Lightweight Session Programming in Scala

Alceste Scalas¹ and Nobuko Yoshida²

¹ Imperial College London, UK  
a.scalas@imperial.ac.uk
₂ Imperial College London, UK  
n.yoshida@imperial.ac.uk

Abstract
Designing, developing and maintaining concurrent applications is an error-prone and time-consuming task; most difficulties arise because compilers are usually unable to check whether the inputs/outputs performed by a program at runtime will adhere to a given protocol specification.

To address this problem, we propose lightweight session programming in Scala: we leverage the native features of the Scala type system and standard library, to introduce (1) a representation of session types as Scala types, and (2) a library, called lchannels, with a convenient API for session-based programming, supporting local and distributed communication. We generalise the idea of Continuation-Passing Style Protocols (CPSPs), studying their formal relationship with session types. We illustrate how session programming can be carried over in Scala: how to formalise a communication protocol, and represent it using Scala classes and lchannels, letting the compiler help spotting protocol violations. We attest the practicality of our approach with a complex use case, and evaluate the performance of lchannels with a series of benchmarks.

1998 ACM Subject Classification  D.1.3 Concurrent Programming, D.3.1 Formal Definitions and Theory, F.3.3 Studies of Program Constructs – Type structure

Keywords and phrases  session types, Scala, concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.21

Supplementary Material  ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.11

1 Introduction and motivation
Concurrent and distributed applications are notoriously difficult to design, develop and maintain. One of the main challenges lies in ensuring that software components interact according to some predetermined communication protocols describing all the valid message exchanges. Such a challenge is typically tackled at runtime, e.g. via testing and message monitoring.

Unfortunately, depending on the number of software components and the complexity of their protocols, tests and monitoring routines can be costly to develop and to maintain, as software and protocols evolve.

Consider the message sequence chart in Figure 1: it is based on an example of “actor protocol” from [26] (slide 42), and schematises the authentication procedure of an application server. A client connects to a frontend, trying to retrieve an active session by its Id; the
frontend queries the application server: if Id is valid, the client gets an Active(S) message with a session handle S, which can be used to perform the command/response loop at the bottom; otherwise, the client must authenticate: the frontend obtains an handle A from an authentication server, and forwards it to the client with a New(A) message. The client must now use A to send its credentials (through an Authenticate message); if they are not valid, the authentication server replies Failure(); otherwise, it retrieves a session handle S and sends Success(S) to the client, who uses S for the session loop (as above). In this example, four components interact with intertwined protocols. Ensuring that messages are sent with the right type and order, and that each component correctly handles all possible responses, can be an elusive and time-consuming task. Runtime monitoring/testing can detect the presence of communication errors, but cannot guarantee their absence; moreover, protocols and code may change during the life cycle of an application – and monitoring/testing procedures will need to be updated. Compile-time checks would allow to reduce this burden, lowering software maintenance costs.

CPS protocols in Scala. The developers of the Akka framework [28] have been addressing these challenges, in the setting of actor-based applications. Standard actors communicate in an untyped way: they can send each other any message, anytime, and must check at runtime whether a given protocol is respected. Akka developers are thus trying to leverage the Scala type system to obtain static protocol definitions and compile-time guarantees on the absence...
case class GetSession(id: Int, 
    replyTo: ActorRef[GetSessionResult])
sealed abstract class GetSessionResult
  case class New(authc: ActorRef[Authenticate]) 
  extends GetSessionResult
  case class Active(service: ActorRef[Command]) 
  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 ...

Figure 2 Akka Typed: protocol of client in Fig. 1.

def client(frontend: ActorRef[GetSession]) = {
  val cont = spawn[GetSessionResult] {
    case New(a) => doAuthentication(a)
    case Active(s) => doSessionLoop(s)
  }
  frontend ! GetSession(42, cont)
}
// ... case classes for the client-server session loop ...

Figure 3 Actor spawning (pseudo code).

of communication errors. Their tentative solution has two parts. The first is Akka Typed [29]: an experimental library with actors that can only receive messages via references of type ActorRef[A], which in turn only allow to send A-typed messages. The second is what we dub Continuation-Passing Style Protocols (CPSPs): sets of message classes that represent sequencing with a replyTo field, of type ActorRef[B]. By convention, replyTo tells where the message recipient should send its B-typed answer: Fig. 2 (based on [26], slide 41) shows the CPSPs of the client in Fig. 1.

In practice, a replyTo field can be instantiated by producing a “continuation actor” that handles the next step of the protocol. Fig. 3 shows a client that, before sending GetSession to the frontend (line 6), spawns a new actor accepting GetSessionResult messages. Then, cont (line 2) has type ActorRef[GetSessionResult], and is sent as replyTo: the frontend should send its New/Active answer there. This creates a conversation between the client and frontend: the message sender produces a “continuation”, and the receiver should use it.

Opportunities and limitations. CPSPs have the appealing feature of being standard Scala types, checked by its compiler, and giving rise to a form of structured interaction in Akka. However, their incarnation seen above has some shortcomings. First and foremost, they are a rather low-level representation, not connected with any established, high-level formalisation of protocols and structured interaction. Hence, non-trivial protocols with branching and recursion (e.g. the one in Fig. 1) can be hard to write and understand in CPS; even message ownership and sequencing may be non-obvious: e.g., determining who sends Failure in Fig. 2, and whether it comes before or after another message, can take some time. Moreover, the CPSPs in Fig. 2 seems to imply that some continuations should be used exactly once – but this intuition is not made explicit in the types. E.g., in Fig. 3, frontend and cont are both ActorRefs – but the actor referred by frontend might accept multiple GetSession
requests, whereas the one referred by cont (spawned on lines 2–5) might just wait for one New/Active message, spawn another continuation actor, and terminate. Arguably, the type of cont should convey whether sending more than one message is an error.

Our contribution: lightweight session programming in Scala. We address the challenges and limitations above by proposing lightweight session programming in Scala – where “lightweight” means that our proposal does not depend on language extensions, nor external tools, nor specific message transport frameworks. We generalise the idea of CPSP, relating it to a well established formalism for the static verification of concurrent programs: session types [19, 20, 39]. We present a library, called lchannels, offering a simplified API for session programming with CPSPs, supporting network-transparent communication. Albeit the Scala type checker does not cater for all the static guarantees provided by session-typed languages (mostly due to the lack of static linearity checks), we show that lchannels and CPSPs allow to represent protocol specifications as Scala types, and write session-based programs in a rather natural way, guaranteeing protocol safety: i.e., once a session starts, no out-of-protocol messages can be sent, and all valid incoming messages are handled. We show that typical protocol errors are detected at compile-time – except for linearity errors: lchannels checks them at runtime, reminding the typical usage of Scala Promises/Futures.

This work focuses on Scala since we leverage several convenient features of the language and its standard library: object orientation, parametric polymorphism with declaration-site variance, first-class functions, labelled union types (case classes), Promises/Futures; yet, our approach could be adapted (at least in part) to any language with similar features.

Outline of the paper. In §2, we summarise session types, explaining the difficulties in their integration in a language like Scala, and how we overcome them by exploiting an encoding into linear types for I/O. In §3 we introduce lchannels, a library for type-safe communication over asynchronous linear channels. In §4 we explain, via several examples, how session programming can be carried over in Scala, by using lchannels and representing session types as CPSPs, according to a session-based software development approach (§4.2). §5 presents optimisations and extensions of lchannels, achieving message transport abstraction and network-transparent communication. In §6 we show the practicality of our approach by implementing the case study in Fig. 1, and evaluating the performance of lchannels – particularly, its message delivery speed w.r.t. other inter-process communication methods. In §7 we give a formal foundation to §4, proving crucial results about duality/subtyping of session types represented in Scala, and overcoming technical difficulties in the transition from a structural to nominal types (e.g., different handling of recursion). We discuss related works in §8, and conclude in §9 – showing how our approach can be adapted to other communication frameworks.

Online resources. Due to space limits, we include proofs, benchmarking details and other materials in http://www.doc.ic.ac.uk/research/technicalreports/2015/#7. For the latest version of lchannels, visit http://alcestes.github.io/lchannels/.

2 Programming with session types: background and challenges

We now summarise the features of languages based on binary session types (§2.1) and their notions of duality and subtyping (§2.2). We then explain their relationship with linear I/O types (§2.3), and give an overview of our strategy for representing them in Scala (§2.4).
2.1 Background: binary session types in a nutshell

Session types regulate the interaction of processes communicating through channels; each channel has two endpoints, and the intuitive semantics is that all values sent on one endpoint can be received on the other in the same order – a bidirectional FIFO model akin e.g. to TCP/IP sockets. A session type says how a process is expected to use a channel endpoint. Let \( \mathbb{B} = \{ \text{Int}, \text{Bool}, \text{Unit}, \ldots \} \) be a set of basic types. A session type \( S \) has the following syntax:

\[
S := \&_{i \in I} ?_i(T_i).S_i \mid \oplus_{i \in I} !_i(T_i).S_i \mid \mu_X.S \mid X \mid \text{end} \quad T := \mathbb{B} \mid S \text{ (closed)}
\]

where \( I \neq \emptyset \), recursion is guarded, and all \( I \) range over pairwise distinct labels. \( T \) denotes a payload type. The branching type (or external choice) \( \&_{i \in I} ?_i(T_i).S_i \) requires the process to receive one input of the form \( 1_i(T_i) \), for any \( i \in I \) chosen at the other endpoint; then, the channel must be used according to the continuation type \( S_i \). The selection type (or internal choice) \( \oplus_{i \in I} !_i(T_i).S_i \), instead, requires the program to choose and perform one output \( 1_i(T_i) \), for some \( i \in I \), and continue using the channel according to \( S_i \). \( \mu_X.S \) is a recursive session type, where \( \mu \) binds \( X \), and \( X \) is a recursion variable. We say that \( S \) is closed iff all its recursion variables are bound. \text{end} is a terminated session with no further inputs/outputs. Note that a payload type \( T \) can be either a basic or a session type: hence, channel endpoints allow to send/receive e.g. integers, strings, or other channel endpoints.

\[ \textbf{Remark 2.1.} \] We use \( \oplus/\& \) as infix operators, omitting them in singleton choices. We often omit \text{end} and Unit: \( ?(\text{Unit}) \oplus !\text{Unit} \) stands for \( \& \{ ?(\text{Unit}) \oplus !\text{Unit}, \text{end} \} \).

