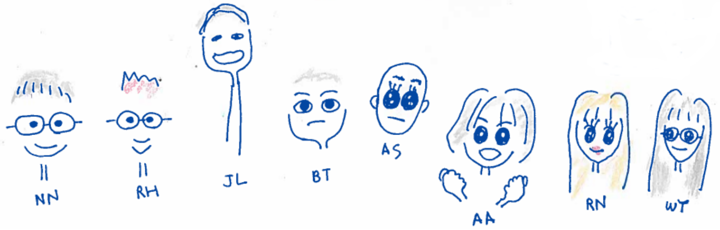
Lightweight Session Programming in Scala

Alceste Scalas
Nobuko Yoshida

Imperial College London

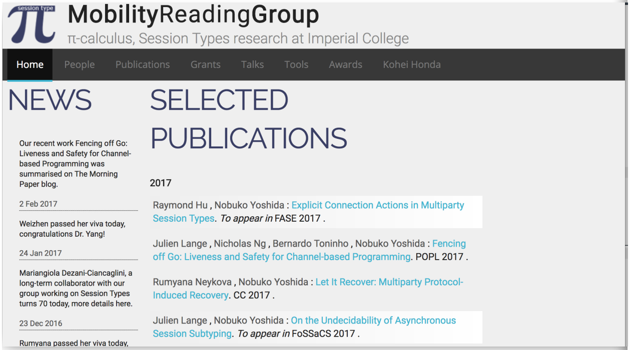
Univerzitet u Novom Sadu
March 27th, 2017

Session Type Mobility Group

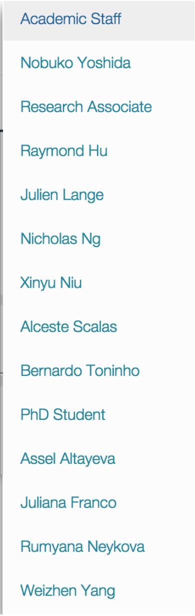


www.mrg.doc.ic.ac.uk

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

www.scribble.org

Home   Getting Started   Downloads   Documentation ▾   Community ▾

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.

Online tool : http://scribble.doc.ic.ac.uk/

```
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;
  }
}
```

Load a sample | Check | Protocol: examples.HelloWorld | Role: Me | Project | Generate Graph

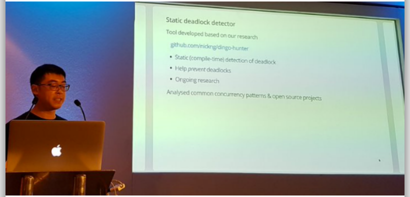
Interactions with Industries

Adam Bowen @adamnbowen · Sep 15
I didn't even know that session types existed an hour ago, but thanks to Nobuko Yoshida's great talk at **#pwlconf**, 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

Click here to add content

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

Interactions with Industries

F#unctional 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... LEARN MORE

Distributed Systems vs. Compositionality

Dr. Roland Kuhn
@rolandkuhn — *CTO of Actyx*

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)

Imperial College London

News

Home | College and Campus | Science | Engineering | Health | Business | Search here... Go

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.
- **[FPL'16]** Xinyu Niu , Nicholas Ng , Tomofumi Yuki , Shaojun Wang , NY, Wayne Luk : EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits.
- **[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.
- **[ESOP'16]** Dimitrios Kouzapas, Jorge A. Pérez, NY: On the Relative Expressiveness of Higher-Order Session Processes.
- **[POPL'16]** Dominic Orchard, NY: Effects as sessions, sessions as effects .

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.
- [FPL'16] Xinyu Niu , Nicholas Ng , Tomofumi Yuki , Shaojun Wang , NY, Wayne Luk: EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits.
- [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.
- [ESOP'16] Dimitrios Kouzapas, Jorge A. Pérez, NY: On the Relative Expressiveness of Higher-Order Session Processes.
- [POPL'16] Dominic Orchard, NY: Effects as Sessions, Sessions as Effects.

