Mixed Sessions

Filipe Casal[a], Andreia Mordido[a], Vasco T. Vasconcelos[a]

[a]*LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal*

Abstract

Session types describe patterns of interaction on communicating channels. Traditional session types include a form of choice whereby servers offer a collection of options, of which each client selects exactly one. Mixed choices blur the distinction between servers and clients (that is, external and internal choice) by allowing options to be both offered and selected in the same choice. We introduce mixed choices in the context of session types and argue that they increase the flexibility of program development at the same time that they reduce the number of synchronisation primitives down to exactly one. We present a type system incorporating subtyping and prove preservation and absence of runtime errors for well-typed processes. We further show that classical (conventional) sessions can be faithfully and tightly embedded in mixed choices, and conversely that there is a minimal encoding from mixed choices to classical sessions. Finally, we discuss algorithmic type checking and a runtime system built on top of a conventional (choice-less) message-passing architecture.

Keywords: Type Systems, Session Types, Mixed Choice.

1. Introduction

Session types provide for describing series of continuous interactions on communication channels [1, 2, 3, 4, 5]. When used in type systems for programming languages, session type systems statically verify that programs follow protocols, and hence that they do not engage in communication mismatches.

In order to motivate mixed sessions, suppose that we want to describe a process that asks for a fixed but unbounded number of integer values from some producer. The consumer may be in two states: happy with the values received so far, or ready to ask the producer for a new value. In the former case it must notify the producer so that this may stop sending numbers. In the latter case, the client must ask the producer for another integer, after which it "goes back to the beginning". Using classical sessions, and looking from the consumer side, the communication channel can be described by a (recursive) session type T of the form

```
⊕{enough:end, more: ?int.T}
```

where $\oplus$ denotes internal choice (the consumer decides), the two branches in the choice are labelled with enough and more, type **end** denotes a channel on which no further interaction is possible, and ?**int** denotes the reception of an integer value. Reception is a prefix to a type, the continuation is T (in this case the "goes back to the beginning" part). The code for the consumer (and the producer as well) features parts that exchange messages in both directions: enough and more selections from the consumer to the producer, and **int** messages from the producer to the consumer. In particular, the consumer must first select option enough (outgoing) and then receive an integer (incoming).

Using mixed sessions one can *invert the direction* of the more selection and write the type of the channel (again as seen from the side of the consumer) as

```
⊕{enough!unit.end, more?int.T}
```

The changes seem merely cosmetic, but now labels are followed by a mandatory polarity (! or ?) and label-polarity pairs constitute the keys of the choice type when seen as a map. The integer value is piggybacked on top of selection more. As a result, the classical session primitive operations: selection and branching (that is, internal and external choice) and communication (output and input) become one only: mixed choice. The producer can be safely written as a recursive process whose body is

```
x (enough?z. 0 + more!n. produce!(x, n+1))
```

offering a choice on channel end x featuring mixed branches with labels enough? and more!, where **0** denotes the terminated process and produce(x, n+1) a recursive call to the producer. The example is further developed in Section 2.

Mixed sessions build on Vasconcelos presentation of session types which we call *classical sessions* [3], by adapting choice and input/output as needed, but keeping everything else unchanged as much as possible. The result is a language with

- a single synchronisation/communication primitive: mixed choice on a given channel that
- allows for duplicated labels in choice processes, leading to non-determinism in a pure linear setting, and
- replicated output processes arising naturally from replicated mixed choices, and that
- enjoys preservation and absence of runtime errors for typable processes,
- provides for embedding classical sessions in a tight type and operational correspondence, and
- that can be embedded in classical sessions up to a minimal encoding [6].

The present paper gathers material from previous work presented at ESOP 2020 and PLACES 2020 [7, 8]. The former paper introduces mixed sessions

together with an embedding of classical into mixed sessions. The latter paper presents an encoding of a sublanguage of mixed choices into classical sessions; here we address the full language.

The rest of the paper is organised as follows: the next section shows mixed sessions in action; Section 3 introduces the technical development of the language, and Section 4 proves the main results (preservation and absence of runtime errors for typable processes). Section 5 recalls classical session types and the encoding criteria. Section 6 presents the classical-to-mixed and Section 7 presents the mixed-to-classical encoding. Section 8 discusses implementation details, and Section 9 explores related work. Section 10 concludes the paper.

2. There Is Room for Mixed Sessions

This section introduces the main ideas behind mixed sessions via examples. We address *mixed choices*, *duplicated labels in choices*, and *unrestricted output*, in this order.

2.1. Mixed Choices

Consider the producer-consumer problem where the producer produces only insofar as so requested by the consumer. Here is the code for a producer that writes numbers starting from `n` on channel end `x`.

```
def produce (x, n) =
  lin x (enough?z. 0 +
         more!n. produce!(x, n+1)
  )
```

Syntax `qx(M+N)` introduces a choice between `M` and `N` on channel end `x`. Qualifier `q` is either **un** or **lin** and controls whether the process is persistent (remains after reduction) or is ephemeral (is consumed in the reduction process). Each branch in a choice is composed of a label (`enough` or `more`), a polarity mark (input `?` or output `!`), a variable or a value (`z` or `n`), and a continuation process (the syntax after the dot). The terminated process is represented by **0**; notation **def** introduces a recursive process. The **def** syntax and its encoding in the base language is from the Pict programming language [9] and taken up by Sepi [10].

A consumer that requests `n` integer values on channel end `y` can be written as follows, where () represents the only value of type **unit**.

```
def consume (y, n) =
  if n == 0
  then lin y (enough!(). 0)
  else lin y (more?z. consume!(x, n−1))
```

Suppose that `x` and `y` are two ends of the same channel. When choices on `x` and on `y` get together, a pair of matching label-polarities pairs is selected and a value transmitted from the output continuation to the input continuation.

Types for the two channel ends ensure that synchronisation succeeds. The type of `x` is **rec** `a.` **lin** `&{enough?unit.end, more!int.a}` where the qualifier **lin** says

that the channel end must be used in exactly one process, & denotes external choice, and each branch is composed of a label, a polarity mark, the type of the communication, and that of the continuation. The type **end** states that no further interaction is possible at the channel and **rec** introduces a recursive type. The type of y is obtained from that of x by inverting views (⊕ and &) and polarities (! and ?), yielding **rec** b. **lin** ⊕{enough!**unit**.**end**, more?**int**.b}. The choice at x in the produce process contains all branches in the type and so we select an external choice view & for x. The choices at y contain only part of the branches, hence the internal choice view ⊕. This type discipline ensures that processes do not engage in runtime errors when trying to find a match for two choices at the two ends of a same channel.

A few type and process abbreviations simplify coding: i) the **lin** qualifier can be omitted, ii) the terminated process **0** together with the trailing dot can be omitted; iii) the terminated type **end** together with the trailing dot can be omitted; and iv) we introduce wildcards (_) in variable binding positions (in input branches).

2.2. Duplicated Labels in Choices for Types and for Processes

Classical session types require distinct identifiers to label distinct branches. Mixed sessions relax this restriction by allowing duplicated labels whenever paired with distinct polarities. The next example describes two processes—countDown and collect—that bidirectionally exchange a fixed number of msg-labelled messages. The number of messages that flow in each direction is not fixed a priori, but instead decided by the non-deterministic operational semantics. The type that describes channel x in the code of process countDown below is **rec** a.⊕{msg!**unit**.a, msg?**unit**.a, done!**unit**}, where one can see the msg label in two distinct branches, but with different polarities.

Process countDown features a parameter n that controls the number of messages exchanged (sent or received). The end of the interaction (when n reaches 0) is signalled by a done message.

```
countDown : (rec a.⊕{msg!unit.a, msg?unit.a, done!unit}, int)
def countDown (x, n) =
  if n == 0
  then x (done!())
  else x (msg!(). countDown!(x, n−1) +
          msg?_. countDown!(x, n−1))
```

Process collect sees the channel from the dual viewpoint, obtained by exchanging ? with ! and ⊕ with &. Parameter n in this case denotes the number of messages received. When done, the process writes the result on channel end r, global to the collect process.

```
collect : (rec b.&{msg!unit.b, msg?unit.b, done?unit}, int)
def collect (y, n) =
  y (msg!(). collect!(y, n+1) +
     msg?_. collect!(y, n) +
     done?_. r (result!n))
```

Mixed sessions allow for duplicated message-polarity pairs permitting a new form of non-determinism that uses exclusively linear channels. A process of the form $(\nu xy)P$ declares a channel with end points x and y to be used in process P. The process

```
(νxy)(
  x (msg!()) |
  y (msg?_. z (m!true) + msg?_. z (m!false))
)
```

featuring two linear choices may reduce to either z (m!**true**) or to z (m!**false**). Non-determinism in the π-calculus without choice ([11, 12] for example) can only be achieved by introducing race conditions on **un** channels. For example, the π-calculus process

```
(νxy)(x!() | y?_.z!true | y?_.z!false))
```

reduces either to (z!**true** | (νxy)y?_.z!**false**)) or to (z!**false** | (νxy)y?_.z!**true**)), leaving for the runtime the garbage collection of the inert residuals. Also note that in this case, channel y cannot remain linear.

Duplicated message-polarities in choices lead to elegant and concise code. A random number generator with a given number n of bits can be written with two processes. The first process sends n messages on channel end x. The contents of the messages are irrelevant (we use value () of type **unit**); what is important is that n more messages are sent, followed by a done message, followed by silence.

```
write : (rec a.⊕{done!unit, more!unit.a}, int)
def write (x, n) =
  if n == 0
  then x(done!())
  else x(more!(). write!(x, n−1))
```

The reader process reads the more messages in two distinct branches and interprets the messages received on one branch as bit 0, and on the other as 1. Upon the reception of a done message, the accumulated random number is conveyed on channel end r, a variable global to the read process.

```
read : (rec b.&{done?unit, more?unit.b}, int)
def read (y, n) =
  y (done?_. r (result!n) +
     more?_. read!(y, 2*n) +
     more?_. read!(y, 2*n+1)
  )
```

Notice that mixed sessions allow duplicated label-polarity pairs in processes but not in types. This point is further discussed in Section 3. Also note that duplicated message labels could be easily added to traditional session types.

