Scala for Multicore

Part 1: Foundations and Message-Passing Concurrency

Philipp Haller, Stanford University and EPFL

Monday, June 20, 2011
Scala for Multicore

Part 1: Foundations and Message-Passing Concurrency

Philipp Haller, Stanford University and EPFL

Resources online at: http://lamp.epfl.ch/~phaller/upmarc

Monday, June 20, 2011
What is Scala?
What is Scala?

Scala is a **statically-typed language** that integrates object-oriented and functional programming.

Resources at http://lamp.epfl.ch/~phaller/upmarc
What is Scala?

Scala is a statically-typed language that integrates object-oriented and functional programming

- Uniform object model

Resources at http://lamp.epfl.ch/~phaller/upmarc
What is Scala?

Scala is a **statically-typed language** that integrates object-oriented and functional programming

- Uniform object model
- Higher-order functions and pattern matching

Resources at [http://lamp.epfl.ch/~phaller/upmarc](http://lamp.epfl.ch/~phaller/upmarc)
What is Scala?

Scala is a **statically-typed language** that integrates object-oriented and functional programming

- Uniform object model
- Higher-order functions and pattern matching
- Novel ways to compose and abstract expressions

Resources at http://lamp.epfl.ch/~phaller/upmarc
What is Scala?

Scala is a **statically-typed language** that integrates object-oriented and functional programming

- Uniform object model
- Higher-order functions and pattern matching
- Novel ways to compose and abstract expressions

Scala runs on the Java Virtual Machine and is completely interoperable with Java

- Compiler preview for Microsoft .NET

Resources at [http://lamp.epfl.ch/~phaller/upmarc](http://lamp.epfl.ch/~phaller/upmarc)
It’s a promising language.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Why are people adopting Scala?
Replaced their Ruby-based back-end services with Scala. Using actors, they could scale their concurrent message queue system to a larger number of users.
"[...] 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."
"[...] 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.**
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

Resources at http://lamp.epfl.ch/~phaller/upmarc
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

Strong support for functional programming.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

- Strong support for functional programming.
- Enables embedded DSLs for concurrency and parallelism.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

- Strong support for functional programming.
- Enables embedded DSLs for concurrency and parallelism.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Why is FP Crucial for Parallel Programming?
Why is FP Crucial for Parallel Programming?

Parallel programming is very hard.

- Data races, deadlocks, memory effects, ...
Why is FP Crucial for Parallel Programming?

Parallel programming is very hard.

- Data races, deadlocks, memory effects, ...

**Reason:** non-deterministic thread interleavings.

- Interleavings observable because of shared state.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Why is FP Crucial for Parallel Programming?

Parallel programming is very hard.
- Data races, deadlocks, memory effects, ...

**Reason:** non-deterministic thread interleavings.
- Interleavings observable because of shared state.

Therefore, by eliminating mutable state we can exclude concurrency hazards!

Resources at http://lamp.epfl.ch/~phaller/upmarc
Why is FP Crucial for Parallel Programming?

Parallel programming is very hard.

- Data races, deadlocks, memory effects, ...

Reason: non-deterministic thread interleavings.

- Interleavings observable because of shared state.

Therefore, by eliminating mutable state we can exclude concurrency hazards!

Functional Programming is the only productive way to work with immutable data structures.
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

- Strong support for functional programming.
- Enables embedded DSLs for concurrency and parallelism.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

Strong support for functional programming.

Enables embedded DSLs for concurrency and parallelism.

Resources at http://lamp.epfl.ch/~phaller/upmarc
Domain-Specific Languages

Scala’s flexible syntax makes it easy to define embedded DSLs

**EXAMPLES:**
Erlang-style actors, X10-style async/finish

```scala
// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}
```

Resources at http://lamp.epfl.ch/~phaller/upmarc
Domain-Specific Languages

Scala’s flexible syntax makes it easy to define embedded DSLs

**Examples:**
Erlang-style actors, X10-style async/finish

// asynchronous message send
actor ! message

// message receive
receive {
    case msgpat₁ => action₁
    ...
    case msgpatₙ => actionₙ
}

Compiler plug-ins enable safety checking

Resources at http://lamp.epfl.ch/~phaller/upmarc
Domain-Specific Languages

Scala’s flexible syntax makes it easy to define embedded DSLs

**EXAMPLES:**
Erlang-style actors, X10-style async/finish

```scala
// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat₁ => action₁
  ...
  case msgpatₙ => actionₙ
}
```

Compiler plug-ins enable **safety checking**

Embedding enables **interoperability**

Resources at http://lamp.epfl.ch/~phaller/upmarc
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

Strong support for functional programming.

Enables embedded DSLs for concurrency and parallelism.
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?

- Strong support for functional programming.
- Enables embedded DSLs for concurrency and parallelism.

These things are a step in the right direction towards Popular Parallel Programming.
Scala and Parallelism.

People are adopting Scala because it is a good basis for parallel programming.

Why?
- Strong support for functional programming.
- Enables embedded DSLs for concurrency and parallelism.