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:

```
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**

Scala

- **Object-oriented** *and* **functional**
- **Declaration-site variance**
- **Case classes** for OO pattern matching

Scala

- **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, . . .)

$$S_h = \mu_X.\left(\begin{array}{l} !\mathtt{Greet}(\mathsf{String}).\left(\begin{array}{l} ?\mathtt{Hello}(\mathsf{String}).X \\ \& \\ ?\mathtt{Bye}(\mathsf{String}).\mathbf{end} \end{array} \right) \\ \oplus \\ !\mathtt{Quit}.\mathbf{end} \end{array} \right)$$

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 π-calculus**: (Honda *et al.*, 1993, 1994, 1998, ...)

$$\overline{S_h} = \mu_X.\left(\begin{array}{l} ?\mathtt{Greet}(\mathsf{String}).\left(\begin{array}{l} !\mathtt{Hello}(\mathsf{String}).X \\ \oplus \\ !\mathtt{Bye}(\mathsf{String}).\mathbf{end} \end{array} \right) \\ \& \\ ?\mathtt{Quit}.\mathbf{end} \end{array} \right)$$

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 π-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 π-calculus**
  (Dardha, Giachino & Sangiorgi, PPDP'12)

Result: **Lightweight Session Programming in Scala**

Session vs. linear types (in pseudo-Scala)

$$S_{\mathrm{h}} = \mu_X.\Big(!\mathtt{Greet}(\mathrm{String}).\big(?\mathtt{Hello}(\mathrm{String}).X \;\&\; ?\mathtt{Bye}(\mathrm{String}).\mathbf{end}\big) \oplus !\mathtt{Quit}.\mathbf{end}\Big)$$

Session vs. linear types (in pseudo-Scala)

$$S_h = \mu_X.\big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \ \& \ ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus !\texttt{Quit}.\textbf{end}\big)$$

"Session 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)

$$S_h = \mu_X.\big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \;\&\; ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus !\texttt{Quit}.\textbf{end}\big)$$

“Session Scala”

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

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

“Linear 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)

$$S_h = \mu_X.\Big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \;\&\; ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus !\texttt{Quit}.\textbf{end}\Big)$$

"Session Scala"

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

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

"Linear 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 `Promise`s/`Future`s

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

Note **input/output co/contra-variance**

lchannels: interface

```
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 `Promise`s/`Future`s

- similar **runtime linearity checks** 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)

}
```

API reminds standard `Promise`s/`Future`s

- similar **runtime linearity checks** 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
}
```

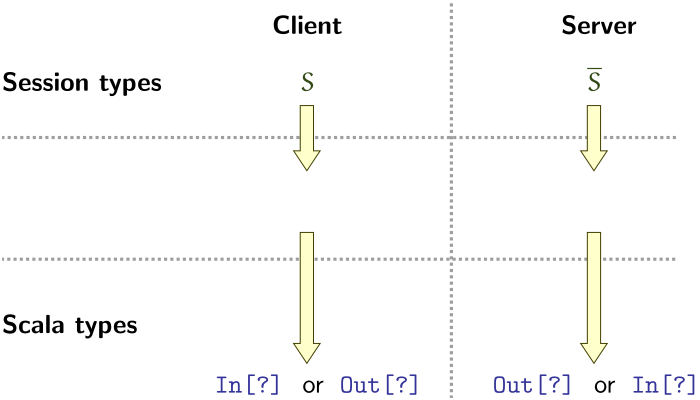
API reminds standard `Promise`s/`Future`s

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

Note **input/output co/contra-variance**

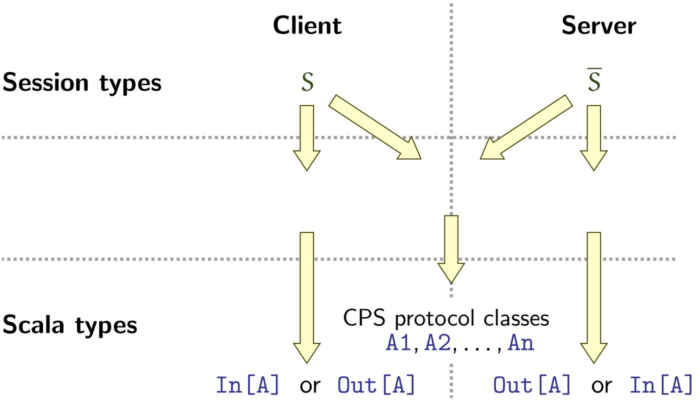
Session programming = In[·]/Out[·] + CPS protocols