2.3. Unrestricted Output

Mixed sessions allow for replicated output processes. The original version of the π-calculus [13, 14] features recursion on arbitrary processes. Subsequent

versions [12] introduce replication but restricted to input processes. When compared to languages with unrestricted input only, unrestricted output allows for more concise programs and fewer message exchanges for the same effect. Here is a process (call it P) containing a pair of processes that exchange msg-labelled messages ad-aeternum,

```
(νxy)(un y (msg!()) | un x (msg?_))
```

where x is of type rec a.un &{msg?unit.a}. The un prefix denotes replication: an un choice survives reduction. Because none of the two sub-processes features a continuation, process P reduces to P in one step. The behaviour of un y (msg!()) can be mimicked by a process without output replication, namely,

```
(νwz) w (ℓ!()) | un z (ℓ?_. y (msg!(). w (ℓ!())))
```

Even if unrestricted output can be simulated with unrestricted input, the encoding requires one extra channel (wz) and an extra message exchange (on channel wz) in order to reestablish the output on channel end y.

It is a fact that unrestricted output can be added to any flavour of the π-calculus (session-typed or not). In the case of mixed sessions it arises naturally: there is only one communication primitive—choice—and this can be classified as lin or un. If an un-choice happens to behave in "output mode", then we have an un-output. It is not obvious how to design the language of mixed choices without allowing unrestricted output, while still allowing unrestricted input (which is mandatory for unbounded behaviour).

3. The Syntax and Semantics of Mixed Sessions

This section introduces the syntax and the semantics of mixed sessions. Inspired in Vasconcelos' formulation of session types for the π-calculus [3, 4], mixed sessions replace input and output, selection and branching (internal and external choice), with a single construct which we call *choice*.

3.1. Syntax

Figure 1 presents the syntax of values and processes. Let x, y, z range over a (countable) set of *variables*, and let l range over a set of *labels*. Metavariable v ranges over *values*. Following the tradition of the π-calculus set up by Milner et al. [13, 14], variables are used both as placeholders for channels and for incoming values in communication. Linearity constraints, central to session types but absent in the π-calculus, dictate that the two ends of a channel must be syntactically distinguished; we use one variable for each end [3]. Even if values can be faithfully encoded in the π-calculus, they become quite handy as primitive constructs. Here we pick the boolean values (so that we may have a conditional process) and unit that plays its role in the embeddings (Sections 6 and 7).

Metavariables P and Q range over processes. Choices are processes of the form $qx\sum_{i\in I} M_i$ offering a choice of M_i alternatives on channel end x, where I denotes a finite, non-empty, set of indices. Qualifier q describes how choice

v	::=	Values:
	x	variable
	$\mathsf{true} \mid \mathsf{false}$	boolean values
	$()$	unit value
P	::=	Processes:
	$qx\sum_{i\in I} M_i$	choice
	$P \mid P$	parallel composition
	$(\nu xx)P$	scope restriction
	$\mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ P$	conditional
	$\mathbf{0}$	inaction
M	::=	Branches:
	$l\star v.P$	branch
$\star$	::=	Polarities:
	$! \mid ?$	out and in
q	::=	Qualifiers:
	$\mathsf{lin} \mid \mathsf{un}$	linear and unrestricted

Figure 1: The syntax of processes

behaves with respect to reduction. If q is lin, then the choice is consumed in reduction, otherwise q must be un, and in this case the choice persists after reduction. The type system in Figure 9 rejects nullary (empty) choices. Choices are composed of two kinds of branches: output $l!v.P$ and input $l?x.P$. An output branch sends value v and continues as P. An input branch receives a value and continues as P with the value replacing variable x. The type system in Figure 9 makes sure that value v in $l?v.P$ is a variable.

The remaining process constructors are standard in the π-calculus. Processes of the form $P \mid Q$ denote the parallel composition of processes P and Q. Scope restriction $(\nu xy)P$ binds together the two channel ends x and y of a same channel in process P. The conditional process $\mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ behaves as process P if v is true and as process Q otherwise. Since we do not have nullary choices, we include $\mathbf{0}$—called inaction—as primitive to denote the terminated process.

3.2. Operational Semantics

The variable bindings in the language are as follows: variables x and y are bound in P in a process of the form $(\nu xy)P$; variable x is bound in P in a choice of the form $l?x.P$. The sets of bound and free variables, as well as substitution, $P[v/x]$, are defined accordingly. We take processes "up to renaming of bound

Structural congruence, $P \equiv P$

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P \qquad (\nu xy)P \equiv (\nu yx)P$$
$$(\nu xy)P \mid Q \equiv (\nu xy)(P \mid Q) \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0} \qquad (\nu wx)(\nu yz)P \equiv (\nu yz)(\nu wx)P$$

Reduction, $P \to P$

$$\mathsf{if\ true\ then}\ P\ \mathsf{else}\ Q \to P \qquad \mathsf{if\ false\ then}\ P\ \mathsf{else}\ Q \to Q \qquad \text{(R-IfT,R-IfF)}$$

$$(\nu xy)(\mathsf{lin}\, x(l!v.P + M) \mid \mathsf{lin}\, y(l?z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid R) \qquad \text{(R-LinLin)}$$

$$(\nu xy)(\mathsf{lin}\, x(l!v.P + M) \mid \mathsf{un}\, y(l?z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}\, y(l?z.Q + N) \mid R) \qquad \text{(R-LinUn)}$$

$$(\nu xy)(\mathsf{un}\, x(l!v.P + M) \mid \mathsf{lin}\, y(l?z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}\, x(l!v.P + M) \mid R) \qquad \text{(R-UnLin)}$$

$$(\nu xy)(\mathsf{un}\, x(l!v.P + M) \mid \mathsf{un}\, y(l?z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}\, x(l!v.P + M) \mid \mathsf{un}\, y(l?z.Q + N) \mid R) \qquad \text{(R-UnUn)}$$

$$\frac{P \to Q}{(\nu xy)P \to (\nu xy)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \qquad \text{(R-Res,R-Par,R-Struct)}$$

Figure 2: Operational semantics

variables", in the sense that we identify processes that differ only in the names of bound variables. In other words, we follow Barendregt *variable convention*: the bound variables of processes that occur in a mathematical context are taken to be different from the free variables [15].

Figure 2 summarizes the operational semantics of mixed sessions. Following the tradition of the π-calculus, a binary relation on processes—*structural congruence*—rearranges processes when preparing these for reduction. Such an arrangement reduces the number of rules to be included in the operational semantics. Structural congruence was introduced by Milner [11, 12]. It is defined as the least congruence relation closed under the axioms in Figure 2. The first three rules state that parallel composition is commutative, associative, and takes inaction as the neutral element. The fourth rule allows exchanging the two channel ends in channel creation; this rule is absent in Vasconcelos [3] but allows reducing the number of otherwise necessary communication axioms in reduction. The fifth rule is commonly known as scope extrusion [13, 14] and allows extending the scope of channel ends x, y to process Q. The side-condition "x and y not free in Q" is redundant in face of the variable convention. The penultimate rule allows collecting channel bindings no longer in use, and the last rule allows for rearranging the order of channel bindings in a process.

Reduction includes six axioms, two for the destruction of boolean values (via

a conditional process), and four for communication. The axioms for communication take processes of a similar nature. Scope restriction (νxy) identifies the two ends of the channel engaged in communication. Under the scope of the channel one finds three processes: the first contains an output process on channel end x, the second contains an input process on channel end y, and the third (R) is an arbitrary process that may contain other references to x and y (the witness process). Communication proceeds by identifying a pair of compatible branches, namely $l!v.P$ and $l?z.Q$. The result contains the continuation process P and the continuation process Q with occurrences of the bound variable z replaced by value v (together with the witness process). The four axioms differ in the treatment of the process qualifiers: lin (ephemeral) and un (persistent). Ephemeral processes are consumed in reduction, persistent processes remain in the contractum.

Choices apart, rules R-LinLin and R-LinUn are already present in the works of Milner and Vasconcelos [12, 3]. Rules R-UnLin and R-UnUn are absent on the grounds of economy: replicated output can be simulated with a new channel and a replicated input. In mixed choices these rules cannot be omitted for there is no distinction between input and output: choice is the only (symmetrical) communication primitive.

We have designed mixed choices in such a way that labels may be duplicated in choices; more: label-polarity pairs may be also be duplicated. This allows for non-determinism in a linear setting. For example, process

$$(\nu xy)(\mathsf{lin}\, x(l!\mathsf{true}.\mathbf{0} + l!\mathsf{false}.\mathbf{0}) \mid \mathsf{lin}\, y(l?z.\mathsf{lin}\, w(m!z.\mathbf{0})))$$

reduces in one step to either $\mathsf{lin}\, w(m!\mathsf{true}.\mathbf{0})$ or $\mathsf{lin}\, w(m!\mathsf{false}.\mathbf{0})$.

The examples in Section 2 take advantage of a def notation, a derived process construct inspired in the SePi [10] and the Pict languages [9]. A process of the form $\mathsf{def}\ x(z) = P\ \mathsf{in}\ Q$ is understood as

$$(\nu xy)(\mathsf{un}\, y(\ell?z.P) \mid Q))$$

and calls to the recursive procedure, of the form $x!v$, are interpreted as $\mathsf{lin}\, x(\ell!v)$, for ℓ an arbitrarily chosen label. The derived syntax hides channel end y and simplifies the syntax of calls to the procedure. Procedures with more than one parameter require tuple passing, a notion that is not primitive to mixed sessions but that is easy to encode; see Vasconcelos [3].

