Lightweight Session Programming in Scala*

Alceste Scalas and Nobuko Yoshida
Imperial College London, UK — \{a.scalas, 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.

Last updated May 3, 2016

Keywords and phrases session types, Scala, concurrency

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 on the right: it is based on an example of “actor protocol” from [27] (slide 42), and schematises the authentication procedure of an application server. A client connects to a frontend, trying to retrieve an active session by its Id; the frontend queries the application server: if Id is valid, the client gets an Active(S) message with a session handle S, which can be used to perform the command/response loop at the bottom; otherwise, the

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

© Alceste Scalas, Nobuko Yoshida licensed under Creative Commons License CC-BY
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, thus reducing software maintenance costs.

### CPS protocols in Scala

The developers of the Scala-based Akka framework [30] 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 of communication errors. Their tentative solution has two parts. The first is Akka Typed [31]: 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 [27], slide 41) shows the CPSPs of the client in Fig. 1.

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

### Opportunities and limitations

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

**Our contribution: lightweight session programming in Scala** We address the challenges and limitations above by proposing lightweight session programming in Scala — where “lightweight” means that our proposal does not depend on language extensions, nor external tools, nor specific message transport frameworks. We generalise the idea of CPSP, relating it to a well established formalism for the static verification of concurrent programs: session types [19, 20, 41]. We present a library, called lchannels, offering a simplified API for session programming with CPSPs, supporting network-transparent communication. Albeit the Scala type checker does not cater for all the static guarantees provided by session-typed languages (mostly due to the lack of static linearity checks), we show that lchannels and CPSPs allow to represent protocol specifications as Scala types, and write session-based programs in a rather natural way, guaranteeing protocol safety: i.e., once a session starts, no out-of-protocol messages can be sent, and all valid incoming messages are handled. We show that typical protocol errors are detected at compile-time — except for linearity errors: lchannels checks them at runtime, reminding the typical usage of Scala Promises/Futures.

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

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

**Online resources** For the latest version of lchannels, visit:

http://alcestes.github.io/lchannels/
Programming with session types: background and challenges

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

2.1 Background: binary session types in a nutshell

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

$$S ::= \{\&_{i \in I} !1_i(T_i).S_i \mid \oplus_{i \in I} !1_i(T_i).S_i \mid \mu_X : S \mid X \mid \text{end} \}$$

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

Remark 2.1. We use $\oplus/\&$ as infix operators, omitting them in singleton choices. We often omit $\text{end}$ and $\text{Unit}$: $\{!\text{Int} \oplus !\text{Unit}\}$ stands for $\{\&_X \{?\text{Int}, \oplus !\text{Unit} \}\}$. For example, the type $S_h$ below describes the client endpoint of a “greeting protocol”:

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

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

Programming languages that support session types are usually based on session-$\pi$ — i.e., a version of $\pi$-calculus [33] extended with session operators. A client respecting $S_h$ would be implemented as $\text{hello}(c)$ in Fig. 4 (left): $c$ is a $S_h$-typed channel endpoint, $!$ is a language primitive for selecting and sending messages, and $?$ for branching (i.e., receiving and pattern matching messages). The type system ensures that $c$ is used according to $S_h$, guaranteeing:

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 ended (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:

$$\overline{\langle \mu_{int} \star \mid ! \langle T \rangle \rangle_S} = \mu_{int} \star \overline{\mid ! \langle T \rangle \rangle_S}$$

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

$$\overline{S_h} = \mu_{int} \star (\mid ? \langle \text{Greet} \rangle \langle \text{String} \rangle, (\mid ! \langle \text{Hello} \rangle \langle \text{String} \rangle, \mid ? \langle \text{Bye} \rangle \langle \text{String} \rangle, \mid ? \langle \text{Quit} \rangle \langle \text{end} \rangle) \rangle \rangle X$$

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

Such a guarantee is made more flexible via session subtyping [13]. Consider the type $S_{h2} = \mid ! \langle \text{Quit} \rangle \rangle$, and its implementation on the right: since $\text{hello2}$ 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 $S_h \preceq S_{h2}^1$—i.e., an $S_h$-typed channel endpoint can always be used in place of an $S_{h2}$-typed one; hence, invoking $\text{hello2}(c)$ is allowed — and such a client program would interact safely and without deadlocks with a server observing $\overline{S_h}$.

2.3 From session-typed to linearly-typed programs

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

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

---

1 This is formalised in §7.1, and proved in Example C.1.
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 \texttt{lHello} is the “linearly encoded” version of \texttt{hello}. Its argument \texttt{c} is a \textit{linear output channel endpoint} that carries a \textit{single} value (whose type is left unspecified, for now). On line 3, it creates a new pair of \textit{linear channels endpoints}, which can carry another single value of some (again unspecified) type: intuitively, what is sent on \texttt{c2out} becomes available in \texttt{c2in}. On line 4, \texttt{c} is used to send a \texttt{Greet} message — which \textit{also carries} \texttt{c2out}. Then, the recipient of \texttt{Greet} and \texttt{c2out} is expected to use the latter to continue the session — i.e., send either \texttt{Hello} or \texttt{Bye}. On line 5, \texttt{c2in} is used to receive such an answer, and the result is matched against \texttt{Hello} and \texttt{Bye}; the latter carries no continuation channel, i.e. the session has ended (line 7); the former, instead, carries a linear (output) channel endpoint \texttt{c3out}, that is used to continue the session with a recursive call (line 6). Note that all channel endpoints received/created in \texttt{lHello} are either used exactly once (\texttt{c}, \texttt{c2in}, \texttt{c3out}), or sent to some other process (\texttt{c2out}).

A crucial difference between \texttt{hello} and \texttt{lHello} is that in the latter, \textit{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 \texttt{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 \textit{linear input and output channels};

P2. we need to suitably \textit{instantiate} each \texttt{?}-type, so to describe the same interactions of \texttt{S_h};

P3. we must \textit{automate} the creation, sending and use of \textit{linear channels}, offering an API that guides the CPS interactions prescribed in [8], and allows to write code similar to \texttt{hello};

P4. we need to handle \textit{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 \textit{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 \textit{S} and \textit{S}'. On the bottom, the same protocol is represented in Scala, as a set of \textit{CPSP classes}, shared between the client and server, and similar to those discussed in §1: they are used as parameters for \texttt{In[A]} and \texttt{Out[A]}, which implement respectively an input/output channel endpoint carrying a \textit{single} value of type \texttt{A}. We extract such CPSP classes from \textit{S} or \textit{S}' , through an \textit{encoding} represented by the arrows; such an encoding exploits an \textit{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.
3.1 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. 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, 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 Example 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. 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

---

2 This matches the output subtyping rule [κ_out] in Def. 7.4.
3 Such a requirement could be enforced by defining the field as val, instead of def; the drawback is that val does not allow type parameters, and this would result in an invariant Out with limited subtyping.
4 This matches the input subtyping rule [κ_in] in Def. 7.4.
message before raising a timeout error. The \(?\) method implements the typical \textit{branching}
operator of session-\(\pi\): it takes a function \(f: \mathit{A} \Rightarrow \mathit{B}\), and once a value \(v\) is \textit{received}, it
returns \(f(v)\). The rationale behind the method signature is clarified in Example 3.1.

\textbf{Example 3.1} (\(!\), \(?\) and selection/branching). Consider the following classes:

\begin{verbatim}
sealed abstract class AorB
case class A() extends AorB; case class B() extends AorB
\end{verbatim}

Let \(c\) be an instance of \textit{Out}[\textit{AorB}]. The \(c.\!\) method can be used as follows:

\begin{verbatim}
c ! A() or c ! B()
\end{verbatim}

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 \textit{AorB}, — e.g., \(\mathit{A}\) or
\(\mathit{B}\): this corresponds, in session-\(\pi\), to the type checking of an internal choice.

Let now \(c\) be an instance of \textit{In}[\textit{AorB}]. The \(c.?\) method can be used as shown below,

\begin{verbatim}
c ? { case A() => println("Got A") case B() => println("Got B") }
\end{verbatim}

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

\textbf{Using \texttt{lchannels} endpoints: static vs. dynamic checks} As seen in Example 3.1, the
Scala compiler can check that an instance of \texttt{lchannels Out} (resp. \texttt{In}) carrying a \textit{sealed
abstract class} is only used under the \textit{safety} and \textit{exhaustiveness} guarantees of a session-
typed channel endpoint with a top-level \& (resp. \(\oplus\))\footnote{This arises from the encoding of session types into linear I/O types with \textit{variants} \cite{8}: we render the \textit{latter} in Scala as \textit{sealed case classes} (as detailed in §7.3).}, i.e., \textbf{S1} and \textbf{S2} in §2.1. Also, an
instance of e.g. \textit{Unit} provides the guarantees of an \textit{end}-typed channel endpoint: it cannot be used for I/O. Unfortunately, the Scala type checker cannot enforce \textit{input/output linearity} (\textbf{S3} and \textbf{S4} in §2.1): hence, \texttt{lchannels} implements the following \textit{runtime linear usage rules}:

\textbf{L1.} each \texttt{Out} instance should be used to perform \textit{exactly one} output. Any further output
will generate a \textit{runtime exception}, forbidding duplicated message transmissions;

\textbf{L2.} each \texttt{In} instance should be used \textit{at least once}. Each use will \textit{retrieve the same value}.

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

\subsection{A local implementation}

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

\footnote{Due to Java legacy, in Scala also \texttt{Null} is a subtype of \texttt{AorB}. This will be explicit in Theorem 7.14.}

\footnote{By design, Scala does not enforce matching on \texttt{null} values, albeit they might be received (see note 5).}
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:

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

Here, `p1` and `p2` are functions taking respectively an input and output channel endpoint carrying `A`, and returning resp. `B1` and `B2`. The `parallel` method creates a pair of `A`-carrying local channel endpoints (line 2), applies `p1` and `p2` on them by spawning separate threads, and returns a pair of `Future` 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 Example 3.2. However, `Promise/Future` instances cannot be serialised, and thus cannot be sent/received over a network: this makes `LocalIn` and `LocalOut` unsuitable for distributed applications. We address this issue later on, in §5.

4 Session programming with `lchannels` and CPS protocols

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

4.1 Representing sequential inputs/outputs

Let us consider the session type `S_{QR} = ?Q(Bool).!R(Int)`, dictating that a channel endpoint must be used first to receive `Q(Bool)`, and then to output `R(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_{QR}`, but, how do we require to send a value of type `R` along the same interaction?

```
case class Q(p: Boolean, cont: Out[R])
case class R(p: Int)
```

![Figure 7](image)

Figure 7 `S_{QR}` and `S_{QR}` in Scala.

Now, consider the dual `S_{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_{QR} and \overline{S}_{QR}. In[Q] is the Scala representation of S_{QR}, while Out[Q] is the representation of \overline{S}_{QR}. We can outline a development approach for session-based applications. For each communication channel:

D1. formalise the two endpoint session types S and \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

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\(^8\). 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), 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:

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

where the type annotation is necessary due to the limited type inference capabilities of Scala\(^9\).

\(^8\) The pattern actually reflects how session-\(\pi\) processes are encoded in standard \(\pi\)-calculus (§2.3).

\(^9\) This limitation is present in Scala 2.11.8, but might be overcome in future versions.
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.

### 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 (Example 4.1), non-singleton external/internal choices (Example 4.2), and multiple channels with higher-order types for *session delegation* (Example 4.3).

