Lightweight Session Programming in Scala

Alceste Scalas
Nobuko Yoshida

Imperial College London

Univerzitet u Novom Sadu
March 27th, 2017
Session Type Mobility Group

www.mrg.doc.ic.ac.uk
Us ∈ Mobility Research Group

http://mrg.doc.ic.ac.uk/
OOI Collaboration

- **TCS’16**: Monitoring Networks through Multiparty Session Types. Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, Nobuko Yoshida
- **LMCS’16**: Multiparty Session Actors. Rumyana Neykova, Nobuko Yoshida
- **FMSD’15**: Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, Nobuko Yoshida
- **TGC’13**: The Scribble Protocol Language. Nobuko Yoshida, Raymond Hu, Rumyana Neykova, Nicholas Ng
Scribble: Describing Multi Party Protocols

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data. However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

---

**Describe**

Scribble is a language for describing multiparty protocols from a global, or endpoint neutral, perspective.

**Verify**

Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that protocols described using the language are sound, and do not suffer from deadlocks or livelocks.

**Project**

Endpoint projection is the term used for identifying the responsibility of a particular role (or endpoint) within a protocol.

**Implement**

Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b) using session type APIs to clearly describe the behaviour, and (c) statically verify the code against the projection.

**Monitor**

Use the endpoint projection for roles defined within a Scribble protocol, to monitor the activity of a particular endpoint, to ensure it correctly implements the expected behaviour.
module examples;

global protocol HelloWorld(role Me, role World) {
    hello() from Me to World;
    choice at World {
        goodMorning1() from World to Me;
    } or {
        goodMorning1() from World to Me;
    }
}
Interactions with Industries

Strange Loop

SEPTEMBER 15-17 2016 / PEABODY OPERA HOUSE / ST. LOUIS, MO

Adam Bowen @adambowen - Sep 15
I didn’t even know that session types existed an hour ago, but thanks to Nobuko Yoshida’s great talk at #pwiconf, I want to learn more.

Nobuko Yoshida
Imperial College, London

DoC researcher to speak at Golang UK conference
by Vicky Kapogianni
20 July 2016

DoC researcher to speak at industry-focused Golang UK conference on results of concurrency research

@nicholaswng rocking on @GolangUKconf about static deadlock detection in #golang #gouk16

The Golang UK Conference
Interactions with Industries

**Functional Londoners Meetup Group**

6 days ago · 6:30 PM  
**Session Types with Fahd Abdeljallal**

43 Members

Synopsis: Session types are a formalism to codify the structure of a communication, using types to specify the communication protocol used. This formalism provides the...

**Distributed Systems vs. Compositionality**

Dr. Roland Kuhn  
@rolandkuhn — CTO of Actyx

**Current State**

- behaviors can be composed both sequentially and concurrently
- effects are not yet tracked
- Scribble generator for Scala not yet there
- theoretical work at Imperial College, London (Prof. Nobuko Yoshida & Alceste Scalas)
Go concurrency verification research at DoC grabs headline

A paper by DoC researchers at POPL on Go concurrency verification was featured in a tech blog and generates a buzz outside of the research community.

A paper by researchers at the department was recently featured in the morning paper, a blog by venture capitalist Adrian Colye, which summarises an important, influential, topical or otherwise interesting paper in the field of computer science every weekday in an easily digestible way by non-researchers. On the 2 Feb 2017 issue of the morning paper, It was highlighted as "the true spirit of POPL (Principles of Programming Languages)".
Selected Publications 2016/2017

- **[FoSSaCS’17]** Julien Lange, NY: On the Undecidability of Asynchronous Session Subtyping.
- **[FASE’17]** Raymond Hu, NY: Explicit Connection Actions in Multiparty Session Types.
- **[CC’17]** Rumyana Neykova, NY: Let It Recover: Multiparty Protocol-Induced Recovery.
- **[POPL’17]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY: Fencing off Go: Liveness and Safety for Channel-based Programming.
- **[ECOOP’16]** Alceste Scala, NY: Lightweight Session Programming in Scala
- **[CC’16]** Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
- **[FASE’16]** Raymond Hu, NY: Hybrid Session Verification through Endpoint API Generation.
- **[TACAS’16]** Julien Lange, NY: Characteristic Formulae for Session Types.
- **[POPL’16]** Dominic Orchard, NY: Effects as sessions, sessions as effects.
Selected Publications 2016/2017

- [CC’16] Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
Lightweight Session Programming in Scala
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

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, …)

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

\[
\overline{S_h} = \mu X. \left( ?\text{Greet}(\text{String}). \right. \\
& \left. \oplus \right. \\
& \left. ?\text{Quit}. \text{end} \right)
\]
Mixing the ingredients

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)
Mixing the ingredients

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)

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

Session Scala

```scala
def client(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")
    c ? { 
      val Hello(name) => client(c)
      val 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() )
  }
}
```
Session vs. linear types (in pseudo-Scala)

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

“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)

