>Context switching overhead</td>
</tr>
<tr>
<td>No inversion of control</td>
<td></td>
</tr>
</tbody>
</table>
Scalability
Scalability

- Scalability of actors fundamentally limited by
  - Resource consumption
  - Context-switching overhead
Scalability

- Scalability of actors fundamentally limited by
  - Resource consumption
  - Context-switching overhead
- Idea of event-driven systems:
  - Minimize both by pooling worker threads using lightweight event handlers
Event-Based Actors
Event-Based Actors

Implementation technique:
Event-Based Actors

Implementation technique:

- **Release thread** whenever actor has to wait
Event-Based Actors

Implementation technique:

- **Release thread** whenever actor has to wait
- Save a **continuation closure**
Event-Based Actors

Implementation technique:

- **Release thread** whenever actor has to wait
- Save a **continuation closure**
- Provide a high-level event-based receive operation that avoids an **inversion of control**
Event-Based Actors

Implementation technique:

- **Release thread** whenever actor has to wait
- Save a **continuation closure**
- Provide a high-level event-based receive operation that avoids an **inversion of control**
Actors for I/O

• Each session is an actor
• I/O events are just messages

```javascript
react {
    case DataReceived(data) => ...
    case SessionClosed => ...
}
```
Actors for I/O

- Each session is an actor
- I/O events are just messages

```reason
react {
  case DataReceived(data) => ...
  case SessionClosed => ...
}
```

Twitter Kestrel message queue
Are Lightweight Threads All We Need?
Are Lightweight Threads All We Need?

• Mainstream managed runtime environments (JVM, .NET CLR) do not provide lightweight threads
Are Lightweight Threads All We Need?

- Mainstream managed runtime environments (JVM, .NET CLR) do not provide lightweight threads

- Goal:
Are Lightweight Threads All We Need?

• Mainstream managed runtime environments (JVM, .NET CLR) do not provide lightweight threads

• Goal:
  • Practical solutions that work on unmodified runtimes
Are Lightweight Threads All We Need?

- Mainstream managed runtime environments (JVM, .NET CLR) do not provide lightweight threads

- Goal:
  - Practical solutions that work on **unmodified runtimes**
  - **Interoperability** with heavyweight threads
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction

- High scalability on mainstream VMs
- Integration with native VM threads
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction

- High scalability on mainstream VMs
- Integration with native VM threads

Haller and Odersky. *Event-based programming without inversion of control*. JMLC’06 (Google Scholar: 98 Citations)
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction

✔ High scalability on mainstream VMs

• Integration with native VM threads
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction

✔ High scalability on mainstream VMs
✔ Integration with native VM threads

Haller and Odersky. *Scala Actors: Unifying thread-based and event-based programming*. TCS’09 (Google Scholar: [239 Citations](#))
Scala Actors

First system that integrates event-based and thread-based programming on managed runtimes through a unified actor abstraction

- High scalability on mainstream VMs
- Integration with native VM threads

No change to the JVM necessary!

Haller and Odersky. *Scala Actors: Unifying thread-based and event-based programming*. TCS’09 (Google Scholar: **239 Citations**)
We saw 5x normal tweets-per-second and about 4x tweets-per-minute as this chart illustrates. Overall, Twitter sailed smoothly through the inauguration [...]”

Source: https://blog.twitter.com/2009/inauguration-day-twitter
Practical Experience

guardian.co.uk:
“[...] used Scala to meet the demanding real-time content searching, indexing or updating. Using actors for example, he explains how they were able to reduce the search index build time from 20 hours to just one. Request patterns, he says, are hard to predict so THE ABILITY TO EASILY SCALE THE SERVICES WAS ESSENTIAL.”

Source: Graham Tackley, Apache Lucene EuroCon 2010
Practical Experience

**guardian.co.uk:**
“[...] used Scala to meet the demanding real-time content searching, indexing or updating. **Using actors** for example, he explains how they were able to **reduce the search index build time from 20 hours to just one.** Request patterns, he says, are hard to predict so **THE ABILITY TO EASILY SCALE THE SERVICES WAS ESSENTIAL.**”

guardian.co.uk has the second highest readership of any on-line news site after the New York Times *

* ACCORDING TO ITS EDITOR

---

*Source: Graham Tackley, Apache Lucene EuroCon 2010*
Impact
Impact

• *Adoption:* actors one of the main concurrency models of Scala
Impact

• *Adoption*: actors one of the main concurrency models of Scala

• *Major influence* on Typesafe’s Akka actor-based event-driven middleware
Impact

