Lightweight session types in Scala

Alceste Scalas and **Nobuko Yoshida**

Imperial College London

BETTY meeting
Valletta (Malta), March 18th, 2016

Session types in Scala: why...

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

Session types in Scala: why...

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

"Session Scala" (pseudo-code)

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

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

Session types in Scala: why... and what we achieve

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

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

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

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

Session types in Scala: why… and what we achieve

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

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

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

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

Scala + `lchannels`

```
def hello(c: Out[Start]): Unit = {
  if (Random.nextBoolean()) {
    val c2 = c !! Greet("Alice")_

    c2 ? {
      case m @ Hello(name) => hello(m.cont)
      case Bye(name)       => ()
    }
  } else {
    c ! Quit()
  }
}
```

Scala

- **Object-oriented** and **functional**
- Immutable `val`s vs. mutable `var`s
- **Case classes** for OO pattern matching

- **Object-oriented** and **functional**
- Immutable `val`s vs. mutable `var`s
- **Case classes** for OO pattern matching

```
sealed abstract class Term(val descr: String)
case class Var(name: String)             extends Term("Variable")
case class Lam(arg: String, body: Term)  extends Term("Lambda")
case class App(f: Term, v: Term)         extends Term("Application")
```

- **Object-oriented** and **functional**
- Immutable `val`s vs. mutable `var`s
- **Case classes** for OO pattern matching

```scala
sealed abstract class Term(val descr: String)
case class Var(name: String)             extends Term("Variable")
case class Lam(arg: String, body: Term)  extends Term("Lambda")
case class App(f: Term, v: Term)         extends Term("Application")
```

```scala
def term2string(term: Term): String = {
  term.descr + ": " + {
    term match {
      case Var(n)    => n
      case Lam(x, b) => f"${x} . ${term2string(b)}"
      case App(f, v) => f"(${term2string(f)}) (${term2string(v)})"
    }
  }
}
```

Promises and futures

From the **standard library**:
concurrent programming via **promises** and **futures**

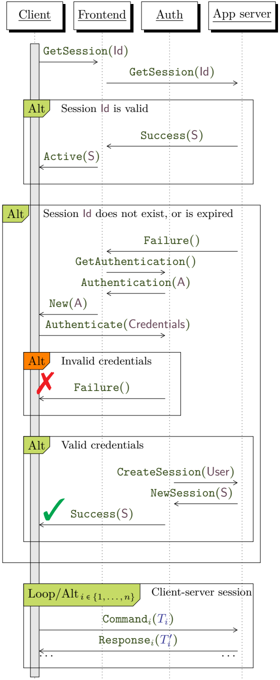
Promises and futures

From the **standard library**:
concurrent programming via **promises** and **futures**

```scala
import scala.concurrent.{Promise, Future, Await}

val p = Promise[Int]
val f = p.future // Type: Future[Int]

// In one thread...
p success 42

// ...and in another thread...
val v = Await.result(f, timeout) // Will be Success(42)
```

Case study: app server with frontend

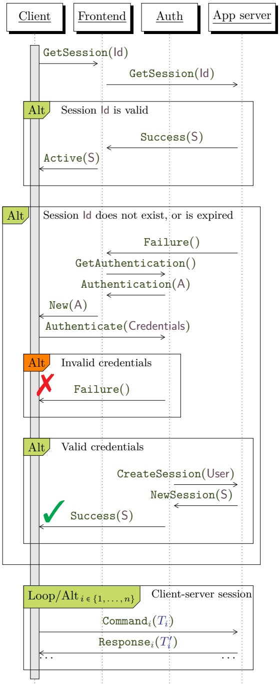
(R. Kuhn, “Project Gålbma: Actors vs Types”, slide 42)



Case study: app server with frontend

(R. Kuhn, “Project Gålbma: Actors vs Types”, slide 42)

Actors interact via **untyped** mailboxes.
What about **protocols** with **sequencing** and **choices**?



Client | Frontend | Auth | App server

`GetSession`(Id)

`GetSession`(Id)

Alt — Session Id is valid

`Success`(S)

`Active`(S)

Alt — Session Id does not exist, or is expired

`Failure`()

`GetAuthentication`()

`Authentication`(A)

`New`(A)

`Authenticate`(Credentials)

Alt — Invalid credentials

✗ `Failure`()

Alt — Valid credentials

`CreateSession`(User)

`NewSession`(S)

✓ `Success`(S)