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

$S_{\text{fifo}} = \mu X. !\text{Datum}(T) . X$ (sending endpoint) $\quad S_{\text{fifo}} = \mu X.?\text{Datum}(T) . X$ (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 $\mu X$… (i.e., Datum), and (ii) continuing with such a name when $X$ occurs (for another case of recursion, see Example 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)
}
```

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

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

- Start for the internal choice of $S_h$ (i.e., the external choice of $S_b$) between Greet, Quit;
- Greeting for the external choice of $S_h$ (i.e., the internal choice of $S_b$) between Hello, Bye.

We obtain the CPSP classes on the right, with $\text{Out[Start]}/\text{In[Start]}$ representing $S_h/S_b$ (by D3). We can write two endpoint processes as:

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

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

```
def server(c: In[Start]): Unit = {
  c ? {
    case m @ Greet(whom) => {
      val c2m = m.cont :: Hello(whom)
      server(c2m)
    } case Quit() => ()
  }
}
```
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 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 at a time), and sleeps again when nobody is waiting. We model this scenario with three components: the customer, the shop and the barber, using session types to formalise their expected interactions, schematised below.

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

\[
S_{\text{cstm}} = \text{Full}?\text{Seat}?\text{Ready}.S_{\text{cut}}
\]

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 \( S_{\text{cut}} \): describes the new hairdo, waits for the Haircut, Pays and leaves. The shop uses the other, dually-typed channel endpoint:

\[
S_{\text{cut}} = \text{Descr(String)}!?\text{Descr(String)}\text{Haircut}!\text{Pay(Int)}
\]

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

\[
S_{\text{barber}} = \mu X.\text{Available}?\text{Serve}(S_{\text{cut}}).
\]

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

Meanwhile, the barber uses its \( S_{\text{barber}} \)-typed channel endpoint to notify that he is Available, and wait for a Serve message — sleeping until he gets one; when it happens, the barber gets a \( S_{\text{cut}} \)-typed channel endpoint in the message payload: he is expected to use it for interacting with the customer, i.e., listen for the hairdo Description, perform the Haircut, and take the Payment. When the customer session terminates, the barber must resume his recursive session with the shop: he notifies that he is Available again, etc.

The CPSP classes extracted from the session types above are shown on the right. As per item D2b of § 4.2, we introduce WaitingRoom as the sealed abstract class corresponding to the external (resp. internal) choice between Full and Seat in \( S_{\text{cstm}} \) (resp. \( S_{\text{cut}} \)).

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 Example 4.1. Once started, Shop creates a $S_{barber}$-typed channel (line 19) and gives the output endpoint to a new Barber (line 20). The enter method returns an input endpoint for interacting according to $S_{cstm}$: after creating two channel endpoints of the suitable type (line 6), enter checks how many people are trying to get a seat, and outputs Full (line 10) or Seat (line 12) before returning the input endpoint (line 15). In the main loop (lines 24–33), the shop waits for an Available message from the barber (line 25), sleeps while retrieving a customer channel from seats (line 26), notifies the customer that the barber is ready, forwards the channel to the barber, and continues its loop.

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

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

As anticipated, our solution for the sleeping barber problem exploits session delegation: the customer starts interacting with the shop, but his session is eventually forwarded to the barber, with a higher-order Serve($S_{cut}$) 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.

---

**Figure 8** Sleeping barber (Example 4.3): shop, barber and customer implementations.
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.\(^\text{10}\) In §3, we introduced the abstract classes In/Out, and LocalIn/LocalOut as simple local implementations for inter-thread communication. The In[[]]/Out[[]] interface can abstract other message transports, allowing lchannels-based programs to achieve faster message delivery, or transparently interact across a network. We discuss 3 examples: queue-, actor- and stream-based channels.

**Optimised queue-based channels** The simple LocalIn/LocalOut classes in Fig. 6 (right) perform all communications through the underlying Future/Promise. However, many applications could mostly use the In.receive/Out.send methods, and could benefit from an optimised implementation of In/Out that (when possible) bypasses In.future/Out.promise. We developed this idea with the QueueIn/QueueOut classes: internally, they deliver messages through Java LinkedTransferQueue (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 [31], 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 (Example 4.2) we can run the former as:

\begin{verbatim}
val (in, out) = ActorChannel.factory[Start]("start"); server(in)
\end{verbatim}

Now, out.path contains the Akka Actor Path [29] of an automatically-generated actor. Such a path can be used, even on a different JVM, to instantiate a proxy for out, as follows:

\begin{verbatim}
val c = ActorOut[Start]("akka.tcp://sys@host.com:5678/user/start"); client(c)
\end{verbatim}

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

\(^{10}\) More features are presented in Appendix A.
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/OutputStream, 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).

For example, suppose that the “greeting protocol” from Example 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\[11\]. 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
val client(c)
```

Note that we did not change the code of client seen in Example 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 Example 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).

\[11\] The implementation of HelloStreamManager is available in Appendix A.1.
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).\footnote{As a minor extension, here we allow a session type payload to be a pair \((T_1, T_2)\). Such an extension could be encoded as a sequence of 2 messages, with \(T_1\) and then \(T_2\) as payloads.}

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

\[
S_{\text{front}} = ℹ\text{getSession}(id).((\text{New}(S_{\text{auth}}) @ \text{Active}(S_{\text{act}})) \cdot S_{\text{auth}} = !\text{Authenticate}(\text{Cred}).(?(\text{Success}(S_{\text{act}}) \& ?\text{Failure}(S_{\text{act}})))
\]

\[
S_{\text{act}} = \mu_X.(\text{Quit} \& ?\text{GetId}(id).X @ !\text{Ping} \cdot \text{Pong} \cdot X @ !\text{Join}(\text{String}).?\text{ChatRoom}((S_{\text{r}}, S_{\text{ctl}})).X)
\]

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

- **New** carries a \text{Sauth}-typed channel endpoint, talking with the auth server: it allows the client to send an Authenticate(\text{Cred}) message (with \text{Cred} being the credentials), and wait for either Success(\text{Sact}) or Failure (the \text{Sact}-typed channel is explained below);

- **Active** carries an \text{Sact}-typed channel endpoint representing the active "session loop" (Fig. 1). When the client receives it, \text{Sact} (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 ChatRoom((S_r, S_ctl)) answer, and the session continues recursively.

The two channels endpoints in the payload allow to interact with the chat room:

- \(S_r = \mu_Y.?(\text{NewMessage}((\text{String}, \text{String})).Y \& ?\text{Quit} \cdot \text{Quit})\). 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);

- \(S_{\text{ctl}} = \mu_Z.!(\text{SendMessage}((\text{String})).Z \& !\text{Quit})\). This endpoint allows the client to send either a message on the chat room (with the payload being the text), or 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\footnote{Their declarations are available in Appendix B.1, Fig. 11 (top).}. In particular, we use Command as the sealed abstract class for the top-level choice in \(S_{\text{act}}\) (this detail will be mentioned again in §6.1.2).

**Internal protocols** Fig. 1 also outlines the internal communications among the frontend, authentication and chat server: they can be formalised as session types, too — as for barbershop interaction in Example 4.3. Here, we only detail the frontend-server interaction type:\footnote{The rest of the internal protocols is described in Appendix B.1.}

\[
S_{\text{FS}} = \mu_X.?(\text{getSession}(id) . ?\text{Success}(S_{\text{act}}).X \& ?\text{Failure}.X)
\]

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

The other declarations are available in Appendix B.1, Fig. 11 (bottom).
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. Example 4.3 is that the delegation appears explicitly in client’s session types, e.g. in **Active** messages with a channel as payload. In CPS protocols, this difference is almost negligible: the **Active** message class\(^\text{15}\) has no continuation, but the client should keep interacting via the **Out** endpoint in the payload — as per rule **L1** in §3.1.

The server-side implementation reuses several solutions from Example 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. Example 4.3 is that requests are queued asynchronously (via **In.future**) and enriched with internal data.

```scala
class ChatServer(...) extends Runnable {
  ...  
  private def createSession(username: String): Out[Command] = {
    val id = allocUniqueSessionId()
    val (in, out) = LocalChannel.factory[Command]()
    in.future.onComplete {
      case Success(cmd) => queueRequest(Success((id, cmd)))
      case Failure(e) => queueRequest(Failure(e))
    }
    // Add the new session to the list of known sessions
    sessions(id) = ... /* session info, including username */
    out
  }
}
```