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



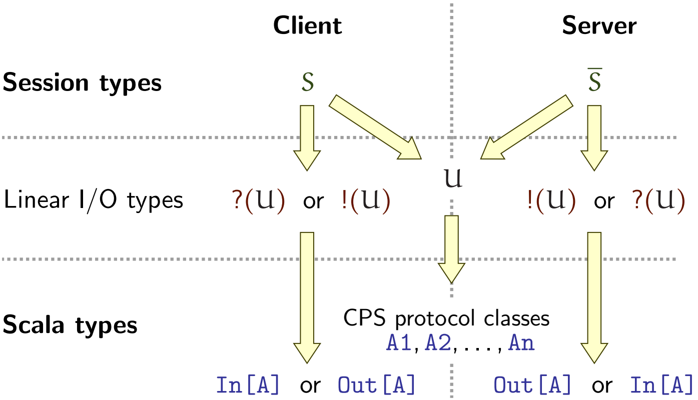
Session programming = In[·]/Out[·] + CPS protocols

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



Session programming = In[·]/Out[·] + CPS protocols

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



Programming with `lchannels` (I)

$$S_h = \mu_X.\Big(!\mathtt{Greet}(\mathsf{String}).\big(?\mathtt{Hello}(\mathsf{String}).X \;\&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\big) \oplus !\mathtt{Quit}.\mathbf{end}\Big)$$

Programming with `lchannels` (I)

$$S_h = \mu_X.\big(!\mathtt{Greet}(\mathrm{String}).\big(?\mathtt{Hello}(\mathrm{String}).X \,\&\, ?\mathtt{Bye}(\mathrm{String}).\mathbf{end}\big) \oplus !\mathtt{Quit}.\mathbf{end}\big)$$

$\mathsf{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ =

```
// Top-level internal choice
case class Greet(p: String)
case class Quit(p: Unit)

// Inner external choice
case class Hello(p: String)
case class Bye(p: String)
```

Programming with `lchannels` (I)

$$S_h = \mu_X.\big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \;\&\; ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus !\texttt{Quit}.\textbf{end}\big)$$

$\text{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}} =$

```
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
```

Programming with `lchannels` (I)

$$S_h = \mu_X.\Big(\mathbf{!}\texttt{Greet}(\text{String}).\big(\mathbf{?}\texttt{Hello}(\text{String}).X \;\&\; \mathbf{?}\texttt{Bye}(\text{String}).\mathbf{end}\big) \oplus \mathbf{!}\texttt{Quit}.\mathbf{end}\Big)$$

$\text{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ =

```
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
```

Programming with `lchannels` (I)

$$S_h = \mu_X.\big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \mathbin{\&} ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus !\texttt{Quit}.\textbf{end}\big)$$

$\text{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ =

```
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
```

Programming with lchannels (I)

$$S_h = \mu_X.\left(!\texttt{Greet}(\text{String}).\left(?\texttt{Hello}(\text{String}).X \;\&\; ?\texttt{Bye}(\text{String}).\textbf{end}\right) \oplus\; !\texttt{Quit}.\textbf{end}\right)$$

$\text{prot}\langle\langle S_h \rangle\rangle_{\mathcal{N}} =$

```
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
```

Programming with `lchannels` (I)

$$S_h = \mu_X.\big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \;\&\; ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus\; !\texttt{Quit}.\textbf{end}\big)$$

$\text{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ =

```
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
```

$\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ = `Out[Start]`

Programming with lchannels (I)

$$S_h = \mu_X.\left(!\mathtt{Greet}(\mathsf{String}).\left(?\mathtt{Hello}(\mathsf{String}).X \,\&\, ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\right) \oplus !\mathtt{Quit}.\mathbf{end}\right)$$

$\mathsf{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ =

```
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
```

