Lightweight Session Programming in Scala*

Alceste Scalas[1] and Nobuko Yoshida[2]

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

* Work partly supported by: EPSRC EP/K011715/1, EP/K034413/1 and EP/L00058X/1, and EU project FP7-612985 *UpScale*.

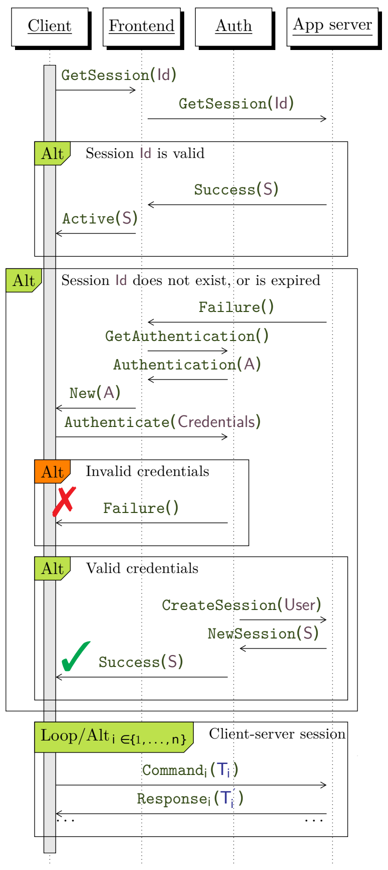


Figure 1 Server with frontend.

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

```scala
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.

```scala
1 def client(frontend: ActorRef[GetSession]) = {
2   val cont = spawn[GetSessionResult] {
3     case New(a) => doAuthentication(a)
4     case Active(s) => doSessionLoop(s)
5   }
6   frontend ! GetSession(42, cont)
7 }
```

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 `ActorRef`s – 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 `Promise`s/`Future`s.

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 class`es), `Promise`s/`Future`s; 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 $\mathbb{B} = \{\mathsf{Int}, \mathsf{Bool}, \mathsf{Unit}, \ldots\}$ be a set of *basic types*. A session type S has the following syntax:

$$S \;::=\; \&_{i\in I} ?\mathtt{l}_i(T_i).S_i \;\big|\; \oplus_{i\in I} !\mathtt{l}_i(T_i).S_i \;\big|\; \mu_X.S \;\big|\; X \;\big|\; \mathbf{end} \qquad T \;::=\; \mathbb{B} \;\big|\; S \text{ (closed)}$$

where $I \neq \emptyset$, recursion is guarded, and all $\mathtt{l}_i$ range over pairwise distinct *labels*. T denotes a *payload type*. The *branching type* (or *external choice*) $\&_{i\in I} ?\mathtt{l}_i(T_i).S_i$ requires the process to receive one input of the form $\mathtt{l}_i(T_i)$, 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*) $\oplus_{i\in I} !\mathtt{l}_i(T_i).S_i$, instead, requires the program to choose and perform one output $\mathtt{l}_i(T_i)$, for some $i \in I$, and continue using the channel according to S_i. $\mu_X.S$ is a *recursive* session type, where μ binds X, and X is a *recursion variable*. We say that S *is closed* iff all its recursion variables are bound. **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.

▶ Remark 2.1. We use $\oplus/\&$ as infix operators, omitting them in singleton choices. We often omit **end** and Unit: $?\mathtt{A}\big(!\mathtt{B}(\mathsf{Int}) \oplus !\mathtt{C}\big)$ stands for $\&\{\,?\mathtt{A}(\mathsf{Unit}).\oplus\{!\mathtt{B}(\mathsf{Int}).\mathbf{end}\,,\, !\mathtt{C}(\mathsf{Unit}).\mathbf{end}\}\,\}$.

For example, the type S_{h} below describes the client endpoint of a "greeting protocol":

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

The client can send either `Quit` and **end** the session, or `Greet`(String); in the second case, it might receive from the server either `Bye`(String) (**end**ing the session), or `Hello`(String): in the second case, the session continues recursively.

Programming languages that support session types are usually based on session-π – i.e., a version of π-calculus [31] extended with session operators. A client respecting S_{h} would be implemented as `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:

S1. *safety:* no out-of-protocol I/O actions are allowed. E.g., `c` can initially be used *only* to send `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 **end**ed (line 6);

S2. *exhaustiveness:* when receiving a message, all outcomes allowed by the type must be covered. E.g., the client must handle *both* `Hello` and `Bye` answers (lines 4–6);

S3. *output linearity:* if S_{h} prescribes an output, it must occur *exactly once*. E.g., after receiving `Hello`, the client *must* send `Greet` or `Quit` (as in the recursive call of line 5);

S4. *input linearity:* similarly, if S_{h} prescribes an input, it must occur *exactly once*. E.g., after sending `Greet`, the client *must* receive the response (as in line 4).

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 *dually*. The *dual of* S, written $\overline{S}$, is defined as:

$$\overline{\&_{i\in I} ?\mathtt{l}_i(T_i).S_i} = \oplus_{i\in I} !\mathtt{l}_i(T_i).\overline{S_i} \qquad \overline{\oplus_{i\in I} !\mathtt{l}_i(T_i).S_i} = \&_{i\in I} ?\mathtt{l}_i(T_i).\overline{S_i}$$

$$\overline{\mu_X.S} = \mu_X.\overline{S} \qquad \overline{X} = X \qquad \overline{\mathbf{end}} = \mathbf{end}$$

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

```
 1 def lHello(c: LinOutChannel[?]): Unit = {
 2   if (...) {
 3     val (c2in, c2out) = createLinChannels[?]()
 4     c.send( Greet("Alice", c2out) )
 5     c2in.receive match {
 6       case Hello(name, c3out) => lHello(c3out)
 7       case Bye(name) => ()
 8     }
 9   } else { c.send( Quit() ) }
10 }
```

Figure 4 Greeting protocol client (pseudo code): session types (left) vs. linear I/O types (right).

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:

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

Duality guarantees the *safe and deadlock-free interaction* of a client and server observing S_{h} and $\overline{S_{\text{h}}}$: 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 $S_{\text{h2}} = !\texttt{Quit}$, and its implementation on the right: since `hello2` only outputs `Quit` on `c2`, it would also behave safely on a S_{h}-typed channel endpoint `c`. In fact, in a session-typed language we have $S_{\text{h}} \leqslant S_{\text{h2}}$– i.e., an S_{h}-typed channel endpoint can always be used in place of an S_{h2}-typed one; hence, invoking `hello2(c)` is allowed – and such a client program would interact safely and without deadlocks with a server observing $\overline{S_{\text{h}}}$.