E.g., the chat server calls the method on the left when the auth server asks to create a new session for **username**: it reserves a session **id** (line 4), creates the channel endpoints **in, out** carrying a **Command** (line 5), keeps **in**, and returns **out** (line 12), that will be the payload of a **NewSession** message. The client **Command** is received asynchronously via **in.future**: in lines 6–9, **cmd** is paired with the session **id**, and queued (line 7). When the pair is later dequeued and processed, **id** tells on which session **cmd** is acting. A similar queuing is performed as the session progresses; e.g., when a **cmd** of type **Ping** is dequeued, the server runs:

```scala
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 [32]) provide channel usage errors and hints, e.g. via typing information and auto-completion suggestions.

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

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

**Unused channel endpoints** Not performing an output can leave a process at the other endpoint stuck, waiting for input — and this could escalate to other processes waiting on other channels; this problem can also arise if a program does not input a message whose continuation/payload is an **output** channel. Spotting this kind of errors can be tricky,

\(^{15}\)See Appendix B.1, Fig. 11 (top), line 7.
especially if channels are dynamically generated, sent, received, stored in collections (as in our case study). \texttt{lcchannels} 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 \texttt{lcchannels} impacts communication speed w.r.t. other inter-thread communication methods: Fig. 9 shows the results. The benchmarks are mainly inspired by [25]; “Streaming” is a parallel blend of “Ring” + “Counting actor”: 16 threads are connected in a ring and a sequence (“stream”) of messages is sent \textit{at once}, measuring the time required for \textit{all} to complete one loop.

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

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

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

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

Notably, the usual “non-CPS” communication we implemented (and measured) over Scala channels / Java queues requires connecting pairs of threads $P_1, P_2$ with pairs of queues (one carrying messages from $P_1$ to $P_2$, the other from $P_2$ to $P_1$). Such queues have type \texttt{Queue[A]}, where \textit{A} must cover \textit{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 \textit{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 $\mu$ binds a distinct variable that actually occurs in its scope. Restriction (i) allows to directly generate a Scala case class from each internal/external choice label. Restriction (ii) is a form of Ottmann/Barendregt’s variable convention [4].

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

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

\[
\begin{align*}
& \forall i \in I : \quad T_i \leq T'_i \quad S_i \leq S'_i \quad \text{[c-Ext]} & \forall i \in I : \quad T'_i \leq T_i \quad S_i \leq S'_i \quad \text{[c-Ext]} \\
& \alpha \leq \beta \quad \text{[c-Ext]} & \alpha \leq \beta \quad \text{[c-Ext]} \\
& \text{end} \leq \text{end} \quad \text{[c-End]} & \text{end} \leq \text{end} \quad \text{[c-End]} \\
& S[\mu \times S'] \leq S' \quad \text{[c-L]} & S \leq \mu \times S' \quad \text{[c-R]} & T \leq T' \quad \text{[c-B]}
\end{align*}
\]

Rule $[\text{c-Ext}]$ 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, $[\text{c-Ext}]$ 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. $[\text{c-End}]$ says that a terminated session has no subtapes. $[\text{c-L}]$ and $[\text{c-R}]$ are standard: a recursive type $S$ is related with $S'$ iff its unfolding is related. $[\text{c-B}]$ extends $\leq$ to basic types.

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

In order to encode session types into Scala types, we exploit an intermediate encoding into linear types for input and output [38]. We focus on a subset of such types, defined below.

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

\[
L ::= \ ?(U) \mid ! (U) \mid \bullet U ::= \{ L_i : \mu X . U \mid X \} \mid \mu X . U \mid X \mid V ::= \mathbb{B} \mid L \text{ (closed)}
\]
where (i) recursion is guarded, and (ii) all \( \lambda_1 \) range over pairwise distinct labels. We also define the carried type of \( L \) as \( \text{carr}(?U) = \text{carr}(l(U)) = U \).

\(?U\) (resp. \( l(U) \)) 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 \( \lambda_1 \)-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 ([38]).** The subtyping relation \( \preceq_{\ell} \) between linear types is coinductively defined by the following rules (where \( \preceq_{\exists} \) is a subtyping between basic types):

\[
\begin{align*}
\frac{\ U \preceq_{\ell} U'}{\mathcal{L} \preceq_{\ell} \mathcal{L}'}, & \quad \frac{U' \preceq_{\ell} U}{\mathcal{L} \preceq_{\ell} \mathcal{L}'}, & \text{[\( \preceq_{\ell},\text{Out} \)]} \\
\frac{\ \forall \ell \in I: V_\ell \preceq_{\ell} V_\ell'}{\mathcal{L} \preceq_{\ell} \mathcal{L}'}, & \quad \frac{\ ! \mathcal{L} \preceq_{\ell} \mathcal{L}'}{\mathcal{L} \preceq_{\ell} \mathcal{L}'}, & \text{[\( \preceq_{\ell},\text{In} \)]}
\end{align*}
\]

The rules in Def. 7.4 are standard: they include the subtyping for variants and records (rule \([\preceq_{\ell},\text{VR}]) and left/right recursion (\([\preceq_{\ell},\text{L/R}]). [\preceq_{\ell},\text{In}]) and \([\preceq_{\ell},\text{Out}]) provide respectively the subtyping for linear inputs (covariant w.r.t. the subtyping of carried types) and outputs (which is instead contravariant): note that they are matched by the variances of In \([\] / Out \([\) (Fig. 6, left). By \([\preceq_{\ell},\text{Eno}], \bullet \) is the only subtype of itself. \([\preceq_{\ell},\exists]) extends \( \preceq_{\ell} \) 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: \( \overline{?U} = !\mathcal{L} \); \( \overline{!U} = ?U \); \( \bullet = \bullet \).

We now introduce our encoding of channel 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:

\[
\begin{align*}
\text{act}(\mathcal{X}_\iota, ?\mathcal{L}(T), S) & = ?
\text{act}(\mathcal{X}_\iota, !\mathcal{L}(T), S) & = !
\text{act}(\mu_X.S) & = \text{act}(S)
\end{align*}
\]

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

\[
\begin{align*}
[\mathcal{L}_\iota, ?\mathcal{L}(T), S]_\Gamma & = \gamma(\mathcal{L}_\iota, T, \mathcal{L}(T), S) & & [\mathcal{L}_\iota, !\mathcal{L}(T), S]_\Gamma & = \mu_{\mathcal{X}_\iota}(\mathcal{L}_\iota, T, \mathcal{L}(T), S) & & [\mathcal{L}_\iota, S]_\Gamma & = \text{act}(S)
\end{align*}
\]

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

Def. 7.6 is inductively defined on the structure of \( S \). 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 \S 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 Def. 7.5) for continuing the session. Recursion is encoded by turning a recursive external (resp. internal) choice into a linear input (resp. output) carrying a recursive variant: this “structural shift” is achieved by collecting open recursion variables in \( \Gamma \), and using the auxiliary encoding \([\gamma]_\Gamma \). E.g., let \( S = \mu_X.\lambda X. [S]_\Gamma \) gives the type \( ![\mu_X.\lambda X. U] \), with \( U \) obtained by letting \( \Gamma U = \Gamma \{ ![\lambda X.U] \} \), and \( U = ![\lambda X.U]_\Gamma = \{ ![\lambda \mathcal{P} Unit.c : [X]_\Gamma] \} = \{ ![\lambda \mathcal{P} Unit.c : ![\lambda X]] \} \) (see Example 7.12).
Our handling of recursion greatly affects our proofs, and is a main difference between Def. 7.6 and the encoding in [6]. Despite this, the crucial Theorem 7.7 still holds.

**Theorem 7.7 (Encoding preserves duality, subtyping).** \([\mathcal{S}] = [S']\), and \(S \leq S'\) iff \([S] \leq [S']\).

### 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 (Def. 7.8), giving a distinct class name to each subterm of \(S\).

**Definition 7.8.** 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 \(S\) if (i) \(\text{dom}(\mathcal{N})\) contains all subterms of \(S\) (except end), (ii) \(\mathcal{N}\) is injective w.r.t. the internal/external choices in its domain, (iii) maps each singleton internal/external choice to its label, (iv) is dually closed, i.e. \(\forall S' \in \text{dom}(\mathcal{N}): \mathcal{N}(S') = \mathcal{N}(\overline{S'})\), 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 Def. 7.11. 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 1channels \(\text{In}[\_]\) (resp. \(\text{Out}[\_]\)) type;
- \(U\) also appears in the dual \(\overline{L}\) (by Def. 7.5), corresponding to \(\overline{S}\) (by Theorem 7.7): it will produce both the type parameter of \(\text{In}/\text{Out}\) above, and the \(\text{CPSP}\) classes of \(\overline{S}/\overline{S}\).

We first formalise the encoding from linear types to Scala types, in Def. 7.9 below.

**Definition 7.9.** 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 \(\text{dom}(\mathcal{M})\) 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 \(\langle L \rangle_\mathcal{M}\), as:

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

\[
\langle U \rangle_\mathcal{M} =
\begin{align*}
\text{case class } l \ (p: \langle V \rangle_\mathcal{M}) \ (\text{val cont: } \langle L \rangle_\mathcal{M}) & \quad \langle U' \rangle_\mathcal{M} \text{ if } U' = \text{cont}(V) \\
& \quad \langle U'' \rangle_\mathcal{M} \text{ if } U'' = \text{cont}(L) \\
\end{align*}
\]

\[
\langle U \rangle_\mathcal{M} =
\begin{align*}
\text{sealed abstract class } \mathcal{M}(U) & \quad \langle U' \rangle_\mathcal{M} \text{ if } U' = \text{cont}(V) \\
& \quad \langle U'' \rangle_\mathcal{M} \text{ if } U'' = \text{cont}(L) \\
\end{align*}
\]

\[
\langle \mu_X.U \rangle_\mathcal{M} = \langle U \rangle_\mathcal{M} \quad \langle X \rangle_\mathcal{M} = \mathcal{M}(X)
\]

The encoding in Def. 7.9 is inductively defined on the structure of \(L\). The first 3 cases turn a top-level \(?()!/()\bullet\) into a corresponding \(\text{In}[\_]/\text{Out}[\_]/\text{Unit}\) type in Scala, and the 4th case keeps basic types unaltered; note that when encoding \(?()\) (resp. \(!()\)), the type parameter of the resulting \(\text{In}[\_]\) (resp. \(\text{Out}[\_]\)) is the Scala class name that \(\mathcal{M}\) maps to \(U\). The remaining cases of Def. 7.9 show how \(U\) originates the session protocol classes. Singleton variants are turned into case classes, while non-singleton variants are turned into sealed abstract classes (with a name given by \(\mathcal{M}\)), extended by one case class per label. Note that if the \(p\) field of a variant consists in some linear type \(?(U')!/?(U')\), the
In a session type $X$, $\mu_X U$ is handled by noticing that, by Def. 7.8, $M(\mu_X U) = M(X) = M(U)$: hence, $X$ is encoded as $M(X) = M(\mu_X U)$.

The last ingredient for our encoding is a way to turn a nominal environment for a session type (Def. 7.8) into one for a linear type (Def. 7.9): this is formalised below.

**Definition 7.10.** We say that $S$ maps $S'$ to $U'$ (in symbols, $S \vdash S' \to U'$) iff, for some $\Gamma$, the computation of $\llbracket S \rrbracket_\Gamma$ involves either (a) an instance of $\llbracket S' \rrbracket_\Gamma$, returning $?U'$ or $!U'$, or (b) an instance of $\llbracket S' \rrbracket_\Gamma'$ returning $U'$. If $\mathcal{N}$ is suitable for $S$, the linear encoding of $\mathcal{N}$ (w.r.t. $S$) is a nominal environment for linear types denoted with $\llbracket \mathcal{N} \rrbracket_S$, such that:

$$
\llbracket \mathcal{N} \rrbracket_S(U) = A \iff \exists S' : S \vdash S' \to U \quad \text{and} \quad \mathcal{N}(S') = A
$$

Intuitively, Def. 7.10 says that if $\mathcal{N}$ maps an internal/external choice $S'$ to some class name $A$, then $\llbracket \mathcal{N} \rrbracket_S$ maps the variant obtained from the encoding of $S'$ to the same $A$.

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

**Definition 7.11.** Given $\mathcal{N}$ suitable for $S$, we define the encoding of $S$ into a Scala type as $\llbracket S \rrbracket_\mathcal{N} = \llbracket \mathcal{N} \rrbracket_S$, and the protocol classes of $S$ as: $\mathit{prot}(\llbracket S \rrbracket_\mathcal{N}) = \{\mathit{cart}(\llbracket S \rrbracket)\}_{\mathcal{N}}$.

Def. 7.11 gives us two pieces of information: $\llbracket S \rrbracket_\mathcal{N}$ is the type $\mathit{In}[1]/\mathit{Out}[1]/\mathit{Unit}$ on which a Scala program can communicate according to $S$, and $\mathit{prot}(\llbracket S \rrbracket_\mathcal{N})$ gives the definitions of all necessary CPSP classes. Technically, $S$ and $\mathcal{N}$ are first linearly encoded (via Definitions 7.6 and 7.10); then, the result is further encoded into Scala types (via Def. 7.9).

**Example 7.12.** The linear encoding of the greeting session type $S_h$ in §2 is:

$$
\llbracket S_h \rrbracket = \llbracket \langle S_h \rangle \rrbracket = \langle \llbracket S_h \rrbracket \rangle
$$

Let us now define $\mathcal{N}$, as described in Example 4.2, making it suitable for $S_h$ (as per Def. 7.8):

$$
\mathcal{N} = \mathcal{N}(\langle \llbracket S_h \rrbracket \rangle)
$$

Now, we can verify that the following mappings hold:

$$
\begin{align*}
S_h & \vdash S_h \to U_h \quad S_h \to X \to X \\
& \Rightarrow \quad \text{Greeting}
\end{align*}
$$

Hence, by Def. 7.10, $\llbracket \mathcal{N} \rrbracket_{S_h}$ maps the first, second and third (recursive) variant types above to $\mathit{Start}$, and the last one to $\mathit{Greeting}$. The encoding $\langle \llbracket S_h \rrbracket \rangle_\mathcal{N} = \langle \mathit{Out}[\mathit{Start}] \rangle$ is $\mathit{Out}[\mathit{Start}]$, while $\mathit{prot}(\langle \llbracket S_h \rrbracket \rangle)_{\mathcal{N}} = \{\mathit{cart}(\llbracket S_h \rrbracket)\}_{\mathcal{N}}$ gives the Scala protocol classes seen in Example 4.2.

We conclude with two results at the roots of our session-based development approach (§4.2). Similarly to Def. 7.5, let the dual of a Scala type be $\mathit{In}[A] = \mathit{Out}[A]$, $\mathit{Out}[A] = \mathit{In}[A]$, and $\mathit{Unit} = \mathit{Unit}$.

**Theorem 7.13.** For all $S$, $\llbracket S \rrbracket_\mathcal{N} = \overline{\llbracket S \rrbracket}_\mathcal{N}$ and $\mathit{prot}(\llbracket S \rrbracket_\mathcal{N}) = \mathit{prot}(\overline{\llbracket S \rrbracket})_\mathcal{N}$.

We conclude with two results at the roots of our session-based development approach (§4.2). Similarly to Def. 7.5, let the dual of a Scala type be $\mathit{In}[A] = \mathit{Out}[A]$, $\mathit{Out}[A] = \mathit{In}[A]$, and $\mathit{Unit} = \mathit{Unit}$.

**Theorem 7.14.** For all $S$, $\langle S \rangle_\mathcal{N} = \overline{\langle S \rangle}_\mathcal{N}$ and $\mathit{prot}(\langle S \rangle_\mathcal{N}) = \mathit{prot}(\overline{\langle S \rangle})_\mathcal{N}$.
Theorem 7.14. For all $A, S, S'$, $A <: \langle S \rangle_N$ implies one of the following:

(a1) $S = \text{end}$, and: $A <: \text{Unit}$ and $\forall B : \{\text{In}[B], \text{Out}[B]\}$;

(a2) act(S) $\neq ?$, and: $A <: \text{Null}$ or $\exists B : A = \text{In}[B]$ and $\langle \text{Null} \not\leq B \rangle$ implies $\exists S', N' : A = \langle S' \rangle_{N'}$ and $S' \subseteq S$;

(a3) act(S) $= \uparrow$, and: $A <: \text{Null}$ or $\exists B : A = \text{Out}[B]$ and $\langle B \not\leq \text{AnyRef} \rangle$ implies $A = \langle S' \rangle_{N'}$.

Moreover, for all $A, S, S'$, $\langle S \rangle_N <: A$ implies one of the following:

(b1) $S = \text{end}$, and: $\forall B : \{\text{In}[B], \text{Out}[B]\}$;

(b2) act(S) $= \uparrow$, and: $\exists B : A = \text{In}[B]$ and $\langle B \not\leq \text{AnyRef} \rangle$ implies $A = \langle S' \rangle_{N'}$;

(b3) act(S) $= ?$, and: $\exists B : A = \text{Out}[B]$ and $\langle B \not\leq \text{AnyRef} \rangle$ implies $\exists S', N' : A = \langle S' \rangle_{N'}$ and $S \subseteq S'$.

Roughly, Theorem 7.14 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 Example 4.3: we have In[Full] $\not<:$ In[WaitingRoom], reflecting the fact that $?\text{Full} \not\leq S_\text{out}$ (by $\langle <\langle \text{Ext} \rangle \rangle$). For item b3, consider Example 4.2: we have Out[Start] $\not<:$ Out[Quit], reflecting the fact that $S_\text{in} \not\leq \text{Quit}$ (by $\langle <\langle \text{in} \text{Ext} \rangle \rangle$). Theorem 7.14 also says that $<$ is stricter than $\leq$ — 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[Null] (b2), or In[Nothing] (a2), or Out[Any] (a3), or Out[Nothing] (b3). This does not compromise safety/exhaustiveness, either: In[Null] 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, 41, 19], as a typing discipline for a variant of the $\pi$-calculus (called session-$\pi$ in §2). They have been studied and developed in multiple directions during the following decades, notably addressing multiparty interactions [20] and logical interpretations [5, 45]. The encoding of session types in linear $\pi$-calculus types has been studied in [9, 8, 6, 7]; our work is mainly based on [8], but our treatment of recursion is novel (see §7.2).

Session types have been mostly implemented on dedicated programming languages with the advanced type-level features outlined in §2 [14, 45, 11, 42, 3]. [34, 36] aim at an integration with Haskell, using monads to enforce linearity (at the price of a restrictive and rather complicated API). [26] adapts [36] to Rust, exploiting its affine types, but showing limitations to binary internal/external choices. [23, 39, 40] 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-$\pi$, is based on the Scribble tool [46], supports multiparty sessions, and generates classes which represent both a channel endpoint and its protocol; hence, in the binary setting, each endpoint has its own hierarchy of generated classes that is different (but “dual”) w.r.t. the other endpoint. Instead, our I/O endpoints are closer to linear types for the $\pi$-calculus [38]: they take the protocol as a type parameter, from a set of CPSP classes which is common between the two endpoints. Other differences are mostly due to the Java type system, which e.g. does not support case classes (complicating exhaustiveness checks) nor declaration-site variance (complicating the handling of I/O co/contra-variance).
The work closer to ours is [35]: 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.

**Type-safe interaction in Scala** 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 [27]. 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.

**Future work** 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 Example 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. [43] 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, 37]) 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

---

16 Further details on the comparison between this paper and [35] are available in Appendix G.