3.3. Typing

Figure 3 summarises the syntax of types. We rely on an extra set, that of *type variables*, $a, b, \ldots$ Types describe values, including boolean and unit values, and channel ends. A type of the form $q\sharp\{U_i\}_{i\in I}$ denotes a channel end. Qualifier q states the number of processes that may contain references to the channel end: exactly one for lin, zero or more for un. View $\sharp$ distinguishes internal ($\oplus$) from external (&) choice. This distinction is not present in processes but is of paramount importance for typing purposes, as we shall see. The branches are

$$
\begin{array}{lll}
T & ::= & \text{Types:} \\
 & q\sharp\{B_i\}_{i\in I} & \text{choice} \\
 & \mathsf{end} & \text{termination} \\
 & \mathsf{unit} & \text{unit} \\
 & \mathsf{bool} & \text{boolean} \\
 & \mu a.T & \text{recursive type} \\
 & a & \text{type variable} \\
B & ::= & \text{Branches:} \\
 & l\star T.T & \text{branch} \\
\sharp & ::= & \text{Views:} \\
 & \oplus \mid \& & \text{internal and external} \\
\Gamma & ::= & \text{Contexts:} \\
 & \cdot & \text{empty} \\
 & \Gamma, x\colon T & \text{entry}
\end{array}
$$

Figure 3: The syntax of types

Coinductive type equivalence, $T \simeq T$

$$
\frac{}{\mathsf{end} \simeq \mathsf{end}} \qquad \frac{}{\mathsf{unit} \simeq \mathsf{unit}} \qquad \frac{}{\mathsf{bool} \simeq \mathsf{bool}}
$$

$$
\frac{T_i \simeq T_i' \quad U_i \simeq U_i'}{q\sharp\{l\star_i T_i.U_i\}_{i\in I} \simeq q\sharp\{l\star_i T_i'.U_i'\}_{i\in I}} \qquad \frac{T[\mu a.T/a] \simeq U}{\mu a.T \simeq U} \qquad \frac{T \simeq U[\mu a.U/a]}{T \simeq \mu a.U}
$$

Figure 4: Coinductive type equivalence

either of output—$l!T.U$—or of input—$l?T.U$—nature. In either case, T denotes the object of communication and U describes the subsequent behaviour of the channel end. Type end denotes the channel end on which no more interaction is possible. Types $\mu a.T$ and a cater for recursive types.

Types are subject to a few syntactic restrictions: i) choices must have at least one branch; ii) label-polarity pairs—$l\star$—are pairwise distinct in the branches of a choice type (unlike in processes); iii) recursive types are assumed contractive (that is, containing no subterm of the form $\mu a_0.\mu a_1 \ldots \mu a_n.a_0$ with $n \geq 0$). New variables, new bindings: type variable a is bound in T in type $\mu a.T$. Again the definitions of bound and free names as well as that of substitution—$T[U/a]$—are defined accordingly.

The coinductive rules for *type equivalence* are presented in Figure 4. By coinductive rules, we are stating that, in fact, the type equivalence relation is

Polarity duality and view duality, $\sharp \perp \sharp$ and $\star \perp \star$

$$! \perp ? \qquad ? \perp ! \qquad\qquad \oplus \perp \& \qquad \& \perp \oplus$$

Coinductive type duality, $T \perp T$

$$\frac{}{\mathsf{end} \perp \mathsf{end}} \qquad \frac{\sharp \perp \flat \qquad \star_i \perp \bullet_i \qquad T_i \simeq T_i' \qquad U_i \perp U_i'}{q\sharp\{l\star_i T_i.U_i\}_{i\in I} \perp q\flat\{l\bullet_i T_i'.U_i'\}_{i\in I}}$$

$$\frac{T[\mu a.T/a] \perp U}{\mu a.T \perp U} \qquad \frac{T \perp U[\mu a.U/a]}{T \perp \mu a.U}$$

Figure 5: Coinductive type duality

defined via a fixed-point construction in the following manner: two types T, U are *equivalent*—$T \simeq U$—if the pair (T, U) belongs to the greatest fixed-point of the monotone operator $F_\simeq$ defined from the rules as

$$\begin{aligned} F_\simeq(R) =& \{(\mathsf{end}, \mathsf{end})\} \\ & \cup \{(\mathsf{unit}, \mathsf{unit})\} \\ & \cup \{(\mathsf{bool}, \mathsf{bool})\} \\ & \cup \{(q\sharp\{l\star_i T_i.U_i\}_{i\in I}, q\sharp\{l\star_i T_i'.U_i'\}_{i\in I}) \mid (U_i, U_i'), (T_i, T_i') \in R\} \\ & \cup \{(\mu a.T, U) \mid (T[\mu a.T/a], U) \in R\} \\ & \cup \{(T, \mu a.U) \mid (T, U[\mu a.U/a]) \in R\} \end{aligned}$$

Similarly to type equivalence, duality and subtyping are also defined coinductively. We refrain from presenting the monotone operator for it can be readily extracted from the rules in Figures 5 and 6.

Duality is a notion central to session types. In order for channel communication to proceed smoothly, the types for the two channel ends must be compatible: if one end says input, the other must say output; if one end says external choice, the other must say internal choice. In presence of recursive types, the problem of building the dual of a given type has been elusive, as works by Bernardi and Hennessy, Bono and Padovani, Gay et al., and Lindley and Morris suggest [16, 17, 18, 19]. Here we eschew the problem by working with a duality relation, as in Gay et al. [18, 20] and other works, such as Kouzapas et al. [6].

We define what it means for two types to be *dual* using a fixed point construction and present the coinductive rules in Figure 5. Type end is the dual of itself. The rule for choice types requires dual views ($\&$ is the dual of $\oplus$, and vice-versa) and dual polarities (? is the dual of !, and vice-versa). Furthermore, the objects of communication must be equivalent and the continuations must be again dual. The other two rules unfold a recursive type on the left and on the right. As an example, we can establish that

$$\mu a.\mathsf{lin}\oplus\{l!\mathsf{bool}.\mathsf{lin}\&\{m?\mathsf{unit}.a\}\} \perp \mathsf{lin}\&\{l?\mathsf{bool}.\mu b.\mathsf{lin}\oplus\{m!\mathsf{unit}.\mathsf{lin}\&\{l?\mathsf{bool}.b\}\}\}$$

Branch subtyping, $B <: B$

$$\frac{T_2 <: T_1 \quad U_1 <: U_2}{l!T_1.U_1 <: l!T_2.U_2} \qquad \frac{T_1 <: T_2 \quad U_1 <: U_2}{l?T_1.U_1 <: l?T_2.U_2}$$

Coinductive subtyping, $T <: T$

$$\frac{}{\mathsf{end} <: \mathsf{end}} \qquad \frac{}{\mathsf{unit} <: \mathsf{unit}} \qquad \frac{}{\mathsf{bool} <: \mathsf{bool}}$$

$$\frac{J \subseteq I \qquad B_j <: C_j}{q\oplus\{B_i\}_{i\in I} <: q\oplus\{C_j\}_{j\in J}} \qquad \frac{I \subseteq J \qquad B_i <: C_i}{q\&\{B_i\}_{i\in I} <: q\&\{C_j\}_{j\in J}}$$

$$\frac{T[\mu a.T/a] <: U}{\mu a.T <: U} \qquad \frac{T <: U[\mu a.U/a]}{T <: \mu a.U}$$

Figure 6: Coinductive subtyping

Lemma 1 (Properties of duality)**.**

1. *Duality is symmetric, that is, if $T \perp U$, then $U \perp T$.*
2. *Duality is an involution, that is, if $S \perp T$ and $T \perp U$, then $S \simeq U$.*

Proof. We sketch the first case. Consider the operator $F_\perp$ defined from the rules of Figure 5 and the set $S = \{(U, T) \mid T \perp U\}$. We need to show that $S \subseteq F_\perp(S)$ to conclude the result using the coinduction principle. We show the choice case: when T is $q\sharp\{l\star_i T_i.U_i\}_{i\in I}$ then U is $q\flat\{l\bullet_i T_i'.U_i'\}_{i\in I}$, which from the duality rule we know that $\sharp \perp \flat, \star_i \perp \bullet_i, T_i \simeq T_i', U_i \perp U_i'$. Given that the view and polarity dualities are symmetric, and the type equivalence is also symmetric, we conclude that $\flat \perp \sharp, \bullet_i \perp \star_i, T_i' \simeq T_i$. Furthermore, given that $U_i \perp U_i'$, by definition of S, $(U_i', U_i) \in S$. Thus, by definition of $F_\perp$, $(q\flat\{l\bullet_i T_i'.U_i'\}_{i\in I}, q\sharp\{l\star_i T_i.U_i\}_{i\in I}) \in F_\perp(S)$. □

Mixed sessions come equipped with a notion of *subtyping*. We define both type and branch subtyping simultaneously, using the coinductive rules in Figure 6. These are essentially the rules of Gay and Hole [20] for algorithmic subtyping. Base types (end, unit, bool) are subtypes to themselves. Subtyping behaves differently in presence of external or internal choice. For external choice we require the branches in the supertype to contain those in the subtype: exercising less options cannot cause difficulties on the receiving side. For internal choice we require the opposite: here offering more choices can not cause runtime errors. For branches we distinguish output from input: output is contravariant on the contents of the message, input is covariant. In either case, the continuation is covariant. Choices, input/output, and recursive types receive no different treatment than those in classical sessions [20]. The operator handles recursion in the exact same way as the operator $F_\perp$. We can show that subtyping is a preorder.