```
def hello2(c2: S_h2): Unit = {
  c2 ! Quit()
}
```

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-π into *standard π-calculus with variants and linear I/O types* [8]: the key idea is that session-π 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 `lHello` is the "linearly encoded" version of `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 `c2out` becomes available in `c2in`. On line 4, `c` is used to send a `Greet` message – which *also carries* `c2out`. Then, the recipient of `Greet` and `c2out` is expected to use the latter to continue the session – i.e., send either `Hello` or `Bye`. On line 5, `c2in` is used to receive such an answer, and the result is matched against `Hello` and `Bye`; the

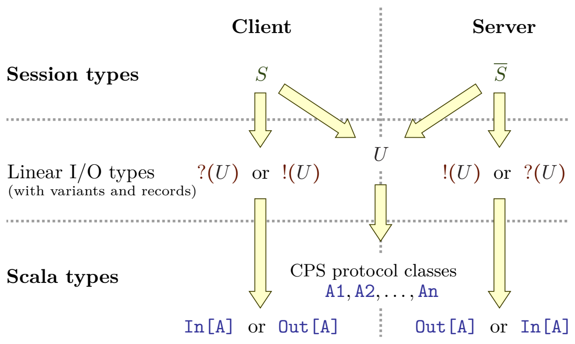


Figure 5 From session types to Scala types.

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 S_{h};
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 $\overline{S}$. 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* $\overline{S}$, 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) –

```scala
 1 abstract class In[+A] {
 2   def future: Future[A]
 3   def receive(implicit d: Duration): A = {
 4     Await.result[A](future, d)
 5   }
 6   def ?[B](f: A => B)(implicit d: Duration): B = {
 7     f(receive)
 8   }
 9 }
10
11 abstract class Out[-A] {
12   def promise[B <: A]: Promise[B]
13   def send(msg: A): Unit    = promise.success(msg)
14   def !(msg: A)                    = send(msg)
15   def create[B](): (In[B], Out[B])
16 }
```

```scala
 1 class LocalIn[+A](val future: Future[A]) extends In[A]
 2
 3 class LocalOut[-A](p: Promise[A]) extends Out[A] {
 4   override def promise[B <: A] = {
 5     p.asInstanceOf[Promise[B]] // Type-safe cast
 6   }
 7   override def create[B]() = LocalChannel.factory[B]()
 8 }
 9
10 object LocalChannel {
11   def factory[A](): (LocalIn[A], LocalOut[A]) = {
12     val promise = Promise[A]()
13     val future = promise.future
14     (new LocalIn[A](future), new LocalOut[A](promise))
15   }
16 }
```

Figure 6 Linear channels in Scala: abstract classes (left) and local implementation (right).

notably, by reflecting their *co*/*contra*-variance. For simplicity, we shape the API and its basic implementation around `Promise`s/`Future`s 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, `Promise`s/`Future`s 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. `A`[1]. 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 constant*[2], 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 `A`[3]. 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 `receive`d, it returns `f(v)`. The rationale behind the method signature is clarified in Theorem 3.1.

[1] This matches the output subtyping rule [$\leqslant_\ell$-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 *in*variant `Out` with limited subtyping.
[3] This matches the input subtyping rule [$\leqslant_\ell$-IN] in Theorem 7.4.

▶ **Example 3.1** (`!`, `?` and selection/branching). Consider the following classes:

```
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:

```
c ! A()
```

or

```
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,

```
c ? { case A() => println("Got A")
      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 `case`s 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 $\oplus$ (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 `Promise`s and `Future`s. 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 `Future`s 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 *in*variance 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 class`es (as detailed in § 7.3).

```
 1 case class Q(p: Boolean, cont: Out[R])
 2 case class R(p: Int)
 3 
 4 def f(c: In[Q]) = {
 5   c ? { q => q.cont ! R(42) }
 6 }
 7 
 8 def g(c: Out[Q]) = {
 9   val (ri,ro) = c.create[R]()
10   c ! Q(true, ro)
11   ri ? { r =>
12     println(f"Got ${r.p}")
13 } }
```

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

```
1 def parallel[A, B1, B2](p1: In[A] => B1, p2: Out[A] => B2): (Future[B1], Future[B2]) = {
2   val (in, out) = LocalChannel.factory[A]();  ( Future { p1(in) }, Future { p2(out) } )
3 }
```

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 $S_{\mathtt{QR}} = ?\mathtt{Q}(\mathsf{Bool}).!\mathtt{R}(\mathsf{Int})$, dictating that a channel endpoint must be used first to receive $\mathtt{Q}(\mathsf{Bool})$, and then to output $\mathtt{R}(\mathsf{Int})$. 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 $S_{\mathtt{QR}}$; but, how do we require to send a value of type R along the same interaction?

```
case class Q(p: Boolean)
case class R(p: Int)
```

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, f uses c to receive a Q-typed value q (line 5); then, uses q.cont to send a value of type R.

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

4.2 A development approach for session-based applications

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

D1. formalise the two endpoint session types S and $\overline{S}$ (assuming they are not trivially **end**);

D2. extract the CPSP classes of S (or, equivalently, of $\overline{S}$). 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 $\overline{S}$) by "connecting" each case class to its "successor" (if any), through the cont field;

D3. let C be the class representing the outermost internal/external choice of S:

- if S starts with an internal choice, its Scala endpoint type is Out[C]. Dually, since $\overline{S}$ starts with an external choice, the Scala type at the other endpoint is In[C];
- otherwise, if S starts with an external choice, its Scala endpoint type is In[C]. Dually, since $\overline{S}$ starts with an internal choice, the Scala type at the other endpoint is Out[C].

The extraction of protocol classes must deal with some subtleties, in particular for determining whether cont should be an In[·] or 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

```
1  abstract class Out[-A] { ...
2    def !![B](h: Out[B] => A): In[B] = {
3      val (cin, cout) = this.create[A]()
4      this ! h(cout)
5      cin
6    }
7    def !![B](h: In[A] => B): Out[B] = {
8      val (cin, cout) = this.create[A]()
9      this ! h(cin)
10     cout
11 } }
```

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 Out (Fig. 6, left) with the method !! above.

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

[7] The pattern actually reflects how session-π processes are encoded in standard π-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 `In[R]`). By letting `h` be an instance of `Q` with a hole in place of `cont`, we can remove line 9 of Fig. 7, and rewrite line 10 as:

```
val ri = c !! Q(true, _: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 `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.