possible way is abstracting under the \texttt{In[]} \texttt{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 (Def. 7.9) to yield \texttt{ActorRef[A]} types instead of \texttt{Out[A]}, obtaining CPSP classes similar to those in Fig. 2. Remarkably, \texttt{Out[A]} and \texttt{ActorRef[A]} are both contravariant w.r.t. \texttt{A}, and enjoy similar subtyping properties (Theorem 7.14). However:

(i) Akka Typed does not offer an input endpoint similar to \texttt{In[.].} Hence, session types whose CPSPs carry input endpoints (e.g., Example 4.1, or \texttt{S_\textit{ctl}} 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 \texttt{ActorRef[A]} raise no errors when used multiple times for sending messages;

(iii) to produce and send a continuation \texttt{ActorRef[A]}, it is customary to cede the control to another actor (possibly a new one, as in Fig. 3); \texttt{1channels}, instead, encourages the creation and use of I/O endpoints along a single thread, in a simple sequential style.

Item (i) is a minor issue; (ii) could be addressed, taking inspiration from the session/linear types theory, by distinguishing unrestricted [44] \texttt{ActorRefs} (allowing 0 or more outputs of the same type) from linear \texttt{ActorRefs} — with the former usable as the latter, but not \textit{vice versa}. Item (iii) 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.

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

Lightweight Session Programming in Scala

24 A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. TOPLAS, 23(3), May 2001.
Appendices
class HelloStreamManager(in: InputStream, out: OutputStream) extends StreamManager(in, out) {
    private val outb = new BufferedWriter(new OutputStreamWriter(out))
    override def streamer(x: scala.util.Try[scala.Any]) = 
        x match {
            case Failure(e) => close() // StreamManager.close() closes in & out
            case Success(v) => 
                v match {
                    case Greet(name) => outb.write(f"GREET $name
                            "); outb.flush()
                    case Quit() => outb.write("QUIT
                            "); outb.flush(); close() // End
                }
        }
    private val inb = new BufferedReader(new InputStreamReader(in))
    private val helloR = "HELLO (.+)\r\n                            ".r // Matches Hello(name)
    private val byeR = "BYE (.+)\r\n                            ".r // Matches Bye(name)
    override def destreamer() = inb.readLine() match {
        case helloR(name) => Hello(name)(StreamOut[Start](this))
        case byeR(name) => close(); Bye(name) // Session end: close streams
        case e => { close(); throw new Exception(f"Bad message: "; e) }
    }
}

Figure 10 Implementation of HelloStreamManager.

A lchannels: further comments and advanced features

A.1 Stream-based channels

Fig. 10 shows a sample implementation of HelloStreamManager, for the “stream-based
channels” introduced in §5.

= The streamer method (lines 5–11) is called whenever a StreamOut.promise is written,
i.e. an output is performed. It converts a Greet/Quit instance into a string, and sends
it through the OutputStream;

= the destreamer method (lines 17–21) is used to complete a StreamIn.future. It reads
a line from the InputStream, and uses the regular expressions in lines 14–15 to recognise
the content and return either Hello or Bye (note that in line 18 we also produce a
continuation channel, as required by Hello, based on the same stream manager).

A.2 Medium-parametric channel endpoints

The different concrete In/Out derivatives seen in §5 allow to abstract lchannels-based
code from the underlying message transport. However, if a programmer wants/needs to
manually create and handle I/O channel endpoints, he/she might introduce a subtle bug in
his/her code: mixing different channel transports, e.g., using an ActorOut endpoint to send
a (non-serialisable) LocalIn instance over the network, as message payload/continuation.

This risk can be avoided by automating the creation of continuation channels, using
Out.!! (§4.3) whenever possible. Otherwise, to avoid channel mixing errors and preserve
transport abstraction, lchannels also provides medium-parametric abstract endpoints, with
types medium.In[M,+A] and medium.Out[M,-A], where M stands for “medium”. All concrete
lchannels classes also instantiate M: e.g., LocalIn extends medium.In[Local, A]17
and ActorIn extends medium.In[Actor, A]. This allows to define medium-parametric CPS
protocols, by threading M along the message classes. E.g., from Example 4.1, a medium-
parametric FIFO type is:

17 For simplicity, this inheritance has been omitted in Fig. 6 (right).
case class Datum[M,T](p: T)(val cont: In[M, Datum[M,T]]) // Note "M"

When creating a Datum[M,T] instance to be sent on a medium.Out[M,...] endpoint, cont is constrained to be an instance of medium.In[M,...] — e.g., if M is Actor, the continuation is allowed to be an ActorIn instance; if a LocalIn instance is provided instead, a compilation error ensues. The price for this additional static check is the new M parameter, which may spread in the code. To avoid the resulting verbosity, the examples in this work avoid manual channel handling, and use medium-generic `lchannels.{In[A],Out[A]}`, defined as in Fig. 6 (left).

Remark A.1 (On tail recursion). For the sake of clarity, throughout this paper we use In.? very liberally. Unfortunately, when such a method is used under recursion, the Scala compiler is unable to optimise the resulting tail-call, thus increasing the stack size at each recursive invocation. This risk can be quickly determined and solved, by (i) annotating recursive methods with @tailrec (as customary when programming in Scala), and (ii) in case of compilation errors, replacing the problematic occurrences of c ? { ... } with c.receive match { ... }. These limitations, that hinder most functional programs in Scala, could be solved either by future improvements of the Scala compiler, or by the addition of tail-call optimizations on future JVM releases.

B Evaluation

B.1 Case study

The frontend also talks with the authentication server, via a SFA-typed channel endpoint:

\[ S_{FA} = \mu X.\!\!GetAuthentication(Id).\?Authentication(S_{auth}).X \]

i.e., the frontend recursively queries for S auth-typed channel endpoints, that will be forwarded to the client with a New message.

Finally, the auth. server talks with the application server, via a SAS-typed endpoint:

\[ S_{AS} = \mu X.\!\!CreateSession(String).\?NewSession(S_{act}).X \]

i.e., the authentication server recursively asks to create a new chat session for an authenticated user (the String payload is the username), and gets a S act-typed channel endpoint that will be forwarded to the client with a NewSession message.

The definitions of all protocol classes used in the case study are shown in Fig. 11.

B.2 Benchmarks

In this section, we provide more details on the benchmarks described in §6.2.

Ping-pong This benchmark measures the time required by ping-pong message exchanges between two threads P and Q, communicating through a channel with endpoint types:

\[ S_{PP} = \mu X.\!\!Ping(String).\{Ping(String).X \& \?Stop\} \quad \overline{S_{PP}} = \mu X.\!\!Ping(String).\{?Ping(String).X \& \Stop\} \]
package chat.protocol

// Session type S_front
case class GetSession(id: Int)(val cont: Out[GetSessionResult])
sealed abstract class GetSessionResult
  case class Active(service: Out[session.Command]) extends GetSessionResult
  case class New(authc: Out[auth.Authenticate]) extends GetSessionResult
package session {
  // Session type S_act
  sealed abstract class Command
    case class GetId()(val cont: Out[Id]) extends Command
    case class Ping(msg: String)(val cont: Out[Pong]) extends Command
    case class Join(chatroom: String)(val cont: Out[ChatRoom]) extends Command
    case class Quit() extends Command
    case class Id(id: Int)(val cont: Out[Command])
    case class Pong(msg: String)(val cont: Out[Command])
    case class ChatRoom(msgs: In[room.Messages], ctl: Out[roomctl.Control])(val cont: Out[Command])
  }
package room {
  // Session type S_r
  sealed abstract class Messages
    case class NewMessage(username: String, text: String)(val cont: In[Messages]) extends Messages
    case class Quit() extends Messages
  }
package roomctl {
  // Session type S_rctl
  sealed abstract class Control
    case class SendMessage(text: String)(val cont: In[Control]) extends Control
    case class Quit() extends Control
  }
package auth {
  // Session type S_auth
    case class Authenticate(username: String, password: String)(val cont: Out[AuthenticateResult])
  }
package chat.protocol.internal

package session {
  // Session type S_FS
  case class GetSession(id: Int)(val cont: Out[GetSessionResult])
  sealed abstract class GetSessionResult
    case class Success(channel: Out[chat.protocol.session.Command])(val cont: Out[GetSessionResult])
    case class Failure()(val cont: Out[GetSessionResult])
  }
  // Session type S_AS
  case class CreateSession(username: String)(val cont: Out[NewSession])
  case class NewSession(channel: Out[chat.protocol.session.Command])(val cont: Out[CreateSession])
  }
package auth {
  // Session type S_FA
    case class GetAuthentication()(val cont: Out[Authenticate])
    case class Authentication(channel: Out[chat.protocol.auth.Authenticate]) (val cont: Out[GetAuthentication])
  }

Figure 11 Chat server: public protocol classes used by clients (top), and internal protocol classes (bottom). We organise them in packages for readability, and to avoid name clashes. We also relax the convention of always using “p” as the name of the field containing the message payload.
**Ring**  This benchmark spawns a ring of $n$ threads $P_0, \ldots, P_{n-1}$, where for each $i \in \{1, \ldots, n-1\}$, $P_i$ is connected to $P_{(i+1) \mod n}$ through a channel with the following endpoint types:

$$S_{\text{ring}} = \mu X. !\text{Fwd}(\text{String}).X \oplus !\text{Stop}$$

Each $P_i$ receives a message from its $S_{\text{ring}}$-typed endpoint, and immediately forwards it to $P_{(i+1) \mod n}$ through the $S_{\text{ring}}$-typed endpoint. The only exception is the “master thread” $P_0$: it sends the first $\text{Fwd}$ message to $P_1$, waits to get it back from $P_{n-1}$ (i.e., after one ring loop), and decides whether to send another $\text{Fwd}$, or $\text{Stop}$ after a certain amount of loops.

**Streaming**  The implementation of this benchmark is similar to “Ring” above, except that the “master thread” $P_0$ sends at once a sequence (“stream”) of $\text{Fwd}$ messages to $P_1$, and then waits to receive all such messages back from $P_{n-1}$. As a consequence, the level of parallelism increases, because (depending on the system scheduling) all threads in the ring can have at the same time one or more messages waiting on their $S_{\text{ring}}$-typed channel endpoint, and thus can run in parallel to receive and forward them.

**Chameneos**  This benchmark is based on the classical peer-to-peer cooperation game [2]: $n$ colour-changing animals (i.e., the “chameneos”) repeatedly enter in a playground, interact with one of their peers, and change their colour.

In the implementation, each chameneo is a thread, and the playground is represented by a singleton broker object, with a method `enter`.

Two chameneos interact by communicating each other the respective name and colour, through a channel with the following endpoint types:

$$S_{\text{cham}} = !\text{Greet}((\text{Name}, \text{Colour})).?\text{Answer}((\text{Name}, \text{Colour}))$$

However, two chameneos can only interact after `entering` the playground. When a chameneo invokes `enter`, it obtains a channel endpoint of the following type, on which it waits for an answer:

$$S_{\text{broker}} = ?\text{Start}(S_{\text{cham}}) \& ?\text{Wait}(S_{\text{cham}}) \& \text{Closed}$$

while the broker/playground keeps the other (dually-typed) channel endpoint:

$$\overline{S}_{\text{broker}} = !\text{Start}(S_{\text{cham}}) \oplus !\text{Wait}(S_{\text{cham}}) \oplus \text{Closed}$$

The broker collects such channel endpoints in an internal queue, and waits for two of them to be available — i.e., for two chameneos to have invoked `enter`. Then, it creates a pair of channel endpoints with types $S_{\text{cham}}, \overline{S}_{\text{cham}}$ and sends them to the two chameneos, respectively as payloads of $\text{Start}$ and $\text{Wait}$ messages. At this point, the two chameneos have met, and interact; then, each one changes its colour, and invokes `enter` again.

The broker/playground counts the total number of meetings, and when a certain amount $m$ is reached, it answers `Closed` to all further requests: when a chameneo receives such an answer, it terminates. The benchmark measures the time required for $n$ chameneos to perform $m$ meetings.
C  Session types

- **Example C.1** (Subtyping with session types). Consider $S_h$ from §2. We have:

  \[
  \text{end} \leq \text{end} \quad [\text{c-End}]
  \]

  \[
  \text{Greet(String)}.(\text{?Hello(String)}).S_h \& \text{?Bye(String).end}) @ \text{!Quit.end} \leq \text{!Quit.end} \quad [\text{c-Int}]
  \]

  \[
  S_h \leq \text{!Quit.end} \quad [\text{c-\mu L}]
  \]

D  Linear types

- **Remark D.1.** The main difference between the linear types in Def. 7.3 and the ones used in [8, 7, 6] are due to our goal of later bridging into Scala types (in §7.3). In a nutshell:

  1. we restrict the types syntax to the exact fragment we will need for our encoding in Def. 7.6. See Example D.2 for more details;
  2. we use records (with p, c labels) instead of tuples;
  3. we cater for recursion (unlike [8]), but we allow the recursion operator $\mu X \ldots$ to **only** bind a variable with a variant types, and **not** with a linear input/output type (unlike [7, 6]). See Example D.3 for more details;
  4. we require the payload type to be closed;
  5. we do not require the syntax of session and linear types include dualized recursion variables (i.e., $\overline{X}$). This is a design goal, and (in part) a consequence of item 4 above.

- **Example D.2.** The linear types syntax in Def. 7.3 does **not** allow to write a type like $!(\text{?[Int]})$, representing a linear output channel carrying an input channel carrying an integer value. However, a “morally” equivalent type can be written as:

  \[
  \text{!}([\text{a} \_\{\text{p} : \text{?([b\_\{\text{p} : \text{Int}, \text{c} : \bullet\})], \text{c} : \bullet\}])}
  \]
  i.e., an output type carrying a variant with a single label a, whose payload is an input type carrying a single-labelled variant (label b) whose payload is an Int.

- **Example D.3.** The linear types syntax in Def. 7.3 does **not** allow to write a type like $\mu X \text{!}([\text{a} \_\{\text{p} : \text{Int}, \text{c} : X\}])$, representing a linear output channel carrying a single-labelled variant (label b) with an integer payload, and a recursive continuation consisting in the type itself. However, a “morally” equivalent type can be written as:

  \[
  \text{!}(\mu X \{\text{a} \_\{\text{p} : \text{Int}, \text{c} : \text{!(X)}\}\})
  \]
  i.e., an output type carrying a recursive, single-labelled variant (label a) with an integer payload, and a continuation consisting in an output type carrying the variant itself.

Thanks to our treatment of recursion and restriction to closed payloads (items 3 and 4 in Remark D.1), our definition of linear type duality is remarkably intuitive, and simpler than [6]: it only requires to turn the top level input into an output (or vice versa), leaving the carried type untouched. Such a property is shared with [8], which however does **not** cater for recursive types. This is a key feature that, in §7.3, will allow us to easily treat the two communication endpoints in Scala.

The encoding in Def. 7.6 ensures that a (possibly recursive) internal (resp. external) choice results in an output (resp. input) linear type, with a (possibly recursive) variant payload featuring a case for each branch: such a property is formalised in Proposition D.4.
Proposition D.4. For all $S$, $[S]$ is either $\bullet$ or, for some $U$, $?U$ or $!U$.

Proof. Straightforward from Def. 7.6.

Remark D.5. Def. 7.6 is inspired to the encodings in [8, 7, 6]. The main differences are due to the goals of our approach and the linear types we use (as outlined in Remark D.1), that lead to a noticeably more complex encoding:

1. the encoding must ensure that top-level recursions in the original session type are “moved” in the payload of the resulting linear input/output type;
2. as a consequence, encoding rules are duplicated: each $[S]_T$ (used until a recursion is met in $S$) is usually paired with $[S]_{T'}$ (used inside a recursion, until an external/internal choice is met in $S$). Some combinations (e.g. $[X]_{T'}$) are never used, and thus left undefined;
3. unlike [6], our encoding of recursion is based on an environment $\Gamma$; by treating duality on such an environment, we do not need dualized recursion variables in the types syntax, as observed in item 5 of Remark D.1.

Example D.6. Consider the following session type, and its encoding:

$$S_r = \mu_X (\langle \text{Msg(String)}, X \oplus \text{!Stop(Unit)} \rangle)$$

We can notice that the main recursive message type appears first within the top-level output type, and then (as $X$) within an input type.

Theorem E.2 also appears in [8, 7, 6]: the technical details of our proof are closer to [8], albeit more complex due to the complexity of Def. 7.6.

Theorem E.3 also appears in [8], which however does not address recursive types. Moreover, it does not appear in [7, 6], and is thus a contribution of this work. The main difference w.r.t. [8] is that our proof is necessarily coinductive (instead of inductive), and more complex due to the complexity of Def. 7.6.

E Proofs for §7.2

Lemma E.1 (Substitution for encoding). $[S\{s'/x\}] = [S\{[s']/x\}]$.

Proof. By structural induction on $S$, proving the following stronger statement, for all $\Gamma$ such that $\text{fv}(S) \subseteq \text{dom}(\Gamma) \setminus \{X\}$:

$$[S\{s'/x\}]_\Gamma = [S]_\Gamma\{[s']/x\} \quad (1)$$


Proof. We prove the following stronger statement, for all (possibly open) $S$ and $\Gamma$ such that $\text{fv}(S) \subseteq \text{dom}(\Gamma)$:

$$[S]_\Gamma = \overline{[\bar{S}]}_\Gamma \quad \text{where} \quad \Gamma(X) = \overline{\Gamma(X)} \quad (2)$$

We proceed by induction on $S$:

- base case $S = \text{end}$. We have:

$$[S]_\Gamma = \bullet \Rightarrow \overline{\bar{S}} = [\bar{S}]_\Gamma$$
\[
\text{base case } S = X. \text{ Then, } S = X = X, \text{ and by hypothesis, } \Gamma(X) = L' \text{ (for some } L'). \\
\text{Therefore, we have:}
\]
\[
[S]_\Gamma = [X]_\Gamma = [X]_\Gamma = L' = [X]_\Gamma = [S]_\Gamma
\]

\text{inductive case } S = \& \{T_i \mid S_i\}. \text{ We have:}
\[
[S]_\Gamma = [\& \{T_i \mid S_i\}]_\Gamma = !\left( \left\{ p : [T_i], c : [S_i]_\Gamma \right\} \right)_{i \in I}