Lemma 2 (Subtyping is a pre-order)**.** *For X, Y, Z either all types or all branches,*

The un predicate, T un

$$\frac{}{\mathsf{end}\ \mathrm{un}} \qquad \frac{}{\mathsf{unit}\ \mathrm{un}} \qquad \frac{}{\mathsf{bool}\ \mathrm{un}} \qquad \frac{}{\mathsf{un}\sharp\{B_i\}_{i\in I}\ \mathrm{un}} \qquad \frac{T\ \mathrm{un}}{\mu a.T\ \mathrm{un}}$$

Figure 7: The un predicate on types

1. $X <: X$.
2. *If* $X <: Y$ *and* $Y <: Z$*, then* $X <: Z$.

Proof. Item 1. Consider the operator $F_{<:}$ defined from the rules of Figure 6 and the set

$$\mathcal{I} = \{(T,T), (\mu a.T, T[\mu a.T/a]) \mid T \text{ is a type}\} \cup \{(B,B) \mid B \text{ is a branch}\}.$$

we need to show that $\mathcal{I} \subseteq F_{<:}(\mathcal{I})$, which via the coinduction principle allows to conclude that $\mathcal{I} \subseteq \nu X.F_{<:}(X)$, the greatest fixed-point of $F_{<:}$. Let $P = (T,T) \in \mathcal{I}$ and consider the possible cases: if T is $\mathsf{end}, \mathsf{unit}$ or bool, then $P \in F_{<:}(\mathcal{I})$ by definition of $F_{<:}$; if T is $q\sharp\{B_i\}_{i\in I}$, since $(B,B) \in \mathcal{I}$ for any branch, we have $(q\sharp\{B_i\}_{i\in\mathcal{I}}, q\sharp\{B_i\}_{i\in\mathcal{I}}) \in F_{<:}(\mathcal{I})$; if T is $\mu a.T$, since $(\mu a.T, T[\mu a.T/a]) \in \mathcal{I}$, by definition of $F_{<:}$, we have $P = (\mu a.T, \mu a.T) \in F_{<:}(\mathcal{I})$. Finally, since $(T[\mu a.T/a], T[\mu a.T/a]) \in \mathcal{I}$, by definition of $F_{<:}$, $(\mu a.T, T[\mu a.T/a]) \in F_{<:}(\mathcal{I})$. The case of branches is also direct. This concludes that $\mathcal{I} \subseteq F_{<:}(\mathcal{I})$, which by the coinduction principle shows that the elements in $\mathcal{I}$ are indeed in the greatest fixed point of $F_{<:}$.

Item 2. Consider the set $\mathcal{T} = \{(X,Z) \mid \exists Y : X <: Y \text{ and } Y <: Z\}$. We sketch the external choice case: consider that Y is a type of the form $q\&\{Y_j\}_{j\in J}$, which makes X of the form $q\&\{X_i\}_{i\in I}$ and Z as $q\&\{Z_k\}_{k\in K}$. Furthermore, from the subtyping rules we know that $I \subseteq J, X_i <: Y_i$ and $J \subseteq K, Y_j <: Z_j$. Thus, we conclude that $I \subseteq K$ and $X_i <: Z_i$, which from the definition of $F_{<:}$, follows that $(X,Z) \in F_{<:}(T)$. □

A similar reasoning shows that $\simeq \subseteq <:$.

The meaning of the un predicate is defined by the rules in Figure 7. Basic types—unit, bool, end—are unrestricted; un-annotated choices are unrestricted; $\mu a.T$ is unrestricted if T is. Contractivity ensures that the predicate is total.

Before presenting the type system, we need to introduce two operations that manipulate typing contexts. The rules in Figure 8 define the meaning of *context split* and *context update*. These two relations are taken verbatim from Vasconcelos [3]; context split is originally from Walker [21] (cf. Kobayashi et al. [22, 23]). Context split is used when type checking processes with two subprocesses. In this case we split the context in two, by copying unrestricted entries to both contexts and linear entries to one only. Context update is used to add to a given context an entry representing the continuation (after a choice operation) of a channel. If the variable in the entry is not in the context, then we add the entry to the context. Otherwise we require the entry to be present in the context and the type to be unrestricted.

Context split, $\Gamma = \Gamma \circ \Gamma$

$$\frac{}{\cdot = \cdot \circ \cdot} \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad T\,\mathsf{un}}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \circ (\Gamma_2, x\colon T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon \mathsf{lin}\, p = (\Gamma_1, x\colon \mathsf{lin}\, p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon \mathsf{lin}\, p = \Gamma_1 \circ (\Gamma_2, x\colon \mathsf{lin}\, p)}$$

Context update, $\Gamma + x\colon T = \Gamma$

$$\frac{x\colon U \notin \Gamma}{\Gamma + x\colon T = \Gamma, x\colon T} \qquad \frac{T\,\mathsf{un}}{(\Gamma, x\colon T) + x\colon T = (\Gamma, x\colon T)}$$

Figure 8: Context split and context update

The rules in Figure 9 introduce the typing system for mixed sessions. We require all instances of the axioms to be built from unrestricted contexts, thus ensuring that linear resources (channel ends) are fully consumed in typing derivations.

The typing rules for values should be straightforward: constants have their own types, the type for a variable is read from the context, and T-Sub is the subsumption rule, allowing a type for a value to be replaced by a supertype.

The rules for branches—T-Out and T-In—follow those for output and input in classical session types. To type an output branch we split the context in two: one part for the value, the other for the continuation process. To type an input branch we add an entry with the bound variable x to the context under which we type the continuation process. Rule T-In rejects branches of the form $l?v.P$ when v not a variable. The continuation type T is not used in neither rule; instead it is incorporated in Γ via the type for the channel of the choice process (cf. rule T-Choice below).

The rules for inaction, parallel composition, and conditional are from Vasconcelos [3]. That for scope restriction is adapted from Gay and Hole [20]. Rule T-Inact follows the general pattern for axioms, requiring an un context. Rule T-Par splits the context in two, providing each subprocess with one part. Rule T-If splits the context and uses one part to type guard v. Because v is unrestricted, we know that Γ_1 contains exactly the un entries in $\Gamma_1 \circ \Gamma_2$ and that Γ_2 is equal to $\Gamma_1 \circ \Gamma_2$. Context Γ_2 is used to type *both* branches of the conditional, for only one of them will ever execute. Rule T-Res introduces in the typing context entries for the two channel ends, x and y, at dual types.

The rule for choice, T-Choice, is new. We first introduce predicates $\Gamma\, q$ on contexts, for $q = \mathrm{un}, \mathrm{lin}$. Predicate $\Gamma\,\mathrm{un}$ is the pointwise extension of the predicate with the same name for types, in Figure 7. Predicate $\Gamma\,\mathrm{lin}$ is true of all Γ. Hence, all contexts are linear and only some contexts are unrestricted.

The incoming context is split in two: one for the subject x of the choice, the other for the various branches in the choice. The process qualifier, q_1, dictates the nature of the incoming context—un or lin—as witnessed by the first

Typing rules for values, $\Gamma \vdash v \colon T$

$$\frac{\Gamma \,\mathsf{un}}{\Gamma \vdash () \colon \mathsf{unit}} \qquad \frac{\Gamma \,\mathsf{un}}{\Gamma \vdash \mathsf{true}, \mathsf{false} \colon \mathsf{bool}} \qquad \frac{\Gamma_1, \Gamma_2 \,\mathsf{un}}{\Gamma_1, x \colon T, \Gamma_2 \vdash x \colon T} \qquad \frac{\Gamma \vdash v \colon T \quad T <: U}{\Gamma \vdash v \colon U}$$

(T-Unit,T-True,T-False,T-Var,T-Sub)

Typing rules for branches, $\Gamma \vdash M \colon B$

$$\frac{\Gamma_1 \vdash v \colon T \qquad \Gamma_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash l!v.P \colon l!T.U} \qquad \frac{\Gamma, x \colon T \vdash P}{\Gamma \vdash l?x.P \colon l?T.U} \qquad \text{(T-Out,T-In)}$$

Typing rules for processes, $\Gamma \vdash P$

$$\frac{\Gamma \,\mathsf{un}}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2} \qquad \text{(T-Inact,T-Par)}$$

$$\frac{\Gamma_1 \vdash v \colon \mathsf{bool} \qquad \Gamma_2 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q} \qquad \frac{\Gamma, x \colon T, y \colon U \vdash P \qquad T \perp U}{\Gamma \vdash (\nu xy)P}$$

(T-If,T-Res)

$$\frac{(\Gamma_1 \circ \Gamma_2)\, q_1 \qquad \Gamma_1 \vdash x \colon q_2 \sharp \{l \star_i T_i.U_i\}_{i \in I} \qquad \Gamma_2 + x \colon U_j \vdash l \star_j v_j.P_j \colon l \star_j T_j.U_j \qquad \{l \star_j\}_{j \in J} = \{l \star_i\}_{i \in I}}{\Gamma_1 \circ \Gamma_2 \vdash q_1 x \sum_{j \in J} l \star_j v_j.P_j} \qquad \text{(T-Choice)}$$

Figure 9: Typing

premise to the rule. This allows for a linear choice to contain channels of an arbitrary nature, but limits unrestricted choices to unrestricted channels only (for one cannot predict how many times such choices will be exercised). The second premise extracts a type $q_2 \sharp \{l_i \star T_i.U_i\}$ for x. The third premise types each branch in turn: type T_j is used to type values v_j in the branches and each type U_j is used to type the corresponding continuation. The rule updates context Γ_2 with the continuation type of x: if q_2 is lin, then x is not in Γ_2 and the update operation simply adds the entry to the context. If, on the other hand, q_2 is un, then x is in Γ_2 and the context update operation (together with rule T-Sub) insists that type U_j is a subtype of $\mathsf{un} \sharp \{l_j \star T_j.U_j\}$, meaning that U_j is a recursive type.