```
case class Q(p: Boolean)
            (val cont: Out[R]) // Curried
case class R(p: Int)

def g(c: Out[Q]) = {
  val ri = c !! Q(true)_ // No type annot.
  ri ? { r => println(f"Got ${r.p}") }
}
```

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:

$$S_{\text{fifo}} = \mu_X.!\texttt{Datum}(T).X \text{ (sending endpoint)} \qquad \overline{S_{\text{fifo}}} = \mu_X.?\texttt{Datum}(T).X \text{ (receiving endpoint)}$$

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

```
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., `Datum`), and *(ii)* `cont`inuing with such a name when X occurs (for another case of recursion, see Theorem 4.2). Note that `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:

```
def sender(fifo: Out[Datum[Int]]): Unit = {
  val cont = fifo !! Datum(1)_ !! Datum(2)_
  sender(cont)
}
```

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

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

▶ **Example 4.2** (Greeting protocol). Consider the "greeting" types S_{h} and $\overline{S_{\text{h}}}$ from § 2. Unlike Theorem 4.1, we now have *non-singleton* internal/external choices. To extract their CPSP classes, we apply item **D2**b 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 $\overline{S_{\text{h}}}$) between `Greet`,`Quit`;
- `Greeting` for the external choice of S_{h} (i.e., the internal choice of $\overline{S_{\text{h}}}$) 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[Start]`/`In[Start]` representing $S_{\mathrm{h}}/\overline{S_{\mathrm{h}}}$ (by **D3**). We can write two endpoint processes as:

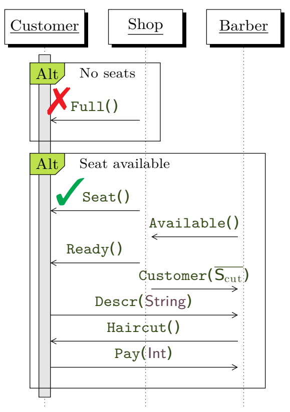
```scala
sealed abstract class Start
case class Greet(p: String)(cont: Out[Greeting]) extends Start
case class Quit(p: Unit)                          extends Start

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

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

```scala
def server(c: In[Start]): Unit = {
  c ? {
    case m @ Greet(whom) => {
      val c2in = m.cont !! Hello(whom)_
      server(c2in)
    }
    case Quit() => ()
} }
```

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_{cstm}-typed channel endpoint:

$S_{\mathrm{cstm}} = ?\mathtt{Full} \,\&\, ?\mathtt{Seat}.?\mathtt{Ready}.S_{\mathrm{cut}} \qquad S_{\mathrm{cut}} = !\mathtt{Descr}(\mathsf{String}).?\mathtt{Haircut}.!\mathtt{Pay}(\mathsf{Int})$

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}: `Descr`ibes the new hairdo, waits for the `Haircut`, `Pay`s and leaves. The shop uses the other, dually-typed channel endpoint:

$$\overline{S_{\mathrm{cstm}}} = !\mathtt{Full} \oplus !\mathtt{Seat}.!\mathtt{Ready}.\overline{S_{\mathrm{cut}}} \qquad \overline{S_{\mathrm{cut}}} = ?\mathtt{Descr}(\mathsf{String}).!\mathtt{Haircut}.?\mathtt{Pay}(\mathsf{Int})$$

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:

$$S_{\mathrm{barber}} = \mu_X.!\mathtt{Available}.?\mathtt{Serve}(\overline{S_{\mathrm{cut}}}).X \quad \text{(barber endp.)} \qquad \overline{S_{\mathrm{barber}}} = \mu_X.?\mathtt{Available}.!\mathtt{Serve}(\overline{S_{\mathrm{cut}}}).X \quad \text{(shop endp.)}$$

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 $\overline{S_{\mathrm{cut}}}$-typed) to the barber, as the payload of `Serve`.

Meanwhile, the barber uses its S_{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 $\overline{S_{\mathrm{cut}}}$-typed channel endpoint in the message payload: he is expected to use it for interacting with the customer, i.e., listen for the hairdo `Descr`iption, perform the `Haircut`, and take the `Pay`ment. 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 **D2**b 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. $\overline{S_{\text{cstm}}}$).

```scala
// Customer <--> shop protocol
sealed abstract class WaitingRoom
case class Full()                         extends WaitingRoom
case class Seat()(val cont: In[Ready])  extends WaitingRoom

case class Ready()(val cont: Out[Description])
case class Description(p: String)(val cont: Out[Cut])
case class Cut()(val cont: Out[Pay])
case class Pay(p: Int)

// Barber <--> shop protocol
case class Available()(val cont: Out[Serve])
case class Serve(p: In[Description])(val cont: Out[Available])
```

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 `Customer`s 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 `Serve`$(\overline{S_{\text{cut}}})$ 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`.

5 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

```
 1 class Shop(nSeats: Int) extends Runnable {
 2   private val seats: Fifo[Out[Ready]] = Fifo() // Customers queue
 3   private val waiting = new AtomicInteger(0) // Customers in shop
 4 
 5   def enter(): In[WaitingRoom] = {
 6     val (in, out) = LocalChannel.factory[WaitingRoom]()
 7     val nPeople = waiting.getAndIncrement() // New person in shop
 8     if (nPeople >= nSeats) { // More people than seats
 9       waiting.getAndDecrement() // Customer must leave
10       out ! Full() // Tell customer that the witing room is full
11     } else {
12       val r = out !! Seat()_ // Tell customer that he got a seat
13       seats.write(r) // Add customer to waiting queue
14     }
15     in // Return input endpoint of customer channel
16   }
17 
18   override def run(): Unit = {
19     val (bIn, bOut) = LocalChannel.factory[Available]()
20     new Barber(bOut).start() // Spawn barber with output endpoint
21     loop(bIn) // Loop on the input endpoint
22   }
23 
24   private def loop(bIn: In[Available]): Unit = {
25     bIn ? { avl => // The barber is available
26       val cust = seats.read // Take 1st customer, sleep if none
27       waiting.getAndDecrement() // Customer is leaving the seat
28       val cust2 = cust !! Ready()_ // Notify that barber is ready
29       // Forward the customer to the barber
30       val bIn2 = avl.cont !! Serve(cust2)_ // bIn2: In[Available]
31       loop(bIn2) // Keep interacting with the barber
32     }
33   }
34 }
```

```
 1 class Barber(c: Out[Available]) extends Runnable {
 2   override def run(): Unit = {
 3     loop(c)
 4   }
 5 
 6   private def loop(c: Out[Available]): Unit = {
 7     (c !! Available()_) ? { srv => // Got customer
 8       val d = srv.p.receive // Got haircut descr
 9       val payC = d.cont !! Haircut()_ // Cut hair
10       val pay = payC.receive // Wait payment
11       // Got pay, no continuation: customer done
12       loop(srv.cont) // Continue shop interaction
13     }
14   }
15 }
```