Loop/Alt$_{i \in \{1,\dots,n\}}$ — Client-server session

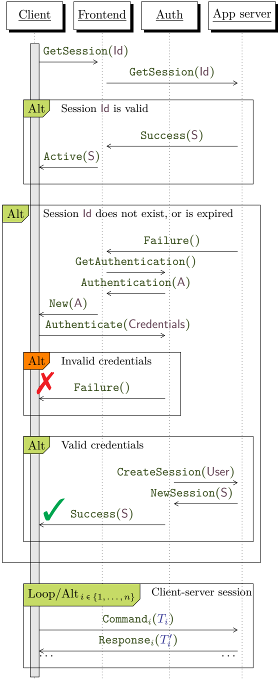
`Command`$_i(T_i)$

`Response`$_i(T'_i)$

… …

Case study: app server with frontend

(R. Kuhn, “Project Gålbma: Actors vs Types”, slide 42)



Client
Frontend
Auth
App server

GetSession(Id)
GetSession(Id)

Alt Session Id is valid
Success(S)
Active(S)

Alt Session Id does not exist, or is expired
Failure()
GetAuthentication()
Authentication(A)
New(A)
Authenticate(Credentials)

Alt Invalid credentials
✗ Failure()

Alt Valid credentials
CreateSession(User)
NewSession(S)
✓ Success(S)

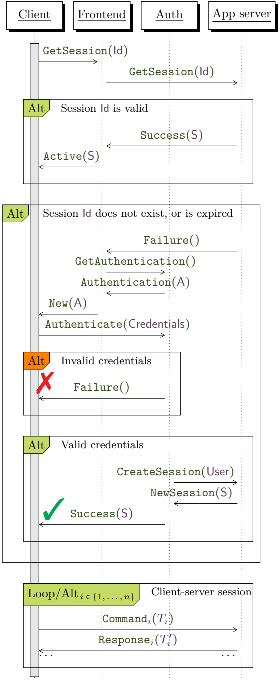
Loop/Alt$_{i \in \{1,\dots,n\}}$ Client-server session
Command$_i$(T_i)
Response$_i$(T'_i)
… …

akka **Actors** interact via **untyped** mailboxes.
What about **protocols** with **sequencing** and **choices**?

Akka Typed proposes typed references `ActorRef[A]`

Case study: app server with frontend

(R. Kuhn, “Project Gålbma: Actors vs Types”, slide 42)



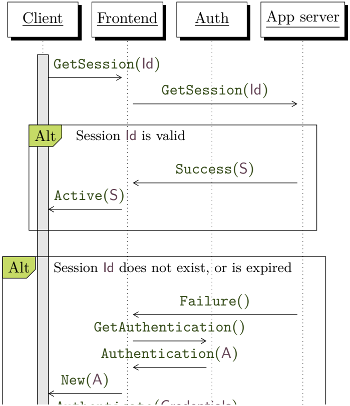
Actors interact via **untyped** mailboxes.
What about **protocols** with **sequencing** and **choices**?

Akka Typed proposes typed references `ActorRef[A]` and **Continuation-Passing Style protocols**

```scala
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult

case class Authenticate(username: String, password: String,
                        replyTo: ActorRef[AuthenticateResult])

sealed abstract class AuthenticateResult
case class Success(service: ActorRef[Command])
      extends AuthenticateResult
case class Failure()  extends AuthenticateResult

sealed abstract class Command
// ... case classes for the client-server session loop ...
```

Case study: app server with frontend (cont'd)

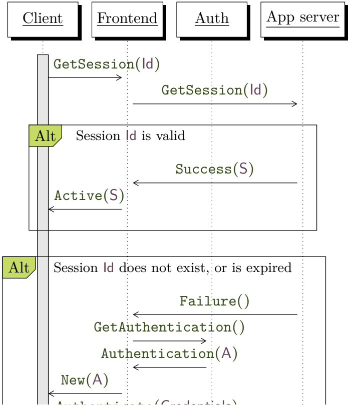
(R. Kuhn, "Project Gålbma: Actors vs Types", slide 42)



```
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult

// ... case classes for authentication, etc ...
```

Case study: app server with frontend (cont'd)

(R. Kuhn, "Project Gålbma: Actors vs Types", slide 42)



```scala
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult

// ... case classes for authentication, etc ...
```

To get a continuation, **spawn a new actor** (pseudo-code follows)

```scala
def client(frontend: ActorRef[GetSession]) = {
  val cont = spawn[GetSessionResult] {
    case New(a)    => doAuthentication(a)
    case Active(s) => doSessionLoop(s)
  }
  frontend ! GetSession(42, cont)
}
```

CPS protocols vs. session types

```
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult

case class Authenticate(username: String, password: String,
                        replyTo: ActorRef[AuthenticateResult])

sealed abstract class AuthenticateResult
case class Success(service: ActorRef[Command])
      extends AuthenticateResult
case class Failure()  extends AuthenticateResult

sealed abstract class Command
// ... case classes for the client-server session loop ...
```