Our Goal: Make “Popular Parallel Programming” possible.
Scala’s Toolbox for Parallel Programming

- ACTORS
- PARALLEL GRAPH PROCESSING
- STM
- PARALLEL DSLs
- FUTURES
- PARALLEL COLLECTIONS
- DISTRIBUTED
Scala’s Toolbox for Parallel Programming

- ACTORS
- PARALLEL GRAPH PROCESSING
- STM
- PARALLEL DSLs
- FUTURES
- PARALLEL COLLECTIONS
- DISTRIBUTED
The Lectures.

**TODAY**

Intro to Scala  
Scala Actors  
Parallel Graph Processing

**TOMORROW**

Parallel Collections  
Parallel DSLs  
PhD Tips
Scala: 
The Basics
An Example Class.

public class Person {
    public final String name;
    public final int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

In Java:
An Example Class.

In Java:

```java
public class Person {
    public final String name;
    public final int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

In Scala:

```scala
class Person(val name: String,
             val age: Int) {}
```
... and its use

In Java:

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{
    ArrayList<Person> minorsList = new ArrayList<Person>();
    ArrayList<Person> adultsList = new ArrayList<Person>();
    for (int i = 0; i < people.length; i++)
        (people[i].age < 18 ? minorsList : adultsList)
            .add(people[i]);
    minors = minorsList.toArray(people);
    adults = adultsList.toArray(people);
}
```
... and its use

In Java:

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{  ArrayList<Person> minorsList = new ArrayList<Person>();
    ArrayList<Person> adultsList = new ArrayList<Person>();
    for (int i = 0; i < people.length; i++)
      (people[i].age < 18 ? minorsList : adultsList).
      .add(people[i]);
    minors = minorsList.toArray(people);
    adults = adultsList.toArray(people);
}
```

In Scala:

```scala
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```
... and its use

In Java:

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{
    ArrayList<Person> minorsList = new ArrayList<Person>();
    ArrayList<Person> adultsList = new ArrayList<Person>();
    for (int i = 0; i < people.length; i++)
        (people[i].age < 18 ? minorsList : adultsList).
            add(people[i]);
    minors = minorsList.toArray(people);
    adults = adultsList.toArray(people);
}
```

In Scala:

```scala
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```
... and its use

In Java:

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{
    ArrayList<Person> minorsList = new ArrayList<Person>();
    ArrayList<Person> adultsList = new ArrayList<Person>();
    for (int i = 0; i < people.length; i++)
    {
        (people[i].age < 18 ? minorsList : adultsList).
            add(people[i]);
    minors = minorsList.toArray(people);
    adults = adultsList.toArray(people);
}
```

In Scala:

```scala
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```
... and its use

In Java:

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{  ArrayList<Person> minorsList = new ArrayList<Person>();
    ArrayList<Person> adultsList = new ArrayList<Person>();
    for (int i = 0; i < people.length; i++)
        (people[i].age < 18 ? minorsList : adultsList)
            .add(people[i]);
    minors = minorsList.toArray(people);
    adults = adultsList.toArray(people);
}
```

In Scala:

```scala
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```

a simple pattern match
Scala unifies class hierarchies and abstract data types (ADTs)
Class Hierarchies and ADTs.

Scala unifies class hierarchies and abstract data types (ADTs)

Introducing pattern matching for objects.
Class Hierarchies and ADTs.

Scala unifies class hierarchies and abstract data types (ADTs)

- Introducing **pattern matching** for objects.
- Concise manipulation of immutable data structures.
Pattern Matching.

Class hierarchy for binary trees:

```scala
abstract class Tree[T]
case object Empty extends Tree[Nothing]
case class Binary[T](elem: T, left: Tree[T], right: Tree[T]) extends Tree[T]
```

In-order traversal:

```scala
def inOrder[T](t: Tree[T]): List[T] = t match {
  case Empty => List()
  case Binary(e, l, r) =>
    inOrder(l) ::: List(e) ::: inOrder(r)
}
```

- Extensibility
- Encapsulation: only constructor params exposed
- Representation independence [ECOOP’07]
Functions and Collections.

First-class functions make collections more powerful

Especially immutable ones

```scala
people.filter(_.age >= 18).
groupBy(_.surname): Map[String, List[Person]]
.values : Iterable[List[Person]]
.count(_.length >= 2)
```
Functions and Collections.

✗ First-class functions make collections more powerful

✗ Especially immutable ones

```scala
people.filter(_.age >= 18)
.groupBy(_.surname): Map[String, List[Person]]
.values: Iterable[List[Person]]
.count(_.length >= 2)
```
Functions are Objects.
Functions are Objects.

Every function is a value

- Values are objects => functions are objects
Functions are Objects.

Every function is a value

Values are objects => functions are objects

The function type \( S \to T \) is equivalent to the class type `java.Function1[S, T]`:

```scala
trait Function1[-S, +T] {
  def apply(x: S): T
}
```
Functions are Objects.