The last premise to rule T-Choice dictates that the set of labels in the choice type coincides with that in the choice process. That does not mean that the label-polarity pairs are in a one-to-one correspondence: label-polarity pairs are pairwise distinct in types (see the syntactic restrictions in Section 3.3), but not in processes. For example, process $\mathsf{lin} x(l?y.\mathbf{0} + l?z.\mathbf{0})$ can be typed against context $x \colon \mathsf{lin} \oplus \{l?\mathsf{bool}.\mathsf{end}\}$. Furthermore, from the fact that the two sets must coincide does not follow that the label-polarity pairs type in the context must coincide with those in the process. Taking advantage of subtyping, the above process can still be typed against context $x \colon \mathsf{lin} \oplus \{l?\mathsf{bool}.\mathsf{end}, m!\mathsf{unit}.\mathsf{end}\}$ because $\mathsf{lin} \oplus \{l?\mathsf{bool}.\mathsf{end}, m!\mathsf{unit}.\mathsf{end}\} <: \mathsf{lin} \oplus \{l?\mathsf{bool}.\mathsf{end}\}$. The opposite phenomenon hap-

pens with external choice, where one may remove branches by virtue of subtyping.

We complete this section by discussing examples that illustrate options taken while building the typing system (we postpone the formal justification to Section 4). Suppose we allow empty choices in the syntax of types. Then process

$$(\nu xy)(x() \mid y())$$

would be typable by taking $x\colon \oplus(), y\colon \&()$, yet the process would not reduce. We could add an extra reduction rule for the effect

$$(\nu xy)(x() \mid y() \mid R) \to (\nu xy)R$$

which would satisfy preservation (Theorem 7). We decided not to include it in the reduction rules as we felt no need for the extra complexity. Including the rule also does not bring any apparent benefit.

The syntax of processes places no restrictions on the label-polarity pairs in choices; yet that of types does. What if we relax the restriction that label-polarities pairs in choice types must be pairwise distinct? Then process

$$(\nu xy)(x(l!\mathsf{true} + l!()) \mid y(l?z.\mathsf{if}\ z\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}))$$

could be typed under the empty context, yet the process might reduce to $\mathsf{if}\ ()\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$ which is a runtime error.

4. Well-typed Mixed Sessions Do Not Lead to Runtime Errors

This section introduces the main results of mixed choices: absence of runtime errors and preservation, both for well-typed processes.

We say that a process is a *runtime error* if it is structurally congruent to:

- a process of the form

$$(\nu x_1 y_1) \dots (\nu x_n y_n)(\nu xy)(qx \sum_{i \in I} l\star_i v_i.P_i \mid q'y \sum_{j \in J} l\star_j w_j.Q_j \mid R)$$

  where $\{l\bullet_i\}_{i \in I} \cap \{l\star_j\}_{j \in J} = \emptyset$ with each $\bullet_i$ is obtained by dualising $\star_i$, or

- a process of the form $qz(M + l?v.P + N)$ and v is not a variable, or

- a process of the form $\mathsf{if}\ v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and v is neither true nor false.

Examples of processes which are runtime errors include:

$$(\nu xy)(\mathsf{lin}\, x(l!\mathsf{true}.\mathbf{0}) \mid \mathsf{lin}\, y(l!\mathsf{true}.\mathbf{0}))$$
$$(\nu xy)(\mathsf{un}\, x(l!\mathsf{true}.\mathbf{0}) \mid \mathsf{lin}\, y(m?z.\mathbf{0}))$$
$$\mathsf{un}\, x(l?\mathsf{false}.\mathbf{0})$$
$$\mathsf{if}\ ()\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$$

Notice that processes of the form $(\nu xy)\mathsf{lin}\, x \sum_{i\in I} M_i$ cannot be classified as runtime errors for they may be typed. Just think of $(\nu xy)\mathsf{lin}\, x(l?z.\mathsf{lin}\, y(l!\mathsf{true}.\mathbf{0}))$, typable under the empty context. Unlike the interpretations of session types in linear logic by Caires, Pfenning and Wadler [24, 25, 26, 27], typable mixed session processes can easily deadlock. Similarly, processes with more than one lin-choice on the same channel end can be typed. For example process $\mathsf{lin}\, x(l!\mathsf{true}.\mathbf{0}) \mid \mathsf{lin}\, x(l?z.\mathbf{0}))$ can be typed under context $x\colon \mu a.\mathsf{un} \oplus \{l!\mathsf{bool}.a, l?\mathsf{bool}.a\}$. Recall the relationship between qualifiers in processes q_1 and those in types q_2 in the discussion of the rules for choice in Section 3.

Theorem 3 (Well-typed processes are not runtime errors)**.** *If $\cdot \vdash P$, then P is not a runtime error.*

Proof. In view of a contradiction, assume that $\cdot \vdash P$ with P of the form

$$(\nu x_1 y_1)\dots(\nu x_n y_n)(q_1 x_n \sum_{i\in I} l\star_i v_i.P_i \mid q_2 y_n \sum_{j\in J} l\star_j w_j.Q_j \mid R)$$

and that $\{l\star_i\}_{i\in I} \cap \{l\bullet_j\}_{j\in J} = \emptyset$ with $\star_k \perp \bullet_k$. From the typing derivation for P, using T-Par and T-Res, we obtain a context $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 = x_1\colon T_1, y_1\colon U_1, \dots, x_n\colon T_n, y_n\colon U_n$ with $T_i \perp U_i$ for all $i = 1, \dots, n$ and that $\Gamma_1 \vdash q_1 x_n \sum_{i\in I} l\star_i v_i.P_i$ and $\Gamma_2 \vdash q_2 y_n \sum_{j\in J} l\star_j w_j.Q_j$ and $\Gamma_3 \vdash R$. From the premises of rule T-Choice we know that $\Gamma_1' \vdash x_n\colon q_1' \sharp_1 \{l\star_{i'} T_{i'}'.T_{i'}''\}_{i'\in I'}$ with $\{l\star_i\}_{i\in I} = \{l\star_{i'}\}_{i'\in I'}$, and $\Gamma_2' \vdash y_n\colon q_2' \sharp_2 \{l\bullet_{j'} U_{j'}'.U_{j'}''\}_{j'\in J'}$ with $\{l\star_j\}_{j\in J} = \{l\star_{j'}\}_{j'\in J'}$, for some Γ_1' and Γ_2'. Given that x_n and y_n have dual types, we know that $\sharp_1 \perp \sharp_2$. Assume that $\sharp_1 = \&$ (the other case is similar) and that set of labels in T_n is $\{l\star_k\}_{k\in K}$. Then, the set of labels for U_n is $\{l\bullet_k\}_{k\in K}$. Subtyping gives $\{l\star_i\}_{i\in I} \subseteq \{l\star_k\}_{k\in K}$ and $\{l\bullet_k\}_{k\in K} \subseteq \{l\bullet_j\}_{j\in J}$. Because the sets of labels are nonempty, there is a label in $\{l\star_i\}_{i\in I} \cap \{l\bullet_j\}_{j\in J}$ contradicting the assumption.

When P is $qz(M + l?v.P + N)$ and v is not a variable, the contradiction is with rule T-Out, which can only be applied when the value v is a variable.

When P is $\mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q$ and v is not a boolean value, then v must be a variable, but P is typed in the empty context, hence v cannot be a variable. □

In order to prepare for the preservation result we introduce a few lemmas.

Lemma 4 (Unrestricted weakening)**.** *Let T un.*

1. *If $\Gamma \vdash v\colon U$, then $\Gamma, x\colon T \vdash v\colon U$.*
2. *If $\Gamma \vdash M\colon B$, then $\Gamma, x\colon T \vdash M\colon B$.*
3. *If $\Gamma \vdash P$, then $\Gamma, x\colon T \vdash P$.*

Proof. Item 1 by rule induction; Items 2 and 3 by mutual rule induction.

□

Lemma 5 (Preservation for $\equiv$)**.** *If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.*

Proof. As in Vasconcelos [3, Lemma 7.4] since we share most of the structural congruence axioms. For $(\nu xy)P \equiv (\nu yx)P$ use the fact that duality is symmetric (Lemma 1). □

Lemma 6 (Substitution). *Let* $\Gamma = \Gamma_1 \circ \Gamma_2$ *and* $\Gamma_1 \vdash v\colon T$.

1. *If* $\Gamma_2, x\colon T \vdash u\colon U$, *then* $\Gamma \vdash u\colon U[v/x]$.
2. *If* $\Gamma_2, x\colon T \vdash M\colon B$, *then* $\Gamma \vdash M\colon B[v/x]$.
3. *If* $\Gamma_2, x\colon T \vdash P$, *then* $\Gamma \vdash P[v/x]$.

Proof. Item 1 by rule induction; Items 2 and 3 by mutal rule induction. □

Theorem 7 (Preservation). *If* $\Gamma \vdash P$ *and* $P \to Q$, *then* $\Gamma \vdash Q$.

Proof. By rule induction on the hypothesis on reduction, making use of weakening, preservation for structural congruence, and substitution (Lemmas 4 to 6). We sketch a couple of cases.

When reduction ends with rule R-Par, the premises to rule T-Par (the only rule that applies) read $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash R$. Then induction gives $\Gamma_1 \vdash Q$ and we conclude the case with rule T-Par.