\]

By the induction hypothesis, we have \( \forall i \in I : [S_i]_\Gamma = [S_i]_\Gamma \) — and therefore, \( \forall i \in I : [S_i]_\Gamma = [S_i]_\Gamma \). Hence, from (3), we conclude:
\[
[S]_\Gamma = !\left( \left\{ p : [T_i], c : [S_i]_\Gamma \right\} \right)_{i \in I} = !\left( \left\{ p : [T_i], c : [S_i]_\Gamma \right\} \right)_{i \in I} = [S]_\Gamma
\]

\text{inductive case } S = \& \{T_i \mid S_i\}. \text{ We have:}
\[
[S]_\Gamma = [\& \{T_i \mid S_i\}]_\Gamma = ?\left( \left\{ p : [T_i], c : [S_i]_\Gamma \right\} \right)_{i \in I}
\]

By the induction hypothesis, we have \( \forall i \in I : [S_i]_\Gamma = [S_i]_\Gamma \). Hence, from (4), we conclude:
\[
[S]_\Gamma = ?\left( \left\{ p : [T_i], c : [S_i]_\Gamma \right\} \right)_{i \in I} = ?\left( \left\{ p : [T_i], c : [S_i]_\Gamma \right\} \right)_{i \in I} = [S]_\Gamma
\]

\text{inductive case } S = \mu X. S'. \text{ We have:}
\[
[S]_\Gamma = \text{act}(S) \mu X. [S']_\Gamma \text{ where } \Gamma' = \Gamma \setminus \{ \text{act}(S)X / X \}
\]

For some \( n \geq 0 \) and \( S_0 \neq \mu X. S''' \), we also have \( S' = \mu Y_1 \cdots Y_k \cdot S_0 \). We can observe that, from (5), by performing \( n \) more encoding steps as per Def. 7.6, we get:
\[
[S]_\Gamma = \text{act}(S) \mu X. [\mu Y_1 \cdots Y_k \cdot S_0]_\Gamma' \\
= \text{act}(S) \mu X. [\mu Y_1 \cdots Y_k \cdot S_0]_\Gamma' \left[ \text{act}(S)Y_1 / Y_1 \right] \\
= \cdots \\
= \text{act}(S) \mu X. [\mu Y_1 \cdots Y_k \cdot [S_0]_{\Gamma' \setminus \{ \text{act}(S)Y_1 / Y_1 \} \cdots \{ \text{act}(S)Y_n / Y_n \}}
\]

We now show two key properties of such encoding steps. Let:
\[
\Gamma'_{n+1} = \Gamma' \\
\forall k \text{ such that } 1 \leq k \leq n, \quad \Gamma'_k = \Gamma'_{k+1} \left[ \text{act}(S_{k-1})Y_k / Y_k \right] \\
\text{and } S'_k = \mu Y_k \cdot S_{k-1} \quad \text{(hence, } S' = S'_n) 
\]

First, we prove that:
\[
\forall h \text{ such that } 0 \leq h \leq n: \text{fv}(S'_h) \subseteq \text{dom}(\Gamma'_h)
\]

We proceed by induction on the number of function updates applied to \( \Gamma'_{n+1} = \Gamma' \) in the definition of \( \Gamma'_{n+1} \) (see (7) above), i.e. on the length of the following sequence:
\[
\left\{ \text{act}(S_0Y_1 / Y_1) \right\} \left\{ \text{act}(S_{n-1})Y_n / Y_n \right\} \cdots \left\{ \text{act}(S_{n-1})Y_n / Y_n \right\}
\]

Such a length is \( n - h \), and it ranges from \( 0 \) (when \( h = n \)) to \( n \) (when \( h = 0 \)), thus covering all the values of \( h \) required in (8). We have the following cases:
Second, we prove the following key statement:

- base case \( n - h = 0 \). In this case, \( n = h \). Since \( \text{fv}(S) \subseteq \text{dom}(\Gamma) \) (by hypothesis), by (5) we have \( \text{fv}(S') \subseteq \text{dom}(\Gamma') \). Thus, since \( S'_n = S'_n = S' \) and \( \Gamma'_{n+1} = \Gamma'_h = \Gamma' \), we conclude \( \text{fv}(S'_0) \subseteq \text{dom}(\Gamma'_h) \);

- inductive case \( n - h = m + 1 \). Therefore, \( h = n - m - 1 \), and we need to prove \( \text{fv}(S'_{n-m-1}) \subseteq \text{dom}(\Gamma'_{n-m}) \). By the induction hypothesis (on statement (8)), the thesis holds for \( h' \) such that \( n - h' = m \), i.e.:

\[
\text{fv}(S''_h) = \text{fv}(S'_{n-m}) = \text{fv}(\mu v_{n-m} \cdot S'_{n-m-1}) \subseteq \text{dom}(\Gamma'_{n-m}) = \text{dom}(\Gamma'_{h+1})
\]

Therefore, we obtain:

\[
\text{fv}(S'_h) = \text{fv}(S'_{n-m-1}) \subseteq \text{dom}(\Gamma'_{n-m}) = \text{dom}(\Gamma'_{h+1})
\]

which concludes the proof of statement (8).

Second, we prove the following key statement:

\[
\forall h \text{ such that } 0 \leq h \leq n: \textstyle \left[ S'_h \right]^{\mu}_{\Gamma_{h+1}} = \textstyle \left[ S'_0 \right]^{\mu}_{\Gamma_{h+1}}
\]

We proceed by induction on \( h \):

- base case \( h = 0 \). Note that \( S_0 \) cannot be a recursion variable, nor \textbf{end}. Therefore, we have two sub-cases:

  * \( S_0 = \left\{ \& c, j, T_j \right\} \cdot S''_h \). Then,

\[
\left[ S_0 \right]^{\mu}_{\Gamma_0} = \left[ \text{\& c, j, T_j \cdot S'} \right]^{\mu}_{\Gamma_0} = \left[ \text{\& c, j, T_j} \right]^{\mu}_{\Gamma_0} \cdot \left[ S''_h \right]^{\mu}_{\Gamma_0}
\]

We now observe that each \( S''_j \) is a subterm of \( S''_h \), which in turn is a subterm of \( S' \); moreover, since \( \text{fv}(S''_j) \subseteq \text{fv}(S''_h) \) and (by (8)) \( \text{fv}(S''_h) \subseteq \text{dom}(\Gamma'_0) \), we have \( \text{fv}(S''_j) \subseteq \text{dom}(\Gamma'_0) \). Therefore, we can apply the main induction hypothesis (statement (2)), and we get \( \forall j \in J: \left[ S''_j \right]^{\mu}_{\Gamma_0} = \left[ S''_j \right]^{\mu}_{\Gamma_0} \cdot \left[ S''_h \right]^{\mu}_{\Gamma_0} \), i.e., \( \forall j \in J: \left[ \text{\& c, j, T_j} \right]^{\mu}_{\Gamma_0} \cdot \left[ S''_h \right]^{\mu}_{\Gamma_0} \). Hence, from (10), we obtain:

\[
\left[ S_0 \right]^{\mu}_{\Gamma_0} = \left[ \text{\& c, j, T_j} \right]^{\mu}_{\Gamma_0} \cdot \left[ S''_h \right]^{\mu}_{\Gamma_0}
\]

* \( S'_0 = \left\{ \& c, j, T_j \right\} \cdot S''_h \). Then,

\[
\left[ S_0 \right]^{\mu}_{\Gamma_0} = \left[ \& c, j, T_j \cdot S''_h \right]^{\mu}_{\Gamma_0} = \left[ \& c, j, T_j \right]^{\mu}_{\Gamma_0} \cdot \left[ S''_h \right]^{\mu}_{\Gamma_0}
\]

Since each \( S''_j \) is a subterm of \( S \) and \( \text{fv}(S''_j) \subseteq \text{fv}(S''_h) \subseteq \text{dom}(\Gamma'_0) \), by the main induction hypothesis (statement (2)) we get \( \forall j \in J: \left[ S''_j \right]^{\mu}_{\Gamma_0} = \left[ S''_j \right]^{\mu}_{\Gamma_0} \). Hence, from (12), we obtain:

\[
\left[ S_0 \right]^{\mu}_{\Gamma_0} = \left[ \text{\& c, j, T_j} \right]^{\mu}_{\Gamma_0} \cdot \left[ S''_h \right]^{\mu}_{\Gamma_0}
\]

Summing up, in both cases (11) and (13) we conclude

\[
\left[ S_h \right]^{\mu}_{\Gamma_{h+1}} = \left[ S'_0 \right]^{\mu}_{\Gamma_{h+1}}
\]
Lightweight Session Programming in Scala

...
Since act(S) = act(S′) = ?,
We have S = \text{1}_i \cdot \text{1}_j, S_i \text{ and } S′ = \text{1}_i \cdot \text{1}_j, S_i′; moreover, for all
i \in I, \, T_i \leq T_i′ \text{ and } S_i \leq S′_i \text{ — i.e.:}
\forall i \in I: ([T_i],[T_i′]) \in \mathcal{R}_i \text{ and } ([S_i],[S_i′]) \in \mathcal{R}_s \text{ (thus, both pairs belong to } \mathcal{R}) \quad (15)

Now, we observe:
\[ [S] = ?(U) \text{ implies } U = [1_{1 \cdot \{p : [T_i],[c : [S_i]]\} }]_{1 \cdot i} \]
\[ [S′] = ?(U′) \text{ implies } U′ = [1_{1 \cdot \{p : [T_i′],[c : [S_i′]]\} }]_{1 \cdot i \cdot j} \quad (16) \]

Hence, from (16) and (15), we conclude that the pair (U,U′) satisfies rule [\text{\ell} \cdot \text{VR}];

We have S = \mu X . S″ \subseteq S′; moreover, from the rule premise, S″\{\mu X . S′/X\} \subseteq S′, which implies:
\[ ([S″\{\mu X . S′/X\}],[S″′]) \text{ belongs to } \mathcal{R}_s \quad (17) \]

We observe that act(S) = act(S″) = ?. Moreover, from
\[ [S] = [\mu X . S″] = \text{act}(S″)\left(\mu X . [S″]\{\omega(\sigma′)(X)\} \right) \]

we have:
\[ [S] = [\mu X . S″] = \text{act}(S″)(\mu X . U″) \quad \text{where } U″ = [S″]\{\omega(\sigma′)(X)\} \quad (18) \]

which also gives us:
\[ U = \mu X . U″ \quad (19) \]

From (18), by Lemma E.1, we know that:
\[ [S″]\{\mu X . S′/X\} = [S″]\{\{\mu X . S″\}\{\omega(\sigma′)(X)\}\} X \quad (20) \]

Note that S″ cannot be a recursion variable, nor end — and the same holds for S″\{\mu X . S′/X\}. Hence, from (20) we also have:
\[ [S″]\{\mu X . S′/X\} = \text{act}(S″)(U″) \quad \text{where } U″ = [S″]\{\omega(\sigma′)(X)\} X \quad (21) \]

Now, we can notice that the unfolding of \( \mu X . U″ \) from (18), is equal to \( U″ \) from (21):
\[ U″\{\mu X . U″/X\} = [S″]\{\omega(\sigma′)(X)\} X \quad (22) \]

Summing up from (21) and (22), we can rewrite (17) as:
\[ \left( \text{act}(S″),\text{act}(S″′)\right) \text{ belongs to } \mathcal{R}_s \quad (23) \]

Since act(S″) = ?, from (23) we have:
\[ (U″\{\mu X . U″/X\}, U″) \text{ belongs to } \mathcal{R}_s, \text{ and thus to } \mathcal{R} \quad (24) \]

Hence, reminding (19), we conclude that \( (\mu X . U″, U″) = (U,U″) \) satisfies rule [\text{\ell} \cdot \text{\mu}L];
We have \( S \subseteq S' = \mu_X.S'' \). The proof is symmetric to that for \([-\mu L]\) above:

1. we develop the unfoldings of \( S' \) and \( U' \) (instead of \( S \) and \( U \));
2. we conclude that the pair \((U, U')\) satisfies rule \([-\mu R]\).

For each pair \((U, U') \in \mathcal{R}_t\), there exists a corresponding pair \(((U'), \top(U)) \in \mathcal{R}_s\), and thus there exist \( S, S' \) such that \([S] = \top(U'), [S'] = \top(U)\) and \( S \subseteq S' \). Hence, \( S, S' \) must be (possibly recursive) internal choices. We proceed by cases on the rule in Def. 7.2 whose conclusion is \( S \subseteq S' \):

\[= [-\mu R].\]

These cases are impossible, because \( \text{act}(S) = \text{act}(S') = \top; \)

\[= [-\mu L].\]

The proof is similar to the case \([-\mu L]\) in the proof for the relation \( \mathcal{R}_t \) above, except that:

1. we have \( \text{act}(S'') = \top \) (instead of \( \text{act}(S'') = ? \));
2. in the step corresponding to (24), we use the relation \( \mathcal{R}_t \) (instead of \( \mathcal{R}_s \));

\[= [-\mu R].\]

The proof is similar to the case \([-\mu R]\) in the proof for the relation \( \mathcal{R}_t \) above, except for the two changes just mentioned (i.e., \( \text{act}(S'') = \top \) and use of \( \mathcal{R}_s \)).

We can now conclude the proof for the \( \Rightarrow \) direction of the statement, by noticing that \( \xi_t \) is the largest relation coinductively defined by the rules in Def. 7.4 — and therefore, \( \mathcal{R} \subseteq \mathcal{R}_t \). Hence, since \( S \subseteq S' \) implies \([S], [S'] \) \( \in \mathcal{R}_s \subseteq \mathcal{R} \subseteq \mathcal{R}_t \), we conclude that \( S \subseteq S' \) implies \([S] \subseteq [S'] \).

We now prove the \( \Leftarrow \) direction of the statement. Consider the relation \( \mathcal{R} = \mathcal{R}_s \cup \mathcal{R}_t \), where:

\[\mathcal{R}_s = \{(S, S') \mid [S] \subseteq [S'] \} \quad \mathcal{R}_t = \{(T, T') \mid [T] \subseteq [T'] \}\]

We prove that \( \mathcal{R} = \mathcal{R}_s \cup \mathcal{R}_t \), by cases on the rule in Def. 7.4 whose conclusion is \([S] \subseteq [S'] \):

\[= [-\mu \text{End}].\]

Then, \([S] = [S'] = \bullet\), i.e. \( S = S' = \end \). Hence, we conclude that the pair \((S, S')\) satisfies rule \([-\mu \text{End}]\);

\[= [-\mu \text{In}].\]

Then, there exist \( U, U' \) such that \([S] = ?(U) \), \([S'] = ?(U') \) and \( U \subseteq U' \). We proceed by cases on the rule in Def. 7.4 whose conclusion is \( U \subseteq U' \):
Thus to ensure that the pair 
and (37) we get:

Since (35), (30) and (32) we know that for all

From (27), by Def. 7.6, we obtain:

Moreover, by Lemma E.1, we get:

We notice that:

Therefore, from (28), (30) and (32) we know that for all \( i \in I \), \( (S_i, S_i') \in \mathcal{R}_s \) and \( (T_i, T_i') \in \mathcal{R}_t \) — i.e., both pairs belong to \( \mathcal{R} \). Hence, from (29) and (31), we conclude that the pair \( (S, S') \) satisfies rule \([s_{\text{Ext}}]\);

We have \( U = \mu X.U'' \) — and thus, by Def. 7.6,

and thus, by Def. 7.6,

We notice that:

Moreover, by Lemma E.1, we get:

Now, we observe that from the premise of rule \([s_{\mu L}]\), we also have:

Since \( \preceq_l \) is the largest relation coinductively defined by the rules in Def. 7.4, by \([s_{\mu L}]\) and (37) we get:

Therefore, from (36), we know that the pair \( (S''\{\mu X.S'/X\}, S') \) belongs to \( \mathcal{R}_s \), and thus to \( \mathcal{R} \). Hence, we conclude that the pair \( (S, S') = (\mu X.S''', S') \) satisfies rule \([s_{\mu L}]\).

We have \( U \preceq_l U'' = \mu_X.U'' \). The proof is symmetric to that for \([s_{\mu L}]\) above:
1. we develop the unfoldings of $U'$ and $S'$ (instead of $U$ and $S$);
2. we conclude that the pair $(S, S')$ satisfies rule $[\varepsilon, \mu R]$;

$\forall \sigma, \text{OUT}$. Then, there exist $U, U'$ such that $[S] = !\langle U \rangle$, $[S'] = !(U')$ and $U' \leq \varepsilon U$. We proceed by cases on the rule in Def. 7.4 whose conclusion is $U' \leq \varepsilon U$:

(a) $[\varepsilon, \text{VR}]$. We have:

$$U = \{ι \cdot (p : V_i, c : L_i)\}_{i \in I \cup J} \quad U' = \{ι' \cdot (p : T_i', c : L_i')\}_{i \in I}$$

such that $\forall i \in I$: $V_i' \leq \varepsilon V_i$ and $L_i' \leq \varepsilon L_i$, i.e. $ι_i \leq L_i$

From (39), by Def. 7.6, we obtain:

$$[S] = !(U) \implies S = \bigoplus_{i \in I \cup J} !ι_i(T_i) . S_i \ (41)$$
$$\forall i \in I \cup J: \exists T_i, S_i: [T_i] = V_i \quad \text{and} \quad [S_i] = L_i \ (42)$$
$$[S'] = !(U') \implies S' = \bigoplus_{i \in I} !ι_i(T'_i) . S'_i \ (43)$$
$$\forall i \in I: \exists T'_i, S'_i: [T'_i] = V'_i \quad \text{and} \quad [S'_i] = L'_i \ (44)$$

Therefore, from (40), (42) and (44) we know that for all $i \in I, (S_i, S'_i) \in \mathcal{R}_s$ and $(T'_i, T_i) \in \mathcal{R}_t$ — i.e., both pairs belong to $\mathcal{R}$. Hence, from (41) and (43), we conclude that the pair $(S, S')$ satisfies rule $[\varepsilon, \mu R]$:

(b) $[\varepsilon, \mu L]$. The proof is similar to the sub-case $[\varepsilon, \mu L]$ in the proof for case $[\varepsilon, \mu R]$ above, except that:

1. we have act($S''$) = $!$ (instead of act($S''$) = $?$);
2. we reach the step corresponding to (38) via rule $[\varepsilon, \text{OUT}]$ (instead of $[\varepsilon, \text{IN}]$);

(c) $[\varepsilon, \mu R]$. The proof is similar to the sub-case $[\varepsilon, \mu R]$ in the proof for case $[\varepsilon, \mu L]$ above, except for the two changes just mentioned (i.e., act($S''$) = $!$ and use of $[\varepsilon, \text{OUT}]$).

For each pair $(T, T') \in \mathcal{R}_t$, we proceed by cases on the rule in Def. 7.4 whose conclusion is $[T] \leq [T']$:

(a) $[\varepsilon, \text{B}]$. Then, $\llbracket T \rrbracket \leq \llbracket T' \rrbracket$, which means $[T], [T'] \in \mathbb{B}$. Therefore, by Def. 7.6, we can only have $[T] = T$ and $[T'] = T'$. Hence, we conclude that the pair $([T], [T'])$ satisfies rule $[\varepsilon, \text{B}]$;

(b) in all the other cases, $[T]$ and $[T']$ must have the form $?()$, $!(c)$ or $\cdot \cdot$ — which, by Def. 7.6, can only be originated if, for some $S$, and $S', T = S$ and $T' = S'$. Thus, we also have $(T, T') \in \mathcal{R}_t$; the proof falls back into the case above.

We can now conclude the proof for the $\iff$ direction of the statement, by noticing that $\leq$ is the largest relation coinductively defined by the rules in Def. 7.2 — and therefore, $\mathcal{R} \subseteq \leq$. Hence, since $[S] \leq [S']$ implies $(S, S') \in \mathcal{R}_s \subseteq \mathcal{R} \subseteq \leq$, we conclude that $[S] \leq [S']$ implies $S \leq S'$.

**Theorem 7.7** (Encoding preserves duality, subtyping). $[S] = [S']$, and $S \leq S'$ iff $[S] \leq [S']$.

**Proof.** Direct consequence of Theorem E.2 and Theorem E.3.

### F Scala types

**Remark F.1.** Note that when encoding $S$ into a linear type, Def. 7.3 inductively maps each internal/external choice subterm of $S$ to some unique variant type. Therefore, a nominal environment $N$ which is suitable for $S$ implicitly gives a distinct name to all such variants, and $[N]$ allows to retrieve it.
- **Proposition F.2.** If $\mathcal{N}$ is suitable for $S$, then $\llbracket S \rrbracket_{\mathcal{N}}$ is suitable for $\llbracket S \rrbracket$.

**Proof.** Follows from Remark F.1. □

- **Lemma F.3.** $\llbracket S \rrbracket_{\mathcal{N}} = \llbracket \llbracket S \rrbracket_{\mathcal{N}} \rrbracket$.

**Proof.** By Def. 7.11, Theorem E.2 and Def. 7.10, $\llbracket S \rrbracket_{\mathcal{N}} = (\llbracket S \rrbracket)_{|N_1} = \llbracket \llbracket S \rrbracket \rrbracket_{|N_1}$. Then, by Def. 7.5, case analysis on Def. 7.9 (first three cases) and applying duality for In/Out/Unit, we get $\llbracket S \rrbracket_{|N_1} = \llbracket \llbracket S \rrbracket \rrbracket_{|N_1}$. By Def. 7.11, we conclude $\llbracket S \rrbracket_{|N_1} = \llbracket S \rrbracket_{\mathcal{N}}$. □

- **Lemma F.4.** $\text{prot}(\llbracket S \rrbracket_{\mathcal{N}}) = \text{prot}(\llbracket S \rrbracket)_{\mathcal{N}}$.

**Proof.** We have:

\[
\begin{align*}
\text{prot}(\llbracket S \rrbracket_{\mathcal{N}}) &= \llbracket \text{carr}(\llbracket S \rrbracket) \rrbracket_{|N_1} \quad \text{(by Def. 7.11)} \\
&= \llbracket \text{carr}(\llbracket S \rrbracket) \rrbracket_{|N_1} \quad \text{(by Def. 7.5 and Def. 7.3)} \\
&= \llbracket \text{carr}(\llbracket S \rrbracket) \rrbracket_{|N_1} \quad \text{(by Theorem E.2)} \\
&= \llbracket \text{carr}(\llbracket S \rrbracket) \rrbracket_{|N_1} \quad \text{(by Def. 7.10 and item (iv) of Def. 7.8)} \\
&= \text{prot}(\llbracket S \rrbracket)_{\mathcal{N}} \quad \text{(by Def. 7.11)}
\end{align*}
\]

- **Theorem 7.13.** For all $S$, $\llbracket S \rrbracket_{\mathcal{N}} = \llbracket S \rrbracket_{\mathcal{N}}$ and $\text{prot}(\llbracket S \rrbracket_{\mathcal{N}}) = \text{prot}(\llbracket S \rrbracket)_{\mathcal{N}}$.

**Proof.** Direct consequence Lemma F.3 and Lemma F.4. □

### F.1 Subtyping in session types and Scala

We define the subtyping relation $\llhd$ in Scala similarly to [24], i.e., through a class table mapping each class name to its declaration. In our case, we consider class tables whose entries have 3 possible forms, corresponding to the declarations generated by Def. 7.9:

\[
A \quad \text{sealed abstract class A} \quad \text{or} \quad B \quad \text{case class B(...) extends C} \quad \text{or} \quad D \quad \text{case class D(...)}
\]

- **Definition F.5 (Scala subtyping).** Given a class table $CT$, $\llhd$ is the smallest relation that contains the reflexive and transitive closure of the immediate subclass relation given by the extends clauses in $CT$, and (ii) is closed forward under the following rules:

\[
\begin{align*}
A &\llhd B \quad \text{In}[A] \llhd \text{In}[B] \quad \text{[\llhd-ln]} \\
B &\llhd A \quad \text{Out}[A] \llhd \text{Out}[B] \quad \text{[\llhd-out]} \\
\text{null} &\llhd A \quad \text{A \notin \{Unit,Nothing\}} \quad \text{[\llhd-null]} \\
\text{A \notin \{Unit,Any\Val\}} &\llhd \text{AnyRef} \quad \text{[\llhd-aref]} \\
\text{Nothing} &\llhd A \quad \text{[\llhd-thing]} \\
\text{A \notin \{Unit,Null\}} &\llhd \text{Any} \quad \text{[\llhd-thing]}
\end{align*}
\]

We write $A \llhd B$ iff $A \llhd B$ and $A \neq B$. We write $\llhd$ (resp. $\llhd$) for the inverse of $\llhd$ (resp. $\llhd$).

Rule [\llhd-ln] in Def. F.5 reflects the covariance of In[A], and [\llhd-out] reflects the contravariance of Out[-A], as per Fig. 6 (left). The rest of the rules reflect the Scala unified types structure [28], shown in Fig. 12: a complete lattice with Any as top element, Nothing at bottom element, AnyVal as LUB of all value types (in this case, just Unit), and AnyRef/Null...
respectively as LUB/GLB of all reference types (in this case, `In[ ]`, `Out[ ]`, and all the classes declared in `CT`).

Example F.6. Consider the following class table:

```
A  sealed abstract class A
B1 case class B1(...) extends A
B2 case class B2(...) extends A
C  case class C(...)
```


Notation F.7. We write $A \in \text{prot}(\langle S \rangle)_{N'}$ iff $A$ is in the domain of the class table of $\text{prot}(\langle S \rangle)_{N'}$.

In Theorem 7.14 below, we intend $<$ to be based on the class table given by $\text{prot}(\langle S \rangle)_{N'}$.

Theorem 7.14. For all $A, S, N'$, $A <\langle S \rangle_{N'}$ implies one of the following:

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

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

\footnote{To avoid cluttering the notation, we are not representing the (curried) class fields: they do not influence the subtyping relation.}
Proof. Note that the leftmost clauses of a1–a3 and b1–b3 are mutually exclusive, and cover all possible shapes of $A$: end, or (possibly recursive) external/internal choice.

For the first part of the statement, assume $A \lessdot \langle S \rangle_{\mathcal{N}}$:

a. if $S = \text{end}$, then by Def. 7.11, $\langle S \rangle_{\mathcal{N}} = \text{Unit}$. The rest follows from Def. F.5;

b. if $\text{act}(S) = \ast$, then $S$ is a (possibly recursive) external choice, and by Def. 7.11, $\langle S \rangle_{\mathcal{N}} = \text{In}[C]$, for some $C$ occurring in the class table given by $\text{prot}(\langle S \rangle_{\mathcal{N}})$. Moreover, by Def. F.5, $A \lessdot \langle S \rangle_{\mathcal{N}}$ can only hold either by rule $\text{[c-in]}$, or $\text{[c-null]}$/$\text{[c-Nothing]}$. The latter two cases give us $A \lessdot \text{Null}$. Otherwise, when $A \lessdot \langle S \rangle_{\mathcal{N}}$ holds by $\text{[c-in]}$, we have $A = \text{In}[A_1]$, for some $A_1$ such that $A_1 \lessdot C$. Now, we prove the existential quantification on $B$ in the statement by letting $B = A_1$. We get $B \lessdot C$, and assuming $\text{Null} \lessdot B$, we proceed by cases on the declaration of $C$ given by $\text{prot}(\langle S \rangle_{\mathcal{N}})$:

- case class C( . . .). Then, besides Null and Nothing, the only subtype of $C$ is $C$ itself, and thus $B = C$, and we get $A = \text{In}[C] = \langle S \rangle_{\mathcal{N}}$. By letting $S' = S$ and $N'' = \mathcal{N}$, we conclude $A = \langle S' \rangle_{\mathcal{N}}$, and $S' \lessdot S$ (by reflexivity of $\lessdot$);

- case class C( . . .) extends D (for some D). Again, $C$ is the only subtype of itself (besides Null and Nothing), and we conclude with $S' = S$, and $N'' = \mathcal{N}$ as above;

- sealed abstract class C. In this case, B’s declaration can only be:

  ```
  case class B( . . .) extends C
  ```

  and by Def. 7.11 and Def. 7.9, this combination is only obtained when $\llbracket S \rrbracket = \langle \text{Out} \rangle$, with $S$ being a (possibly recursive) non-singleton external choice, and $U$ being a (possibly recursive) non-singleton variant with a case $B \llbracket p : [T_1], c : [S_1] \rrbracket$, corresponding to a branch $\text{?B}(T_1).S_1$ of $S$. Now, we can define $S'$ by removing the other external choice branches of $S$, and define $N'$ by reflecting the prominent on the elements of $\text{dom}(\mathcal{N})$: we obtain $S' \lessdot S$ (by $\text{[c-Ext]}$, possibly preceded by applications of $\text{[c-\muL]}$/\text{[c-\muR]}), and $A = \langle S' \rangle_{\mathcal{N}}$;

a. if $\text{act}(S) = \downarrow$, then $S$ is a (possibly recursive) internal choice, and by Def. 7.11, $\langle S \rangle_{\mathcal{N}} = \text{Out}[C]$, for some $C$ occurring in the class table given by $\text{prot}(\langle S \rangle_{\mathcal{N}})$. Moreover, by Def. F.5, $A \lessdot \langle S \rangle_{\mathcal{N}}$ can only hold either by rule $\text{[c-out]}$, or $\text{[c-null]}$/$\text{[c-Nothing]}$. The latter two cases give us $A \lessdot \text{Null}$. Otherwise, when $A \lessdot \langle S \rangle_{\mathcal{N}}$ holds by $\text{[c-out]}$, we have $A = \text{Out}[A_1]$, for some $A_1$ such that $C \lessdot A_1$. Now, we prove the existential quantification on $B$ in the statement by letting $B = A_1$. We get $C \lessdot B$, and assuming $B \lessdot \text{AnyRef}$, we proceed by cases on the declaration of $C$ given by $\text{prot}(\langle S \rangle_{\mathcal{N}})$:

- case class C( . . .). Then, besides AnyRef and Any, the only supertype of $C$ is $C$ itself, and thus $B = C$, and we get $A = \text{Out}[C] = \langle S \rangle_{\mathcal{N}}$. By letting $S' = S$ and $N'' = \mathcal{N}$, we conclude $A = \langle S' \rangle_{\mathcal{N}}$, and $S' \lessdot S$ (by reflexivity of $\lessdot$);

- case class C( . . .) extends D (for some D). This case is absurd: it could only be obtained from $\text{prot}(\langle S \rangle_{\mathcal{N}})$ if $C$ is a label of some non-singleton internal choice mapped to $D$ in $\mathcal{N}$, i.e. $C$ appears as a proper subterm of $S$: but then, we must conclude $\langle S \rangle_{\mathcal{N}} \neq \text{Out}[C]$ (contradiction);

- sealed abstract class C. Then, C is the only supertype of itself (besides AnyRef and Any), and we conclude with $S' = S$ and $N'' = \mathcal{N}$ as above.

We now prove the second part of the statement. Assume $\langle S \rangle_{\mathcal{N}} \lessdot A$:
b1. if \( S = \text{end} \), then by Def. 7.11, \( \langle S \rangle_N = \text{Unit} \). The rest follows from Def. F.5;
b2. if \( \text{act}(S) = \_ \), then \( S \) is a (possibly recursive) external choice, and by Def. 7.11, \( \langle S \rangle_N = \text{In}[C] \), for some \( C \) occurring in the class table given by \( \text{proto}(\langle S \rangle_N) \). Moreover, by Def. F.5, \( \langle S \rangle_N < A \) can only hold either by rule \( \langle \text{-In} \rangle \), or \( \langle \text{-AnyRef} \rangle / \langle \text{-Any} \rangle \). The latter two cases give us \( A < \text{AnyRef} \). Otherwise, when \( \langle S \rangle_N < A \) holds by \( \langle \text{-Out} \rangle \), we have \( A = \text{In}[A1] \), for some \( A1 \) such that \( C < A1 \). Now, we prove the existential quantification on \( B \) in the statement by letting \( B = A1 \). We get \( C < B \), and assuming \( B \neq \text{Null} \), we proceed by cases on the declaration of \( C \) given by \( \text{proto}(\langle S \rangle_N) \):

- case class \( C(\ldots) \). Then, besides \text{AnyRef} and \text{Any}, the only supertype of \( C \) itself, and thus \( B = C \), we get \( A = \text{In}[C] = \langle S \rangle_N \). By letting \( S' = S \) and \( N' = N \), we conclude \( A = \langle S' \rangle_N \), and \( S \leq S' \) (by reflexivity of \( \leq \));
- case class \( C(\ldots) \text{ extends } D \) (for some \( D \)). This case is absurd: it could only be obtained from \( \text{proto}(\langle S \rangle_N) \) if \( C \) is a label of some non-singleton external choice mapped to \( B \) in \( N \), i.e. \( C \) appears as a proper subterm of \( S \); but then, we must conclude \( \langle S \rangle_N \neq \text{In}[C] \) (contradiction);
- sealed abstract class \( C \). Then, \( C \) is the only supertype of itself (besides \text{AnyRef} and \text{Any}), and we conclude with \( S' = S \) and \( N' = N \) as above.

b3. if \( \text{act}(S) = \_! \), then \( S \) is a (possibly recursive) internal choice, and by Def. 7.11, \( \langle S \rangle_N = \text{Out}[C] \), for some \( C \) occurring in the class table given by \( \text{proto}(\langle S \rangle_N) \). Moreover, by Def. F.5, \( \langle S \rangle_N < A \) can only hold either by rule \( \langle \text{-Out} \rangle \), or \( \langle \text{-AnyRef} \rangle / \langle \text{-Any} \rangle \). The latter two cases give us \text{AnyRef} < A. Otherwise, when \( \langle S \rangle_N < A \) holds by \( \langle \text{-Out} \rangle \), we have \( A = \text{Out}[A1] \), for some \( A1 \) such that \( A1 < C \). Now, we prove the existential quantification on \( B \) in the statement by letting \( B = A1 \). We get \( B < C \), and assuming \text{Null} \neq B, we proceed by cases on the declaration of \( C \) given by \( \text{proto}(\langle S \rangle_N) \):

- case class \( C(\ldots) \). Then, besides \text{Null} and \text{Nothing}, the only subtype of \( C \) itself, and thus \( B = C \), we get \( A = \text{Out}[C] = \langle S \rangle_N \). By letting \( S' = S \) and \( N' = N \), we conclude \( A = \langle S' \rangle_N \), and \( S \leq S' \) (by reflexivity of \( \leq \));
- case class \( C(\ldots) \text{ extends } D \) (for some \( D \)). Again, \( C \) is the only subtype of itself (besides \text{Null} and \text{Nothing}), and we conclude with \( S' = S \) and \( N' = N \) as above;
- sealed abstract class \( C \). In this case, \( B \)'s declaration can only be:

```java
case class B(\ldots) extends C
```

and by Def. 7.11 and Def. 7.9, this combination is only obtained when \( \llbracket S \rrbracket = \_!(U) \), with \( S \) being a (possibly recursive) non-singleton internal choice, and \( U \) being a (possibly recursive) non-singleton variant with a case \( B \{ p : [T1] , c : [S1] \} \), corresponding to a selection \( B(T1),S1 \) of \( S \). Now, we can define \( S' \) by removing the other internal choice branches of \( S \), and define \( N' \) by reflecting the pruning on the elements of \( \text{dom}(N) \): we obtain \( S \leq S' \) (by \( \text{-Int} \)), possibly preceded by applications of \( \langle \text{-\mu L} \rangle / \langle \text{-\mu R} \rangle \), and \( A = \langle S' \rangle_N \).

---

**G  Related and future work**

We give a more detailed comparison between our work and [35].

The main similarities are that both works are based on [8], and overcome the lack of static linearity guarantees in the host language via runtime checks, and focus on session safety and easy-to-use APIs.
Technically, [35] focuses on structural types in an equi-recursive, coinductive framework; as we target Scala’s nominal type system and we want to extract CPS protocol classes, we need a more delicate treatment of recursion. [35] focuses on duality (with Theorem E.2 as common result), while we address subtyping more thoroughly. For API design, [35] focuses on type inference in OCaml, while we deal with its limited availability in Scala. [35] implements ad hoc runtime linearity checks, while we shape lchannels around Scala’s Promises/Futures, trying to leverage a notion of “linear” usage that is already familiar to Scala programmers. [35] adopts from [14] the idea of send/receive and select/branch operations returning continuation channels; instead, we merge send/select and receive/branch (with a simpler API for n-ary choices) and represent continuations as part of CPS messages (providing a natural form of session delegation, see Example 4.3), with a simplified API (§4.3).