Every function is a value

- Values are objects => functions are objects

The function type \( S \Rightarrow T \) is equivalent to the class type `scala.Function1[S, T]`

```scala
trait Function1[-S, +T] {
  def apply(x: S): T
}
```

For example, the anonymous successor function `(x: Int) => x + 1` (short `_ + 1`) is expanded to:

```scala
new Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
```
Arrays are Objects.

Arrays = mutable functions over integer ranges

Syntactic sugar:
\[
a(i) = a(i) + 2 \text{ for } a\text{.update}(i, a\text{.apply}(i) + 2)
\]

```scala
definal class Array[T](_length: Int) extends java.io.Serializable
    with java.lang.Cloneable {

    def length: Int = ...
    def apply(i: Int): T = ...
    def update(i: Int, x: T): Unit = ...
    override def clone: Array[T] = ...
}
```
Partial Functions.

Functions that are defined only for some objects

Test using `isDefinedAt`

```scala
trait PartialFunction[-A, +B] extends (A => B) {
  def isDefinedAt(x: A): Boolean
}
```

Blocks of pattern-matching cases are instances of partial functions

This lets one write control structures that are not easily expressible otherwise
Actors in Scala.

Send/receive constructs adopted from **Erlang**

Send is asynchronous: messages are buffered in actor’s **mailbox**

Receive picks the first message in the mailbox that matches one of the patterns `msgpat_i`

If no pattern matches the actor suspends

```
// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}
```

Partial function of the type, `PartialFunction[Msg, Action]`
Actors in Scala.

Send/receive constructs adopted from Erlang

Send is asynchronous: messages are buffered in actor’s mailbox

Receive picks the first message in the mailbox that matches one of the patterns msgpat_i

If no pattern matches the actor suspends

// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}

Partial function of the type, PartialFunction[Msg, Action]
Actors in Scala.

Send/receive constructs adopted from **Erlang**

Send is asynchronous: messages are buffered in actor’s **mailbox**

Receive picks the first message in the mailbox that matches one of the patterns msgpat\(_i\)

If no pattern matches the actor suspends

---

// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat\(_1\) => action\(_1\)
  ...
  case msgpat\(_n\) => action\(_n\)
}

Partial function of the type, `PartialFunction[Msg, Action]`
A Simple Actor.

```scala
val summer = actor {
    var sum = 0
    loop {
        receive {
            case ints: Array[Int] =>
                sum += ints.reduceLeft((a, b) => (a+b)%7)
            case from: Actor =>
                from ! sum
        }
    }
}
```
Erlang-style Actors.
Erlang-style Actors.

- No inversion of control
- Message reception is explicit and blocking
Erlang-style Actors.

- No inversion of control
  - Message reception is explicit and blocking
- Fine-grained message filtering
  - Messages are filtered upon reception
Erlang-style Actors.

- No inversion of control
  - Message reception is explicit and blocking
- Fine-grained message filtering
  - Messages are filtered upon reception
- **NOT** Erlang-style actors: E, Kilim, ActorFoundry
Erlang-style Actors.

- No inversion of control
  - Message reception is explicit and blocking

- Fine-grained message filtering
  - Messages are filtered upon reception

- **NOT** Erlang-style actors: E, Kilim, ActorFoundry

- Incentive: programmer productivity
Goal of Scala Actors?

Programming system for Erlang-style actors that:
Goal of Scala Actors?

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
Goal of Scala Actors?

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
- integrates with thread-based code;
Goal of Scala Actors?

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
- integrates with thread-based code;
- provides safe and efficient message passing.
Implementing Actors.

Thread-based implementation:
Implementing Actors.

Thread-based implementation:

✗ One thread per actor
Implementing Actors.

Thread-based implementation:

- One thread per actor
- JVM maps threads to OS processes
Implementing Actors.

Thread-based implementation:

- One thread per actor
- JVM maps threads to OS processes
- Receive blocks thread while waiting for message
Implementing Actors.

Thread-based implementation:
- One thread per actor
- JVM maps threads to OS processes
- Receive blocks thread while waiting for message

**Pros**
- No inversion of control.
- Interoperability with threads.

**Cons**
- High memory consumption.
- Context switching overhead.
Event-Based Actors.
Event-Based Actors.

**Main Problem** of thread-per-actor model:

Actors consume a lot of resources while waiting for messages.
Event-Based Actors.

**Main Problem** of thread-per-actor model:

Actors consume a lot of resources while waiting for messages.

**Idea:** Suspend actor by saving continuation closure and releasing current thread

Transparent thread pooling
Event-Based Actors.

