Lightweight Session
Programming in Scala

Alceste Scalas
Nobuko Yoshida

Imperial College
London

S-REPLS 6
University College London, 25 May 2017
Troubles with session programming

Consider a simple “greeting” client/server session protocol:

1. the client can ask to greet someone, or quit
2. if asked to greet, the server can either:
   2.1 say hello, and go back to 1
   2.2 say bye, and end the session
Troubles with session programming

Consider a simple “greeting” client/server session protocol:

1. the client can ask to greet someone, or quit
2. if asked to greet, the server can either:
   2.1 say hello, and go back to 1
   2.2 say bye, and end the session

Typical approach:

- describe the protocol informally
- develop ad hoc protocol APIs to avoid protocol violations
- find bugs via runtime testing/monitoring
Troubles with session programming

Consider a simple “greeting” client/server session protocol:

1. the client can ask to *greet* someone, or *quit*
2. if *asked to greet*, the server can either:
   2.1 say *hello*, and go back to 1
   2.2 say *bye*, and end the session

Typical approach:

- describe the protocol *informally*
- develop *ad hoc* protocol APIs to avoid *protocol violations*
- find bugs via *runtime testing/monitoring*

Impact on *software evolution and maintenance*
Lightweight Session Programming in Scala

This talk: we show how in Scala + lchannels we can write:

```scala
def client(c: Out[Start]): Unit = {
  if (Random.nextBoolean()) {
    val c2 = c !! Greet("Alice")
    c2 ? {
      case m @ Hello(name) => client(m.cont)
      case Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```

...with a clear theoretical basis, giving a general API with static protocol checks and message transport abstraction
- Object-oriented and functional
- Declaration-site variance
- Case classes for OO pattern matching
- Object-oriented and functional
- Declaration-site variance
- **Case classes** for OO pattern matching

```scala
sealed abstract class Pet

case class Cat(name: String) extends Pet

case class Dog(name: String) extends Pet

def says(pet: Pet) = {
  pet match {
    case Cat(name) => name + " says: meoow"
    case Dog(name) => name + " says: woof"
  }
}
```
Session types

Consider again our “greeting” client/server session protocol:

1. the client can ask to greet someone, or quit
2. if asked to greet, the server can either:
   2.1 say hello, and go back to 1
   2.2 say bye, and end the session
Session types

Consider again our “greeting” client/server session protocol:

1. the client can ask to greet someone, or quit
2. if asked to greet, the server can either:
   2.1 say hello, and go back to 1
   2.2 say bye, and end the session

We can formalise the client viewpoint as a session type for the session π-calculus:  (Honda et al., 1993, 1994, 1998, ...)

\[ S_h = \mu X. (\text{!Greet(String).} \oplus \text{!Quit.end}) \oplus (\text{?Hello(String).} X \& \text{?Bye(String).end}) \]
Session types

Consider again our “greeting” client/server session protocol:

1. the client can ask to greet someone, or quit
2. if asked to greet, the server can either:
   2.1 say hello, and go back to 1
   2.2 say bye, and end the session

We can formalise the server viewpoint as a (dual) session type for the session $\pi$-calculus: (Honda et al., 1993, 1994, 1998, . . .)

$$\overline{S_h} = \mu X. \left( ?\text{Greet}(String) \cdot \left( \right) \right) \oplus \left( !\text{Hello}(String).X \right) \oplus \left( !\text{Bye}(String).\text{end} \right)$$
From theory to practice

Desiderata:
- find a **formal link** between **Scala types** and **session types**
- represent **sessions** in a language **without session primitives**
  - **lightweight**: no language extensions, minimal dependencies

**Inspiration** (from concurrency theory):
- **encoding of session types into linear types for $\pi$-calculus**
  (Dardha, Giachino & Sangiorgi, PPDP’12)
From theory to practice

Desiderata:

- find a **formal link** between *Scala types* and *session types*
- represent *sessions* in a language **without** *session primitives*
  - lightweight: no language extensions, minimal dependencies

**Inspiration** (from concurrency theory):

- encoding of session types into linear types for $\pi$-calculus
  (Dardha, Giachino & Sangiorgi, PPDP’12)

Result: **Lightweight Session Programming in Scala**
Session vs. linear types (in pseudo-Scala)

\[ S_h = \mu X. (\text{!Greet(String)}, (\text{?Hello(String)}, X \& \text{?Bye(String)}, \text{end}) \oplus \text{!Quit.end}) \]

Defining and implementing linear in/out channels
**Session vs. linear types (in pseudo-Scala)**

\[
S_h = \mu X. \left( !\text{Greet}(\text{String}) . ( ?\text{Hello}(\text{String}).X \ & \ ?\text{Bye}(\text{String}).\text{end} ) \oplus !\text{Quit}.\text{end} \right)
\]

“Session Scala”