For example, the type \( S_h \) below describes the client endpoint of a “greeting protocol”:

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

The client can send either \text{Quit} and \text{end} the session, or \text{Greet(String)}; in the second case, it might receive from the server either \text{Bye(String) (ending the session)}, or \text{Hello(String)}: in the second case, the session continues recursively.

Programming languages that support session types are usually based on session-\( \pi \)-i.e., a version of \( \pi \)-calculus [31] extended with session operators. A client respecting \( S_h \) would be implemented as \text{hello(c)} in Fig. 4 (left): \( c \) is a \( S_h \)-typed channel endpoint, \( ! \) is a language primitive for selecting and sending messages, and \( ? \) for branching (i.e., receiving and pattern matching messages). The type system ensures that \( c \) is used according to \( S_h \), guaranteeing:

\begin{enumerate}
  \item \textbf{S1. safety:} no out-of-protocol I/O actions are allowed. E.g., \( c \) can initially be used only to send \text{Greet/Quit} (lines 3,8), no outputs are allowed when \( S_h \) expects \( c \) to receive (line 4), no inputs when \( S_h \) expects \( c \) to send (lines 3,8), no I/O when \( S_h \) has ended (line 6);
  \item \textbf{S2. exhaustiveness:} when receiving a message, all outcomes allowed by the type must be covered. E.g., the client must handle both \text{Hello} and \text{Bye} answers (lines 4–6);
  \item \textbf{S3. output linearity:} if \( S_h \) prescribes an output, it must occur exactly once. E.g., after receiving \text{Hello}, the client must send \text{Greet} or \text{Quit} (as in the recursive call of line 5);
  \item \textbf{S4. input linearity:} similarly, if \( S_h \) prescribes an input, it must occur exactly once. E.g., after sending \text{Greet}, the client must receive the response (as in line 4).
\end{enumerate}

2.2 Background: safe, deadlock-free interaction via duality/subtyping

A session-typed language ensures correct run-time interaction by statically checking that the two endpoints of a channel are used \textit{dually}. The \textit{dual} of \( S \), written \( \overline{S} \), is defined as:

\[
\overline{\&_{i \in I} ?_i(T_i).S_i} = \oplus_{i \in I} !_i(T_i).S_i \quad \overline{\oplus_{i \in I} !_i(T_i).S_i} = \&_{i \in I} ?_i(T_i).S_i
\]

\[
\mu_X.\overline{S} = \mu_X.X \quad \overline{X} = X \quad \text{end} = \text{end}
\]
Intuitively, the internal/external choices of $S$ are swapped in $\overline{S}$; hence, each client-side output is matched by a server-side input, and vice versa. In our example, $c$ is a client-side endpoint that must be used according to $S_h$; the server-side dual channel endpoint has type:

$$\overline{S}_h = \mu_X.(?\text{Greet}(\text{String}).(\Pi\text{Hello}(\text{String}).X \oplus \Pi\text{Bye}(\text{String}).\text{end}) & ?\text{Quit}.\text{end})$$

Duality guarantees the safe and deadlock-free interaction of a client and server observing $S_h$ and $\overline{S}_h$: no unexpected messages are sent/received, and the session progresses until its end.

Such a guarantee is made more flexible via session subtyping [13]. Consider the type $S_{h2} = \Pi\text{Quit}$, and its implementation on the right: since $\text{hello2}$ only outputs $\text{Quit}$ on $c_2$, it would also behave safely on a $S_{h1}$-channel endpoint $c$. In fact, in a session-typed language we have $S_h \preceq S_{h2}$ — i.e., an $S_h$-typed channel endpoint can always be used in place of an $S_{h2}$-typed one; hence, invoking $\text{hello2}(c)$ is allowed — and such a client program would interact safely and without deadlocks with a server observing $\overline{S}_h$.

### 2.3 From session-typed to linearly-typed programs

Unfortunately, integrating session types into a “mainstream” programming language is not trivial: they require sophisticated type system features. Safety/exhaustiveness can be achieved by letting $c$’s type evolve according to $S_h$ after each I/O action — but most type systems assign a fixed type to each variable; I/O linearity checks require linearity analysis; internal/external choices, session subtyping and duality need dedicated type-level machinery.

In this paper, we show how session programming can be carried over in Scala, recovering part of the static guarantees provided by session types. We take inspiration from the encoding of session-$\pi$ into standard $\pi$-calculus with variants and linear I/O types [8]: the key idea is that session-$\pi$ and session types can be encoded in a more basic language and type system that do not natively support session primitives (e.g., internal/external choices and duality), by adopting a “continuation-passing style” interaction over linear input/output channel endpoints that are used exactly once. In particular, [8] (Theorems 1, 2) proves that a process using variants, linear I/O types and CPS interaction can precisely mirror the typing and the runtime communications of a session typed process.

An intuition of our approach is given in Fig. 4 (right), where $\Pi\text{Hello}$ is the “linearly encoded” version of $\text{hello}$. Its argument $c$ is a linear output channel endpoint that carries a single value (whose type is left unspecified, for now). On line 3, it creates a new pair of linear channels endpoints, which can carry another single value of some (again unspecified) type: intuitively, what is sent on $c_2\text{out}$ becomes available in $c_2\text{in}$. On line 4, $c$ is used to send a $\text{Greet}$ message — which also carries $c_2\text{out}$. Then, the recipient of $\text{Greet}$ and $c_2\text{out}$ is expected to use the latter to continue the session — i.e., send either $\Pi\text{Hello}$ or $\Pi\text{Bye}$. On line 5, $c_2\text{in}$ is used to receive such an answer, and the result is matched against $\Pi\text{Hello}$ and $\Pi\text{Bye};$ the

```scala
def hello(c: $h_h$): Unit = {
  if (...) {
    case c ! Greet("Alice") => 
      case c ? { h2 ! Greet("Alice") } 
      case c2in.receive {
        case c2out.send( Greet("Alice", c2out) ) => 
          } 
        } 
    } else { c ! Quit() }
}

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

*Figure 4* Greeting protocol client (pseudo code): session types (left) vs. linear I/O types (right).
latter carries no continuation channel, i.e. the session has ended (line 7); the former, instead, carries a linear (output) channel endpoint c3out, that is used to continue the session with a recursive call (line 6). Note that all channel endpoints received/created in lHello are either used exactly once (c, c2in, c3out), or sent to some other process (c2out).

A crucial difference between hello and lHello is that in the latter, each variable has a constant type. This suggests that, although the Scala type checker cannot check linearity, it might be leveraged to obtain a form of session typing, offering safety and exhaustiveness for programs written in “linear CPS”, like lHello. Then, as seen in §2.2, we could also obtain safe and deadlock-free interaction – provided that a program creates, uses or sends its linear channel endpoints according to [8], and the other program involved in a session interacts in a “dual” way. However, the pseudo-code of Fig. 4 (right) highlights four Problems:

P1. we need to represent and implement linear input and output channels;
P2. we need to suitably instantiate each ?-type, so to describe the same interactions of Sh;
P3. we must automate the creation, sending and use of linear channels, offering an API that guides the CPS interactions prescribed in [8], and allows to write code similar to hello;
P4. we need to handle session subtyping and duality in the Scala type system.

2.4 From session types to session programming in Scala: an outline

In the rest of the paper, we demonstrate how to tackle Problems P1–P4, staying close to the session/linear types theory, and yet achieving practical session programming in Scala. Our approach is summarised in Fig. 5. On top, we have a client and a server that should interact through a channel, whose protocol is described with dual session types S and S. On the bottom, the same protocol is represented in Scala, as a set of CPSP classes, shared between the client and server, and similar to those discussed in §1: they are used as parameters for In[A] and Out[A], which implement respectively an input/output channel endpoint carrying a single value of type A. We extract such CPSP classes from S or S, through an encoding represented by the arrows; such an encoding exploits an intermediate generation of linear I/O types (middle of Fig. 5), as detailed in §7. We address P1 in §3, P2 in §4, P3 in §4.3, and P4 in §7.3.

3 lchannels, a (small) library for type-safe interaction

We now introduce lchannels, a Scala library providing typed linear channels. We designed the programmer interface to be close to the formal definition of linear channels (§7.2) –
notably, by reflecting their co/contra-variance. For simplicity, we shape the API and its basic implementation around Promises/Futures from the Scala standard library [16], since they are familiar to Scala developers, and remarkably close to the expected usage of linear channel endpoints (§2.3): (i) a Promise[A] must be completed exactly once with an A-typed value v, and (ii) after completion, v becomes available on the corresponding Future[A]. Moreover, Promises/Futures provide asynchronous message passing.

We present the lchannels API in §3.1, and a simple implementation in §3.2. We give further details, examples and extensions after showing the representation of session types as CPSP classes (§4) – which constitute the principal use case for lchannels.

### 3.1 The programmer interface

The cornerstones of lchannels are the abstract classes Out[-A] and In[+A], representing channel endpoints allowing respectively to send and receive one A-typed value. Their slightly simplified declarations are shown in Fig. 6 (left).

The class Out[-A] is contravariant w.r.t. A¹. Its promise (line 12) is expected to be eventually completed with the value to be sent; a crucial requirement is that promise must be implemented as a constant², to ensure that it will be completed only once. Note that due to the contravariance of A, the type of promise cannot be simply Promise[A]: the reason is that the latter is invariant w.r.t. A; the bounded type parameter B <: A allows to overcome this limitation. send(msg) and its alias ! offer a simplified interface above promise, representing the selection/output operator of session-π (see Theorem 3.1). Finally, Out’s abstract method create[B]() returns a new pair of input/output channels carrying B; this method is used to create continuation endpoints, as seen in Fig. 4 (right, line 3).

The class In[+A] is covariant w.r.t. its type parameter A³. Its future will contain the value sent from the corresponding Out endpoint. The receive method offers a simplified interface over future: the implicit parameter d specifies how long to wait for an incoming message before raising a timeout error. The ? method implements the typical branching operator of session-π: it takes a function f: A => B, and once a value v is received, it returns f(v). The rationale behind the method signature is clarified in Theorem 3.1.

---

¹ This matches the output subtyping rule [κ-Out] in Theorem 7.4.
² Such a requirement could be enforced by defining the field as val, instead of def; the drawback is that val does not allow type parameters, and this would result in an invariant Out with limited subtyping.
³ This matches the input subtyping rule [κ-Ins] in Theorem 7.4.
Example 3.1 (!, ? and selection/branching). Consider the following classes:

```scala
sealed abstract class AorB

case class A() extends AorB;
case class B() extends AorB
```

Let `c` be an instance of `Out[AorB]`. The `c.!` method can be used as follows:

```scala
1  c ! A() 
2  or 
3  c ! B() 
```

Note that `!` resembles the output/selection operator seen in Fig. 4 (left). Moreover, the Scala compiler ensures that the argument of `!` belongs to a subtype of `AorB`, – e.g., `A` or `B`⁴: this corresponds, in session-π, to the type checking of an internal choice.

Let now `c` be an instance of `In[AorB]`. The `c.?` method can be used as shown below,

```scala
1  c ? { 
2    case A() => println("Got A") 
3    case B() => println("Got B") } 
```

where the `{...}` block, as per usual Scala syntax, is a function from `AorB` to `Unit`. This reminds the branching operator seen in Fig. 4 (left). Moreover, since `AorB` is a sealed abstract class, the Scala compiler can check exhaustiveness, warning if the `cases` do not cover both `A` and `B`⁵: this corresponds, in session-π, to the type checking of an external choice.

Using lchannels endpoints: static vs. dynamic checks. As seen in Theorem 3.1, the Scala compiler can check that an instance of `lchannels Out` (resp. `In`) carrying a sealed abstract class is only used under the safety and exhaustiveness guarantees of a session-typed channel endpoint with a top-level ⊕ (resp. &)⁶, i.e., `S1` and `S2` in §2.1. Also, an instance of e.g. `Unit` provides the guarantees of an end-typed channel endpoint: it cannot be used for I/O. Unfortunately, the Scala type checker cannot enforce input/output linearity (`S3` and `S4` in §2.1); hence, lchannels implements the following runtime linear usage rules:

L1. each `Out` instance should be used to perform exactly one output. Any further output will generate a runtime exception, forbidding duplicated message transmissions;

L2. each `In` instance should be used at least once. Each use will retrieve the same value.

L1 and L2 reflect the typical usage of Scala’s Promises and Futures. The lack of static linearity checks impacts deadlock-freedom guarantees: we will discuss this topic in §6.1.3. Note that L1 matches S3, while L2 is more relaxed than S4. The latter is not a technical necessity, since In could be easily designed to raise an exception if used twice for input; we adhere to the familiar behaviour of Futures for simplicity of presentation, and to readily apply some common programming patterns, e.g. registering one or more input callbacks.

3.2 A local implementation

Fig. 6 (right) shows a simple local implementation of `In[A]/Out[A]`, as a thin layer over a `Promise[A]/Future[A]` pair (created in lines 12–14): a value written in the former becomes available on the latter. The `A`-cast in line 5 (due to the invariance of `Promise[A]`) is safe: the type bound on `B` ensures that `Promise[B]` can only be written with a subtype of `A`.

Example 3.2 (Spawning interacting threads). Two threads that communicate through a local (linear) channel can be created with a method similar to the following:

---

⁴ Due to Java legacy, in Scala also `null` is a subtype of `AorB`. This will be explicit in §7.3.

⁵ By design, Scala does not enforce matching on `null` values, albeit they might be received (see note 4).

⁶ This arises from the encoding of session types into linear I/O types with variants [8]; we render the latter in Scala as sealed case classes (as detailed in §7.3).
```scala
case class Q(p: Boolean, cont: Out[R])
case class R(p: Int)
def f(c: In[Q]) = {
c ? { q => q.cont ! R(42) }
def g(c: Out[Q]) = {
val (ri, ro) = c.create[R]()
c ! Q(true, ro)
ri ? { r => println(f"Got ${r.p}"")
}
```

![Figure 7](image)

*Figure 7* $S_{QR}$ and $\overline{S_{QR}}$ in Scala.

```scala
val (in, out) = LocalChannel.factory[A](); ( Future { p1(in) }, Future { p2(out) })
}
```

Here, p1 and p2 are functions taking respectively an input and output channel endpoint carrying A, and returning resp. B1 and B2. The `parallel` method creates a pair of A-carrying local channel endpoints (line 2), applies p1 and p2 on them by spawning separate threads, and returns a pair of Futures that will be completed with their return value (line 3).

Actually, `parallel` is a method of the `LocalChannel` object in Fig. 6. Most of the examples in the rest of the paper feature two endpoint functions with the signature of p1 and p2, and they can be executed concurrently (and type-safely) via `LocalChannel.parallel`.

Our local implementation of `lchannels` is suitable for type-safe inter-thread communication, as suggested in Theorem 3.2. However, `Promise/Future` instances cannot be serialised, and thus cannot be sent/received over a network: this makes `LocalIn` and `LocalOut` unsuitable for distributed applications. We address this issue later on, in §5.

## 4 Session programming with lchannels and CPS protocols

We now address Problem P2 in §2.3: given a session type $S$, how to instantiate the type parameters of `In[.]`/`Out[.]`, to represent the (possibly recursive) sequencing of internal/external choices of $S$. The answer lies in representing the states of $S$ as CPS protocol classes, as outlined in §2.4. We give an example-driven intuition of such a representation, and the resulting session-based software development approach (§4.2). The formalisation is in §7.

### 4.1 Representing sequential inputs/outputs

Let us consider the session type $S_{QR} = ?Q(Bool).!R(Delete)$, dictating that a channel endpoint must be used first to receive $Q(Bool)$, and then to output $R(Delete)$. In Scala, we could define the two case classes on the right (where the field p stands for “payload”), and we can instantiate a linear input endpoint of type `In[Q]`, which allows to perform the first input of $S_{QR}$; but, how do we require to send a value of type $R$ along the same interaction?

Inspired by the encoding of session types into linear types [8], we can instead define the case classes in Fig. 7 (lines 1–2), where `cont` stands for “continuation” (and recalls `replyTo` in §1). Now, the value received from `In[Q]` also carries an `Out[R]` endpoint for continuing the interaction; the value received from `In[R]`, instead, does not have a `cont` field, since the
protocol ends there. In lines 4–6, \( c \) uses \( c \) to receive a \( Q \)-typed value \( q \) (line 5) then uses \( q \cdot \text{cont} \) to send a value of type \( R \).

Now, consider the dual \( S_{QR} = \pi Q(\text{Bool}) :: R(\text{Int}) \): we can represent it in Scala simply by reusing \( Q \) and \( R \) in Fig. 7, and instantiating a linear output endpoint \( \text{Out}[Q] \). Its usage is shown in lines 8–13. To produce a value of type \( Q \), \( g \) must also produce a channel endpoint \( \text{Out}[R] \): for this reason, the two continuation endpoints \( \text{ri}, \text{ro} \) are created (line 9), respectively with types \( \text{In}[R], \text{Out}[R] \). On line 10, \( c \) is used to send a \( Q \)-typed value, carrying \( \text{ro} \): the recipient is expected to use it for continuing the interaction; on line 11, \( \text{ri} \) is used to receive the value \( r \) (of type \( R \)) sent on \( \text{ro} \).

4.2 A development approach for session-based applications

In our last example, \( Q \) and \( R \) are the CPSP classes of both \( S_{QR} \) and \( S_{QR} \). \( \text{In}[Q] \) is the Scala representation of \( S_{QR} \), while \( \text{Out}[Q] \) is the representation of \( S_{QR} \). We can outline a development approach for session-based applications. For each communication channel:

D1. formalise the two endpoint session types \( S \) and \( S \) (assuming they are not trivially end);
D2. extract the CPSP classes of \( S \) (or, equivalently, of \( S \)). Roughly, it means:
   a. convert each internal/external choice into a set of case classes (one per label);
   b. when a choice has multiple labels, let each case class above extend a common sealed abstract class, representing the multiple choice itself;
   c. recover the sequencing in \( S \) (and \( S \)) by “connecting” each case class to its “successor” (if any), through the \( \text{cont} \) field;
D3. let \( C \) be the class representing the outermost internal/external choice of \( S \):
   a. if \( S \) starts with an internal choice, its Scala endpoint type is \( \text{Out}[C] \). Dually, since \( S \) starts with an external choice, the Scala type at the other endpoint is \( \text{In}[C] \);
   b. otherwise, if \( S \) starts with an external choice, its Scala endpoint type is \( \text{In}[C] \). Dually, since \( S \) starts with an internal choice, the Scala type at the other endpoint is \( \text{Out}[C] \).

The extraction of protocol classes must deal with some subtleties, in particular for determining whether \( \text{cont} \) should be an \( \text{In}[] \) or \( \text{Out}[] \) endpoint, and for representing recursion. We will formally address these issues in §7.3; now, we proceed with more examples.

4.3 Interlude: automating channel creation

Before proceeding, we take a quick detour to address Problem P3 of §2.3. In Fig. 7 (line 9), we can notice a case of manual creation of channel endpoints, as in Fig. 4 (right, line 3). This is a key pattern for “CPS interactions”: when sending a message that does not conclude a session, it is necessary to create a pair of channels, send one of them, and use the other to continue interacting\(^7\). This “create-send-continue” pattern ensures session progress, but is an error-prone burden for the programmer; so, we automate it by extending \( \text{Out} \) (Fig. 6, left) with the method \( !! \) above.

Take \( c \) of type \( \text{Out}[Q] \) from Fig. 7 (lines 8–13), and let \( h \) be a function from \( \text{Out}[R] \) to \( Q \): \( c \cdot h \) creates a pair of channel endpoints \( (\text{cin}, \text{cout}) \) of type \( \text{In}[R], \text{Out}[R] \) (line 3 above),

\(^7\) The pattern actually reflects how session-\( \pi \) processes are encoded in standard \( \pi \)-calculus (§2.3).
applies \( h \) to \( cout \), sends the result via \( c \) (line 4), and returns \( cin \) for continuing the session (the other case of !! is “dual”, when \( h \)’s domain is \( \text{In}[R] \)). By letting \( h \) be an instance of \( Q \) with a hole in place of \( \text{cont} \), we can remove line 9 of Fig. 7, and rewrite line 10 as:

\[
\text{val ri = c !! Q(true,(_:\text{Out}[R]))},
\]

where the type annotation is necessary due to the limited type inference capabilities of Scala\(^8\).

We can address this last inconvenience by defining \( Q \) as a curried case class, and placing the hole in the curried \( \text{cont} \) field: the Scala compiler can now infer its type. The resulting code is shown on the right (with \( f \) unchanged w.r.t. Fig. 7). We will adopt this style for the rest of the paper.

### 4.4 Examples

We now discuss some examples of the session-based approach outlined in §4.2. We proceed by increasing complexity, showing how to instantiate CPSP classes to represent recursion (Theorem 4.1), non-singleton external/internal choices (Theorem 4.2), and multiple channels with higher-order types for session delegation (Theorem 4.3).

▶ **Example 4.1 (FIFO).** An unidirectional FIFO channel, with endpoints for sending/receiving values of type \( T \), can be represented with the following recursive session types:

\[
S_{\text{fifo}} = \mu X. !\text{Datum}(T).X \quad \overline{S_{\text{fifo}}} = \mu X. ?\text{Datum}(T).X
\]

The corresponding CPSP classes consist in just one (parametric) declaration:

```scala
case class Datum[T](p: T)(val cont: In[Datum[T]])
```

i.e., we represent the recursion on \( X \) by (i) taking the name of the class corresponding to the outermost internal/external choice under \( \mu X \ldots \) (i.e., \( \text{Datum} \)), and (ii) continuing with such a name when \( X \) occurs (for another case of recursion, see Theorem 4.2). Note that \( \text{cont} \) is an input endpoint, used by the recipient to receive a further value, while the sender keeps the output endpoint to produce a value. The endpoint processes can be written as:

```scala
def sender(fifo: Out[Datum[Int]]): Unit = {
  val cont = fifo !! Datum(1) !! Datum(2) !! cont
}
def receiver(fifo: In[Datum[Int]]): Unit = {
  val v = fifo.receive
  println(f"Got \$\{v.p\}"); receiver(v.cont)
}
```

Here, \( \text{sender} \) performs two outputs in a row (line 2): this is allowed since each application of !! returns a channel of type \( \text{Out}([\text{Datum}[T]]) \) (cf. declaration of \( \text{Datum}[T] \) above).

▶ **Example 4.2 (Greeting protocol).** Consider the “greeting” types \( S_h \) and \( \overline{S_h} \) from §2. Unlike Theorem 4.1, we now have non-singleton internal/external choices. To extract their CPSP classes, we apply item D2b of §4.2: add a sealed abstract class for each internal/external choice, extending it with one case class per label. In this case, we add:

- **Start** for the internal choice of \( S_h \) (i.e., the external choice of \( \overline{S_h} \)) between Greet, Quit;
- **Greeting** for the external choice of \( S_h \) (i.e., the internal choice of \( S_h \)) between Hello, Bye.

\(^8\) This limitation is present in Scala 2.11.8, but might be overcome in future versions.
We obtain the CPSP classes on the right, with $Out[\text{Start}] / In[\text{Start}]$ representing $S_0 / S_0$ (by D3). We can write two endpoint processes as:

``` scala
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
```

Note that `client` is similar to the pseudo code of `hello` in Fig. 4 (left).

> Example 4.3 (Sleeping barber with session delegation). We address a classical problem in concurrency theory [10]: a barber waits for customers in his shop, sleeping when there is nobody to serve. When a customer enters in the shop, he goes through a waiting room with $n$ chairs: if all chairs are taken, he leaves; otherwise, he sits. If the barber is sleeping, he wakes up, serves all sitting customers (one a time), and sleeps again when nobody is waiting. We model this scenario with three components: the customer, the shop and the barber, using session types to formalise their expected interactions, schematised below.

In this example, we show how multiple concurrent sessions (one per customer) can be handled by single-threaded programs (shop and barber). We also show how to exploit session delegation by leveraging higher-order session types (i.e., channel endpoints that send/receive other channel endpoints). When a customer enters in the shop, he gets a $S_\text{cstm}$-typed channel endpoint:

$S_\text{cstm} = \mu X. \text{Full} \& \text{Seat} \Rightarrow X \cdot \text{Ready} \cdot S_\text{cut}$

$S_\text{cut} = \nu Y. \text{Descr}(Y) \cdot \text{Haircut} \cdot \text{Pay}(\text{Int})$

He might receive either a `Full` message (when no seats are available), or a `Seat`: in the first case, the session ends; in the second case, he waits for the barber to be `Ready`. Then, he continues with `S_cut`: describes the new hairdo, waits for the `Haircut`, `Pays` and leaves. The shop uses the other, dually-typed channel endpoint:

$S_\text{cut} = \nu Y. \text{Descr}(Y) \cdot \text{Haircut} \cdot \text{Pay}(\text{Int})$

and keeps track of the $n$ seats to choose whether to send `Full` or `Seat`. When the customer gets a `Seat`, the shop interacts with the barber, through a channel with endpoint types:

$S_\text{barber} = \mu X. \text{Available} \cdot \text{Serve}(\text{Seat}) \cdot X$ (barber endp.)

$S_\text{barber} = \mu X. \text{Available} \cdot \text{Serve}(\text{Seat}) \cdot X$ (shop endp.)

i.e., the shop recursively waits for the barber to be `Available`; when it happens, it picks a sitting customer (i.e., one that has received a `Seat`), sends a `Ready` message to him, and forwards the channel endpoint (now $S_\text{cut}$-typed) to the barber, as the payload of `Serve`.

Meanwhile, the barber uses its $S_\text{barber}$-typed channel endpoint to notify that he is `Available`, and wait for a `Serve` message – sleeping until he gets one; when it happens, the barber gets a $S_\text{cut}$-typed channel endpoint in the message payload: he is expected to use it for interacting with the customer, i.e., listen for the hairdo Description, perform the Haircut, and take the Payment. When the customer session terminates, the barber must resume his recursive session with the shop: he notifies that he is `Available` again, etc.
The CPSP classes extracted from the session types above are shown on the right. As per item D2b of §4.2, we introduce WaitingRoom as the sealed abstract class corresponding to the external (resp. internal) choice between Full and Seat in $S_{cstm}$ (resp. $S_{cstm}$).

### Implementation

The code of the shop, barber and customer is shown in Fig. 8. They are supposed to run as concurrent threads, and thus implement the Runnable interface.

Shop is parametric in the number of seats. It collects the channel endpoints of the waiting customers in its private seats field, which may be any FIFO-like container with a blocking read method: we could use e.g. scala.concurrent.Channel[Out[Ready]], or a FIFO based on Theorem 4.1. Once started, Shop creates a $S_{barber}$-typed channel (line 19) and gives the output endpoint to a new Barber (line 20). The enter method returns an input endpoint for interacting according to $S_{cstm}$: after creating two channel endpoints of the suitable type (line 6), enter checks how many people are trying to get a seat, and outputs Full (line 10) or Seat (line 12) before returning the input endpoint (line 15). In the main loop (lines 24–33), the shop waits for an Available message from the barber (line 25), sleeps while retrieving a customer channel from seats (line 26), notifies the customer that the barber is Ready, forwards the channel to the barber, and continues its loop.

Barber, in line 7, notifies the shop that he is Available, and uses the channel endpoint returned by !! (whose type is In[Serve]) to wait for a Serve message. Then, he interacts with the customer using the In[Descr]-typed endpoint received as payload (lines 8–11); after being paid, he continues the session with the shop (line 11).

The code for Customer is simple: he invokes the enter method of the Shop given as parameter (line 3), and uses the returned channel to interact according to $S_{cstm}$. If the waiting room is Full, he retries later (lines 5–7). To model multiple customers competing for the seats, it is sufficient to start multiple Customers referring to the same Shop.

As anticipated, our solution for the sleeping barber problem exploits session delegation: the customer starts interacting with the shop, but his session is eventually forwarded to the barber, with a higher-order $\text{Serve}(S_{cstm})$ message. Delegation is transparent: no dedicated code is required in Customer’s implementation. Moreover, delegation is safe: e.g., the Scala type checker ensures that only Out[Ready]-typed channel endpoints are stored in Shop.seats, and that the barber picks up the session only after the shops sends Ready.

### Optimisations, transport abstraction and error handling

In this section, we demonstrate how lchannels allows to abstract from the underlying message transport medium, and to handle communication errors. In §3, we introduced the abstract classes In/Out, and LocalIn/LocalOut as simple local implementations for inter-thread communication. The In[]/Out[] interface can abstract other message transports, allowing lchannels-based programs to achieve faster message delivery, or transparently interact across a network. We discuss 3 examples: queue-, actor- and stream-based channels.

#### Optimised queue-based channels.

The simple LocalIn/LocalOut classes in Fig. 6 (right) perform all communications through the underlying Future/Promise. However, many
applications could mostly use the `In.receive`/`Out.send` methods, and could benefit from an optimised implementation of `In/Out` that (when possible) bypasses `In.future`/`Out.promise`. We developed this idea with the `QueueIn/QueueOut` classes: internally, they deliver messages through Java `LinkedTransferQueues` (under the runtime linearity constraints L1/L2 of §3.1) – and only allocate and use a `Future/Promise` when the `.future/.promise` methods are explicitly invoked. Moreover, we optimised the `QueueOut.!!` method to reuse queues when continuing a session. The resulting performance improvements are shown in §6.2.

**Network-transparent actor-based channels.** We implemented proof-of-concept network-transparent subclasses of `In/Out`, called `ActorIn/ActorOut`: they deliver messages by automatically spawning Akka Typed actors [29], which in turn can communicate over a network.

Using such actor-based channels, a local process can interact with a remote one through a local actor-based endpoint that proxies a remote endpoint. E.g., to obtain a remote interaction between two instances `server` and `client` (Theorem 4.2) we can run the former as:

```
val (in, out) = ActorChannel.factory[Start]("start");   server(in)
```

Now, `out.path` contains the Akka Actor Path [27] of an automatically-generated actor. Such a path can be used, even on a different JVM, to instantiate a proxy for `out`, as follows:

```
val c = ActorOut[Start]("akka.tcp://sys@host.com:5678/user/start");   client(c)
```

where `ActorOut`’s argument matches `out.path`. Then, the `client` and `server` will interact over a network, without changing their code.

All the examples in this paper can also run on `ActorChannels`, simply by replacing the calls to `LocalChannel.factory[A]()` with `ActorChannel.factory[A]()` (e.g. in Fig. 8, Shop, line 6). To achieve complete transport-independence, `factory` can be parameterised.

We choose Akka as a message transport medium due to its widespread availability, using Akka Typed to obtain stronger static typing guarantees throughout the implementation. The main challenges were related to making `ActorIn/ActorOut` instances `serializable`: this
is a crucial requirement, as channel endpoints might appear (as payloads or continuations) in messages sent/received over a network. In particular, sending an \texttt{ActorOut[A]} roughly corresponds to sending an \texttt{ActorRef[A]} instance (which is serializable out-of-the-box) – but sending an \texttt{ActorIn[A]} has no Akka equivalent, and requires some internal machinery.

**Network-transparent stream-based channels.** Often, programs interacting over a network are implemented with different languages, and use bare TCP/IP sockets without a common higher-level networking framework. Still, such programs might need to observe complicated protocols (e.g. RFC-based ones like POP3, SMTP, etc.) that can be abstractly represented as session types [12, 21]. To address this scenario, we extended l\texttt{Channels} with channel endpoints that send/receive messages through Java \texttt{InputStream/OutputStream}s, obtained e.g. from a network socket. The main classes are \texttt{StreamIn/StreamOut} (extending resp. \texttt{In/Out}), and can only be instantiated by providing a protocol-specific \texttt{StreamManager} which can serialize/deserialize messages to/from a stream of bytes (tracking the session status if needed).

<table>
<thead>
<tr>
<th>Message</th>
<th>Text format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greet('Alice')</td>
<td>GREET Alice</td>
</tr>
<tr>
<td>Hello('Alice')</td>
<td>HELLO Alice</td>
</tr>
<tr>
<td>Bye('Alice')</td>
<td>BYE Alice</td>
</tr>
<tr>
<td>Quit()</td>
<td>QUIT</td>
</tr>
</tbody>
</table>

For example, suppose that the “greeting protocol” from Theorem 4.2 abstracts a textual protocol as shown on the left, and we want our client to interact with a third-party server using that textual format over TCP/IP sockets. We first need to derive the \texttt{StreamManager} class, implementing a \texttt{HelloStreamManager} that suitably serializes/deserializes the textual messages. Then, we can let our client talk with a remote server, via TCP/IP, using the textual format:

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

Note that we did not change the code of \texttt{client} seen in Theorem 4.2: we leverage l\texttt{Channels} and protocol classes to represent and type-check the high-level protocol structure (sequencing, choices, recursion), while separating the low-level details from the logic of the program.

**Error handling.** The methods of \texttt{In[A]} seen in Fig. 6 do not handle errors; e.g., \texttt{receive} throws an exception if no message arrives within the (implicit) \texttt{Duration} \texttt{d}. However, input errors are quite common in real-world applications: e.g., the process at the other endpoint might not timely send a message, or may send a wrong message that a \texttt{StreamManager} cannot deserialize, or a network problem may occur. As typical for Scala APIs, we extended \texttt{In[A]} to capture failures as \texttt{Try[A]} values, via 2 additional methods: \texttt{tryReceive} and \texttt{??}.

```
c ?? { case Success(m) => _match {
  case A() => println("Got A")
  case B() => println("Got B")
  case Failure(e) => /* Inspect e */
}
```

E.g., the branching on \texttt{AorB} in Theorem 3.1 can be made error-resilient by using \texttt{c.??}, as shown on the left: the top-level matching is now on \texttt{Try[AorB]}.

### 6 Evaluation

We now assess the practicality of the approach in §4.2 with a case study based the “client with frontend” in Fig. 1 (§6.1), and a performance evaluation of l\texttt{Channels} (§6.2).

#### 6.1 A case study: application server with frontend

This section shows how our approach can address the “server with frontend” scenario in §1. We consider an application server that is a \textit{chat server} allowing users to join/leave chat rooms, and send/receive messages to/from them. We formalise the protocols of the
application (§6.1.1), and illustrate some characteristics of the implementation (§6.1.2), and discuss how development was aided by CPS protocols and lchannels (§6.1.3).

6.1.1 The protocols

We formalise the protocols in Fig. 1 as session types, dividing them in two groups: public (used by clients), and internal (used for frontend/auth/chat server interaction).

**Public protocols.** The session type $S_{\text{front}}$ formalises the usage of the channel endpoint that the frontend handles while interacting with a client. It is defined as follows:

$$S_{\text{front}} = {?}\text{getSession}(l d). (\{\text{New}(S_{\text{auth}}) @ !\text{Active}(S_{\text{act}})\}) \quad S_{\text{auth}} = !\text{Authenticate}(\text{Cred}). ( {?}\text{Success}(S_{\text{act}}) & {?}\text{Failure})$$

$$S_{\text{act}} = \mu X. \{ \text{Quit} @ !\text{getId}(l d). X @ !\text{Ping}. Y @ !\text{Join}(\text{String}). ?\text{ChatRoom}((S_{\text{act}}, S_{\text{ctl}})). Y \}$$

The service implementing $S_{\text{front}}$ waits for a $\text{getSession}(l d)$ request from a client; then, with an internal choice @, it might answer by sending either $\text{New}(S_{\text{auth}})$ or $\text{Active}(S_{\text{act}})$:

- **New** carries a $S_{\text{auth}}$-typed channel endpoint, talking with the auth server; it allows the client to send an $\text{Authenticate}(\text{Cred})$ message (with Cred being the credentials), and wait for either $\text{Success}(S_{\text{act}})$ or $\text{Failure}$ (the $S_{\text{act}}$-typed channel is explained below);
- **Active** carries an $S_{\text{act}}$-typed channel endpoint representing the active “session loop” (Fig. 1). When the client receives it, $S_{\text{act}}$ (which is recursive) allows to choose among:
  - **Quit.** In this case, the chat session ends;
  - **GetId.** Then, the client receives an $\text{Id}(l d)$ answer whose payload is the current session identifier, and continues the session recursively;
  - **Ping(String).** Then, the client receives a $\text{Pong(String)}$, and continues recursively;
  - **Join(String),** with the payload being a chat room name. Then, the client joins a chat room, gets a $\text{ChatRoom}((S_{\text{act}}, S_{\text{ctl}}))$ answer, and the session continues recursively. The two channels endpoints in the payload allow to interact with the chat room:
    - $S_{\text{act}} = \mu Y. {?}\text{NewMessage}((\text{String}, \text{String})). Y & {?}\text{Quit}$. This recursive endpoint allows the client to receive either a $\text{NewMessage}$ from the chat room (with the payload being the sender username and the message text), or $\text{Quit}$ (ending the interaction);
    - $S_{\text{ctl}} = \mu Z. !\text{SendMessage}(\text{String}). Z @ !\text{Quit}$. This endpoint allows the client to send either a message on the chat room (with the payload being the text), or $\text{Quit}$.

The CPS protocol classes of the session types above are extracted as in the examples of §4.4, and are almost identical to Fig. 2. In particular, we use Command as the sealed abstract class for the top-level choice in $S_{\text{act}}$ (this detail will be mentioned again in §6.1.2).

**Internal protocols.** Fig. 1 also outlines the internal communications among the frontend, authentication and chat server: they can be formalised as session types, too – as for barbershop interaction in Theorem 4.3. Here, we only detail the frontend-server interaction type:

$$S_{\text{PS}} = \mu X. \text{getSession}(l d). (?\text{Success}(S_{\text{act}})). X & ?\text{Failure}. X)$$

The frontend recursively queries for active sessions (passing the identifier received from a client), getting either Success or Failure. In the first case, the message payload is a $S_{\text{act}}$-typed channel endpoint, that will be forwarded to the client with an Active message.
6.1.2 The implementation

This case study uses higher-order session types to naturally model the “handles” mentioned in §1. A difference w.r.t. Theorem 4.3 is that the delegation appears explicitly in client’s session types, e.g. in Active messages with a channel as payload. In CPS protocols, this difference is almost negligible: the Active message class has no continuation, but the client should keep interacting via the Out endpoint in the payload – as per rule L1 in §3.1.

The server-side implementation reuses several solutions from Theorem 4.3 – e.g., internal FIFOs for storing and later processing requests: this happens e.g. when the single-threaded chat server manages multiple client sessions. The main difference w.r.t. Theorem 4.3 is that requests are queued asynchronously (via in.future) and enriched with internal data.

```java
class ChatServer(...) extends Runnable {
  ... 
  private def createSession(username: String): Out[Command] = {
    val id = allocUniqueSessionId()
    val (in, out) = LocalChannel.factory[Command]()
    in.future.onComplete {
      // Using scala.util.{Success, Failure}
      case Success(cmd) => queueRequest(Success((id, cmd)))
      case Failure(e) => queueRequest(Failure(e))
    }
    // Add the new session to the list of known sessions
    sessions(id) = ... /* session info, including username */
    out
  }
  // E.g., the chat server calls the method on the left when the auth server asks to create a new session for username: it reserves a session id (line 4), creates the channel endpoints in, out carrying a Command (line 5), keeps in, and returns out (line 12), that will be the payload of a NewSession message. The client Command is received asynchronously via in.future: in lines 6–9, cmd is paired with the session id, and queued (line 7). When the pair is later dequeued and processed, id tells on which session cmd is acting. A similar queuing is performed as the session progresses; e.g., when a cmd of type Ping is dequeued, the server runs:
    val in2 = cmd.cont !! Pong(cmd.msg) // cmd’s type: Ping; in2’s type: In[Command]
  }