- **Adoption**: actors one of the main concurrency models of Scala

- **Major influence** on Typesafe’s Akka actor-based event-driven middleware

  - **Production use**: Amazon, VMware, Autodesk, UBS, Credit Suisse, Klout, IGN, TDC, CSC, Blizzard, and others
Impact

- **Adoption:** actors one of the main concurrency models of Scala

- **Major influence** on Typesafe’s Akka actor-based event-driven middleware

  - **Production use:** Amazon, VMware, Autodesk, UBS, Credit Suisse, Klout, IGN, TDC, CSC, Blizzard, and others

- **Education**
Impact

• **Adoption**: actors one of the main concurrency models of Scala

• **Major influence** on Typesafe’s Akka actor-based event-driven middleware

  • *Production use*: Amazon, VMware, Autodesk, UBS, Credit Suisse, Klout, IGN, TDC, CSC, Blizzard, and others

• **Education**

  • Undergraduate teaching, e.g., [SIGCSE’13, SIGCSE’14] (http://scalaworkshop.com)
Impact

• **Adoption**: actors one of the main concurrency models of Scala

• **Major influence** on Typesafe’s Akka actor-based event-driven middleware

  • **Production use**: Amazon, VMware, Autodesk, UBS, Credit Suisse, Klout, IGN, TDC, CSC, Blizzard, and others

• **Education**

  • Undergraduate teaching, e.g., [SIGCSE’13, SIGCSE’14] (http://scalaworkshop.com)

  • Graduate teaching, e.g., “Programming Paradigms for Concurrency” at IST Austria
## Research Topics

<table>
<thead>
<tr>
<th>Concurrent programming</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven actors [JMLC’06]</td>
<td>Scala Actors</td>
</tr>
<tr>
<td>Integrating threads and events [TCS’09, Artima’12]</td>
<td></td>
</tr>
</tbody>
</table>

Types + generic programming
## Research Topics

<table>
<thead>
<tr>
<th>Concurrent programming</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven actors</td>
<td>Scala Actors</td>
</tr>
<tr>
<td>[JMLC’06]</td>
<td></td>
</tr>
<tr>
<td>Integrating threads and events</td>
<td></td>
</tr>
<tr>
<td>Event-driven concurrency</td>
<td>Scala Async</td>
</tr>
<tr>
<td>[SIP’13]</td>
<td></td>
</tr>
</tbody>
</table>

**Types + generic programming**
Generalization
Generalization

• Event-based code benefits not only actor-based programming models
Generalization

• Event-based code benefits not only actor-based programming models

• Generalization to other concurrency models possible via *cooperative multitasking*
Generalization

• Event-based code benefits not only actor-based programming models

• Generalization to other concurrency models possible via *cooperative multitasking*

• Challenge in *virtual machine-based languages*
Generalization

• Event-based code benefits not only actor-based programming models

• Generalization to other concurrency models possible via cooperative multitasking

• Challenge in virtual machine-based languages
  • Low-level stack manipulation impossible
Scala Async

• Research project at Typesafe
Scala Async

• Research project at Typesafe
Scala Async

- Research project at Typesafe

- Goals:

  - Common substrate for event-based concurrency based on *suspendible computations*

  - *Industrial-strength implementation*
Scala Async

- Research project at Typesafe

- Goals:
  - Common substrate for event-based concurrency based on *suspendible computations*
  - *Industrial-strength implementation*

- Requirements:
  - Support for the full Scala language
  - High performance
Async: Primitives
Async: Primitives

- `async { ... }` *delimits* a computation
Async: Primitives

• `async { ... }` *delimits* a computation

• `await(obj)` *suspends* a computation until the "awaitable" `obj` has been completed
Async: Primitives

- `async { … }` *delimits* a computation

- `await(obj)` *suspends* a computation until the “awaitable” `obj` has been completed

  - `obj` receives closure to resume computation
Async Programming Model
Async Programming Model

• Support for *full Scala language*
Async Programming Model

• Support for *full Scala language*

  • `async { ... }` may contain *nested definitions*
    (functions, classes/traits), lazy values, etc.
Async Programming Model

• Support for **full Scala language**

  • `async { ... }` may contain **nested definitions** (functions, classes/traits), lazy values, etc.

  • `await` may occur within **any Scala expression** (loops, pattern matching, try-catch-finally, etc.)
Async Programming Model

- Support for **full Scala language**

  - `async { ... }` may contain **nested definitions** (functions, classes/traits), lazy values, etc.