**Main Problem** of thread-per-actor model:
Actors consume a lot of resources while waiting for messages.

**Idea:** Suspend actor by saving continuation closure and releasing current thread

Transparent thread pooling

```scala
def act() {
  react { case Put(x) =>
    react { case Get(from) =>
      from ! x
      act()
    }
  }
}
```
Invocations of `react` do not return
Must provide continuation in body of `react`

Does this mean we have to write code in continuation-passing style?
No, control-flow combinators enable modular composition

```scala
a andThen b // runs a followed by b
```
```scala
def loop(body: => Unit) = body andThen loop(body)
```
Programming with react

Invocations of react do not return
- Must **provide continuation** in body of react

Does this mean we have to write code in continuation-passing style?
- No, control-flow combinators enable modular composition

```
a andThen b //runs a followed by b
```
```
def loop(body: => Unit) = body andThen loop(body)
```
Thread-based Programming

Actors should be able to block their thread temporarily:

- When interacting with thread-based code
- When it is difficult to provide the continuation

```scala
val tasks: List[Task]
tasks foreach { task => worker ! task }
val results = tasks map { task =>
  receive {
    case Done(result) => result
  }
}
```

Blocks current thread if actor has to wait for a message
Managing Blocking.

Actors (many)

Thread Pool

worker threads (few)
Managing Blocking.

A diagram illustrating the relationship between actors and a thread pool. The diagram shows multiple threads with task queues, which are managed by the thread pool. The thread pool contains a limited number of worker threads (few) that manage the task queues for many actors (many).
Managing Blocking.

Actor A:
- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool

worker threads (few)
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool

Task Queue

Worker threads (few)
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  receive {
    case Next =>
  }
Managing Blocking.

Actor A:
- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool:
- Task queue
- Task queue
- Task queue
- Task queue

worker threads (few)

Actors (many)
Managing Blocking.

Actor A:

- Start 3 actors
- Then: receive {
  case Next =>
}

Thread Pool:

- receive {
  case Put(x) =>
}

worker threads (few)

Actors (many)
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  receive {
    case Next =>
  }

### Thread Pool

- task queue
- task queue
- task queue
- worker threads (few)

Actors (many)

```scala
receive {
  case Put(x) =>
}
```
Managing Blocking.

Actor A:
- Start 3 actors
- Then:
  - receive { case Next => }
  - receive { case Put(x) => }
  - receive { case Put(x) => }

Thread Pool:
- Tasks (many)
- Worker threads (few)
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread Pool:

- Actors (many)
- Receive:
  - case Put(x) =>

Worker threads (few)
Managing Blocking.

Actor A:
- Start 3 actors
- Then:
  - receive { case Next => }

Thread Pool:
- Actor A!
  - receive { case Put(x) => }
- Actors (many)
  - receive { case Put(x) => }
- worker threads (few)
Actor A:

• Start 3 actors
• Then:
  receive {
    case Next =>
  }

Thread Pool

worker threads (few)

Actors (many)

actor {
  A ! Next
}
receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}
Managing Blocking.

Actor A:

- Start 3 actors
- Then:
  - receive {
    - case Next =>
  }

Thread Pool:

- worker threads (few)

Actors (many)

- receive {
  - case Put(x) =>
}

- receive {
  - case Put(x) =>
}

- receive {
  - case Put(x) =>
}
Managing Blocking.

Actor A:

- Start 3 actors
- Then: receive { case Next => }

thread pool

worker threads (few)

Actors (many)

actor {
  A ! Next
} receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}
Managing Blocking.

Actor A:
- Start 3 actors
- Then: receive { case Next => }

Thread pool locked up!

Actor (many)

worker threads (few)

receive { case Put(x) => }

receive { case Put(x) => }

receive { case Put(x) => }

receive { case Put(x) => }

actor {
  A ! Next
}
receive { case Put(x) => }

receive { case Put(x) => }

receive { case Put(x) => }

周一，2011年6月20日
Managing Blocking.

Actor A:
- Start 3 actors
- Then:
  receive {
    case Next =>
  }

Thread pool locked up!

MUST AVOID situation where:
- all worker threads blocked.
- there is a task in some task queue.

Worker threads (few)

Actors (many)

actor {
  A ! Next
}
receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}

receive {
  case Put(x) =>
}
def receive[R](f: PartialFunction[Any, R]): R = {
  ...
  val elem = mailbox.extractFirst(msg => f.isDefinedAt(msg))
  if (elem == null) {
    synchronized {
      waitingFor = f
      isSuspended = true
      scheduler.managedBlock(blocker)
    }
  }
  else {
    // process message...
  }
  ...
}
def receive[R](f: PartialFunction[Any, R]): R = {
  ...
  val elem = mailbox.extractFirst(msg => f.isDefinedAt(msg))
  if (elem == null) {
    synchronized {
      waitingFor = f
      isSuspended = true
      scheduler.managedBlock[blocker]
    }
  } else {
    // process message...
  }
}

object blocker extends ManagedBlocker {
  def block() = {
    Actor.this.suspendActor()
    true
  }
  def isReleasable =
    !Actor.this.isSuspended
}
“We saw 5x normal tweets-per-second and about 4x tweets-per-minute as this chart illustrates. Overall, Twitter sailed smoothly through the inauguration [...]”
Goal of Scala Actors?

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
- integrates with thread-based code;
- provides safe and efficient message passing.

UNIFIED actors
Goal of Scala Actors? **REVISITED.**

Programming system for Erlang-style actors that:
- offers high scalability on mainstream platforms;
- integrates with thread-based code;
- provides safe and efficient message passing.