```scala
def client(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")

    c ? {
      Hello(name) => client(c)
      Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```

Session vs. linear types (in pseudo-Scala)

\[ S_h = \mu X. (!\text{Greet}(\text{String}). (?\text{Hello}(\text{String}). X \& ?\text{Bye}(\text{String}). \text{end}) \oplus !\text{Quit}. \text{end}) \]

"Session Scala"

```scala
def client(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")
    c ? {
      Hello(name) => client(c)
      Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```

"Linear Scala"

```scala
def client(c: LinOutChannel[a]): Unit = {
  if (...) {
    if (...) {
      val (c2in, c2out) = createLinChannels[a]()
      c.send( Greet("Alice", c2out) )
      c2in.receive match {
        case Hello(name, c3out) => client(c3out)
        case Bye(name) => ()
      }
    }
  } else {
    c.send( Quit() )
  }
}
```
Session vs. linear types (in pseudo-Scala)

\[ S_h = \mu_X.(!Greet(String).(?Hello(String).X & ?Bye(String).end) \oplus !Quit.end) \]

“Session Scala”

```scala
def client(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")
    c ? {
      Hello(name) => client(c)
      Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```

“Linear Scala”

```scala
def client(c: LinOutChannel[?]): Unit = {
  if (...) {
    val (c2in, c2out) = createLinChannels[?]()
    c.send( Greet("Alice", c2out) )
    c2in.receive match {
      case Hello(name, c3out) => client(c3out)
      case Bye(name) => ()
    }
  } else {
    c.send( Quit() )
  }
}
```

Goals:

- define and implement linear in/out channels
- instantiate the “?" type parameter
- automate continuation channel creation
lchannels: **interface**

```scala
abstract class In[+A] {

  def receive(implicit d: Duration): A

}

abstract class Out[-A] {

  def send(msg: A): Unit

}
```

API offers **typed send/receive**

- with **runtime checks** for **linear use** and **error handling**

Note **input/output co/contra-variance**
**lchannels: interface**

```scala
abstract class In[+A] {

  def receive(implicit d: Duration): A

  def ?[B](f: A => B)(implicit d: Duration): B = {
    f(receive)
  }
}

abstract class Out[-A] {

  def send(msg: A): Unit
  def !(msg: A) = send(msg)
}
```

API offers **typed** send/receive, plus **syntactic sugar**

> with **runtime checks** for **linear use** and **error handling**

Note **input/output co/contra-variance**
**lchannels: interface**

```scala
abstract class In[+A] {
  def future: Future[A]
  def receive(implicit d: Duration): A = {
    Await.result[A](future, d)
  }
  def ?[B](f: A => B)(implicit d: Duration): B = {
    f(receive)
  }
}

abstract class Out[-A] {
  def promise[B <: A]: Promise[B] // Impl. must be constant
  def send(msg: A): Unit = promise.success(msg)
  def !(msg: A) = send(msg)
}
```