$\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ = `Out[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)

$$S_h = \mu_X.\big(!\mathtt{Greet}(\mathsf{String}).\big(?\mathtt{Hello}(\mathsf{String}).X \;\&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\big) \oplus !\mathtt{Quit}.\mathbf{end}\big)$$

$\mathsf{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ =

```
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
```

$\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ = `Out[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() )
  }
}
```

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[·]:

```
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
  }

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

Programming with `lchannels` (II)

$$S_h = \mu_X.\Big(\mathbf{!}\texttt{Greet}(\mathsf{String}).\big(\mathbf{?}\texttt{Hello}(\mathsf{String}).X \;\&\; \mathbf{?}\texttt{Bye}(\mathsf{String}).\mathbf{end}\big) \oplus \mathbf{!}\texttt{Quit}.\mathbf{end}\Big)$$

Programming with `lchannels` (II)

$$S_h = \mu_X.\Big(!\mathtt{Greet}(\mathsf{String}).\big(?\mathtt{Hello}(\mathsf{String}).X \;\&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end}\big) \oplus !\mathtt{Quit}.\mathbf{end}\Big)$$

"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)

$$S_h = \mu_X.\Big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \,\&\, ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus !\texttt{Quit}.\textbf{end}\Big)$$

$\text{prot}\langle\langle S_h \rangle\rangle_{\mathcal{N}}$ =

```
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
```

"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)

$$S_h = \mu_X.\Big(!\texttt{Greet}(\text{String}).\big(?\texttt{Hello}(\text{String}).X \;\&\; ?\texttt{Bye}(\text{String}).\textbf{end}\big) \oplus !\texttt{Quit}.\textbf{end}\Big)$$

$\text{prot}\langle\!\langle S_h \rangle\!\rangle_{\mathcal{N}}$ =

```
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
```

"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	**Compile-time**
Exhaustive input / ext. choice	**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)*.

$\langle\!\langle \overline{S} \rangle\!\rangle_{\mathcal{N}} = \overline{\langle\!\langle S \rangle\!\rangle_{\mathcal{N}}}$ (where $\overline{\texttt{In[A]}} = \texttt{Out[A]}$ and $\overline{\texttt{Out[A]}} = \texttt{In[A]}$).

Formal properties

Theorem *(Preservation of duality)*.

$\langle\!\langle \overline{S} \rangle\!\rangle_{\mathcal{N}} = \overline{\langle\!\langle S \rangle\!\rangle_{\mathcal{N}}}$ (where $\overline{\texttt{In[A]}} = \texttt{Out[A]}$ and $\overline{\texttt{Out[A]}} = \texttt{In[A]}$).

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

$\text{prot}\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} = \text{prot}\langle\!\langle \overline{S} \rangle\!\rangle_{\mathcal{N}}$.

Formal properties

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

Theorem *(Dual session types have the same CPS protocol classes).*
$\mathsf{prot}\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} = \mathsf{prot}\langle\!\langle \overline{S} \rangle\!\rangle_{\mathcal{N}}$.

Theorem *(Scala subtyping implies session subtyping).*
For all $S, \mathcal{N}$:

- if $\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} = \texttt{In[A]}$ and $\texttt{B} <: \texttt{In[A]}$,
  then $\exists S', \mathcal{N}'$ such that $\texttt{B} = \langle\!\langle S' \rangle\!\rangle_{\mathcal{N}'}$ and $S' \leqslant S$;
- if $\langle\!\langle S \rangle\!\rangle_{\mathcal{N}} = \texttt{Out[A]}$ and $\texttt{Out[A]} <: \texttt{B}$,
  then $\exists S', \mathcal{N}'$ such that $\texttt{B} = \langle\!\langle S' \rangle\!\rangle_{\mathcal{N}'}$ and $S \leqslant S'$.

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**

- *B. Joseph. "Session Metaprogramming in Scala". MSc Thesis, 2016*

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

- *A. Scalas, O. Dardha, R. Hu, N. Yoshida.*
  "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**

- *B. Joseph. "Session Metaprogramming in Scala". MSc Thesis, 2016*

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

- *A. Scalas, O. Dardha, R. Hu, N. Yoshida. "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