```
 1 class Customer(shop: Shop) extends Runnable {
 2   override def run(): Unit = {
 3     val s = shop.enter() // Type: In[WaitingRoom]
 4     s ? {
 5       case Full() => { // No seats
 6         Thread.sleep(...) // Random wait
 7         run() // Try taking a seat again
 8       }
 9       case m @ Seat() => { // Got a seat
10         val r = m.cont.receive // Barber ready
11         val cutC = r.cont !! Descr("Fancy cut")_
12         val cut = cutC.receive // Wait for cut
13         cut.cont ! Pay(42) // Cut done, paying
14       }
15     }
16   }
17 }
```

Figure 8 Sleeping barber (Theorem 4.3): shop, barber and customer implementations.

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 greeting `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` above. 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 `ActorOut[A]` roughly corresponds to sending an `ActorRef[A]` instance (which is serializable out-of-the-box) – but sending an `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 `lchannels` with channel endpoints that send/receive messages through Java `InputStream`/`OutputStreams`, obtained e.g. from a network socket. The main classes are `StreamIn`/`StreamOut` (extending resp. `In`/`Out`), and can only be instantiated by providing a protocol-specific `StreamManager` which can serialize/deserialize messages to/from a stream of bytes (tracking the session status if needed).

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

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 `StreamManager` class, implementing a `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:

```scala
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 `client` seen in Theorem 4.2: we leverage `lchannels` 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 `In[A]` seen in Fig. 6 do not handle errors; e.g., `receive` throws an exception if no message arrives within the (implicit) `Duration 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 `StreamManager` cannot deserialize, or a network problem may occur. As typical for Scala APIs, we extended `In[A]` to capture failures as `Try[A]` values, via 2 additional methods: `tryReceive` and `??`.

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

E.g., the branching on `AorB` in Theorem 3.1 can be made error-resilient by using `c.??`, as shown on the left: the top-level matching is now on `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 `lchannels` (§ 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 *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_{front} formalises the usage of the channel endpoint that the frontend handles while interacting with a client. It is defined as follows:

$$S_{\text{front}} = ?\texttt{GetSession}(\mathsf{Id}).\big(!\texttt{New}(S_{\text{auth}}) \oplus !\texttt{Active}(S_{\text{act}})\big) \quad S_{\text{auth}} = !\texttt{Authenticate}(\mathsf{Cred}).\big(?\texttt{Success}(S_{\text{act}}) \,\&\, ?\texttt{Failure}\big)$$
$$S_{\text{act}} = \mu_X.\big(!\texttt{Quit} \oplus !\texttt{GetId}.?\texttt{Id}(\mathsf{Id}).X \oplus !\texttt{Ping}.?\texttt{Pong}.X \oplus !\texttt{Join}(\mathsf{String}).?\texttt{ChatRoom}((S_{\text{r}}, S_{\text{rctl}})).X\big)$$

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

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

$$S_{\text{FS}} = \mu_X.!\texttt{GetSession}(\mathsf{Id}).\big(?\texttt{Success}(S_{\text{act}}).X \,\&\, ?\texttt{Failure}.X\big)$$

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_{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 classhas no `cont`inuation, 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.

```
1  class ChatServer(...) extends Runnable {
2    ...
3    private def createSession(username: String): Out[Command]) = {
4      val id = allocUniqueSessionId()
5      val (in, out) = LocalChannel.factory[Command]()
6      in.future.onComplete { // Using scala.util.{Success, Failure}
7        case Success(cmd) => queueRequest(Success((id, cmd)))
8        case Failure(e) => queueRequest(Failure(e))
9      }
10     // Add the new session to the list of known sessions
11     sessions(id) = ... /* session info, including username */
12     out
13 } }
```

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 `Promise`s, 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

(§ 5): 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 `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 `Out.send`/`In.receive` for inter-thread communication, instantiating them with `LocalChannel`s, `QueueChannel`s and `ActorChannel`s (columns 1, 5, 7). As a comparison, we adapted such implementations to interact via `Promise`s/`Future`s (column 2), and also to interact "non-CPS" via `scala.concurrent.Channel`s, and Java `ArrayBlockingQueue`s / `LinkedTransferQueue`s (columns 3, 4, 6).

The overhead of `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* `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 `Promise`s/`Future`s 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 `LocalChannel`s add a small slowdown to the underlying `Promise`s/`Future`s. `QueueChannel`s 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, `QueueChannel`s add a perceivable overhead on the underlying `LinkedTransferQueue`s. `ActorChannel`s are slower, especially with many threads and low parallelism (as in "Ring"): it is due to the (currently unoptimised) internal machinery that makes `ActorChannel`s *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 `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.

▶ Remark 7.1 (Syntactic restrictions). For all S, *(i)* each label is unique, and also a valid Scala class name, and *(ii)* each μ binds a *distinct* variable that actually occurs in its scope.

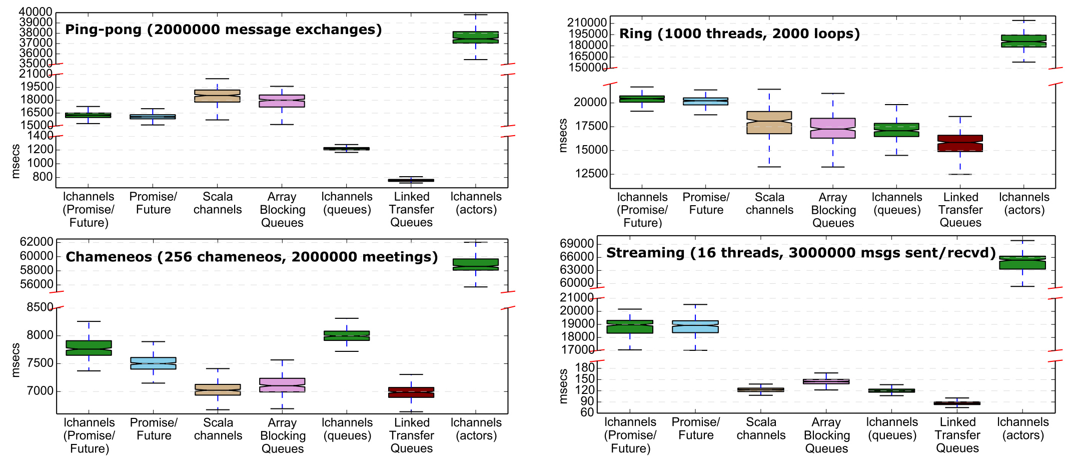


Figure 9 Benchmark results (box&whisker plot): 30 runs × 10 JVM invocations, Intel Core i7-4790 (4 cores, 3.6 GHz), 16 GB RAM, Ubuntu 14.04, Oracle JDK 64-bit 8u72, Scala 2.11.7, Akka 2.4.2.