CPS protocols are **Scala types** providing **structured interaction** in Akka

CPS protocols vs. session types

```
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult

case class Authenticate(username: String, password: String,
                        replyTo: ActorRef[AuthenticateResult])

sealed abstract class AuthenticateResult
case class Success(service: ActorRef[Command])
      extends AuthenticateResult
case class Failure()  extends AuthenticateResult

sealed abstract class Command
// ... case classes for the client-server session loop ...
```

CPS protocols are **Scala types** providing **structured interaction** in Akka

But they are also:

- **low-level**, cumbersome to write (and read)
- not related with any **high-level protocol formalism**
- ambiguous about **linearity of (typed) actor references**

CPS protocols vs. session types

```
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult

case class Authenticate(username: String, password: String,
                        replyTo: ActorRef[AuthenticateResult])

sealed abstract class AuthenticateResult
case class Success(service: ActorRef[Command])
      extends AuthenticateResult
case class Failure()  extends AuthenticateResult

sealed abstract class Command
// ... case classes for the client-server session loop ...
```

Idea: if you squint a bit, CPS protocols remind the **encoding of session types into linear and variant types for standard π-calculus**
— i.e., *represent sessions in a language without session primitives*
(Dardha, Giachino & Sangiorgi, PPDP'12; Dardha, BEAT'14)

CPS protocols vs. session types

```
case class GetSession(id: Int,
                      replyTo: ActorRef[GetSessionResult])

sealed abstract class GetSessionResult
case class Active(service: ActorRef[Command])
    extends GetSessionResult
case class New(authc: ActorRef[Authenticate])
    extends GetSessionResult

case class Authenticate(username: String, password: String,
                        replyTo: ActorRef[AuthenticateResult])

sealed abstract class AuthenticateResult
case class Success(service: ActorRef[Command])
      extends AuthenticateResult
case class Failure()  extends AuthenticateResult

sealed abstract class Command
// ... case classes for the client-server session loop ...
```

Idea: if you squint a bit, CPS protocols remind the **encoding of session types into linear and variant types for standard π-calculus**
— i.e., *represent sessions in a language without session primitives*
(Dardha, Giachino & Sangiorgi, PPDP'12; Dardha, BEAT'14)

We develop this idea to obtain **lightweight session types 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)

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

“Session Scala”

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

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

Session vs. linear types (in pseudo-Scala)

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

“Session Scala”

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

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

“Linear Scala”

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

Session vs. linear types (in pseudo-Scala)

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

"Session Scala"

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

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

"Linear Scala"

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

Goals:

- define and implement linear in/out channels
- instantiate the "?" type parameter
- automate channel endpoint creation

`lchannels`: **interface**

```scala
abstract class In[+A] {
  def future: Future[A]






}

abstract class Out[-A] {
  def promise[B <: A]: Promise[B] // Impl. must be constant



}
```

Based on standard `Promise`s/`Future`s

- reuse **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)
  }




}

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


}
```

Based on standard `Promise`s/`Future`s

- reuse **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)

}
```

Based on standard `Promise`s/`Future`s

- reuse **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])
}
```

Based on standard `Promise`s/`Future`s

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

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

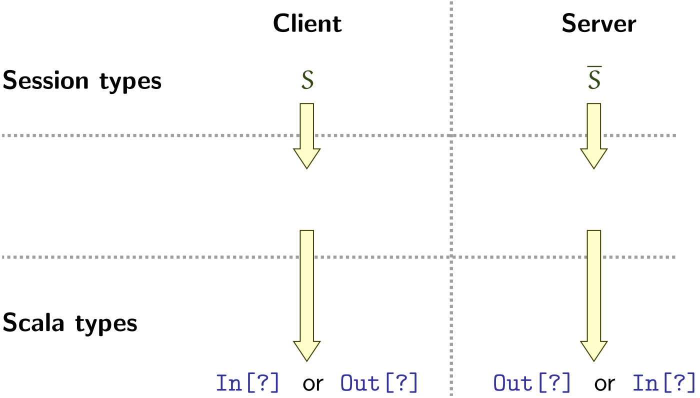
`lchannels`: non-distributed implementation