  // and in2.future is used for queuing the next client command, like in.future in lines 6–9.
}
```

6.1.3 Lessons learned

As expected, CPS protocols and lchannels allow the Scala type checker to detect protocol errors that usually arise on untyped channels, e.g., trying to send the wrong type of message, or forgetting to consider some cases when branching with In.? This greatly simplified the present case study, where multiple channels with various protocols are handled concurrently. Since we leverage the existing Scala type system, modern Scala IDEs (such as [30]) provide channel usage errors and hints, e.g. via typing information and auto-completion suggestions.

However, as seen in §3.1, Scala and lchannels cannot perform static linearity checks: hence, they cannot spot two kinds of errors, illustrated below, that impact session progress.

**Double usages of output endpoints.** They occur when an Out[A] instance is used twice to send A-typed values: then, by L1 in §3.1, an exception is thrown, and the extra message is not sent. This kind of error never occurred in our experience: the CPS interaction guided by lchannels seems to naturally shape programs where output endpoints are discarded after used. Moreover, as for Scala Promises, double outputs causes an immediate runtime error, that (we believe) should usually arise in proximity of the code requiring a fix.

**Unused channel endpoints.** Not performing an output can leave a process at the other endpoint stuck, waiting for input – and this could escalate to other processes waiting on other channels; this problem can also arise if a program does not input a message whose continuation/payload is an output channel. Spotting this kind of errors can be tricky, especially if channels are dynamically generated, sent, received, stored in collections (as in our case study). lchannels mitigates this issue via timeouts on the receiving side.
they allow to see which channel is stuck in which state – and thus, which process is not producing an output. In our case study, a few issues of this kind were easily fixed.

6.2 Benchmarks

We implemented several micro-benchmarks to evaluate how \texttt{lchannels} impacts communication speed w.r.t. other inter-thread communication methods: Fig. 9 shows the results. The benchmarks are mainly inspired by [24]; “Streaming” is a parallel blend of “Ring”+ “Counting actor”: 16 threads are connected in a ring and a sequence (“stream”) of messages is sent at once, measuring the time required for all to complete one loop.

We wrote an implementation of each benchmark using \texttt{Out.send/In.receive} for inter-thread communication, instantiating them with \texttt{LocalChannels}, \texttt{QueueChannels} and \texttt{ActorChannels} (columns 1, 5, 7). As a comparison, we adapted such implementations to interact via \texttt{Promises/Futures} (column 2), and also to interact “non-CPS” via \texttt{scala.concurrent.Channels}, and Java \texttt{ArrayBlockingQueues / LinkedTransferQueues} (columns 3, 4, 6).

The overhead of \texttt{lchannels} w.r.t. “non-CPS” queue-based interaction has two origins:

1. runtime linearity checks, i.e. inspecting/setting a flag when a channel endpoint is used;
2. repeated creation of \texttt{In/Out continuation pairs} (§4.3): in comparison, our “non-CPS” benchmarks create Scala channels / Java queues just once at the beginning of each session.

Hardware/JVM settings highly influence the measurements: queues or \texttt{Promises/Futures} can become relatively faster/slower, or show more/less variance, depending on the benchmark. Still, the results tend to be consistent with Fig. 9. It can be seen that \texttt{LocalChannels} add a small slowdown to the underlying \texttt{Promises/Futures}. \texttt{QueueChannels} are considerably faster, except when many short-lived sessions are rapidly created (this scenario is stressed by “Chameneos”, against the optimisations seen in §5); still, \texttt{QueueChannels} add a perceivable overhead on the underlying \texttt{LinkedTransferQueues}. \texttt{ActorChannels} are slower, especially with many threads and low parallelism (as in “Ring”): it is due to the (currently unoptimised) internal machinery that makes \texttt{ActorChannels} network-transparent, and more suitable for distributed settings where network latency can make the slowdown less relevant.

Notably, the usual “non-CPS” communication we implemented (and measured) over Scala channels / Java queues requires connecting pairs of threads $P_1,P_2$ with pairs of queues (one carrying messages from $P_1$ to $P_2$, the other from $P_2$ to $P_1$). Such queues have type \texttt{Queue[A]}, where $A$ must cover all the message types that could be sent/received: for protocols with sequencing and branching, this leads to loose static type checks, that combined with the lack of runtime monitoring, increase the risk of protocol violations errors.

7 A formal foundation

We now explain the formal foundations of our approach (as outlined in §4.2), by detailing how to extract CPSP classes from session types, and studying how Scala’s type system handles session subtyping/duality. We summarise session subtyping (§7.1), and we introduce our encoding from session to linear types (§7.2), and then into Scala types (§7.3).

7.1 Session types and subtyping

We defined session types and duality in §2; to ease the treatment, we adopt 2 restrictions.

\begin{remark}
\textbf{(Syntactic restrictions).} For all $S$, (i) each label is unique, and also a valid Scala class name, and (ii) each $\mu$ binds a distinct variable that actually occurs in its scope.
\end{remark}
Figure 9 Benchmark results (box & whisker plot): 30 runs × 10 JVM invocations, Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB RAM, Ubuntu 14.04, Oracle JDK 64-bit 8u72, Scala 2.11.7, Akka 2.4.2.

Restriction (i) allows to directly generate a Scala `case class` from each internal/external choice label. Restriction (ii) is a form of Ottmann/Barendregt’s variable convention [4].

The session subtyping relation $\leq$ allows to safely replace a $S'$-typed channel endpoint with an $S$-typed one, provided that $S \subseteq S'$ holds. The relation is defined as follows.

**Definition 7.2 (Session subtyping [13]).** The subtyping relation between session types is coinductively defined by the following rules (where $\leq_B$ is a subtyping between basic types):

$$
\begin{align*}
\forall i \in I: &\quad T_i \leq T'_i \quad S_i \leq S'_i & [\text{<Ext}] \\
\& \end{align*}
$$

$$
\begin{align*}
\forall i \in I: &\quad T'_i \leq T_i \quad S'_i \leq S_i & [\text{<Int}] \\
\& \end{align*}
$$

$$
\begin{align*}
\text{end} \leq \text{end} [\text{<End}] &\quad S(\mu X.S/X) \leq S' \\
\mu X.S \leq S' & [\text{<muL}] \\
V \leq S' (\mu X.S/X) & [\text{<muR}] \\
T \leq_B T & [\text{<3}]
\end{align*}
$$

Rule [<Ext>] says that an external choice $S$ is smaller than another external choice $S'$ iff $S$ offers a subset of the labels, and for all common labels, the payload and continuation types are in the relation. The rationale is that a program which correctly uses an $S'$-typed channel endpoint supports all its inputs – hence, the program also supports the more restricted inputs of an $S$-typed endpoint. Dually, [<Int>] says that an internal choice $S$ is smaller than another internal choice $S'$ iff $S$ offers a superset of the labels, and for all common labels, the payload and continuation types are in the relation. The rationale is that a program which correctly uses an $S'$-typed channel endpoint might only perform one of the allowed outputs, that is also allowed by the more liberal $S$-typed endpoint. [<End>] says that a terminated session has no subtypes. [<muL> and [<muR>] are standard. [<3>] extends $\leq$ to basic types.

### 7.2 Linear I/O types (with records and variants)

In order to encode session types into Scala types, we exploit an intermediate encoding into linear types for input and output [30]. We focus on a subset of such types, defined below.

**Definition 7.3.** Let $\mathbb{B}$ be a set of basic types ($\S 2$). A linear type $L$ is defined as:

$$
L ::= ?(U) \mid !(U) \mid \text{•} \quad U ::= \{l_i \{p : V_i, c : L_j\}\}_{i\in I} \mid \mu X.U \mid X \quad V ::= \mathbb{B} \mid L (\text{closed})
$$

where (i) recursion is guarded, and (ii) all $l_i$ range over pairwise distinct labels. We also define the carried type of $L$ as $\text{carr}(?(U)) = \text{carr}(!(U)) = U$. 

\(?U\) (resp. \(!U\)) is the type of a linear channel endpoint that must be used to input (resp. output) one value of type \(U\); \(\bullet\) denotes an endpoint that cannot be used for I/O. \(U\) is a (possibly recursive) variant type where each \(1_l\)-labelled element is a record with 2 fields: \(p\) (mapped to a basic value or a linear channel endpoint) and \(c\) (mapped to a linear endpoint).

**Definition 7.4 ([36]).** The subtyping relation \(\leq\) between linear types is inductively defined by the following rules (where \(\leq\) is a subtyping between basic types):

\[
\begin{align*}
U & \leq U' & \text{[Lin]} \\
U' & \leq U & \text{[Lin]}
\end{align*}
\]

\[
\begin{align*}
V & \leq V' & \text{[Lin]} \\
V' & \leq V & \text{[Lin]}
\end{align*}
\]

\[
\begin{align*}
U & \leq U' (x: I) & \text{[Lin-Get]} \\
U' & \leq U & \text{[Lin-Put]}
\end{align*}
\]

\[
\begin{align*}
U & \leq U' \{x: I\} & \text{[Lin-Get]} \\
U' & \leq U & \text{[Lin-Put]}
\end{align*}
\]

The rules in Theorem 7.4 are standard: they include the subtyping for variants and records (rule \([\text{Lin-VR}]\)) and left/right recursion (\([\text{Lin-UR}]\)). \([\text{Lin-Get}]\) and \([\text{Lin-Put}]\) provide respectively the subtyping for linear inputs (covariant w.r.t. the subtyping of carried types) and outputs (which is instead contravariant): note that they are matched by the variances of \(\text{In}\) / \(\text{Out}\) (Fig. 6, left). By \([\text{Lin-VR}]\), \(\bullet\) is the only subtype of itself. \([\text{Lin-Get}]\) extends \(\leq\) to basic types.

In the linear types world, the duality between two channel endpoints is very simple: it holds when they are both \(\bullet\), or they are an input and an output carrying the same type.

**Definition 7.5 ([8]).** The dual of \(L\) (written \(\overline{L}\)) is: \(\overline{U} = !\!(U)\); \(\overline{!\!(U)} = ?(U)\); \(\bullet = \bullet\).

We now introduce our encoding of session types into linear types. Albeit inspired by [8, 6], it features a different treatment of recursion, allowing us to bridge into Scala types.