**EVENT-BASED** actors
- no inversion of control
- no changes to the JVM
- no CPS transform

Goal of Scala Actors?  

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
- integrates with thread-based code;
- provides safe and efficient message passing.

- Temporarily & safely monopolize thread
- Interact with thread-based code


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.
- (Deep) copying messages upon sending is safe but inefficient.
Safe and Efficient Message Passing.

- Sending mutable objects by reference may lead to data races.
- (Deep) copying messages upon sending is safe but inefficient.
- 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.

(Deep) copying messages upon sending is safe but inefficient.

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

Lightweight type-based approach to enforce uniqueness.
Internal vs. External Aliases

local variable

internal alias of x

external alias of x

Region of x

Monday, June 20, 2011
Separate Uniqueness.
Separate Uniqueness.

A reference is unique if it is the only reference pointing into some region.
Separate Uniqueness.

- A reference is unique if it is the only reference pointing into some region

- Unique references may only have temporary external aliases
Separate Uniqueness.

A reference is unique if it is the only reference pointing into some region.

Unique references may only have temporary external aliases.

A region may be transferred between actors using a unique reference; transferring invalidates the unique reference.
Annotation System

@unique
Unique variable/parameter/result

@transient
Non-consumable (borrowed) unique parameter

@peer(x)
Parameter/result in the same region as x

x capturedBy y
Alias of x in region of y; consumes x

swap(x.f, y)
Return unique x.f and replace with unique y

No explicit regions/owners
No static alias analysis
Supports closures and nested classes
Unique Variables and Regions

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

def report(logList: LogList @unique) {
  master ! new Results(logList)
}
```

logFile in disjoint region of logList
Mutating Unique Objects.