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* $\leqslant$ allows to safely replace a S'-typed channel endpoint with a S-typed one, provided that $S \leqslant S'$ 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 $\leqslant_{\mathbb{B}}$ is a subtyping between basic types):

$$\dfrac{\forall i \in I:\quad T_i \leqslant T'_i \quad S_i \leqslant S'_i}{\&_{i\in I} ?\mathtt{l}_i(T_i).S_i \;\leqslant\; \&_{i\in I\cup J} ?\mathtt{l}_i(T'_i).S'_i}\ [\leqslant\text{-Ext}] \qquad \dfrac{\forall i \in I:\quad T'_i \leqslant T_i \quad S_i \leqslant S'_i}{\oplus_{i\in I\cup J} !\mathtt{l}_i(T_i).S_i \;\leqslant\; \oplus_{i\in I} !\mathtt{l}_i(T'_i).S'_i}\ [\leqslant\text{-Int}]$$

$$\mathbf{end} \leqslant \mathbf{end}\ [\leqslant\text{-End}] \qquad \dfrac{S\{^{\mu_X.S}/_X\} \;\leqslant\; S'}{\mu_X.S \;\leqslant\; S'}\ [\leqslant\text{-}\mu\text{L}] \qquad \dfrac{S \;\leqslant\; S'\{^{\mu_X.S'}/_X\}}{S \;\leqslant\; \mu_X.S'}\ [\leqslant\text{-}\mu\text{R}] \qquad \dfrac{T \leqslant_{\mathbb{B}} T'}{T \leqslant T'}\ [\leqslant\text{-}\mathbb{B}]$$

Rule [$\leqslant$-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, [$\leqslant$-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. [$\leqslant$-End] says that a terminated session has no subtypes. [$\leqslant$-μL] and [$\leqslant$-μR] are standard. [$\leqslant$-$\mathbb{B}$] extends $\leqslant$ 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 [36]*. We focus on a subset of such types, defined below.

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

$$L \;::=\; ?(U) \;\big|\; !(U) \;\big|\; \bullet \qquad U \;::=\; [\mathtt{l}_i\_\{\mathtt{p} : V_i, \mathtt{c} : L_i\}]_{i\in I} \;\big|\; \mu_X.U \;\big|\; X \qquad V \;::=\; \mathbb{B} \;\big|\; L \text{ (closed)}$$

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

$?(U)$ (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 $\mathtt{l}_i$-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* $\leqslant_\ell$ *between linear types* is coinductively defined by the following rules (where $\leqslant_{\mathbb{B}}$ is a subtyping between basic types):

$$\frac{U \leqslant_\ell U'}{?(U) \leqslant_\ell ?(U')}\,[\leqslant_\ell\text{-In}] \qquad \frac{U' \leqslant_\ell U}{!(U) \leqslant_\ell !(U')}\,[\leqslant_\ell\text{-Out}] \qquad \bullet \leqslant_\ell \bullet\,[\leqslant_\ell\text{-End}] \qquad \frac{V \leqslant_{\mathbb{B}} V'}{V \leqslant_\ell V'}\,[\leqslant_\ell\text{-}\mathbb{B}]$$

$$\frac{\forall i \in I\colon\ \ V_i \leqslant_\ell V_i' \quad L_i \leqslant_\ell L_i'}{[\mathtt{l}_i\_\{\mathtt{p} : V_i, \mathtt{c} : L_i\}]_{i\in I} \leqslant_\ell [\mathtt{l}_i\_\{\mathtt{p} : V_i', \mathtt{c} : L_i'\}]_{i\in I\cup J}}\,[\leqslant_\ell\text{-VR}] \qquad \frac{U\{{}^{\mu_X.U}/_X\} \leqslant_\ell U'}{\mu_X.U \leqslant_\ell U'}\,[\leqslant_\ell\text{-}\mu\text{L}] \qquad \frac{U \leqslant_\ell U'\{{}^{\mu_X.U'}/_X\}}{U \leqslant_\ell \mu_X.U'}\,[\leqslant_\ell\text{-}\mu\text{R}]$$

The rules in Theorem 7.4 are standard: they include the subtyping for variants and records (rule $[\leqslant_\ell\text{-VR}]$) and left/right recursion ($[\leqslant_\ell\text{-}\mu\text{L}]/[\leqslant_\ell\text{-}\mu\text{R}]$). $[\leqslant_\ell\text{-In}]$ and $[\leqslant_\ell\text{-Out}]$ provide respectively the subtyping for linear inputs (*co*variant w.r.t. the subtyping of carried types) and outputs (which is instead *contra*variant): note that they are matched by the variances of `In[·]`/`Out[·]` (Fig. 6, left). By $[\leqslant_\ell\text{-End}]$, $\bullet$ is the only subtype of itself. $[\leqslant_\ell\text{-}\mathbb{B}]$ extends $\leqslant_\ell$ 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: $\overline{?(U)} = !(U)$; $\overline{!(U)} = ?(U)$; $\overline{\bullet} = \bullet$.

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:

$$\mathrm{act}(\&_{i\in I}?\mathtt{l}_i(T_i).S_i) = ? \qquad \mathrm{act}(\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i) = ! \qquad \mathrm{act}(\mu_X.S) = \mathrm{act}(S)$$

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

$$\begin{array}{ll}
[\![\&_{i\in I}?\mathtt{l}_i(T_i).S_i]\!]_\Gamma = ?([\mathtt{l}_i\_\{\mathtt{p} : [\![T_i]\!], \mathtt{c} : [\![S_i]\!]_\Gamma\}]_{i\in I}) & [\![\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i]\!]_\Gamma = !\left(\left[\mathtt{l}_i\_\left\{\mathtt{p} : [\![T_i]\!], \mathtt{c} : \overline{[\![S_i]\!]_\Gamma}\right\}\right]_{i\in I}\right) \\
[\![\&_{i\in I}?\mathtt{l}_i(T_i).S_i]\!]^\mu_\Gamma = \quad [\mathtt{l}_i\_\{\mathtt{p} : [\![T_i]\!], \mathtt{c} : [\![S_i]\!]_\Gamma\}]_{i\in I} & [\![\oplus_{i\in I}!\mathtt{l}_i(T_i).S_i]\!]^\mu_\Gamma = \quad \left[\mathtt{l}_i\_\left\{\mathtt{p} : [\![T_i]\!], \mathtt{c} : \overline{[\![S_i]\!]_\Gamma}\right\}\right]_{i\in I} \\
[\![\mu_X.S]\!]_\Gamma = \mathrm{act}(S)\left(\mu_X.[\![S]\!]^\mu_{\Gamma\{\mathrm{act}(S)(X)/X\}}\right) & [\![\mathbf{end}]\!]_\Gamma = \bullet \\
[\![\mu_X.S]\!]^\mu_\Gamma = \qquad\qquad \mu_X.[\![S]\!]^\mu_{\Gamma\{\mathrm{act}(S)(X)/X\}} & [\![X]\!]_\Gamma = \Gamma(X) \qquad [\![T]\!]_\Gamma = T \text{ (if } T \in \mathbb{B}\text{)}
\end{array}$$

The *encoding of* S *into a linear type* is $[\![S]\!]_\varnothing$, also abbreviated $[\![S]\!]$.