**Definition 7.6** (Encoding of session into linear types). Let the action of a session type be:

\[
\text{act}(\{T\},S) = ? \quad \text{act}(\{T\},S) = !
\]

Moreover, let \(\Gamma\) be a partial function from session type variables to linear types. The encoding of \(S\) into a linear type w.r.t. \(\Gamma\), written \([S]_{\Gamma}\), is defined as:

\[
\begin{align*}
[S]_{\Gamma} & = \{T\} \quad \text{if } \Gamma(T) = ?(U) \quad \text{act}(\{T\},S) = ?
\end{align*}
\]

The encoding of \(S\) into a linear type is \([S]_{\overline{\Gamma}}\), also abbreviated \([S]\).

Theorem 7.6 is inductively defined on \(S\). Intuitively, it turns \text{end} into \(\bullet\), and external (resp. internal) choices into linear input (resp. output) types. In the latter case, each choice label becomes a label of the carried variant, its payload is encoded into the \(p\) field of the corresponding record, and its continuation into the \(c\) field. Crucially, when encoding an internal choice, \(c\) carries the dual of the encoding of the original continuation: this is because, as seen in §4.3, sending a value requires to allocate a new pair of I/O channel endpoints, keep one of them, and send the other (i.e., the dual, by Theorem 7.5) for continuing the session. Recursion is encoded by turning a recursive external (resp. internal) choice into a linear input (resp. output) carrying a recursive variant: this “structural shift” is achieved by collecting open recursion variables in \(\Gamma\), and using the auxiliary encoding \([1]_{\overline{\Gamma}}\). E.g., let \(S = \mu X. ?A.X\); \([S]_{\overline{\Gamma}}\) gives the type \(\{X\} U\), with \(U\) obtained by letting \(\Gamma' = \Gamma\{X: X\}\), and \(U = \{A.X\} = \{A.X\} U\).

Our handling of recursion greatly affects our proofs, and is a main difference between Theorem 7.6 and the encoding in [6]. Despite this, the crucial Theorem 7.2 still holds.

[Encoding preserves duality, subtyping] \([S] = [\overline{S}]\), and \(S \leq S'\) iff \([S] \leq [\overline{S'}]\).
7.3 From session types to Scala types

We now present our encoding of session types into Scala types. Since Scala has a nominal type system but session types are structural, our encoding requires a nominal environment (Theorem 7.7), giving a distinct class name to each subterm of \( S \).

▶ Definition 7.7. A nominal environment for session types \( N \) is a partial function from (possibly open) session types to Scala class names. \( N \) is suitable for \( S \) if (i) \( \text{dom}(N) \) contains all subterms of \( S \) (except end), (ii) is injective w.r.t. the internal/external choices in its domain, (iii) maps each singleton internal/external choice to its label, (iv) is dually closed, i.e. \( \forall S' \in \text{dom}(N) : N(S') = N(S'') \), and (v) if \( N(\mu_X.S') \) is defined, then \( N(\mu_X.S') = N(X) = N(S') \).

Our encoding of a session type \( S \) into a Scala type is given in Theorem 7.10. It relies on an intermediate encoding of \( S \) into a linear type \( L \), which is further encoded into Scala classes. Such an intermediate step will allow us to exploit the fact that \( L \) is either \( \bullet \), or a linear input/output \?/(U)/!(U), for some (possibly recursive) \( U \). We will see that:

- if \( L \) is an input (resp. output), it will result in a `lchannels In[\_]` (resp. `Out[\_]`) type;
- \( U \) also appears in the dual \( L' \) (by Def. 7.5), corresponding to \( S' \) (by §7.2): it will produce both the type parameter of \( \text{In/Out} \) above, and the CPS classes of \( S/S' \).

We first formalise the encoding from linear types to Scala types, in Theorem 7.8 below.

▶ Definition 7.8. A nominal environment for linear types \( M \) is a partial function from (possibly open) variant types to Scala class names. \( M \) is suitable for \( L \) if \( \text{dom}(M) \) contains all subterms of \( L \) (except \( \bullet \)), is injective w.r.t. the variants in its domain, maps each singleton variant to its label, and if \( M(\mu_X.U) \) is defined, then \( M(\mu_X.U) = M(X) = M(U) \). Given \( M \) suitable for \( L \), we define the encoding of \( L \) into Scala types w.r.t. \( M \), written \( (L)_M \), as:

\[
\begin{align*}
\{?\}(U)_M &= \text{In}[M(U)] \\
\{!(U\_)_M &= \text{Out}[M(U)] \quad \{\bullet\}_M = \text{Unit} \\
\{V\}_M &= \text{V} & (\text{if } V \in \mathbb{B})
\end{align*}
\]

\[
\begin{align*}
\{U\}_M &= \left\{ \begin{array}{ll}
\text{case class } 1\ (p:\{V\}_M)\{(\text{val cont:\(L\)_M})
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\{U\}'_M &= \{U\}_M \\
\{U''\}_M &= \{U\}'_M & \text{if } U' = \text{carr}(V) \\
\end{align*}
\]

\[
\begin{align*}
\{U\}_M &= \left\{ \begin{array}{ll}
\text{sealed abstract class } L\_M \text{ extends } M(U) \\
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\{U\}_M &= \left\{ \begin{array}{ll}
\text{case class } 1\ (p:\{V\}_M)\{(\text{val cont:\(L\)_M})
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\{U\}'_M &= \{U\}_M \\
\{U''\}_M &= \{U\}'_M & \text{if } U' = \text{carr}(V) \\
\end{align*}
\]

\[
\begin{align*}
(\mu_X.U)_M &= (U)_M \\
\{X\}_M &= M(X)
\end{align*}
\]

The encoding in Theorem 7.8 is inductively defined on \( L \). The first 3 cases turn a top-level \?/(\_)/\bullet into a corresponding `In[\_]`/`Out[\_]`/`Unit` type in Scala, and the 4\(^{th}\) case keeps basic types unaltered; note that when encoding \?/(U) (resp. !(U)), the type parameter of the resulting `In[\_]` (resp. `Out[\_]`) is the Scala class name that \( M \) maps to \( U \). The remaining cases of Theorem 7.8 show how \( U \) originates the session protocol classes. Singleton variants are turned into `case classes`, while non-singleton variants are turned into `sealed abstract classes` (with a name given by \( M \)), extended by one `case class` per label. Note that if the \( p \) field of a variant consists in some linear type \?/(U)/!(U), the CPS classes of \( U' \) are generated as well – and similarly for the \( c \) field. A recursive term \( \mu_X.U \) is handled by noticing that, by Theorem 7.7, \( M(\mu_X.U) = M(X) = M(U) \): hence, \( X \) is encoded as \( M(X) = M(\mu_X.U) \).

The last ingredient for our encoding is a way to turn a nominal environment for a session type (Theorem 7.7) into one for a linear type (Theorem 7.8): this is formalised below.
Definition 7.9. We say that $S$ maps $S'$ to $U'$ (in symbols, $S \rightarrow S' \Rightarrow U'$) if, for some $\Gamma$, the computation of $[S]$ involves either (a) an instance of $[S']_\Gamma$ returning $?U'$ or $!(U')$, or (b) an instance of $?S'_{\Gamma'}$, returning $U'$. If $\mathcal{N}$ is suitable for $S$, the linear encoding of $\mathcal{N}$ (w.r.t. $S$) is a nominal environment for linear types denoted with $[\mathcal{N}]_S$, such that:

$$[\mathcal{N}]_S(U) = A \ 	ext{iff} \ \exists S': S \rightarrow S' \Rightarrow U \ 	ext{and} \ \mathcal{N}(S') = A$$

Intuitively, Theorem 7.9 says that if $\mathcal{N}$ maps an internal/external choice $S'$ to some class name $A$, then $[\mathcal{N}]_S$ maps the variant obtained from the encoding of $S'$ to the same $A$.

We are now ready to define our encoding of session types into Scala types.

Definition 7.10. Given $\mathcal{N}$ suitable for $S$, we define the encoding of $S$ into a Scala type as $\langle S \rangle_{\mathcal{N}} = \{[S]\}_{[\mathcal{N}]_S}$, and the protocol classes of $S$ as: $\text{prot}\langle S \rangle_{\mathcal{N}} = \{\text{carr}(\{S\})\}_{[\mathcal{N}]_S}$.

Theorem 7.10 gives us two pieces of information: $\langle S \rangle_{\mathcal{N}}$ is the type $\text{In}[\cdot]/\text{Out}[\cdot]/\text{Unit}$ on which a Scala program can communicate according to $S$, and $\text{prot}\langle S \rangle_{\mathcal{N}}$ gives the definitions of all necessary CPSP classes. Technically, $S$ and $\mathcal{N}$ are first linearly encoded (via Definitions 7.6 and 7.9); then, the result is further encoded into Scala types (via Theorem 7.8).

Example 7.11. The linear encoding of the greeting session type $S_h$ in §2 is:

$$[S_h] = \{U_h\} \quad \text{where} \quad U_h = \mu X . \langle \text{Greet}_h \{p : \text{String}, c : !\text{String} : (\text{Hello}_h \{p : \text{String}, c : !\text{String} : (\text{GoodNight}_h \{p : \text{String}, c : !\text{String} : (\text{Quit}_h \{p : \text{Unit}, c : !\text{String} : X \rangle]))\} \rangle \rangle = \text{Start}$$

Let us now define $\mathcal{N}$, as described in Theorem 4.2, making it suitable for $S_h$ (as per Theorem 7.7):

$$\mathcal{N} \langle \text{Greet}(\text{String}) \rangle \langle \text{GoodNight}(\text{String}) \rangle \langle \text{Quit}(\text{Unit}) \rangle \langle \text{Hello}(\text{String}) \rangle \langle \text{Bye}(\text{String}) \rangle = \text{Start}$$

Now, we can verify that the following mappings hold:

$$S_h \rightarrow \langle \text{Greet}(\text{String}) \rangle \langle \text{Hello}(\text{String}) \rangle \langle \text{Bye}(\text{String}) \rangle \langle \text{Quit}(\text{Unit}) \rangle$$

Hence, by Theorem 7.9, $[\mathcal{N}]_{S_h}$ maps the first, second and third (recursive) variant types above to $\text{Start}$, and the last one to $\text{Greeting}$. The encoding $\langle S_h \rangle_{\mathcal{N}} = \{[S_h]\}_{[\mathcal{N}]_{S_h}}$ is $\text{Out}[\text{Start}]$, while $\text{prot}\langle S_h \rangle_{\mathcal{N}} = \{\text{carr}(\{S_h\})\}_{[\mathcal{N}]_{S_h}}$ gives the Scala protocol classes seen in Theorem 4.2.

We conclude with two results at the roots of our session-based development approach (§4.2). Let the dual of a Scala type be $\text{In}[\cdot] = \text{Out}[\cdot]$, $\text{Out}[\cdot] = \text{In}[\cdot]$, and $\text{Unit} = \text{Unit}$.

For all $S$, $\langle S \rangle_{\mathcal{N}} = \langle S \rangle_{\mathcal{N}'}$ and $\text{prot}\langle S \rangle_{\mathcal{N}} = \text{prot}\langle S \rangle_{\mathcal{N}'}$.

§7.3 says that a session type and its dual are encoded as dual Scala types, and dual session types have the same protocol classes: this justifies steps D1–D3 in §4.2.

Finally, let $\prec$ be the Scala subtyping (the full definition is available in the on-line technical report, see §1). Suppose that we encode a session type $S$, getting $B$, and write a program using $A$ such that $A \prec B$ or $B \prec A$; by §7.3, this is sound. For all $A, S, N$, $A \prec \langle S \rangle_{\mathcal{N}}$ implies one of the following:

1. $S = \text{end}$, and: $A \prec \text{Unit}$ and $\forall B : A \not\in \{\text{In}[B], \text{Out}[B]\}$;
2. $\text{act}(S) = \?$, and: $A \not\in \text{Null}$ or $\exists B : A \in \{\text{In}[B], \text{Out}[B] \}$ implies $\exists S', N' : A \equiv \langle S' \rangle_{\mathcal{N}'}$ and $S' \prec S$;
3. $\text{act}(S) = !$, and: $A \not\in \text{Null}$ or $\exists B : A \in \{\text{In}[B], \text{Out}[B] \}$ and $B \not\in \text{AnyRef}$ implies $A \equiv \langle S \rangle_{\mathcal{N}}$.

Moreover, for all $A, S, N$, $\langle S \rangle_{\mathcal{N}} \prec A$ implies one of the following:
(b1) $S = \text{end}$, and: $\text{Unit} < A$ and $\forall B.A \not\equiv \{\text{In}[B], \text{Out}[B]\};$
(b2) $\text{act}(S) = ?$, and: $\text{AnyRef} < A$ or $\exists B.A = \text{In}[B]$ and $(B \not\subseteq \text{AnyRef} \implies A \not\subseteq \{S\}_{X});$
(b3) $\text{act}(S) = !$, and: $\text{AnyRef} < A$ or $\exists B.A = \text{Out}[B]$ and $(\text{null} \not\subseteq B \implies 3S', N': A \equiv \{S\}^4_{X},$ and $S \subseteq S').$

Roughly, § 7.3 says that Scala subtyping reflects session subtyping, thus preserving its safety/exhaustiveness guarantees ($S1$ and $S2$ in §2.1). When end is encoded, items $a1/b1$ say that its Scala sub/super-types cannot be In/Out, i.e. their instances do not allow I/O. For item $a2$, consider Theorem 4.3: we have $\text{In}[\text{Full}] < : \text{In}[\text{WaitingRoom}]$, reflecting the fact that $?\text{Full} \subseteq S_{\text{estm}}$ (by $\langle \cdot, \text{Ext} \rangle$). For item $b3$, consider Theorem 4.2: we have $\text{Out}[\text{Start}] < : \text{Out}[\text{Quit}]$, reflecting the fact that $S_1 \subseteq !\text{Quit}$ (by $\langle \cdot, \mu \rangle$ and $\langle \cdot, \text{Inv} \rangle$). § 7.3 also says that $<$ is stricter than $\subseteq$ – e.g., by item $a3$, the Scala encoding of an internal choice has no subtypes, and by item $b2$, an external choice has no supertypes. However, Scala allows for sub/super-types that do not correspond to any session type: besides the unavoidable $\text{null}$ cases (items $a2, a3, b3$), it is possible e.g. to write a method $f$ with a parameter of type $\text{In}[\text{Any}]$ (b2), or $\text{In}[\text{Nothing}]$ (a2), or $\text{Out}[\text{Any}]$ (a3), or $\text{Out}[\text{Nothing}]$ (b3). This does not compromise safety/exhaustiveness, either: $\text{In}[\text{Any}]$ makes $f$ accept any message, $\text{Out}[\text{Nothing}]$ forbids $f$ to send, while $\text{In}[\text{Nothing}] / \text{Out}[\text{Any}]$ are subtypes of all $\text{In/Out}$ types – thus making $f$ non-applicable to any channel endpoint obtained by encoding a session type. Notably, this holds by co/contra-variance of $\text{In}[+\text{A}] / \text{Out}[-\text{A}]$ (Fig. 6, left).

8 Related work

Session types and their implementation. Session types were introduced by Honda et al. in [18, 39, 19], as a typing discipline for a variant of the $\pi$-calculus (called session-$\pi$ in §2). They have been studied and developed in multiple directions during the following decades, notably addressing multiparty interactions [20] and logical interpretations [5, 43]. The encoding of session types in linear $\pi$-calculus types has been studied in [9, 8, 6, 7]; our work is mainly based on [8], but our treatment of recursion is novel (see §7.2).

Session types have been mostly implemented on dedicated programming languages with the advanced type-level features outlined in §2 [14, 43, 11, 40, 3]. [32, 34] aim at an integration with Haskell, using monads to enforce linearity (at the price of a restrictive and rather complicated API). [25] adapts [34] to Rust, exploiting its affine types, but showing limitations to binary internal/external choices. [23, 37, 38] are based on a Java language extension and runtime/session-type-inspired primitives for I/O and branching. [22] integrates session types in Java via automatic generation of classes representing session-typed channel endpoints, with run-time linearity checks. The main differences w.r.t. our work are that [22] is closer to session-$\pi$, is based on the Scribble tool [44], supports multiparty sessions, and generates classes which represent both a channel endpoint and its protocol; hence, in the binary setting, each endpoint has its own hierarchy of generated classes that is different (but “dual”) w.r.t. the other endpoint. Instead, our I/O endpoints are closer to linear types for the $\pi$-calculus [36]: they take the protocol as a type parameter, from a set of CPSP classes which is common between the two endpoints. Other differences are mostly due to the Java type system, which e.g. does not support case classes (complicating exhaustiveness checks) nor declaration-site variance (complicating the handling of I/O co/contra-variance).

The work closer to ours is [33]: it presents an encoding of session types in a ML-like language, and an OCaml library reminiscent of lchannels. We share several ideas and features, including the theoretical basis of [8]. The differences are at technical and API design levels, due to different languages and goals (type inference vs. CPSP extraction);given the wide adoption of Scala, we focus on practical validation with use cases and benchmarks.

Strong typing guarantees for concurrent applications have been a longstanding goal for
the Scala and Akka communities. In the actor realm, Akka Typed (§1) is remarkably close to [17]: both propose `ActorRef[A]`-typed actor references. We drew inspiration from them and CPSPs, merging the theoretical basis of [8]. Some (non-linear) channel APIs have been tentatively introduced in Akka, e.g. `channels` (Akka 1.2) and macro-based `typed channels` (Akka 2.1); however, they were later deprecated, mainly due to design and maintainability issues [26]. `lchannels` is based on a clear and well-established theory, adapted to the Scala setting: thus, the implementation is fairly simple and maintainable, not requiring macros.

9 Conclusions

We showed how session programming can be carried over in Scala, by representing protocols as types that the compiler can check. We based our approach on a lightweight integration of session types, based on CPSP classes and the `lchannels` library. We showed that our approach supports local and distributed interaction, has a formal basis (the encoding of session types into linear I/O types), and attested its viability with use cases and benchmarks.

We plan to extend our approach to multiparty session types (MPSTs), by extracting CPSP classes from a global type [20], rather than addressing multiple binary sessions separately (as in Theorem 4.3 and §6.1). Just as binary session typing guarantees safe and deadlock free interaction for two parties involved in one session (§2.2), MPSTs extend such a guarantee to two or more parties; the main challenge is that encoding MPSTs into Scala types might be complex, and require a tool akin to [22].

The Scala landscape is fast-moving, and recent developments may influence the evolution of our work. [41] introduces customisable effect for Scala: by extending the `lchannels` I/O operations with an effect, we could obtain stronger linearity guarantees — e.g., ensuring that a program does not “forget” a session (§6.1.3). [15] studies capabilities for borrowing object references: they could ensure that a channel endpoint is never used if sent (§3.1). Similar guarantees could be achieved by examining the program call graph [1]. Recent results on Scala’s type system (e.g. on path-dependant and structural types [2, 35]) might improve our encoding, removing the limitation on the uniqueness of choice labels (Remark 7.1).

We will further extend and optimise `lchannels` and its API: many improvements are possible, and the transport abstraction allows to easily compare different implementations, under different settings and uses. We also plan to extend our approach to other languages: one candidate is C#, due to its support for first-class functions and declaration-site variance.

Towards session types for Akka Typed (and other frameworks). This work focuses on `lchannels`, but our approach can be generalised to other communication frameworks. One possible way is abstracting under the `In[]/Out[]` API, as in §5; another way is directly using the I/O endpoints offered by other frameworks. Consider e.g. Akka Typed: we can adapt CPSP extraction (Theorem 7.8) to yield `ActorRef[A]` types instead of `Out[A]`, obtaining CPSP classes similar to those in Fig. 2. Remarkably, `Out[A]` and `ActorRef[A]` are both contravariant w.r.t. A, and enjoy similar subtyping properties (§7.3). However:

(i) Akka Typed does not offer an input endpoint similar to `In[]`. Hence, session types whose CPSPs carry input endpoints (e.g., Theorem 4.1, or `S_\text{ext}` in §6.1.1) must be adapted (i.e., sequences of two outputs or two inputs must be replaced with input-output alternations);

(ii) instances of `ActorRef[A]` raise no errors when used multiple times for sending messages;

(iii) to produce and send a continuation `ActorRef[A]`, it is customary to cede the control to another actor (possibly a new one, as in Fig. 3); `lchannels`, instead, encourages the creation and use of I/O endpoints along a single thread, in a simple sequential style.
Item 1 is a minor issue; 2 could be addressed, taking inspiration from the session/linear types theory, by distinguishing unrestricted \[42\] \text{ActorRefs} (allowing 0 or more outputs of the same type) from linear \text{ActorRefs} – with the former usable as the latter, but not vice versa. Item 3 marks a crucial difference between reactive, actor-based concurrent programming (where the protocol flow is decomposed into multiple input-driven handlers), and thread-based programming. We plan to study the formal foundations for applying “session types as CPSPs” in the reactive setting, and their feasibility w.r.t. software industry practices.

Acknowledgements. Thanks to Roland Kuhn, Julien Lange and the anonymous reviewers for their helpful remarks on earlier versions of this paper. Thanks to Julien Lange and Nicholas Ng for their feedback during artifact testing, and to the anonymous artifact reviewers for their detailed remarks and suggestions.

References