API offers **typed** `send/receive`, plus **syntactic sugar**
- with **runtime checks** for **linear use** and **error handling**

Note **input/output co/contra-variance**
lchannels: interface

abstract class In[+A] {
  def future: Future[A]
  def receive(implicit d: Duration): A = {
    Await.result[A](future, d)
  }
  def ?[B](f: A => B)(implicit d: Duration): B = {
    f(receive)
  }
}

abstract class Out[-A] {
  def promise[B <: A]: Promise[B] // Impl. must be constant
  def send(msg: A): Unit = promise.success(msg)
  def !(msg: A) = send(msg)
  def create[B](): (In[B], Out[B]) // Used to continue a session
}

API offers typed send/receive, plus syntactic sugar
  ▶ with runtime checks for linear use and error handling
Note input/output co/contra-variance
Session programming = $\text{In}[\cdot]/\text{Out}[\cdot] + \text{CPS protocols}$

How do we instantiate the $\text{In}[\cdot]/\text{Out}[\cdot]$ type parameters?

### Session types

- **Client**
  - $S$

- **Server**
  - $\bar{S}$

### Scala types

- **Client**
  - $\text{In}[^?]$ or $\text{Out}[^?]$

- **Server**
  - $\text{Out}[^?]$ or $\text{In}[^?]$
Session programming $= \text{In}[\cdot] / \text{Out}[\cdot] + \text{CPS protocols}$

How do we **instantiate** the In[\cdot]/Out[\cdot] type parameters?

<table>
<thead>
<tr>
<th>Session types</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$\bar{S}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scala types</th>
<th>CPS protocol classes</th>
<th>In[A] or Out[A]</th>
<th>Out[A] or In[A]</th>
</tr>
</thead>
</table>

$A_1, A_2, \ldots, A_n$
Session programming = \( \text{In} [\cdot]/\text{Out} [\cdot] + \text{CPS protocols} \)

How do we instantiate the \( \text{In} [\cdot]/\text{Out} [\cdot] \) type parameters?

**Session types**

- Client
  - \( S \)
  - Linear I/O types: \(?(U)\) or \(! (U)\)
- Server
  - \( \bar{S} \)
  - Linear I/O types: \(! (U)\) or \(? (U)\)

**Scala types**

- CPS protocol classes: \( A_1, A_2, \ldots, A_n \)
- \( \text{In} [A] \) or \( \text{Out} [A] \)
- \( \text{Out} [A] \) or \( \text{In} [A] \)
Programming with \textit{l}channels (I)

\[ S_h = \mu_X.(!\text{Greet(String)}.(?\text{Hello(String)}.X \& ?\text{Bye(String)}.\text{end}) \oplus !\text{Quit}.\text{end}) \]
Programming with lchannels (I)

\[ S_h = \mu X. (!\text{Greet}(\text{String}).(?\text{Hello}(\text{String}).X \& ?\text{Bye}(\text{String}).\text{end}) \oplus !\text{Quit}.\text{end}) \]

\[
\text{prot}\langle S_h \rangle_N = \begin{cases} 
// \text{Top-level internal choice} \\
\text{case class Greet}(p: \text{String}) \\
\text{case class Quit}(p: \text{Unit}) \\
\end{cases}

// \text{Inner external choice} \\
\text{case class Hello}(p: \text{String}) \\
\text{case class Bye}(p: \text{String})
\]
Programming with \texttt{lchannels (I)}

\[ S_h = \mu X. (!\text{Greet}(\text{String}).(?\text{Hello}(\text{String}).X \& ?\text{Bye}(\text{String}).\text{end}) \oplus !\text{Quit}\text{.end}) \]