Theorem 7.6 is inductively defined on S. Intuitively, it turns **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 Γ, and using the auxiliary encoding $[\![\cdot]\!]^\mu_\Gamma$. E.g., let $S = \mu_X.?\mathtt{A}.X$: $[\![S]\!]_\Gamma$ gives the type $?(\mu_X.U)$, with U obtained by letting $\Gamma' = \Gamma\{{}^{?(X)}/_X\}$, and $U = [\![?\mathtt{A}.X]\!]^\mu_{\Gamma'} = [\mathtt{A}\_\{\mathtt{p} : \mathsf{Unit}, \mathtt{c} : [\![X]\!]_{\Gamma'}\}] = [\mathtt{A}\_\{\mathtt{p} : \mathsf{Unit}, \mathtt{c} : ?(X)\}]$ (see Theorem 7.11).

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] $\overline{[\![S]\!]} = [\![\overline{S}]\!]$, and $S \leqslant S'$ iff $[\![S]\!] \leqslant_\ell [\![S']\!]$.

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* $\mathcal{N}$ is a partial function from (possibly open) session types to Scala class names. $\mathcal{N}$ *is suitable for* S iff *(i)* $\mathrm{dom}(\mathcal{N})$ 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. $\forall S' \in \mathrm{dom}(\mathcal{N}) : \mathcal{N}(S') = \mathcal{N}(\overline{S'})$, and *(v)* if $\mathcal{N}(\mu_X.S')$ is defined, then $\mathcal{N}(\mu_X.S') = \mathcal{N}(X) = \mathcal{N}(S')$.

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 $\overline{L}$ (by Def. 7.5), corresponding to $\overline{S}$ (by § 7.2): it will produce both the type parameter of `In`/`Out` above, and the *CPSP classes* of $S/\overline{S}$.

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* $\mathcal{M}$ is a partial function from (possibly open) variant types to Scala class names. $\mathcal{M}$ *is suitable for* L iff $\mathrm{dom}(\mathcal{M})$ 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 $\mathcal{M}(\mu_X.U)$ is defined, then $\mathcal{M}(\mu_X.U) = \mathcal{M}(X) = \mathcal{M}(U)$. Given $\mathcal{M}$ suitable for L, we define the *encoding of* L *into Scala types w.r.t.* $\mathcal{M}$, written $\langle\!\langle L \rangle\!\rangle_{\mathcal{M}}$, as:

$$\langle\!\langle ?(U) \rangle\!\rangle_{\mathcal{M}} = \texttt{In}[\mathcal{M}(U)] \qquad \langle\!\langle !(U) \rangle\!\rangle_{\mathcal{M}} = \texttt{Out}[\mathcal{M}(U)] \qquad \langle\!\langle \bullet \rangle\!\rangle_{\mathcal{M}} = \texttt{Unit} \qquad \langle\!\langle V \rangle\!\rangle_{\mathcal{M}} = V \ (\text{if } V \in \mathbb{B})$$

$$\langle\!\langle U \rangle\!\rangle_{\mathcal{M}} = \left[\begin{array}{ll} \multicolumn{2}{l}{\texttt{case class l (p:}\langle\!\langle V \rangle\!\rangle_{\mathcal{M}}\texttt{)(val cont:}\langle\!\langle L \rangle\!\rangle_{\mathcal{M}}\texttt{)}} \\ \langle\!\langle U' \rangle\!\rangle_{\mathcal{M}} & \text{if } U' = \mathrm{carr}(V) \\ \langle\!\langle U'' \rangle\!\rangle_{\mathcal{M}} & \text{if } U'' = \mathrm{carr}(L) \end{array}\right. \qquad \text{if } U = [\texttt{l\_}\{\texttt{p}:V, \texttt{c}:L\}]$$

$$\langle\!\langle U \rangle\!\rangle_{\mathcal{M}} = \left[\begin{array}{l} \texttt{sealed abstract class } \mathcal{M}(U) \\ \left.\begin{array}{ll} \multicolumn{2}{l}{\texttt{case class l}_i\texttt{ (p:}\langle\!\langle V_i \rangle\!\rangle_{\mathcal{M}}\texttt{)(val cont:}\langle\!\langle L_i \rangle\!\rangle_{\mathcal{M}}\texttt{) extends } \mathcal{M}(U)} \\ \langle\!\langle U' \rangle\!\rangle_{\mathcal{M}} & \text{if } U' = \mathrm{carr}(V_i) \\ \langle\!\langle U'' \rangle\!\rangle_{\mathcal{M}} & \text{if } U'' = \mathrm{carr}(L_i) \end{array}\right\}_{i \in I} \end{array}\right. \quad \begin{array}{l} \text{if } U = [\texttt{l}_i\texttt{\_}\{\texttt{p}:V_i, \texttt{c}:L_i\}]_{i \in I} \\ \text{and } |I| > 1 \end{array}$$

$$\langle\!\langle \mu_X.U \rangle\!\rangle_{\mathcal{M}} = \langle\!\langle U \rangle\!\rangle_{\mathcal{M}} \qquad \langle\!\langle X \rangle\!\rangle_{\mathcal{M}} = \mathcal{M}(X)$$

The encoding in Theorem 7.8 is inductively defined on L. The first 3 cases turn a top-level $?(\cdot)/!(\cdot)/\bullet$ into a corresponding `In[·]`/`Out[·]`/`Unit` type in Scala, and the 4th 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 $\mathcal{M}$ maps to U. The remaining cases of Theorem 7.8 show how U originates the session protocol classes. Singleton variants are turned into `case class`es, while non-singleton variants are turned into `sealed abstract class`es (with a name given by $\mathcal{M}$), extended by one `case class` per label. Note that if the `p` field of a variant consists in some linear type $?(U')/!(U')$, the CPSP 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, $\mathcal{M}(\mu_X.U) = \mathcal{M}(X) = \mathcal{M}(U)$: hence, X is encoded as $\mathcal{M}(X) = \mathcal{M}(\mu_X.U)$.