```
class LocalIn[+A](val future: Future[A]) extends In[A]

class LocalOut[-A](p: Promise[A]) extends Out[A] {
  override def promise[B <: A] = {
    p.asInstanceOf[Promise[B]] // Type-safe cast
  }
  override def create[B]() = LocalChannel.factory[B]()
}

object LocalChannel {
  def factory[A](): (LocalIn[A], LocalOut[A]) = {
    val promise = Promise[A]()
    val future = promise.future
    (new LocalIn[A](future), new LocalOut[A](promise))
  }
}
```

Just a **thin abstraction layer on top of a** `Promise`/`Future` **pair**

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

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



Client

Server

Session types

S

$\overline{S}$

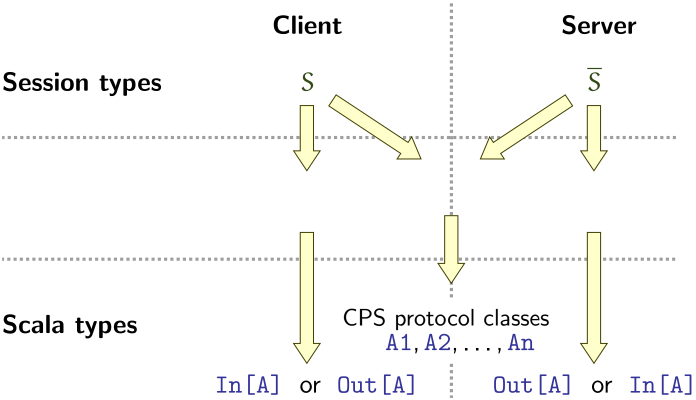
Scala types

`In[?]` or `Out[?]`

`Out[?]` or `In[?]`

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

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



Client

Server

Session types

S

$\overline{S}$

Scala types

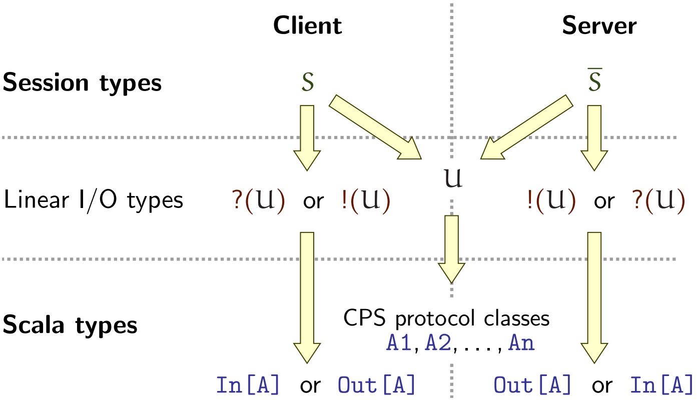
CPS protocol classes
`A1`, `A2`, . . . , `An`

`In[A]` or `Out[A]`

`Out[A]` or `In[A]`

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

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



Client | **Server**

Session types S $\overline{S}$

Linear I/O types $?(U)$ or $!(U)$ U $!(U)$ or $?(U)$

Scala types

CPS protocol classes
`A1`, `A2`, . . . , `An`

`In[A]` or `Out[A]` | `Out[A]` or `In[A]`

Programming with `lchannels` (I)

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

Programming with `lchannels` (I)

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

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

```
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
```

```
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)

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

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

```scala
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
```

```scala
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 channel endpoint creation ✗

Automating channel endpoint creation

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

Automating channel endpoint creation

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

Programming with `lchannels` (II)

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

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 hello(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")

    c ? {
      case Hello(name) => hello(c)
      case 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 hello(c: S_h): Unit = {
  if (...) {
    c ! Greet("Alice")

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

Programming with `lchannels` (II)

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

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

```
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
```

“Session Scala” (pseudo-code)

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

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

Scala + `lchannels`

```
def hello(c: Out[Start]): Unit = {
  if (Random.nextBoolean()) {
    val c2 = c !! Greet("Alice")_

    c2 ? {
      case m @ Hello(name) => hello(m.cont)
      case Bye(name)       => ()
    }
  } else {
    c ! Quit()
  }
}
```

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 (disallowed)**
Double use of linear input endp.	**Run-time (allowed, constant)**

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 (disallowed)**
Double use of linear input endp.	**Run-time (allowed, constant)**
"Forgotten" output	**Run-time (timeout on input side)**
"Forgotten" input	**Unchecked**

`lchannels` for distributed interaction

We have seen a **local** implementation of `lchannels`, allowing **thread interaction**

```
val (cin, cout) = LocalChannel.factory[Start]()
// cin, cout  have type  LocalIn[Start], LocalOut[Start]

spawnThread( server(cin) )
spawnThread( client(cout) )
```

`lchannels` for distributed interaction

We have seen a **local** implementation of `lchannels`,
allowing **thread interaction**

```
val (cin, cout) = LocalChannel.factory[Start]()
// cin, cout  have type  LocalIn[Start], LocalOut[Start]

spawnThread( server(cin) )
spawnThread( client(cout) )
```

However, `LocalIn`/`LocalOut` instances **cannot be serialised**,
and thus **cannot be sent/received** over a network

`lchannels` for distributed interaction

We have seen a **local** implementation of `lchannels`, allowing **thread interaction**