```scala
class LogList {
  var elem: LogFile = null
  var next: LogList = this
  @transient def add(file: LogFile @peer(this)) =
  if (isEmpty) {
    elem = file; next = new LogList
  } else next.add(file)
}
```

- Receiver must remain unique after adding file
- `@transient` is equivalent to `@unique` except it does not consume the receiver
- file must point into the same region as the receiver, expressed using `@peer(this)`
Transferring Unique Objects.

How can we transfer a separately-unique object from one region to another?

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

Returns alias of `logFile` in region of `logList`

Consumes `logFile`
Transferring Unique Objects.

How can we transfer a separately-unique object from one region to another?

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

- Returns alias of `logFile` in region of `logList`
- Consumes `logFile`
Alias Invariant

Two variables x, y are separate (in heap H) iff there is no object reachable from both x and y.

Definition (Separate Uniqueness):

A variable x is separately-unique in heap H iff for all y != x, y is live => separate (H, x, y)

Definition (Alias Invariant):

Unique parameters are separately-unique
Formal Type System

- Class-based object calculus with capabilities and capturedBy/swap
- A unique variable has type $\rho \triangleright C$
- Capability $\rho = \text{access permission to a region in heap}$

**Definition (Capability Type Invariant):**

Let $x: \rho \triangleright C$ and $x': \rho' \triangleright C'$ be local variables ($\rho \neq \rho'$). If there is a heap $H$ at program point $P$ such that both $x$ and $y$ are live at $P$, then separate($H, x, y$)
Type Checking

Typing judgment: \( \Gamma ; \Delta \mathcal{X} t : T ; \Delta' \)

Type rules consume capability set \( \Delta \) and produce capability set \( \Delta' \)

Capabilities in \( \Delta \) grant access to variables in \( t \)

A variable of type \( \rho \triangleright C \) can only be accessed if \( \rho \) is contained in \( \Delta \)

Capabilities in \( \Delta' \) available after type checking \( t \)
Type Checking

Typing judgment: $\Gamma ; \Delta \times t : T ; \Delta'$

Type rules consume capability set $\Delta$ and produce capability set $\Delta'$

Capabilities in $\Delta$ grant access to variables in $t$

- A variable of type $\rho \rightarrow C$ can only be accessed if $\rho$ is contained in $\Delta$

Capabilities in $\Delta'$ available after type checking $t$
Capability Creation/Consumption

Instance creation:

\[
\Gamma ; \Delta \vdash y : \rho \triangleright D \\
\Delta = \Delta' \oplus \bar{\rho} \\
\text{fields}(C) = \alpha l : D \\
\rho' \text{ fresh} \\
\Gamma ; \Delta \vdash \text{new } C(y) : \rho' \triangleright C ; \Delta' \oplus \rho'
\]
Separation and Internal Aliasing

Field assignment:

\[
\begin{align*}
\Gamma; \Delta \vdash y : \rho \triangleright C \\
\text{fields}(C) = \alpha l : D \\
\alpha_i \neq \text{unique} \\
\Gamma; \Delta \vdash y.l_i := z : \rho \triangleright C ; \Delta
\end{align*}
\]
Separate Uniqueness

- Assume \( x \) has type \( \rho \triangleright C \)

- **Capability type invariant**: if there is a heap \( H \) where \( \neg \text{separate}(H, x, y) \), then \( y \) has type \( \rho \triangleright D \)

- Consuming \( \rho \) makes all variables of type \( \rho \triangleright D \) unusable

- Consuming \( \rho \) makes all external aliases of \( x \) unusable

- Invoking a method consumes capabilities of unique arguments
Soundness

- Small-step operational semantics
- Soundness established using syntactic Wright-Felleisen Technique
  - Preservation: Reduction preserves uniqueness and separation invariants
  - Progress: Well-typed programs do not get stuck because of missing capabilities
Immutable Types

- Instances of *immutable classes* are deeply immutable
- Allow immutable objects to be reachable from two different regions
- Capabilities guarding *immutable instances are not consumed*
Immutable Types

- Instances of **immutable classes** are deeply immutable
- Allow immutable objects to be reachable from two different regions
- Capabilities guarding **immutable instances are not consumed**

\[
\Gamma; \Delta \vdash y : \rho \triangleright C \quad \Gamma; \Delta \vdash z : \rho' \triangleright C'
\]

\[
\Delta = \begin{cases} 
\Delta' & \text{if } C \in I \\
\Delta' \oplus \rho & \text{otherwise}
\end{cases}
\]

\[
\Gamma; \Delta \vdash y \text{ capturedBy } z : \rho' \triangleright C ; \Delta'
\]
Actors and Concurrency

Add !, receive, and actor creation expressions

**Reduction**
- Actor = sequential execution state + mailbox
- Rules for reducing a set of actors in the context of a shared heap

**Typing**
- Actors are instances of `Actor` subclasses
- `Send` consumes non-immutable arguments
- `Receive` returns unique references
Actor Isolation

*Isolation theorem:*

Variables accessible by different actors are separate up to immutable objects

→ *Corollary (with progress):*
  only immutable objects are accessed concurrently
Implementation and Experience

Plug in for Scala compiler
- Erases capabilities and `capturedBy` for code generation

Practical experience:

<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>collections self-contained</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>
Implementation and Experience

Plug in for Scala compiler

- Erases capabilities and `capturedBy` for code generation

Practical experience:

<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>collections self-contained</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></td>
</tr>
</tbody>
</table>

**Types:** `DoubleLinkedList`, `ListBuffer`, and `HashMap` including all transitively extended traits.
External vs. Separate Uniqueness

**EXTERNAL UNIQUENESS**
- No external aliases
- No unique method receivers
- Deep/full encapsulation

**SEPARATE UNIQUENESS**
- Local external aliases
- Unique method receivers (self transfer)
- Full encapsulation

[Clarke, Wrigstad 2003; Müller, Rudich 2007; Clarke et al. 2008]
Goal of Scala Actors?

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
- integrates with thread-based code;
- provides safe and efficient message passing.

REVISITED. (AGAIN)
Goal of Scala Actors?

REVISITED. (AGAIN)

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
- integrates with thread-based code;
- provides safe and efficient message passing.

CAPABILITIES FOR UNIQUENESS

- Lightweight pluggable type system.
- Race-freedom through actor isolation.

Haller and Odersky. Capabilities for uniqueness and borrowing, Proc. ECOOP, 2010
Summary: Actors

- Scalable Erlang-style actors
- Integration of thread-based and event-based programming
- Used in large-scale production systems
- Lightweight uniqueness types for actor isolation
  - No explicit regions/owners
  - Soundness and actor isolation proofs
Data is growing. At the same time, there is a growing desire to do MORE with that data.

143 days

That is how long I must wait for my 5400 simulations to finish running. I started this process more than 50 hours ago, thinking it would be done Tuesday. Maleki and Donoho are not kidding when they write, it would have required several years to complete our study on a single modern desktop computer.
As an example, **MACHINE LEARNING (ML)** has provided elegant and sophisticated solutions to many complex problems on a small scale,
As an example,

**MACHINE LEARNING (ML)**

has provided elegant and sophisticated solutions to many complex problems on a small scale,

could open up **NEW APPLICATIONS + NEW AVENUES OF RESEARCH** if ported to a larger scale
As an example,

**MACHINE LEARNING (ML)**

has provided elegant and sophisticated solutions to many complex problems on a small scale,

but efforts are routinely limited by complexity and running time of **SEQUENTIAL** algorithms.
As an example,

**Machine Learning (ML)**

- has provided elegant and sophisticated solutions to many complex problems on a small scale,
- but efforts are routinely limited by complexity and running time of **sequential** algorithms.

*described as,*

- a community full of “**entrenched procedural programmers**”

typically focus on optimizing sequential algorithms when faced with scaling problems.
As an example,

**Machine Learning (ML)**

has provided elegant and sophisticated solutions to many complex problems on a small scale,

but efforts are routinely limited by complexity and running time of **sequential** algorithms.

**described as,**

a community full of “**ENTRENCHED PROCEDURAL PROGRAMMERS**”

typically focus

need to make it easier to

experiment with parallelism

when faced

Monday, June 20, 2011
What about MapReduce?
What about MapReduce?

Poor support for iteration.

MapReduce instances must be chained together in order to achieve iteration.

⚠️ Not always straightforward.
Even building non-cyclic pipelines is hard (e.g., FlumeJava, PLDI’10).

⚠️ Overhead is significant.
Communication, serialization (e.g., Phoenix, IISWC’09).
Menthor...
Menthor... is a framework for parallel graph processing. (But it is not limited to graphs.)
Menthor...

is a framework for parallel graph processing.
(But it is not limited to graphs.)

is inspired by BSP.
With functional reduction/aggregation mechanisms.
**Menthor**...

- is a framework for parallel graph processing. (But it is not limited to graphs.)
- is inspired by BSP.
  - With functional reduction/aggregation mechanisms.
- avoids an inversion of control of other BSP-inspired graph-processing frameworks.
**Menthor...**

- is a framework for parallel graph processing.
  (But it is not limited to graphs.)

- is inspired by BSP.
  With functional reduction/aggregation mechanisms.

- avoids an inversion of control
  of other BSP-inspired graph-processing frameworks.

- is implemented in Scala,
  and there is a preliminary experimental evaluation.
Menthor’s Model of Computation.
Data.
Data.

Split into data items managed by vertices.
and sizes range from primitives to large matrices.
Data.

Split into data items managed by *vertices*.
Relationships expressed using *edges* between vertices.
Algorithms.
Data items stored inside of vertices *iteratively* updated.
Algorithms.

Data items stored inside of vertices \textit{iteratively} updated.

Iterations happen as \texttt{SYNCHRONIZED SUPERSTEPS}. 

(inspired by the BSP model)
Algorithms.

Data items stored inside of vertices \textit{iteratively} updated. Iterations happen as SYNCHRONIZED SUPERSTEPS.
Algorithms.

Data items stored inside of vertices \textit{iteratively} updated. Iterations happen as \textbf{Synchronous Supersteps}.

\[ \text{update each vertex in parallel.} \]
Algorithms.

Data items stored inside of vertices iteratively updated. Iterations happen as SYNCHRONIZED SUPERSTEPS.

1. update each vertex in parallel.

2. update produces outgoing messages to other vertices
Algorithms.

Data items stored inside of vertices \textit{iteratively} updated. Iterations happen as \textit{Synchronized Supersteps}.

1. update each vertex in \textit{parallel}.
2. \textit{update} produces \textit{outgoing} messages to other vertices
3. \textit{incoming} messages available at the beginning of the next \textit{Superstep}. 
Substeps. (and Messages)

SUBSTEPS are computations that,
**Substeps. (and Messages)**

**SUBSTEPS** are computations that,

\[I.\] update the value of **this Vertex**
**Substeps. (and Messages)**

**Substeps** are computations that,

1. update the value of **this** Vertex
2. return a list of messages:

```scala
case class Message[Data](source: Vertex[Data], dest: Vertex[Data], value: Data)
```
**Substeps. (and Messages)**

**Substeps** are computations that,

1. update the value of this Vertex
2. return a list of messages:

```scala
case class Message[Data](source: Vertex[Data], dest: Vertex[Data], value: Data)
```

**Examples...**

```scala```
{
  value = ...
  List()
}
```
**Substeps. (and Messages)**

**Substeps** are computations that,

1. update the value of *this* Vertex
2. return a list of messages:
   ```scala
case class Message[Data](source: Vertex[Data], dest: Vertex[Data], value: Data)
```

**Examples...**

```scala
{  
  value = ...
  List()
}
```

```scala
{  
  ...
  for (nb <- neighbors)  
    yield Message(this, nb, value)
}
```
Substeps. (and Messages)

**Substeps** are computations that,

1. update the value of *this* Vertex
2. return a list of messages:
   
   ```scala
   case class Message[Data](source: Vertex[Data],
                             dest: Vertex[Data],
                             value: Data)
   ```

**Examples...**

Each is *implicitly* converted to a `Substep[Data]`
Some Examples...
class PageRankVertex extends Vertex[Double](0.0d) {
  def update() = {
    var sum = incoming.foldLeft(0)(_ + _.value)
    value = (0.15 / numVertices) + 0.85 * sum

    if (superstep < 30) {
      for (nb <- neighbors) yield 
        Message(this, nb, value / neighbors.size)
    } else
      List()
  }
}
Another Example.

class PhasedVertex extends Vertex[MyData] {
  var phase = 1

  def update() = {
    if (phase == 1) {
      ...
      if (condition)
        phase = 2
    } else if (phase == 2) {
      ...
    }
  }
}
Another Example.

class PhasedVertex extends Vertex[MyData] {
  var phase = 1

  def update() = {
    if (phase == 1) {
      ...
      if (condition)
        phase = 2
    } else if (phase == 2) {
      ...
    }
  }
}

INVERSION OF CONTROL!!
Thus, manual stack management...

Monday, June 20, 2011
Inverting the Inversion.

Reducers: Use high-level combinators to build expressions of type `Substep[Data]`

```scala
class PhasedVertex extends Vertex[MyData] {
  def update() = {
    thenUntil(condition) {
      ...
    } then {
      ...
    }
  }
}
```
Inverting the Inversion.

Use high-level combinators to build expressions of type `Substep[Data]`.

class PhasedVertex extends Vertex[MyData] {
  def update() = {
    thenUntil(condition) {
      ...
    } then {
      ...
    }
  }
}
Inverting the Inversion.

Use high-level combinators to build expressions of type \texttt{Substep[Data]}
Thus avoiding manual stack management.