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, $S \vdash S' \mapsto U'$) iff, for some Γ, the computation of $[\![S]\!]$ involves either (a) an instance of $[\![S']\!]_{\Gamma}$ returning $?(U')$ or $!(U')$, or (b) an instance of $[\![S']\!]_{\Gamma}^{\mu}$ 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:

$$[\![\mathcal{N}]\!]_S(U) \;=\; \texttt{A} \qquad \text{iff} \qquad \exists S'\colon S \vdash S' \mapsto U \;\text{ and }\; \mathcal{N}(S') = \texttt{A}$$

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

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 $\langle\!\langle S\rangle\!\rangle_{\mathcal{N}} = \langle\!|[\![S]\!]|\!\rangle_{[\![\mathcal{N}]\!]_S}$, and the *protocol classes of* S as: $\text{prot}\langle\!\langle S\rangle\!\rangle_{\mathcal{N}} = \langle\!|\text{carr}([\![S]\!])|\!\rangle_{[\![\mathcal{N}]\!]_S}$.

Theorem 7.10 gives us two pieces of information: $\langle\!\langle S\rangle\!\rangle_{\mathcal{N}}$ is the type $\texttt{In[}\cdot\texttt{]}/\texttt{Out[}\cdot\texttt{]}/\texttt{Unit}$ on which a Scala program can communicate according to S, and $\text{prot}\langle\!\langle S\rangle\!\rangle_{\mathcal{N}}$ 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:

$$[\![S_h]\!] \;=\; !(U_h) \qquad \text{where } U_h \;=\; \mu_X.\begin{bmatrix} \texttt{Greet\_}\left\{\mathsf{p} : \mathsf{String}, \mathsf{c} : !\left(\begin{bmatrix} \texttt{Hello\_}\{\mathsf{p} : \mathsf{String}, \mathsf{c} : !(X)\}, \\ \texttt{GoodNight\_}\{\mathsf{p} : \mathsf{String}, \mathsf{c} : \bullet\} \end{bmatrix}\right)\right\}, \\ \texttt{Quit\_}\{\mathsf{p} : \mathsf{Unit}, \mathsf{c} : \bullet\} \end{bmatrix}$$

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

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

Now, we can verify that the following mappings hold:

$$S_h \vdash \left(\begin{array}{l} !\texttt{Greet}(\mathsf{String}).\left(\begin{array}{l} ?\texttt{Hello}(\mathsf{String}).X \\ \&\ ?\texttt{Bye}(\mathsf{String}).\mathbf{end} \end{array}\right) \\ \oplus\ !\texttt{Quit}(\mathsf{Unit}) \end{array}\right) \mapsto \begin{bmatrix} \texttt{Greet\_}\left\{\mathsf{p} : \mathsf{String}, \mathsf{c} : !\left(\begin{bmatrix} \texttt{Hello\_}\{\mathsf{p} : \mathsf{String}, \mathsf{c} : !(X)\}, \\ \texttt{Bye\_}\{\mathsf{p} : \mathsf{String}, \mathsf{c} : \bullet\} \end{bmatrix}\right)\right\}, \\ \texttt{Quit\_}\{\mathsf{p} : \mathsf{Unit}, \mathsf{c} : \bullet\} \end{bmatrix}$$

$$S_h \vdash S_h \mapsto U_h \qquad S_h \vdash X \mapsto X \qquad S_h \vdash \left(\begin{array}{l} ?\texttt{Hello}(\mathsf{String}).X \\ \&\ ?\texttt{Bye}(\mathsf{String}).\mathbf{end} \end{array}\right) \mapsto \begin{bmatrix} \texttt{Hello\_}\{\mathsf{p} : \mathsf{String}, \mathsf{c} : !(X)\}, \\ \texttt{Bye\_}\{\mathsf{p} : \mathsf{String}, \mathsf{c} : \bullet\} \end{bmatrix}$$

Hence, by Theorem 7.9, $[\![\mathcal{N}]\!]_{S_h}$ maps the first, second and third (recursive) variant types above to $\texttt{Start}$, and the last one to $\texttt{Greeting}$. The encoding $\langle\!\langle S_h\rangle\!\rangle_{\mathcal{N}} = \langle\!|[\![S_h]\!]|\!\rangle_{[\![\mathcal{N}]\!]_{S_h}}$ is $\texttt{Out[Start]}$, while $\text{prot}\langle\!\langle S_h\rangle\!\rangle_{\mathcal{N}} = \langle\!|\text{carr}([\![S_h]\!])|\!\rangle_{[\![\mathcal{N}]\!]_{S_h}}$ 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). Letthe *dual of a Scala type* be $\overline{\texttt{In[A]}} = \texttt{Out[A]}$, $\overline{\texttt{Out[A]}} = \texttt{In[A]}$, and $\overline{\texttt{Unit}} = \texttt{Unit}$.

For all S, $\langle\!\langle \overline{S}\rangle\!\rangle_{\mathcal{N}} = \overline{\langle\!\langle S\rangle\!\rangle_{\mathcal{N}}}$ and $\text{prot}\langle\!\langle S\rangle\!\rangle_{\mathcal{N}} = \text{prot}\langle\!\langle \overline{S}\rangle\!\rangle_{\mathcal{N}}$.
§ 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 $<:$ 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 $\texttt{B}$, and write a program using $\texttt{A}$ such that $\texttt{A} <: \texttt{B}$ or $\texttt{B} <: \texttt{A}$: by § 7.3, this is sound. For all $\texttt{A}, S, \mathcal{N}$, $\texttt{A} <: \langle\!\langle S\rangle\!\rangle_{\mathcal{N}}$ implies one of the following:

(a1) $S = \mathbf{end}$, and: $\texttt{A} <: \texttt{Unit}$ and $\forall \texttt{B}\colon \texttt{A} \notin \{\texttt{In[B]}, \texttt{Out[B]}\}$;
(a2) $\text{act}(S) = ?$, and: $\texttt{A} <: \texttt{Null}$ or $\exists \texttt{B}\colon \texttt{A} = \texttt{In[B]}$ and ($\texttt{Null} \lneq\colon \texttt{B}$ implies $\exists S', \mathcal{N}' : \texttt{A} = \langle\!\langle S'\rangle\!\rangle_{\mathcal{N}'}$ and $S' \leqslant S$);
(a3) $\text{act}(S) = !$, and: $\texttt{A} <: \texttt{Null}$ or $\exists \texttt{B}\colon \texttt{A} = \texttt{Out[B]}$ and ($\texttt{B} \lneq\colon \texttt{AnyRef}$ implies $\texttt{A} = \langle\!\langle S\rangle\!\rangle_{\mathcal{N}}$).