\hspace{1cm}

\begin{center}
\begin{verbatim}
sealed abstract class Start
case class Greet(p: String) extends Start
case class Quit(p: Unit) extends Start

sealed abstract class Greeting
case class Hello(p: String) extends Greeting
case class Bye(p: String) extends Greeting
\end{verbatim}
\end{center}
Programming with lchannels (I)

\[ S_h = \mu_X.(!\text{Greet}(\text{String}).(?\text{Hello}(\text{String}).X \& ?\text{Bye}(\text{String}).\text{end}) \oplus !\text{Quit}.\text{end}) \]

```
sealed abstract class Start

case class Greet(p: String)(val cont: Out[Greeting]) extends Start

case class Quit(p: Unit) extends Start

sealed abstract class Greeting

case class Hello(p: String)(val cont: Out[Start]) extends Greeting

case class Bye(p: String) extends Greeting
```

```
prot\{S_h\}_N =
```

```
sealed abstract class Start

case class Greet(p: String)(val cont: Out[Greeting]) extends Start

case class Quit(p: Unit) extends Start

sealed abstract class Greeting

case class Hello(p: String)(val cont: Out[Start]) extends Greeting

case class Bye(p: String) extends Greeting
```
Programming with lchannels (I)

\[ S_h = \mu X. (!\text{Greet(String)}. (?\text{Hello(String)}. X \& ?\text{Bye(String)}. \text{end}) \oplus !\text{Quit.end}) \]

\[
\text{prot}\langle S_h \rangle_\mathcal{N} = \\
\text{sealed abstract class Start} \\
\text{case class Greet(p: String)(val cont: Out[Greeting]) extends Start} \\
\text{case class Quit(p: Unit)} \text{ extends Start} \\
\text{sealed abstract class Greeting} \\
\text{case class Hello(p: String)(val cont: Out[Start]) extends Greeting} \\
\text{case class Bye(p: String)} \text{ extends Greeting}
\]
Programming with lchannels (I)

\[ S_h = \mu X. (\text{!Greet(String)}.(?\text{Hello(String)}.X \& ?\text{Bye(String)}.\text{end}) \oplus \text{!Quit}.\text{end}) \]

\[
\text{prot}\langle S_h \rangle_N = \\
\text{sealed abstract class Start} \\
\text{case class Greet}(p: \text{String})(\text{val cont: Out[Greeting]}) \text{ extends Start} \\
\text{case class Quit}(p: \text{Unit}) \text{ extends Start}
\]

\[
\text{sealed abstract class Greeting} \\
\text{case class Hello}(p: \text{String})(\text{val cont: Out[Start]}) \text{ extends Greeting} \\
\text{case class Bye}(p: \text{String}) \text{ extends Greeting}
\]
Programming with lchannels (I)

\[ S_h = \mu X. (! \text{Greet(String)}. (? \text{Hello(String)}.X \& ? \text{Bye(String)}. \text{end}) \oplus ! \text{Quit}. \text{end}) \]

\[ \text{prot} \langle S_h \rangle_N = \text{sealed abstract class Greeting} \]
\[ \quad \text{case class Hello(p: String)(val cont: Out[Start]) extends Greeting} \]
\[ \quad \text{case class Bye(p: String)} \]

\[ \langle S_h \rangle_N = \text{Out}[\text{Start}] \]
Programming with lchannels (I)

\[ S_h = \mu X. (!Greet(String).(?Hello(String).X & ?Bye(String).end) \oplus !Quit.end) \]

\[ \text{prot}\langle S_h \rangle_N = \]

\[ \langle S_h \rangle_N = \text{Out}[\text{Start}] \]

def client(c: Out[Start]): Unit = {
  if (Random.nextBoolean()) {
    val (c2in, c2out) = c.create[Greeting]()
    c.send( Greet("Alice", c2out) )
    c2in.receive match {
      case Hello(name, c3out) => client(c3out)
      case Bye(name) => ()
    }
  } else {
    c.send( Quit() )
  }
}
Programming with lchannels (I)

\[ S_h = \mu_X( !\text{Greet}(\text{String}).( ?\text{Hello}(\text{String}).X \& ?\text{Bye}(\text{String}).\text{end}) \oplus !\text{Quit}.\text{end}) \]