  - `await` may occur within **any Scala expression** (loops, pattern matching, try-catch-finally, etc.)

- Main restriction:

  - No `await` within closures or nested classes
Async vs. Continuations
Async vs. Continuations

• CPS supports powerful delimited continuations
Async vs. Continuations

• CPS supports powerful delimited continuations

• Translation of `async/await` based on ANF transform and state machines
  - No CPS transform
Async vs. Continuations

- CPS supports powerful *delimited continuations*

- Translation of `async/await` based on ANF transform and state machines

- No CPS transform

- **CPS transform not practical** for industrial-strength implementation

  - Closure allocation *expensive on virtual machines*

  - Closures generated by Scala compiler *capture too much* of their environment (memory leaks, etc.)
Practical Experience
Practical Experience

• New standard Scala module
  • *Scala Improvement Proposal SIP-22*
Practical Experience

- New standard Scala module
  - *Scala Improvement Proposal SIP-22*
- Heavy use by Typesafe customers
Practical Experience

- New standard Scala module
  - *Scala Improvement Proposal SIP-22*
- Heavy use by Typesafe customers
- Used in *Coursera MOOC* “Principles of Reactive Programming” (Odersky, Meijer, Kuhn)
## Research Topics

<table>
<thead>
<tr>
<th>Concurrent programming</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Event-driven actors</strong> [JMLC’06]</td>
<td><strong>Scala Actors</strong></td>
</tr>
<tr>
<td><strong>Integrating threads and events</strong> [TCS’09, Artima’12]</td>
<td></td>
</tr>
<tr>
<td><strong>Event-driven concurrency</strong> [SIP’13]</td>
<td><strong>Scala Async</strong></td>
</tr>
</tbody>
</table>

| **Types + generic programming**                            |                              |

**Philipp Haller**
## Research Topics

<table>
<thead>
<tr>
<th>Concurrent programming</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven actors [JMLC’06]</td>
<td>Scala Actors</td>
</tr>
<tr>
<td>Integrating threads and events [TCS’09, Artima’12]</td>
<td></td>
</tr>
<tr>
<td>Event-driven concurrency [SIP’13]</td>
<td>Scala Async</td>
</tr>
</tbody>
</table>

### Types + generic programming

Zero-copy messaging [ECOOP’10]
Safe and Efficient Message Passing
Safe and Efficient Message Passing

Goals:
Safe and Efficient Message Passing

Goals:

- **Zero-copy message passing**
  
  - Examples: data-heavy pipelines, protocol stacks, etc.
Safe and Efficient Message Passing

Goals:

- **Zero-copy message passing**
  
  - Examples: data-heavy pipelines, protocol stacks, etc.

- **High-performance sequential code**, i.e., support for mutable objects
Safe and Efficient Message Passing
Safe and Efficient Message Passing

Sending mutable objects by reference may lead to data races
Safe and Efficient Message Passing

Sending mutable objects by reference may lead to data races

Use unique references to enable efficient by-reference message passing without races
Safe and Efficient Message Passing

Sending mutable objects by reference may lead to data races

Use unique references to enable efficient by-reference message passing without races

Lightweight type-based approach to enforce uniqueness
Safe and Efficient Message Passing

- Sending mutable objects by reference may lead to data races
- Use unique references to enable **efficient by-reference message passing without races**
- Lightweight type-based approach to enforce uniqueness

The challenge: few source code annotations, usable by normal developers
Example

```scala
val sink = actor {
  react {
    case b: ArrayBuffer[Int] =>
      val first = b.remove(0)
      ...
  }
}

actor {
  val buf: ArrayBuffer[Int] @unique =
    new ArrayBuffer[Int]
  buf += 5
  sink ! buf
  buf.remove(0)
}
```
Example

```scala
val sink = actor {
  react {
    case b: ArrayBuffer[Int] =>
      val first = b.remove(0)
      ..
  }
}

actor {
  val buf: ArrayBuffer[Int] @unique =
    new ArrayBuffer[Int]
  buf += 5
  sink ! buf
  buf.remove(0)
}
```