When reduction ends with rule R-LinLin, we know that rule T-Res introduces $x\colon T, y\colon U$ with $T \perp U$ in the context Γ. From there, with applications of T-Par and T-Choice, we have $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$ and $\Gamma_1 \vdash \mathsf{lin}\, x(M + l!v.P + M')$ and $\Gamma_2 \vdash \mathsf{lin}\, y(N + l?z.Q + N')$ and $\Gamma_3 \vdash R$. Furthermore, $\Gamma_1 = \Gamma_1' \circ \Gamma_1''$ and $\Gamma_1' \vdash x\colon \mathsf{lin} \oplus \{B, l!T'.T'', B'\}$ and $\Gamma_1'', x\colon T'' \vdash l!v.P\colon l!T'.T''$. From the T-Out rule, $\Gamma_v \vdash v\colon T'$ and $\Gamma_4 \vdash P$. For the y side, $\Gamma_2' \vdash y\colon \mathsf{lin}\&\{N, l?U'.U'', N'\}$ and $\Gamma_2'', y\colon U'' \vdash l?z.Q\colon l?U'.U''$. From the T-In rule, $\Gamma_z, y\colon U'', z\colon U' \vdash Q$. We also have that $T' = U'$ from the duality of T and U. Using the substitution lemma, we get $\Gamma_z, y\colon U'', \Gamma_v \vdash Q[v/z]$. Using T-Par and T-Res we build a derivation for the conclusion of R-LinLin. □

5. Classical Session Types and the Encoding Criteria

This section introduces the syntax and semantics of classical session types and the Kouzapas et al. [6] correctness criteria for typed encodings.

5.1. Classical Session Types

The syntax and semantics of classical session types are in Figure 10; we follow Vasconcelos [3]. The syntax and the rules for the various judgements extend those of Figures 1 to 9, where we remove choice both from grammar productions (for processes and types) and from the various judgements (operational semantics, subtyping, duality, and typing). On what concerns the syntax of processes, the choice construct of Figure 1 is replaced by new process constructors: output, linear (lin) and replicated (un) input, selection (internal choice) and branching (external choice). The four reduction axioms in Figure 2 that pertain to choice (R-LinLin, R-LinUn, R-UnLin, R-UnUn) are replaced by the three axioms in Figure 10. Rule R-LinCom describes the output against ephemeral-input interaction, rule R-UnCom the output against replicated-input interaction, and rule R-Case selects a label in the menu at the other channel end.

Typing for classical session types is in Figure 11. The syntax of types features new constructs—linear and unrestricted input and output, and linear and

Classical syntactic forms (replaces some productions in Figure 1)

$$
\begin{array}{rll}
P & ::= \ldots & \text{Processes:} \\
& x!v.P & \text{output} \\
& qx?x.P & \text{input} \\
& x \lhd l.P & \text{selection} \\
& x \rhd \{l_i \colon P_i\}_{i \in I} & \text{branching}
\end{array}
$$

Classical reduction rules, $P \to P$, (plus R-Res, R-Par, and R-Struct from Figure 2)

$$
\begin{array}{rl}
(\nu xy)(x!v.P \mid \mathsf{lin}\, y?z.Q \mid R) \to (\nu xy)(P \mid Q[v/z] \mid R) & \text{(R-LinCom)} \\
(\nu xy)(x!v.P \mid \mathsf{un}\, y?z.Q \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}\, y?z.Q \mid R) & \text{(R-UnCom)} \\
\dfrac{j \in I}{(\nu xy)(x \lhd l_j.P \mid y \rhd \{l_i \colon Q_i\}_{i \in I} \mid R) \to (\nu xy)(P \mid Q_j \mid R)} & \text{(R-Case)}
\end{array}
$$

Figure 10: Classical session types: syntax and reduction

unrestricted external and internal choice—replacing the choice construct in Figure 3. The un predicate for the classical types is true of un qualified types. The lin predicate is as in mixed choices (Figure 7). The subtyping and duality rules for the new type constructors are taken from Gay and Hole [20]. The new rules for type equivalence—$T \simeq U$—should be easy to extract from the new syntax in Figure 11 and are therefore omitted.

The typing rule for choice in Figure 9 is replaced by the four typing rules in Figure 11. All rules but T-OutC are taken verbatim from Vasconcelos [3]. Rule T-OutC increases typability and ensures proper alignment with the translation from mixed sessions to classical sessions discussed in Section 7. Let T be the type $\mu a.\mathsf{lin}!a.a$, hence $T = \mathsf{lin}!T.T$. Process $x!x$ is not typable under context $x \colon T$ using the original rule for output [3], but becomes typable with rule T-OutC, as the following derivation shows.

$$
\dfrac{\dfrac{}{x \colon T \vdash x \colon \mathsf{lin}!T.T}\ \text{T-Var} \qquad \dfrac{}{x \colon T \vdash x \colon T}\ \text{T-Var} \qquad \dfrac{}{\cdot \vdash \mathsf{end}}\ \text{T-Inact}}{x \colon T \vdash x!x.\mathbf{0}}\ \text{T-OutC}
$$

5.2. Minimal Encodings

A *typed calculus* is a five tuple composed of a set of types $\mathcal{T}$, a set of processes $\mathcal{P}$, a reduction relation $\to$ on processes, an equivalence relation $\approx$ on processes, and a typing relation $\vdash$. Given two typed calculi $\langle \mathcal{T}_i, \mathcal{P}_i, \to_i, \approx_i, \vdash_i \rangle$, $i = 1, 2$, a *typed encoding* is a pair of maps, one on types $[\![\cdot]\!]_{\mathcal{T}} \colon \mathcal{T}_1 \to \mathcal{T}_2$, the other on processes $[\![\cdot]\!]_{\mathcal{P}} \colon \mathcal{P}_1 \to \mathcal{P}_2$. We assume that both calculi include the inaction

Classical syntactic forms (replaces some productions in Figure 3)

$$\begin{array}{rrl} T & ::= & \ldots & \text{Types:} \\ & & q \star T.T & \text{communication} \\ & & q\sharp\{l_i\colon T_i\}_{i\in I} & \text{choice} \end{array}$$

Classical un *predicate,* $T\,\mathrm{un}$ *(replaces some of the rules in Figure 7)*

$$\frac{}{\mathsf{un} \star T.U\ \mathrm{un}} \qquad \frac{}{\mathsf{un}\sharp\{l_i\colon T_i\}_{i\in I}\ \mathrm{un}}$$

Classical coinductive subtyping rules, $T <: T$

$$\frac{U <: T \qquad T' <: U'}{q!T.T' <: q!U.U'} \qquad \frac{T <: U \qquad T' <: U'}{q?T.T' <: q?U.U'}$$

$$\frac{J \subseteq I \qquad T_j <: U_j \qquad (\forall j \in J)}{q\oplus\{l_i\colon T_i\}_{i\in I} <: q\oplus\{l_j\colon U_j\}_{j\in J}} \qquad \frac{I \subseteq J \qquad T_i <: U_i \qquad (\forall i \in I)}{q\&\{l_i\colon T_i\}_{i\in I} <: q\&\{l_j\colon U_j\}_{j\in J}}$$

Classical coinductive type duality rules, $T \perp T$

$$\frac{\bullet \perp \star \quad T_1 \simeq T_2 \quad U_1 \perp U_2}{q\bullet T_1.U_1 \perp q\star T_2.U_2} \qquad \frac{\sharp \perp \flat \qquad T_i \perp U_i \quad (\forall i \in I)}{q\sharp\{l_i\colon T_i\}_{i\in I} \perp q\flat\{l_i\colon U_i\}_{i\in I}}$$

Classical typing rules, $\Gamma \vdash P$ *(replaces some of the rules in Figure 9)*

$$\frac{\Gamma_1 \vdash x\colon q\,!T.U \qquad \Gamma_2 + x\colon U = \Gamma_3 \circ \Gamma_4 \qquad \Gamma_3 \vdash v\colon T \qquad \Gamma_4 \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x!v.P} \quad \text{(T-OutC)}$$

$$\frac{(\Gamma_1 \circ \Gamma_2)\, q_1 \qquad \Gamma_1 \vdash x\colon q_2?T.U \qquad (\Gamma_2 + x\colon U), y\colon T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash q_1 x?y.P} \quad \text{(T-InC)}$$

$$\frac{\Gamma_1 \vdash x\colon q\&\{l_i\colon T_i\}_{i\in I} \qquad \Gamma_2 + x\colon T_i \vdash P_i \qquad (\forall i \in I)}{\Gamma_1 \circ \Gamma_2 \vdash x \rhd \{l_i\colon P_i\}_{i\in I}} \quad \text{(T-Branch)}$$

$$\frac{\Gamma_1 \vdash x\colon q\oplus\{l\colon T\} \qquad \Gamma_2 + x\colon T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x \lhd l.P} \quad \text{(T-Sel)}$$

Figure 11: Classical session types: typing

process $\mathbf{0}$, parallel composition $P \mid Q$, scope restriction $(\nu xy)P$ and that typing contexts are equipped with a context splitting operation. We denote by $\Rightarrow$ the reflexive and transitive closure of the reduction relation $\rightarrow$. A *barb* $P \downarrow_x$ is an observable on an output or selection prefix with subject x in process P [28]. A *weak barb* $P \Downarrow_x$ is a barb after zero or more reduction steps.

Given two typed calculi $\mathcal{L}_1, \mathcal{L}_2$, and a typed encoding from $\mathcal{L}_1$ to $\mathcal{L}_2$, we say that the encoding is:

Syntax preserving: If it is

Homomorphic wrt parallel composition: If $[\![\Gamma \circ \Gamma']\!] \vdash_2 [\![P \mid P']\!]$, then $[\![\Gamma]\!] \circ [\![\Gamma']\!] \vdash_2 [\![P]\!] \mid [\![P']\!]$.