\[
\begin{aligned}
\text{sealed abstract class } & \text{Start} \\
\text{case class } & \text{Greet}(p: \text{String})(\text{val cont: Out[Greeting]}) \text{ extends Start} \\
\text{case class } & \text{Quit}(p: \text{Unit}) \text{ extends Start} \\
\text{sealed abstract class } & \text{Greeting} \\
\text{case class } & \text{Hello}(p: \text{String})(\text{val cont: Out[Start]}) \text{ extends Greeting} \\
\text{case class } & \text{Bye}(p: \text{String}) \text{ extends Greeting} \\
\end{aligned}
\]

\[
\begin{aligned}
def \text{client}(c: \text{Out[Start]}): \text{Unit} = \{ \\
\text{if (Random.nextBoolean())} \{ \\
\text{val (c2in, c2out) = c.create[Greeting]()} \\
\text{c.send( Greet("Alice", c2out) )} \\
\text{c2in.receive match} \{ \\
\text{case Hello(name, c3out) => \text{client(c3out)}} \\
\text{case Bye(name) => ()} \\
\text{\} else} \{ \\
\text{c.send( Quit() )} \\
\text{\} } \\
\end{aligned}
\]

Goals:

- define and implement linear in/out channels ✓
- instantiate the "?" type parameter ✓
- automate continuation channel creation ❌
The “create-send-continue” pattern

We can observe that **In/Out** channel pairs are usually created for **continuing a session after sending a message**
The “create-send-continue” pattern

We can observe that In/Out channel pairs are usually created for continuing a session after sending a message

Let us add the `!!` method to `Out[A]`:

```scala
class Out[A] {
  ...
  def !![B](h: Out[B] => A): In[B] = {
    val (cin, cout) = this.create[A]()  // Create...
    this ! h(cout)  // ...send...
    cin  // ...continue
  }

    val (cin, cout) = this.create[A]()  // Create...
    this ! h(cin)  // ...send...
    cout  // ...continue
  }
}
```
Programming with lchannels (II)

\[ S_h = \mu_X.(!Greet(String).(?Hello(String).X \& ?Bye(String).end) \oplus !Quit.end) \]
Programming with lchannels (II)

\[ S_h = \mu X. (\texttt{!Greet(String)}. (?\texttt{Hello(String)}.X \& ?\texttt{Bye(String)}.\texttt{end}) \oplus !\texttt{Quit.end}) \]

“Session Scala” (pseudo-code)

```scala
def client(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")

    c ? {
      Hello(name) => client(c)
      Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```
Programming with lchannels (II)

\[ S_h = \mu X (\text{!Greet(String)}. (?\text{Hello(String)}. X \& ?\text{Bye(String)}. \text{end}) \oplus \text{!Quit}. \text{end}) \]

```
sealed abstract class Start
  case class Greet(p: String)(val cont: Out[Greeting]) extends Start
  case class Quit(p: Unit) extends Start

sealed abstract class Greeting
  case class Hello(p: String)(val cont: Out[Start]) extends Greeting
  case class Bye(p: String) extends Greeting
```

“Session Scala” (pseudo-code)

```scala
def client(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")

    c ? {
      Hello(name) => client(c)
      Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```
Programming with lchannels (II)

\[ S_h = \mu_X (\text{!Greet(String)} \cdot (\text{?Hello(String)} \cdot X \& \text{?Bye(String)} \cdot \text{end}) \oplus \text{!Quit.end}) \]

```
sealed abstract class Start
  case class Greet(p: String)(val cont: Out[Greeting]) extends Start
  case class Quit(p: Unit) extends Start

sealed abstract class Greeting
  case class Hello(p: String)(val cont: Out[Start]) extends Greeting
  case class Bye(p: String)
```

```
"Session Scala" (pseudo-code)
```

```
def client(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")
    c ? {
      Hello(name) => client(c)
      Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```