Unique reference
val sink = actor {
  react {
    case b: ArrayBuffer[Int] =>
      val first = b.remove(0)
      ..
  }
}

actor {
  val buf: ArrayBuffer[Int] @unique = new ArrayBuffer[Int]
  buf += 5
  sink ! buf
  buf.remove(0)
}
val sink = actor {
    react {
        case b: ArrayBuffer[Int] =>
            val first = b.remove(0)
            ..
    }
}

actor {
    val buf: ArrayBuffer[Int] @unique =
        new ArrayBuffer[Int]
    buf += 5
    sink ! buf
    buf.remove(0)
}
Example

val sink = actor {
  react {
    case b: ArrayBuffer[Int] =>
      val first = b.remove(0)
      ...
  }
}

actor {
  val buf: ArrayBuffer[Int] @unique = new ArrayBuffer[Int]
  buf += 5
  sink ! buf
  buf.remove(0)
}
Formal Type System
Formal Type System

• Class-based object calculus
Formal Type System

- Class-based object calculus
- Unique types $\rho \triangleright C$ are guarded by capabilities $\rho$
Formal Type System

• Class-based object calculus

• Unique types $\rho \triangleright C$ are guarded by capabilities $\rho$

• $\rho = \text{access permission to a region in the heap}$
Formal Type System

- Class-based object calculus
- Unique types $\rho \triangleright C$ are guarded by capabilities $\rho$
- $\rho$ = access permission to a region in the heap
- Capabilities statically tracked $\Gamma ; \Delta \vdash t : T ; \Delta'$
Formal Type System

\[
\begin{align*}
\Gamma; \Delta \vdash e : T' ; \Delta' & \quad T' \ll T \\
\Gamma; \Delta \vdash e : T; \Delta' \\
\Gamma; \Delta \vdash e : T; \Delta' & \quad \Gamma, x : T; \Delta' \vdash t : T' ; \Delta'' \\
\Gamma; \Delta \vdash \text{let } x = e \text{ in } t : T' ; \Delta'' \\
\Gamma(y) = \rho \triangleright C & \quad \rho \in \Delta \\
\Gamma; \Delta \vdash y : \rho \triangleright C ; \Delta \\
\Gamma; \Delta \vdash y : \rho \triangleright C & \quad \text{fields}(C) = \alpha l : D \quad \alpha_i \neq \text{unique} \\
\Gamma; \Delta \vdash y.l_i : \rho \triangleright D_i ; \Delta \\
\Gamma; \Delta \vdash y : \rho \triangleright C & \quad \Gamma; \Delta \vdash z : \rho \triangleright D_i & \quad \text{fields}(C) = \alpha l : D \quad \alpha_i \neq \text{unique} \\
\Gamma; \Delta \vdash y.l_i := z : \rho \triangleright C ; \Delta \\
\Gamma; \Delta \vdash y : \rho \triangleright D & \quad \Delta = \Delta' \oplus \bar{\rho} & \quad \text{fields}(C) = \alpha l : D \quad \rho' \text{ fresh} \\
\Gamma; \Delta \vdash \text{new } C(\bar{y}) : \rho' \triangleright C ; \Delta' \oplus \rho'
\end{align*}
\]

(T-SUB)  
(T-LET)  
(T-VAR)  
(T-SELECT)  
(T-ASSIGN)  
(T-NEW)
Soundness

- Complete soundness proof

- Soundness implies that if a program type-checks it is guaranteed that at run time
  - *no consumed objects are accessed*
  - *no objects reachable from a consumed object are accessed*
Practical Experience

Plug-in for Scala compiler
## Practical Experience

### Plug-in for Scala compiler

<table>
<thead>
<tr>
<th></th>
<th>size [LOC]</th>
<th>changes [LOC]</th>
<th>property checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable collections</td>
<td>2046</td>
<td>60</td>
<td>transferable</td>
</tr>
<tr>
<td>partest</td>
<td>4182</td>
<td>61</td>
<td>actor isolation</td>
</tr>
<tr>
<td>ray tracer</td>
<td>414</td>
<td>18</td>
<td>actor isolation</td>
</tr>
</tbody>
</table>
Example (partest)

```scala
val logList: LogList @unique = new LogList
for (test <- tests) {
    val logFile: LogFile @unique =
        createLogFile(test, kind)
    // run test...
    logList.add(capture(logFile, logList))
}
report(logList)

def report(logList: LogList @unique) {
    master ! logList
}
```
Capabilities for Uniqueness and Borrowing
Capabilities for Uniqueness and Borrowing

• New approach to unique references
  • Simple capability tokens are enough
Capabilities for Uniqueness and Borrowing