Compositional wrt channel restriction: If $[\![\Gamma]\!] \vdash_2 [\![(\nu xy)P]\!]$, then $[\![\Gamma]\!] \vdash_2 (\nu xy)[\![P]\!]$.

Name invariant: If $[\![\sigma(\Gamma)]\!] \vdash_2 [\![\sigma(P)]\!]$, then $\sigma([\![\Gamma]\!]) \vdash_2 \sigma([\![P]\!])$, for any injective renaming of variables σ.

Type sound: If $\Gamma \vdash_1 P$, then $[\![\Gamma]\!] \vdash_2 [\![P]\!]$.

Barb preserving: If $\Gamma \vdash_1 P \downarrow_x$, then $[\![\Gamma]\!] \vdash_2 [\![P]\!] \Downarrow_x$.

Operationally complete: If $\Gamma \vdash_1 P$ and $P \to_1 P'$, then $[\![P]\!] \Rightarrow_2 \approx_2 [\![P']\!]$.

Operationally sound: If $[\![\Gamma]\!] \vdash_2 [\![P]\!]$ and $[\![P]\!] \to_2 Q$, then $P \to_1 P'$ and $Q \Rightarrow_2 \approx_2 [\![P']\!]$.

Minimal: If it is syntax preserving, barb preserving, and operationally complete.

5.3. The Classical and the Mixed Typed Calculi

The mixed typed calculus $\mathcal{M}$ is composed of the type language in Figure 3, the process language in Figure 1, the reduction relation in Figure 2, syntactic equality as the equivalence relation, and the typing relation in Figure 9.

The classical typed calculus $\mathcal{C}$ is composed of the type language in Figure 11, the process language and the reduction relation Figure 10, the equivalence relation defined below, and the typing relation in Figure 11.

The behavioural equivalence $\approx_{\mathcal{C}}$ for classical sessions we are interested in extends structural congruence $\equiv$ with the following rule

$$(\nu xy) \prod_{i \in I} x \lhd l_i.\mathbf{0} \;\approx_{\mathcal{C}}\; \mathbf{0}. \qquad \text{(C-Equiv)}$$

The new rule allows collecting processes that are left by the encoding of nondeterministic choice. We call it *extended structural congruence*. This equation could be an axiom in a suitable strong bisimulation (cf. Honda [29]), but extended structural congruence suffices for our purposes. It should be easy to see that the axiom preserves typability by choosing a type $\mu a.\mathsf{un} \oplus \{l_i \colon a\}_{i \in I}$ for channel end x, and that $\approx_{\mathcal{C}}$ is an equivalence relation, given that $\equiv$ is.

We now turn our attention to *barbs*. A typed mixed session process P *has a barb in* x, notation $\Gamma \vdash_{\mathcal{M}} P \downarrow_x$, if $\Gamma \vdash_{\mathcal{M}} P$ and

- $P \equiv (\nu x_n y_n) \ldots (\nu x_1 y_1)(q x \sum_{i \in I} M_i \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$ and $\Gamma \vdash_{\mathcal{M}} x \colon q \oplus \{B_i\}_{i \in I}$.

Notice that only types can reveal barbs in processes since internal choice is indistinguishable from external choice at the process level in $\mathcal{M}$.

On the other hand, a typed classical session process P *has a barb in* x, written $\Gamma \vdash_{\mathcal{C}} P \downarrow_x$, if $\Gamma \vdash_{\mathcal{C}} P$ and

Process translation

$$\begin{aligned}
[\![x!v.P]\!] &= \mathsf{lin}\, x(\mathsf{msg}!v.[\![P]\!]) \\
[\![qx?y.P]\!] &= q\, x(\mathsf{msg}?y.[\![P]\!]) \\
[\![x \lhd l.P]\!] &= \mathsf{lin}\, x(l!().[\![P]\!]) \\
[\![x \rhd \{l_i\colon P_i\}_{i\in I}]\!] &= \mathsf{lin}\, x \sum_{i\in I} l_i?y_i.[\![P_i]\!] \quad (y_i \notin \mathrm{fv}(P_i))
\end{aligned}$$

(Homomorphic for $\mathbf{0}$, $P \mid Q$, $(\nu xy)P$, and $\mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q$)
Type translation

$$\begin{aligned}
[\![q!S.T]\!] &= q{\oplus}\{\mathsf{msg}![\![S]\!].[\![T]\!]\} \\
[\![q?S.T]\!] &= q\&\{\mathsf{msg}?[\![S]\!].[\![T]\!]\} \\
[\![q{\oplus}\{l_i\colon T_i\}_{i\in I}]\!] &= q{\oplus}\{l_i!\mathsf{unit}.[\![T_i]\!]\}_{i\in I} \\
[\![q\&\{l_i\colon T_i\}_{i\in I}]\!] &= q\&\{l_i?\mathsf{unit}.[\![T_i]\!]\}_{i\in I}
\end{aligned}$$

(Homomorphic for end, unit, bool, $\mu a.T$, and a)

Figure 12: The translation of classical sessions into mixed sessions

- either $P \equiv (\nu x_n y_n) \ldots (\nu x_1 y_1)(x!v.Q \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$
- or $P \equiv (\nu x_n y_n) \ldots (\nu x_1 y_1)(x \lhd l.Q \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$.

Similarly to Kouzapas et al. [6], we consider selections as barbs.

6. Classical Sessions as Mixed Sessions

This section shows that the language of classical sessions can be embedded in that of mixed sessions. The embedding is defined in Figure 12. It consists of two maps, one for processes, the other for types. These maps act as homomorphisms on all process and type constructors not explicitly shown. For example $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$. We distinguish one *fresh* label, denoted by msg, and use it to encode input and output (both in processes and types). Input and output processes are encoded in choices with one only msg-labelled branch. The output process is qualified as lin (it does not survive reduction) and the input process reads its qualifier q from the incoming process. Choice processes in classical sessions are encoded as choices in mixed sessions. The value transmitted on the mixed session is irrelevant: we pick () of type unit for the output side, and a fresh variable y_i on the input side. Both types are linear.

Input and output types are translated into choice types. For output we *arbitrarily* pick an external choice ($\oplus$), and conversely for the input. The label in the only branch is msg in order to match our pick for processes, and the qualifier is read from the incoming type. For classical choices, we read the qualifier and the view from the incoming type. The type of the communication

in the branches of the mixed choice is unit, again so that it matches our pick for processes.

The rest of this section discusses the properties of classical-to-mixed encoding, based on the criteria discussed in Section 5.2.

Theorem 8 (Syntax Preservation)**.** *The classical-to-mixed encoding is*

1. *Homomorphic wrt parallel composition: if* $[\![\Gamma \circ \Gamma']\!] \vdash_{\mathcal{M}} [\![P \mid P']\!]$*, then* $[\![\Gamma]\!] \circ [\![\Gamma']\!] \vdash_{\mathcal{M}} [\![P]\!] \mid [\![P']\!]$*.*
2. *Compositional wrt channel restriction: if* $[\![\Gamma]\!] \vdash_{\mathcal{M}} [\![(\nu xy)P]\!]$*, then* $[\![\Gamma]\!] \vdash_{\mathcal{M}} (\nu xy)[\![P]\!]$*.*
3. *Name invariant: if* $[\![\sigma(\Gamma)]\!] \vdash_{\mathcal{M}} [\![\sigma(P)]\!]$*, then* $\sigma([\![\Gamma]\!]) \vdash_{\mathcal{M}} \sigma([\![P]\!])$*, for any injective renaming of variables* σ*.*

Proof. Items 1 and 2: immediate from the definition of the translation of processes in Figure 12. Item 3: the encoding transforms each channel in itself, hence is trivially name invariant. □

The type soundness result for the classical-to-mixed encoding is the Item 9 of Theorem 9 below. The remaining items are intermediate results necessary to prove type soundness.

Theorem 9 (Type Soundness)**.**

1. *If* $T \,\mathrm{un}_{\mathcal{C}}$*, then* $[\![T]\!] \,\mathrm{un}_{\mathcal{M}}$*.*
2. *If* $\Gamma \,\mathrm{un}_{\mathcal{C}}$*, then* $[\![\Gamma]\!] \,\mathrm{un}_{\mathcal{M}}$*.*
3. *If* $T <: U$*, then* $[\![T]\!] <: [\![U]\!]$
4. *If* $\Gamma \vdash_{\mathcal{C}} v\colon T$*, then* $[\![\Gamma]\!] \vdash_{\mathcal{M}} v\colon [\![T]\!]$*.*
5. *If* $\Gamma = \Gamma_1 \circ_{\mathcal{C}} \Gamma_2$*, then* $[\![\Gamma]\!] = [\![\Gamma_1]\!] \circ_{\mathcal{M}} [\![\Gamma_2]\!]$*.*
6. *If* $\Gamma = \Gamma_1 + x\colon T$*, then* $[\![\Gamma]\!] = [\![\Gamma_1]\!] + x\colon [\![T]\!]$*.*
7. *If* $T \simeq U$*, then* $[\![T]\!] \simeq [\![U]\!]$*.*
8. *If* $T \perp U$*, then* $[\![T]\!] \perp [\![U]\!]$*.*
9. *If* $\Gamma \vdash_{\mathcal{C}} P$*, then* $[\![\Gamma]\!] \vdash_{\mathcal{M}} [\![P]\!]$*.*

Proof. Item 1: Immediate from Figure 12; the encoding preserves the qualifiers.

Item 2 By rule induction on the hypothesis using Item 1.

Item 3: By coinduction using the set $\mathcal{T} = \{([\![T]\!], [\![U]\!]) \mid T <: U\}$.

Item 4: By rule induction on the hypothesis, using Items 2 and 3.