```
val (cin, cout) = LocalChannel.factory[Start]()
// cin, cout  have type  LocalIn[Start], LocalOut[Start]

spawnThread( server(cin) )
spawnThread( client(cout) )
```

However, `LocalIn`/`LocalOut` instances **cannot be serialised**, and thus **cannot be sent/received** over a network

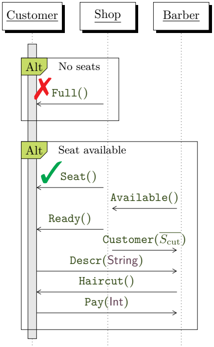
Still, `In`/`Out` can **abstract a distributed communication medium**

- we implemented interaction via **Akka actors** and **TCP/IP sockets**

Examples and case studies

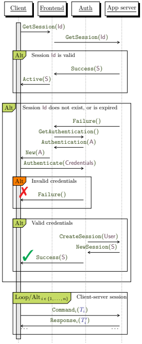
We used session types, CPS protocols and `lchannels` to implement several examples and case studies, including e.g.:

Sleeping barber



(types: ~20 LOC)

Chat server with frontend

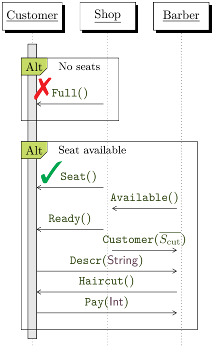


(types: ~100 LOC)

Examples and case studies

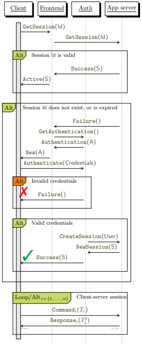
We used session types, CPS protocols and `lchannels` to implement several examples and case studies, including e.g.:

Sleeping barber



(types: ~20 LOC)

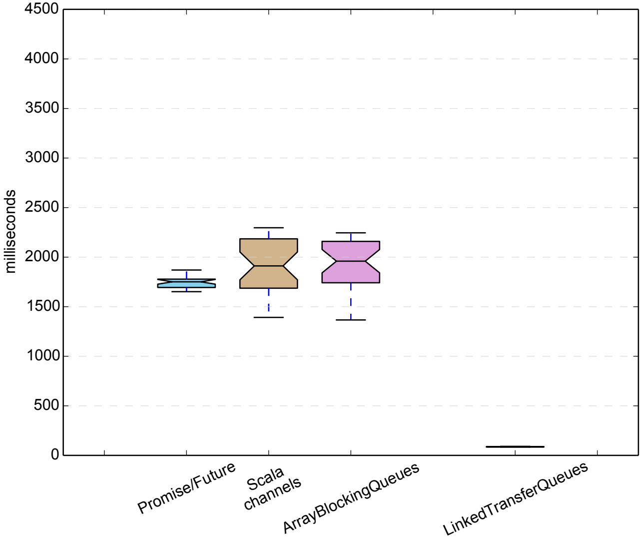
Chat server with frontend



(types: ~100 LOC)

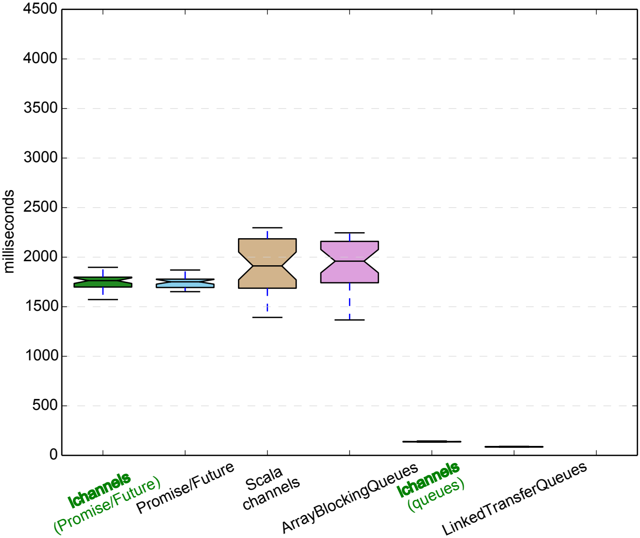
- Most typical protocol errors are **statically ruled out**
- **Timeouts** and **linearity exceptions** take care of the rest
- **Session delegation** is supported — even encouraged!

Benchmark: ping-pong (448,000 message transmissions, 30 runs)



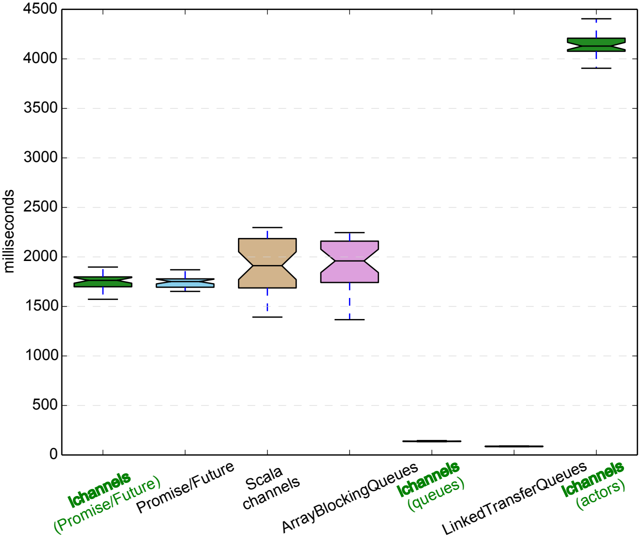
Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB of RAM, Ubuntu 14.04, Oracle JDK 8u72, Scala 2.11.8, Akka 2.4.2

Benchmark: ping-pong (448,000 message transmissions, 30 runs)



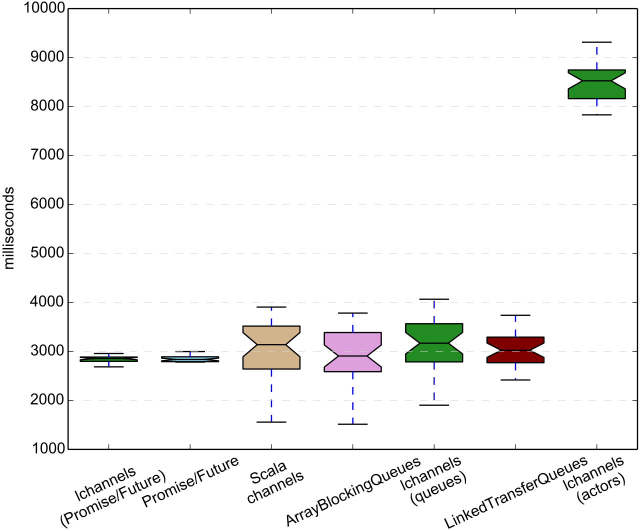
Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB of RAM, Ubuntu 14.04, Oracle JDK 8u72, Scala 2.11.8, Akka 2.4.2

Benchmark: ping-pong (448,000 message transmissions, 30 runs)



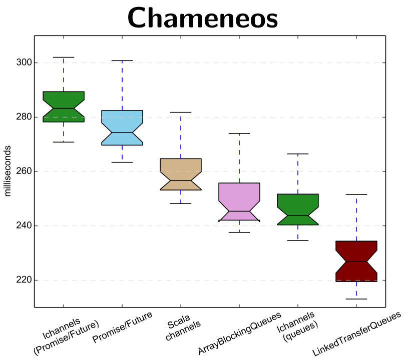
Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB of RAM, Ubuntu 14.04, Oracle JDK 8u72, Scala 2.11.8, Akka 2.4.2

Benchmark: ring (448,000 message transmissions, 30 runs)

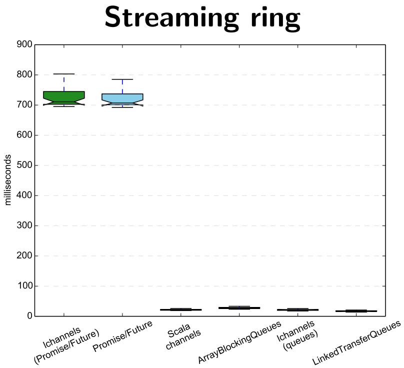


Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB of RAM, Ubuntu 14.04, Oracle JDK 8u72, Scala 2.11.8, Akka 2.4.2

More benchmarks (448,000 message transmissions, 30 runs)



Chameneos



Streaming ring

Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB of RAM, Ubuntu 14.04, Oracle JDK 8u72, Scala 2.11.8, Akka 2.4.2

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 $\overline{\mathtt{In[A]}} = \mathtt{Out[A]}$ and $\overline{\mathtt{Out[A]}} = \mathtt{In[A]}$).

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