```
Scala + lchannels
```

```
def client(c: Out[Start]): Unit = {
  if (Random.nextBoolean()) {
    val c2 = c !! Greet("Alice")
    c2 ? {
      case m @ Hello(name) => client(m.cont)
      case Bye(name) => ()
    }
  } else {
    c ! Quit()
  }
}
```
Demo
Run-time and compile-time checks

Well-typed output / int. choice
Exhaustive input / ext. choice

Compile-time Compile-time
Run-time and compile-time checks

Well-typed output / int. choice  Compile-time
Exhaustive input / ext. choice  Compile-time

Double use of linear output endp.  Run-time
Double use of linear input endp.  Run-time
Run-time and compile-time checks

Well-typed output / int. choice  Compile-time
Exhaustive input / ext. choice  Compile-time

Double use of linear output endp.  Run-time
Double use of linear input endp.  Run-time

“Forgotten” output  Run-time (timeout on input side)
“Forgotten” input  Unchecked
Formal properties

Theorem \((Preservation \ of \ duality)\).
\[
\langle \overline{S} \rangle_N = \langle S \rangle_N \quad \text{(where } \text{In}[A] = \text{Out}[A] \text{ and } \text{Out}[A] = \text{In}[A]).
\]
Formal properties

**Theorem** (*Preservation of duality*).
\[
\llbracket \bar{S} \rrbracket_N = \llbracket S \rrbracket_N \quad \text{(where} \quad \text{In}[A] = \text{Out}[A] \quad \text{and} \quad \text{Out}[A] = \text{In}[A])
\]

**Theorem** (*Dual session types have the same CPS protocol classes*).
\[
\text{prot}\llbracket S \rrbracket_N = \text{prot}\llbracket \bar{S} \rrbracket_N
\]
Formal properties

**Theorem (Preservation of duality).**

\[ \langle \overline{S} \rangle_N = \langle S \rangle_N \quad (\text{where } \text{In}[A] = \text{Out}[A] \text{ and } \text{Out}[A] = \text{In}[A]). \]

**Theorem (Dual session types have the same CPS protocol classes).**

\[ \text{prot} \langle S \rangle_N = \text{prot} \langle \overline{S} \rangle_N. \]

**Theorem (Scala subtyping implies session subtyping).**

For all \( S, N \):

- if \( \langle S \rangle_N = \text{In}[A] \) and \( B <: \text{In}[A] \),
  then \( \exists S', N' \) such that \( B = \langle S' \rangle_{N'} \) and \( S' \leq S \);
- if \( \langle S \rangle_N = \text{Out}[A] \) and \( \text{Out}[A] <: B \),
  then \( \exists S', N' \) such that \( B = \langle S' \rangle_{N'} \) and \( S \leq S' \).
Conclusions

We presented a **lightweight integration of session types in Scala** based on a **formal link** between CPS protocols and session types.

We leveraged **standard Scala features** (from its type system and library) with a **thin abstraction layer** (**lchannels**)

- low **cognitive overhead, integration** and **maintenance** costs
- naturally supported by **modern IDEs** (e.g. Eclipse)

We validated our session-types-based programming approach with **case studies** (from literature and industry) and **benchmarks**
Ongoing and future work

Automatic generation of CPS protocol classes from session types, using Scala macros


Extension to multiparty session types, using Scribble

Ongoing and future work

Automatic generation of CPS protocol classes from session types, using **Scala macros**


Extension to **multiparty session types**, using **Scribble**


Generalise the approach to other frameworks beyond **lchannels**, and study its properties.
Natural candidates: **Akka Typed, Reactors.IO**

Investigate other programming languages. Possible candidate: **C#** (declaration-site variance and FP features)
Try `lchannels` and Scribble!

http://alcestes.github.io/lchannels

http://scribble.org

ECOOP 2016

ECOOP 2017