Moreover, for all $\texttt{A}, S, \mathcal{N}$, $\langle\!\langle S\rangle\!\rangle_{\mathcal{N}} <: \texttt{A}$ implies one of the following:

(b1) $S = \textbf{end}$, and: `Unit <: A` and $\forall$`B`: `A` $\notin$ {`In[B]`, `Out[B]`};
(b2) $\text{act}(S) = ?$, and: `AnyRef <: A` or $\exists$`B`: `A = In[B]` and (`B` $\lneq$: `AnyRef` implies `A` $= \langle\!\langle S \rangle\!\rangle_{\mathcal{N}}$);
(b3) $\text{act}(S) = !$, and: `AnyRef <: A` or $\exists$`B`: `A = Out[B]` and (`Null` $\lneq$: `B` implies $\exists S', \mathcal{N}'$: `A` $= \langle\!\langle S' \rangle\!\rangle_{\mathcal{N}'}$ and $S \leqslant S'$).

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 `In[Full] <: In[WaitingRoom]`, reflecting the fact that `?Full` $\leqslant S_{\text{cstm}}$ (by [$\leqslant$-Ext]). For item b3, consider Theorem 4.2: we have `Out[Start] <: Out[Quit]`, reflecting the fact that $S_{\text{h}} \leqslant$ `!Quit` (by [$\leqslant$-μL] and [$\leqslant$-Int]). § 7.3 also says that `<:` is stricter than $\leqslant$ – 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 `Null` cases (items a2, a3, b3), it is possible e.g. to write a method `f` with a parameter of type `In[Any]` (b2), or `In[Nothing]` (a2), or `Out[Any]` (a3), or `Out[Nothing]` (b3). This does not compromise safety/exhaustiveness, either: `In[Any]` makes `f` accept any message, `Out[Nothing]` forbids `f` to send, while `In[Nothing]`/`Out[Any]` are subtypes of all `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 `In[+A]`/`Out[-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 π-calculus (called session-π 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 π-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 with 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-π, 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 π-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 class`es (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 session 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_{rctl} 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] `ActorRef`s (allowing 0 or more outputs of the same type) from *linear* `ActorRef`s – 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

1 Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. Constructing call graphs of Scala programs. In *ECOOP*, 2014. `doi:10.1007/978-3-662-44202-9_3`.

2 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014. `doi:10.1145/2660193.2660216`.

3 Stephanie Balzer and Frank Pfenning. Objects as session-typed processes. In *AGERE!*, 2015. `doi:10.1145/2824815.2824817`.

4 Hendrik Pieter Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1985.

5 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, 2010. `doi:10.1007/978-3-642-15375-4_16`.

6 Ornela Dardha. Recursive session types revisited. In *BEAT*, 2014. `doi:10.4204/EPTCS.162.4`.

7 Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*. Phd thesis, Università degli studi di Bologna, May 2014.

8 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, 2012. `doi:10.1145/2370776.2370794`.

9 Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, 2011.

10 Edsger W Dijkstra. *Cooperating sequential processes*. Springer, 1965.

11 Juliana Franco and Vasco Thudichum Vasconcelos. A concurrent programming language with refined session types. In *SEFM*, 2013. `doi:10.1007/978-3-319-05032-4_2`.

12 Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *ESOP*. Springer Berlin Heidelberg, 1999. `doi:10.1007/3-540-49099-X_6`.

13 Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 2005. `doi:10.1007/s00236-005-0177-z`.

14 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1), January 2010. `doi:10.1017/S0956796809990268`.

15 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.

16 Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and Promises. URL: `http://docs.scala-lang.org/overviews/core/futures.html`.

17 Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From Akka to TAkka. In *SCALA'14*, 2014. `doi:10.1145/2637647.2637651`.

18 Kohei Honda. Types for dyadic interaction. In *CONCUR*, 1993. `doi:10.1007/3-540-57208-2_35`.

19 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998. `doi:10.1007/BFb0053567`.

20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, 2008. `doi:10.1145/1328438.1328472`.

21 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *ECOOP*, 2010. `doi:10.1007/978-3-642-14107-2_16`.

22 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, 2016.

23 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP*, 2008. `doi:10.1007/978-3-540-70592-5_22`.

24 Shams M. Imam and Vivek Sarkar. Savina – an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents &#38; Decentralized Control*, AGERE!, 2014. `doi:10.1145/2687357.2687368`.

25 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP, 2015. `doi:10.1145/2808098.2808100`.

26 Roland Kuhn. Project Gålbma, actors vs types, 2015. Slides (available on `slideshare.net`).

27 Lightbend, Inc. Actor paths, 2016. `http://doc.akka.io/.../addressing.html`.

28 Lightbend, Inc. The Akka toolkit and runtime, 2016. URL: `http://akka.io/`.

29 Lightbend, Inc. Akka Typed, 2016. `http://doc.akka.io/.../typed.html`.

30 Lightbend, Inc. The Scala IDE, 2016. URL: `http://scala-ide.org/`.

31 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Inf. & Comput.*, 1992. `doi:10.1016/0890-5401(92)90008-4`.

32 Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, 2004. `doi:10.1007/978-3-540-24836-1_5`.

33 Luca Padovani. A Simple Library Implementation of Binary Sessions. `hal:01216310`, 2015.

34 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Haskell*, 2008. `doi:10.1145/1411286.1411290`.

35 Tiark Rompf and Nada Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University and EPFL, 2015. Unpublished. `http://arxiv.org/abs/1510.05216`. `arXiv:1510.05216`.

36 Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes.* Cambridge University Press, 2003.

37 K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. Efficient session type guided distributed interaction. In *COORDINATION*, 2010. `doi:10.1007/978-3-642-13414-2_11`.

38 K. C. Sivaramakrishnan, Mohammad Qudeisat, Lukasz Ziarek, Karthik Nagaraj, and Patrick Eugster. Efficient sessions. *Sci. Comput. Program.*, 78(2), 2013. `doi:10.1016/j.scico.2012.03.004`.

39 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, 1994. `doi:10.1007/3-540-58184-7_118`.

40 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *ESOP*, 2013. `doi:10.1007/978-3-642-37036-6_20`.

41 Matías Toro and Éric Tanter. Customizable gradual polymorphic effects for Scala. In *OOPSLA*, 2015. `doi:10.1145/2814270.2814315`.

42 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. & Comput.*, 217, 2012. `doi:10.1016/j.ic.2012.05.002`.

43 Philip Wadler. Propositions as sessions. In *ICFP*, 2012. `doi:10.1145/2364527.2364568`.

44 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC*, 2013. `doi:10.1007/978-3-319-05119-2_3`.