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**

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

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

We found a **formal connection** between **CPS protocols** and session types, based on their **encoding into linear/variant types** for standard π-calculus

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

Future work

Lightweight integration of **multiparty session types**, using **Scribble** to generate (more complicated) CPS protocol classes

Generalise the approach to other frameworks beyond `lchannels`, and study its properties.
Natural candidate: **Akka Typed**

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

Thanks!

(questions?)

Session types = $\texttt{In}[\cdot]/\texttt{Out}[\cdot]$ + CPS protocols (cont'd)

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

Session types = $\mathtt{In}[\cdot]$/$\mathtt{Out}[\cdot]$ + CPS protocols (cont'd)

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

Let us **give a name to each non-singleton int/ext choice**:

$$\mathcal{N}\left(\begin{array}{l} !\mathtt{Greet}(\mathsf{String}).\left(\begin{array}{l} ?\mathtt{Hello}(\mathsf{String}).X \\ \&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end} \end{array}\right) \\ \oplus\; !\mathtt{Quit}(\mathsf{Unit}) \end{array}\right) = \mathtt{Start} \qquad \mathcal{N}\left(\begin{array}{l} ?\mathtt{Hello}(\mathsf{String}).X \\ \&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end} \end{array}\right) = \mathtt{Greeting}$$

Session types = $\texttt{In[}\cdot\texttt{]/Out[}\cdot\texttt{]}$ + CPS protocols (cont'd)

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

Let us **give a name to each non-singleton int/ext choice**:

$$\mathcal{N}\left(\begin{array}{l}\texttt{!Greet}(\mathrm{String}).\left(\begin{array}{l}\texttt{?Hello}(\mathrm{String}).X\\ \mathbin{\&} \texttt{?Bye}(\mathrm{String}).\mathbf{end}\end{array}\right)\\ \oplus\ \texttt{!Quit}(\mathrm{Unit})\end{array}\right) = \texttt{Start} \qquad \mathcal{N}\left(\begin{array}{l}\texttt{?Hello}(\mathrm{String}).X\\ \mathbin{\&} \texttt{?Bye}(\mathrm{String}).\mathbf{end}\end{array}\right) = \texttt{Greeting}$$

We get the **CPS protocol** of S_{h}

$\mathrm{prot}\langle\langle S_{\mathrm{h}} \rangle\rangle_{\mathcal{N}}$ =

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

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

Session types = In[·]/Out[·] + CPS protocols (cont'd)

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

Let us **give a name to each non-singleton int/ext choice**:

$$\mathcal{N}\left(\begin{array}{l} !\mathtt{Greet}(\mathsf{String}).\left(\begin{array}{l} ?\mathtt{Hello}(\mathsf{String}).X \\ \&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end} \end{array}\right) \\ \oplus\; !\mathtt{Quit}(\mathsf{Unit}) \end{array}\right) = \mathtt{Start} \qquad \mathcal{N}\left(\begin{array}{l} ?\mathtt{Hello}(\mathsf{String}).X \\ \&\; ?\mathtt{Bye}(\mathsf{String}).\mathbf{end} \end{array}\right) = \mathtt{Greeting}$$

We get the **CPS protocol** of S_h and its **linear channel endpoint type**:

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

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

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

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

Session types = $\texttt{In}[\cdot]$ / $\texttt{Out}[\cdot]$ + CPS protocols (cont'd)

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

Let us **give a name to each non-singleton int/ext choice**:

$$\mathcal{N}\left(\begin{array}{l} !\texttt{Greet}(\texttt{String}).\left(\begin{array}{l} ?\texttt{Hello}(\texttt{String}).X \\ \&\; ?\texttt{Bye}(\texttt{String}).\textbf{end} \end{array}\right) \\ \oplus\; !\texttt{Quit}(\texttt{Unit}) \end{array}\right) = \texttt{Start} \qquad \mathcal{N}\left(\begin{array}{l} ?\texttt{Hello}(\texttt{String}).X \\ \&\; ?\texttt{Bye}(\texttt{String}).\textbf{end} \end{array}\right) = \texttt{Greeting}$$

Intermediate encoding into **linear types with variants and records**:

$$[\![S_h]\!] = !\left(\mu_X.\left[\begin{array}{l} \texttt{Greet}\_\left\{\mathrm{p} : \mathrm{String}, \mathrm{c} : !\left(\left[\begin{array}{l} \texttt{Hello}\_\{\mathrm{p} : \mathrm{String}, \mathrm{c} : !(X)\}, \\ \texttt{GoodNight}\_\{\mathrm{p} : \mathrm{String}, \mathrm{c} : \bullet\} \end{array}\right]\right)\right\}, \\ \texttt{Quit}\_\{\mathrm{p} : \mathrm{Unit}, \mathrm{c} : \bullet\} \end{array}\right]\right)$$

We get the **CPS protocol** of S_h and its **linear channel endpoint type**:

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

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

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

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

Actor-based channels

`ActorIn` and `ActorOut` extend `In`/`Out`, and send/receive messages by **automatically spawning Akka Typed actors**

- they are **both serialisable** (unlike `LocalIn`/`LocalOut`)
- **distributed interaction for free!** (on top of Akka Remoting)

Actor-based channels

`ActorIn` and `ActorOut` extend `In`/`Out`, and send/receive messages by **automatically spawning Akka Typed actors**

- they are **both serialisable** (unlike `LocalIn`/`LocalOut`)
- **distributed interaction for free!** (on top of Akka Remoting)

For example, on one JVM:

```
val (in, out) = ActorChannel.factory[Start]()
// in, out  have type  ActorIn[Start], ActorOut[Start]
server(in)
```

Actor-based channels

`ActorIn` and `ActorOut` extend `In`/`Out`, and send/receive messages by **automatically spawning Akka Typed actors**

- they are **both serialisable** (unlike `LocalIn`/`LocalOut`)
- **distributed interaction for free!** (on top of Akka Remoting)

For example, on one JVM:

```
val (in, out) = ActorChannel.factory[Start]()
// in, out  have type  ActorIn[Start], ActorOut[Start]
server(in)
```

On **another** JVM, we can **proxy** `out` **through its Actor Path**:

```
val c = ActorOut[Start]("akka.tcp://sys@host.com/user/start")
// c  has type  ActorOut[Start]
client(c)
```

Stream-based channels

`StreamIn`/`StreamOut` allow **interoperability**
by **reading/writing messages from/to byte streams**

Stream-based channels

`StreamIn`/`StreamOut` allow **interoperability**
by **reading/writing messages from/to byte streams**

Suppose that our "greeting protocol" has a **textual format** (e.g., from an RFC):

Message	Text format
`Greet`("Alice")	`GREET Alice`
`Hello`("Alice")	`HELLO Alice`
`Bye`("Alice")	`BYE Alice`
`Quit`()	`QUIT`

Stream-based channels

`StreamIn`/`StreamOut` allow **interoperability**
by **reading/writing messages from/to byte streams**

Suppose that our "greeting protocol" has a **textual format** (e.g., from an RFC):

Message	Text format
`Greet("Alice")`	`GREET Alice`
`Hello("Alice")`	`HELLO Alice`
`Bye("Alice")`	`BYE Alice`
`Quit()`	`QUIT`

We define a `StreamManager` to read/write it (on the right)

```
class HelloStreamManager(in: InputStream, out: OutputStream)
      extends StreamManager(in, out) {
  private val outb = new BufferedWriter(new OutputStreamWriter(out))

  override def streamer(x: scala.util.Try[Any]) = x match {
    case Failure(e) => close() // StreamManager.close() closes in & out
    case Success(v) => v match {
      case Greet(name) => outb.write(f"GREET ${n}\n"); outb.flush()
      case Quit() => outb.write("QUIT\n"); outb.flush(); close() // End
    }
  }

  private val inb = new BufferedReader(new InputStreamReader(in))
  private val helloR = """HELLO (.+)""".r // Matches Hello(name)
  private val byeR = """BYE (.+)""".r     // Matches Bye(name)

  override def destreamer() = inb.readLine() match {
    case helloR(name) => Hello(name)(StreamOut[Greeting](this))
    case byeR(name) => close(); Bye(name) // Session end: close streams
    case e => { close(); throw new Exception(f"Bad message: '${e}'") }
  }
}
```

Stream-based channels

`StreamIn`/`StreamOut` allow **interoperability**
by **reading/writing messages from/to byte streams**

Suppose that our "greeting protocol" has a **textual format** (e.g., from an RFC):

Message	Text format
Greet("Alice")	GREET Alice
Hello("Alice")	HELLO Alice
Bye("Alice")	BYE Alice
Quit()	QUIT

We define a `StreamManager` to read/write it (on the right)

```
class HelloStreamManager(in: InputStream, out: OutputStream)
      extends StreamManager(in, out) {
  private val outb = new BufferedWriter(new OutputStreamWriter(out))

  override def streamer(x: scala.util.Try[Any]) = x match {
    case Failure(e) => close() // StreamManager.close() closes in & out
    case Success(v) => v match {
      case Greet(name) => outb.write(f"GREET ${n}\n"); outb.flush()
      case Quit() => outb.write("QUIT\n"); outb.flush(); close() // End
    }
  }

  private val inb = new BufferedReader(new InputStreamReader(in))
  private val helloR = """HELLO (.+)""".r // Matches Hello(name)
  private val byeR = """BYE (.+)""".r     // Matches Bye(name)

  override def destreamer() = inb.readLine() match {
    case helloR(name) => Hello(name)(StreamOut[Greeting](this))
    case byeR(name) => close(); Bye(name) // Session end: close streams
    case e => { close(); throw new Exception(f"Bad message: '${e}'") }
  }
}
```

```
val conn = new Socket("host.com", 1337) // Hostname and port of greeting server
val strm = new HelloStreamManager(conn.getInputStream, conn.getOutputStream)
val c = StreamOut[Start](strm) // Output channel endpoint, to host.com:1337
client(c)
```