• New approach to unique references
  • Simple capability tokens are enough

• Lightweight type system
  • No explicit regions/owners

• Syntactic soundness proof

• Applied to real-world, concurrent Scala code

Haller and Odersky. Capabilities for Uniqueness and Borrowing. ECOOP’10 (Google Scholar: 56 Citations)
<table>
<thead>
<tr>
<th>Research Topics</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Concurrent programming</strong></td>
<td><strong>Scala Actors</strong></td>
</tr>
<tr>
<td>Event-driven actors [JMLC’06]</td>
<td>Systems</td>
</tr>
<tr>
<td>Integrating threads and events [TCS’09, Artima’12]</td>
<td>Scala Asyc</td>
</tr>
<tr>
<td>Event-driven concurrency [SIP’13]</td>
<td>Systems</td>
</tr>
<tr>
<td><strong>Types + generic programming</strong></td>
<td><strong>Scala Actors</strong></td>
</tr>
<tr>
<td>Zero-copy messaging [ECOOP’10]</td>
<td>Systems</td>
</tr>
</tbody>
</table>
# Research Topics

<table>
<thead>
<tr>
<th>Concurrent programming</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven actors</td>
<td>Scala Actors</td>
</tr>
<tr>
<td>[JMLC’06]</td>
<td></td>
</tr>
<tr>
<td>Integrating threads and events</td>
<td>Scala Async</td>
</tr>
<tr>
<td>[TCS’09, Artima’12]</td>
<td></td>
</tr>
<tr>
<td>Event-driven concurrency</td>
<td>[SIP’13]</td>
</tr>
<tr>
<td>Types + generic programming</td>
<td></td>
</tr>
<tr>
<td>Zero-copy messaging</td>
<td>Pickling</td>
</tr>
<tr>
<td>[ECOOP’10]</td>
<td>Spores</td>
</tr>
<tr>
<td>Fast serialization</td>
<td></td>
</tr>
<tr>
<td>[OOPSLA’13]</td>
<td></td>
</tr>
<tr>
<td>Safe closure passing</td>
<td></td>
</tr>
<tr>
<td>[ECOOP’14]</td>
<td></td>
</tr>
</tbody>
</table>
Serialization: Desiderata
Serialization: Desiderata

• Type safety
Serialization: Desiderata

- Type safety
- Performance
Serialization: Desiderata

- Type safety
- Performance
- Extensibility
  - Exchange external representation
Insights
Insights

**Insight 1**: In many cases, static type information of serialized objects enables
Insights

**Insight 1**: In many cases, static type information of serialized objects enables

- checking for transitive serializability;
Insights

**Insight 1**: In many cases, **static type information** of serialized objects enables

- checking for transitive serializability;
- generating efficient serialization code.
Insights

**Insight 1**: In many cases, *static type information* of serialized objects enables

- checking for transitive serializability;
- generating efficient serialization code.

**Insight 2**: Existing language features of Scala are powerful enough to achieve all our requirements
Insights

**Insight 1:** In many cases, static type information of serialized objects enables
- checking for transitive serializability;
- generating efficient serialization code.

**Insight 2:** Existing language features of Scala are powerful enough to achieve all our requirements

Enables a new serialization framework without changing the Scala language or the underlying JVM!
Pickling
Pickling

- *Object-oriented pickler combinators*
Pickling

- *Object-oriented pickler combinators*
- A framework that
Pickling

- **Object-oriented pickler combinators**
- A framework that
  - enables *retrofitting existing types* with serializability
Pickling

- **Object-oriented pickler combinators**
- A framework that
  - enables *retrofitting existing types* with serializability
  - allows *exchanging external representations*
Pickling: Performance
Pickling: Performance

Which datatypes are used in practical distributed frameworks?

<table>
<thead>
<tr>
<th></th>
<th>primitives/primitive arrays</th>
<th>value-like types</th>
<th>collections</th>
<th>case classes</th>
<th>type descriptor</th>
<th>generics</th>
<th>subtyping polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoTrellis (Akka)</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Evactor (Akka)</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Spark</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Storm</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>N/A</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Twitter Chill</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

Legend: ●: Heavy Use  ○: Light Use  ○: No Use
Pickling: Performance

- Pickling/Unpickling Spark Datatypes, Linear Regression
- Pickling/Unpickling Geotrellis Datatypes
Pickling: Performance

Miller, Haller, Burmako, and Odersky. *Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization*. OOPSLA’13
Pickling: Performance

Miller, Haller, Burmako, and Odersky. *Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization*. OOPSLA’13
Serializing Functions?
Serializing Functions?