```scala
class PhasedVertex extends Vertex[MyData] {
  def update() = {
    thenUntil(condition) {
      ...
    } then {
      ...
    }
  }
}
```
Reduction Combinators: **crunch** steps.
Reduction Combinators:

**crunch** steps.

⇆ Reduction operations important.
  - Replacement for shared data.
  - Global decisions.
Reduction Combinators:

\textit{crunch} steps.

Reduction operations important.
- Replacement for shared data.
- Global decisions.

Provided as just another kind of \texttt{Substep[Data]}
Reduction Combinators: crunch steps.

def update() = {
  then {
    value = ...
  } crunch ((v1: Double, v2: Double) => v1 + v2) then {
    incoming match { case List(reduced) => ...
    } ...
  }
  ...
}
Menthor’s Implementation
Actors.

Implementation based upon Actors.

Central GRAPH instance is an actor, which manages a set of WORKER actors.
Actors.

Implementation based upon Actors.

Central GRAPH instance is an actor, which manages a set of WORKER actors.
Actors.

Implementation based upon Actors.

Central `GRAPH` instance is an actor, which manages a set of `WORKER` actors

`GRAPH` synchronizes workers using supersteps.
Actors.

Implementation based upon Actors.

Each WORKER manages a partition of the graph’s vertices,

- Deliver incoming messages that were sent in the previous superstep;
- Select and execute update step on each vertex in its partition;
- Forward outgoing messages generated by its vertices in the current superstep.
Implementing Reduction.

GRAPH

FOREMEN

WORKERS

Monday, June 20, 2011
Implementing Reduction.

I. WORKER reduces the values of all vertices in its partition.
Implementing Reduction.

1. WORKER reduces the values of all vertices in its partition.

2. The result and the closure that was used to compute it is sent to the GRAPH actor, which computes the final reduced value.
Implementing Reduction.

1. WORKER reduces the values of all vertices in its partition.
2. The result and the closure that was used to compute it is sent to the GRAPH actor, which computes the final reduced value.
3. The final result is passed to all WORKERS which make it available to their vertices as incoming messages (at the beginning of the next superstep)
Implementation Principles.
Implementation Principles.

✗ A pure Scala library
  - No staging and code generation.
  - No dependency on language virtualization.
Implementation Principles.

✗ A pure Scala library
- No staging and code generation.
- No dependency on language virtualization.

✗ Benefits
- Compatible with mainline Scala compiler.
- Fast compilation.
- Simple debugging and troubleshooting.
Implementation Principles.

✗ A pure Scala library
- No staging and code generation.
- No dependency on language virtualization.

✗ Benefits
- Compatible with mainline Scala compiler.
- Fast compilation.
- Simple debugging and troubleshooting.

✗ Drawbacks
- No aggressive optimizations.
- No support for heterogeneous hardware platforms.
Related Work.

**Google’s Pregel**
- **Main Inspiration**: Graphs/BSP
- **Control**: Inverted
- **Async Execution**: Non-determinism

**GraphLab**
- **Signal/Collect**

**OptiML**
- **Aggressive Optimizations**
- **Requires Staging**
- **Debugging**: Not optimal, yet

**Spark**
- Designed for Iteration
- Cluster support
- No graph support
- Non-determinism

Monday, June 20, 2011
Related Work.

**Google’s Pregel**
- **Main Inspiration**: Graphs/BSP
- **Control**: Inverted
- **Async Execution**: Non-determinism

**GraphLab**

**Signal/Collect**

**OptiML**
- Aggressive **Optimizations**
- Requires Staging
- Debugging: Not optimal, yet

**Spark**
- Designed for Iteration
- Cluster support
- No graph support
- Non-determinism

(Many more discussed in a workshop paper.)
Conclusions
Conclusions

✗ Can avoid inversion of control in vertex-based BSP using closures.
Conclusions

Can avoid inversion of control in vertex-based BSP using closures.

Higher-order functions useful for reductions, in an imperative model.
Conclusions

Can avoid inversion of control in vertex-based BSP using closures.

Higher-order functions useful for reductions, in an imperative model.

Explicit parallelism feasible if computational model simple (cf. MapReduce)
Conclusions

- Can avoid inversion of control in vertex-based BSP using closures.
- Higher-order functions useful for reductions, in an imperative model.
- Explicit parallelism feasible if computational model simple (cf. MapReduce)
- The puzzle pieces are there to make analyzing bigger data easier.

http://lamp.epfl.ch/~phaller/menthor/