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

"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() )
  }
}
```
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[?]): 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

abstract class In[+A] {
    def receive(implicit d: Duration): A
}

abstract class Out[-A] {
    def send(msg: A): Unit
}

API reminds standard Promises/Futures

- similar runtime linearity checks 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 reminds standard **Promises/Futures**

- similar **runtime linearity checks** and **error handling**

Note **input/output co/contra-variance**
Ichannels: 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)
}

API reminds standard Promises/Futures
  ▶ similar runtime linearity checks 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 ?(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 reminds standard Promises/Futures
- similar runtime linearity checks and error handling

Note input/output co/contra-variance
Session programming = \( \text{In}[\cdot]/\text{Out}[\cdot] \) + CPS protocols

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

Session types

- Client: \( S \)
- Server: \( \overline{S} \)

Scala types

- Client: \( \text{In}[	ext{?}] \) or \( \text{Out}[	ext{?}] \)
- Server: \( \text{Out}[	ext{?}] \) or \( \text{In}[	ext{?}] \)
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  

Server

**Scala types**

$S$, $\overline{S}$

$\text{In}[A]$ or $\text{Out}[A]$  

$\text{Out}[A]$ or $\text{In}[A]$  

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

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

### Diagram

**Session types**

$S$  \quad \text{Client} \quad \text{Server} \quad \bar{S}$

**Linear I/O types**

$\text{Client}: \ ?(U)$ or $!(U)$

$\text{Server}: \ !?(U)$ or $!(U)$

$\text{U}$

**Scala types**

**CPS protocol classes**

$A_1, A_2, \ldots, A_n$

$\text{Client}: \ \text{In}[A]$ or $\text{Out}[A]$

$\text{Server}: \ \text{Out}[A]$ or $\text{In}[A]$
Programming with lchannels (I)

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

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

\[
\text{prot}\left\langle S_h \right\rangle_N = \quad \begin{array}{l}
\text{// Top-level internal choice} \\
\text{case class Greet(p: String)} \\
\text{case class Quit(p: Unit)} \\
\text{// Inner external choice} \\
\text{case class Hello(p: String)} \\
\text{case class Bye(p: String)}
\end{array}
\]
Programming with lchannels (l)

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

\[
\text{prot}\langle S_h \rangle_N = \\
\text{sealed abstract class Start} \\
\text{case class Greet}(p: \text{String}) \hspace{1cm} \text{extends Start} \\
\text{case class Quit}(p: \text{Unit}) \hspace{1cm} \text{extends Start} \\
\text{sealed abstract class Greeting} \\
\text{case class Hello}(p: \text{String}) \hspace{1cm} \text{extends Greeting} \\
\text{case class Bye}(p: \text{String}) \hspace{1cm} \text{extends Greeting}
\]
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)
```

```
prot\{S_h\}_N =

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

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

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

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

prot\(\langle S_h \rangle_N = \)

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)

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

\[ \text{prot} \langle S_h \rangle_{\mathcal{N}} = \]

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

\[ \langle S_h \rangle_{\mathcal{N}} = \text{Out}[\text{Start}] \]
Programming with \texttt{lchannels (I)}

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

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

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

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

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

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

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

**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
abstract 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)

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

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

"Session Scala" (pseudo-code)

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

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

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

```
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)

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

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

```
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
```

**“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
Exhaustive input / ext. choice
Double use of linear output endp.
Double use of linear input endp.

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

- Well-typed output / int. choice
- Exhaustive input / ext. choice
- Double use of linear output endp.
- Double use of linear input endp.
- “Forgotten” output
- “Forgotten” input

**Compile-time**
- Compile-time

**Run-time**
- Run-time

**Unchecked**
- Unchecked

**Compile-time**
- Compile-time

**Run-time**
- Run-time (timeout on input side)
Theorem *(Preservation of duality).*
\[\llbracket S \rrbracket_N = \llbracket S \rrbracket_N \] (where \( \text{In}[A] = \text{Out}[A] \) and \( \text{Out}[A] = \text{In}[A] \)).
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*).
\[
\text{prot}\langle S \rangle_N = \text{prot}\langle \overline{S} \rangle_N .
\]
Formal properties

Theorem *(Preservation of duality).*
\[ \langle S \rangle_{\mathcal{N}} = \langle S \rangle_{\mathcal{N}} \] (where \( \text{In}[A] = \text{Out}[A] \) and \( \text{Out}[A] = \text{In}[A] \)).

Theorem *(Dual session types have the same CPS protocol classes).*
\[ \text{prot}\langle S \rangle_{\mathcal{N}} = \text{prot}\langle S \rangle_{\mathcal{N}}. \]

Theorem *(Scala subtyping implies session subtyping).*
For all \( S, \mathcal{N} \):
\begin{itemize}
  \item if \( \langle S \rangle_{\mathcal{N}} = \text{In}[A] \) and \( B <: \text{In}[A] \),
    then \( \exists S', \mathcal{N}' \) such that \( B = \langle S' \rangle_{\mathcal{N}'} \) and \( S' \leq S \);
  \item if \( \langle S \rangle_{\mathcal{N}} = \text{Out}[A] \) and \( \text{Out}[A] <: B \),
    then \( \exists S', \mathcal{N}' \) such that \( B = \langle S' \rangle_{\mathcal{N}'} \) and \( S \leq S' \).
\end{itemize}
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


Extend to multiparty session types, using Scribble

  “A Linear Decomposition of Multiparty Sessions”.
  https://www.doc.ic.ac.uk/~ascalas/mpst-linear/
Ongoing and future work

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


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

  “A Linear Decomposition of Multiparty Sessions”.
  https://www.doc.ic.ac.uk/~ascalas/mpst-linear/

**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!

http://alcestes.github.io/lchannels