• Common in data analytics frameworks (Apache Spark, Twitter’s Scalding, etc.)
Serializing Functions?

• Common in data analytics frameworks (Apache Spark, Twitter’s Scalding, etc.)

• Even without language support in widespread systems like Hadoop MapReduce (verbose class definition syntax)
Serializing Functions?

• Common in data analytics frameworks (Apache Spark, Twitter’s Scalding, etc.)

• Even without language support in widespread systems like Hadoop MapReduce (verbose class definition syntax)

• **Problem: source of safety hazards**
Hazards
Hazards

• Runtime serialization errors due to capturing non-serializable objects
Hazards

• Runtime serialization errors due to capturing non-serializable objects

• Race conditions due to capturing mutable references
Hazards

- Runtime serialization errors due to capturing non-serializable objects
- Race conditions due to capturing mutable references
- Memory leaks
Hazards

• Runtime serialization errors due to capturing non-serializable objects

• Race conditions due to capturing mutable references

• Memory leaks

• ...

Philipp Haller
Example: Apache Spark

class MyCoolRddApp {
  val param = 3.14
  val log = new Log(...)
  ...
  def work(rdd: RDD[Int]) {
    rdd.map(x => x + param)
      .reduce(...)
  }
}
Example: Apache Spark

class MyCoolRddApp {
  val param = 3.14
  val log = new Log(...)
  ...
  def work(rdd: RDD[Int]) {
    rdd.map(x => x + param)
      .reduce(...)
  }
}
Example: Akka

```scala
def receive = {
  case Request(data) =>
    future {
      val result = transform(data)
      sender ! Response(result)
    }
}
```
Spores

• Spores: “safe” closures

• Avoid hazards through **explicit, typed closure environments**

```scala
val s = spore {
  val h = helper
  (x: Int) => {
    val result = x + " " + h.toString
    println("The result is: " + result)
  }
}
```
Spores

• Spores: “safe” closures

• Avoid hazards through explicit, typed closure environments

val s = spore {
  val h = helper
  (x: Int) => {
    val result = x + " " + h.toString
    println("The result is: " + result)
  }
}

Miller, Haller and Odersky. *Spores: a type-based foundation for closures in the age of concurrency and distribution*. ECOOP’14
Ongoing Work: Safe Closure Passing
Ongoing Work: Safe Closure Passing

• “Make programming interfaces of data analytics frameworks safer”
Ongoing Work: Safe Closure Passing

• “Make programming interfaces of data analytics frameworks safer”

• Functional processing of distributed data
Ongoing Work: Safe Closure Passing

• “Make programming interfaces of data analytics frameworks safer”

• Functional processing of distributed data

```scala
val persons: SiloRef[List[Person]] = ...
val adults = persons.map(spore {
  ps => ps.filter(p => p.age >= 18)
})
```
Acknowledgments

- **Scala Actors**
  - [JMLC’06, TCS’09] by Philipp Haller and Martin Odersky
  - [Artima’12] by Philipp Haller and Frank Sommers
  - [Coordination’08] by Philipp Haller and Tom Van Cutsem

- **Capabilities for Uniqueness and Borrowing**
  - [ECOOP’10] by Philipp Haller and Martin Odersky

- **Asynchronous and Reactive Programming**
  - [LCPC’12] by Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky
  - [SIP’13] by Philipp Haller and Jason Zaugg

- **Data-Centric Programming**
  - [OOPSLA’13] by Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky
  - [ECOOP’14] by Heather Miller, Philipp Haller, and Martin Odersky
Future Directions
Future Directions

• Type systems
  • Uniqueness types “for the masses”
  • Generic type and effect systems
Future Directions

• *Type systems*
  • Uniqueness types “for the masses”
  • Generic type and effect systems

• *Concurrent and distributed programming*
  • Failures, persistence, consistency, etc.
Future Directions

• **Type systems**
  • Uniqueness types “for the masses”
  • Generic type and effect systems

• **Concurrent and distributed programming**
  • Failures, persistence, consistency, etc.

• **Programming languages for Big Data**
  • Safe, high-level functional abstractions for efficient “big” memory management
Future Directions

• *Type systems*
  • Uniqueness types “for the masses”
  • Generic type and effect systems

• *Concurrent and distributed programming*
  • Failures, persistence, consistency, etc.

• *Programming languages for Big Data*
  • Safe, high-level functional abstractions for efficient “big” memory management