Items 5 and 6: By rule induction on the hypothesis.

Item 7: By coinduction using the set $\mathcal{E} = \{([\![T]\!], [\![U]\!]) \mid T \simeq U\}$.

Item 8: By coinduction using the set $\mathcal{D} = \{([\![T]\!], [\![U]\!]) \mid T \perp U\}$ and Item 7.

Item 9: By rule induction on the hypothesis. We sketch a few cases. When the derivation ends with T-InC, we use Item 1, Item 6, and induction. When the derivation ends with T-Branch, we obtain $([\![\Gamma_2]\!] + x\colon [\![T_i]\!]), y_i\colon \mathsf{unit} \vdash [\![P_i]\!]$ from the induction hypothesis $[\![\Gamma_2]\!] + x\colon [\![T_i]\!] \vdash [\![P_i]\!]$ using Item 6 and weakening (Lemma 4). □

Theorem 10 (Barb Preservation)**.** *The typed encoding* $[\![\cdot]\!] : \mathcal{C} \longrightarrow \mathcal{M}$ *preserves barbs, that is, if* $\Gamma \vdash_{\mathcal{C}} P \downarrow_x$*, then* $[\![\Gamma]\!] \vdash_{\mathcal{M}} [\![P]\!] \Downarrow_x$*.*

Proof. By case analysis for a classical typed barb.

Case $P \equiv (\nu x_n y_n) \dots (\nu x_1 y_1)(x!v.Q \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$. Then, $[\![P]\!] = (\nu x_n y_n) \dots (\nu x_1 y_1)(\mathsf{lin}\, x(\mathsf{msg!}v.[\![Q]\!]) \mid [\![R]\!])$ where $[\![\Gamma]\!] \vdash_{\mathcal{M}} x \colon q \oplus \{\mathsf{msg!}[\![S]\!].[\![T]\!]\}$ from type soundness, and $x \notin \{x_i, y_i\}_{i=1}^n$, which makes $[\![P]\!]$ a typed mixed barb.

Case $P \equiv (\nu x_n y_n) \dots (\nu x_1 y_1)(x \triangleleft l.Q \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$. Then $[\![P]\!] = (\nu x_n y_n) \dots (\nu x_1 y_1)(\mathsf{lin}\, x(l!().[\![Q]\!]) \mid [\![R]\!])$ where $[\![\Gamma]\!] \vdash_{\mathcal{M}} x \colon q \oplus \{l_i!\mathsf{unit}.[\![T_i]\!]\}_{i \in I}$ and $x \notin \{x_i, y_i\}_{i=1}^n$, which makes $[\![P]\!]$ a typed mixed barb. □

Operational correspondence states that the embedding preserves and reflects reduction. In our case, the embedding is quite tight: one reduction step in classical sessions corresponds to one reduction step in mixed sessions. There is no runtime penalty in running classical sessions on a mixed sessions machine. Further notice that we do not rely on any equivalence relation on mixed sessions to establish the result: mixed-sessions images leave no "junk" in the process of simulating classical sessions.

Theorem 11 (Operational correspondence)**.** *Let P, P' be classical session processes and Q a mixed session process.*

Completeness *If $P \to_{\mathcal{C}} P'$, then $[\![P]\!] \to_{\mathcal{M}} [\![P']\!]$.*

Soundness *If $[\![P]\!] \to_{\mathcal{M}} Q$, then $P \to_{\mathcal{C}} P'$ and $[\![P']\!] = Q$, for some P'.*

Proof. Straightforward rule induction on the hypotheses, relying on the fact that $[\![P]\!][v/x] = [\![P[v/x]]\!]$ and $x_i \notin \mathrm{fv}(P_i)$ in the translation of $x \rhd \{l_i \colon P_i\}_{i \in I}$. □

7. Mixed Sessions as Classical Sessions

This section shows that the language of mixed sessions can be embedded in that of classical sessions.

7.1. Motivation

One of the novelties in mixed sessions is the possible presence of duplicated label-polarity pairs in choices. This introduces a form of non-determinism that can be easily captured in classical sessions. Adapted from the non-deterministic summation of Honda [29], the NDChoice classical session process creates a race condition on a new channel with endpoints s, t featuring multiple selections on the s endpoint, for only one branch on the t endpoint. This guarantees that exactly one of the branches is non-deterministically selected. The remaining selections must eventually be garbage collected. We assume that $\prod_{1 \le i \le n} Q_i$ denotes the process $Q_1 \mid \dots \mid Q_n$ for $n > 0$, and that Π binds tighter than the parallel composition operator.

$$\mathrm{NDChoice}\{P_i\}_{i \in I} = (\nu st)\left(\prod_{i \in I} s \triangleleft l_i.\mathbf{0} \mid t \rhd \{l_i \colon P_i\}_{i \in I}\right)$$

```
type Tx = lin&{m: !int.end,
               n: ?bool.end}

new x y: Tx

// lin x (m!3.0 + n?w.0)
case x of
  m → new s1 t1: *+{ℓ}
      s1 select ℓ |
      case t1 of
        ℓ → x!3
  n → new s2 t2: *+{ℓ}
      s2 select ℓ |
      case t2 of
        ℓ → x?w
|
// lin y (m?z.0)
new s3 t3: *+{ℓ}
s3 select ℓ |
case t3 of
  ℓ → y select m .
      new s4 t4: *+{ℓ}
      s4 select ℓ |
      case t4 of
        ℓ → y?z
```

Figure 13: Translation of (new xy)(lin x (m!3.0 + n?w.0)| lin y (m?z.0))

The type of channel end s is $\mu a.\mathsf{un}\oplus\{l_i\colon a\}_{i\in I}$. The qualifier must be un because s occurs in multiple threads in NDChoice; recursion arises because of the typing rules for processes reading or writing in unrestricted channels.

Equipped with the NDChoice operator we describe the translation of mixed sessions to classical sessions via a few examples, all of which fully type check and run in SePi [10]. To handle duplicated label-polarity pairs in choices, we organize choice processes by label-polarity fragments. Each such fragment represents a part of a choice operation where all possible outcomes have the same label and polarity. When a reduction occurs, one of the branches is taken, non-deterministically, using the NDChoice operator. After a non-deterministic choice of the branch, and depending on the polarity of the fragment, the process continues by either writing on or reading from the original channel.

The translation of choice processes is guided by their types. For each choice we need to know its qualifier (lin, un) and its view ($\oplus$, &), and this information is present in types alone.

Figure 13 shows the translation of the mixed process (new xy)(lin x (m!3.0 + n?w.0)| lin y (m?z.0)), where x is of type lin&{m!int.end, n?bool.end}. The corresponding type in classical sessions is lin&{m:!int.end, n:?bool.end}, which should not come as a surprise. Because channel end x is of an external choice type (&), the choice on x is encoded as a case process. The other end of the channel, y, is typed as an internal choice ($\oplus$) and is hence translated as a select process. Occurrences of the NDChoice process appear in a degenerate form, always applied to a single branch. We have four of them: three for each of the branches in case processes ($\mathsf{s_1t_1}$, $\mathsf{s_2t_2}$, and $\mathsf{s_4t_4}$) and one for the external choice in the mixed session process ($\mathsf{s_3t_3}$).

In general, an external choice is translated into a classical branching (case) over the unique labels of the fragments of the process, but where the polarity

```
type Tx = lin&{m: !int.end}

new x y: Tx

// lin x (m!3.0 + m!5.0)
case x of
    m → new s1 t1: *+{ℓ1,ℓ2}
         s1 select ℓ1 |
         s1 select ℓ2 |
         case t1 of
              ℓ1 → x!3
              ℓ2 → x!5
|
// lin y (m?z.0)
new s2 t2: *+{ℓ}
s2 select ℓ |
case t2 of
  ℓ → y select m.
       new s3 t3: *+{ℓ}
       s3 select ℓ |
       case t3 of
            ℓ → y?z
```

Figure 14: Translation of (new xy)(lin x (m!3.0 + m!5.0) | lin y (m?z.0))

of each label is inverted. The internal choice, in turn, is translated as (possibly nondeterministic collection of) classical **select** process but keeps the label polarity. This preserves the behaviour of the original process: in mixed choices, a reduction occurs when a branch $l!v.P$ matches against another branch $l?z.Q$ with the same label but with dual polarity ($l!$ against $l?$), while in a classical session the labels alone must match (l against l). Needless to say, we could have followed the strategy of dualizing internal choices rather than external.

If we label reduction steps with the names of the channel ends on which they occur, we can see that, in this case a $\xrightarrow{xy}$ reduction step in mixed sessions is mimicked by a long series of classical reductions, namely $\xrightarrow{s_3t_3}\xrightarrow{xy}\xrightarrow{s_1t_1}\xrightarrow{s_4t_4}\xrightarrow{xy}$ or $\xrightarrow{s_3t_3}\xrightarrow{xy}\xrightarrow{s_4t_4}\xrightarrow{s_1t_1}\xrightarrow{xy}$. Notice the three reductions to resolve non-determinism (on s_it_i) and the two reductions on xy to encode branching followed by message passing, an atomic operation in mixed sessions.

Figure 14 shows an example of a mixed choice process with a duplicated label-polarity pair, m!. If we assign type lin &{m!int} to x, then we know that the choice on x is encoded as **case** and that on y as **select**. In this case, the NDChoice operator is applied in a non-degenerate manner to decide whether to send the values 3 or 5 on x channel end, by means of channel $\mathsf{s_1t_1}$. Again we can see that the one step reduction on channel xy in the original mixed session process originates a sequence of five reduction steps in classical sessions, namely $\xrightarrow{s_2t_2}\xrightarrow{xy}\xrightarrow{s_1t_1}\xrightarrow{s_3t_3}\xrightarrow{xy}$ or $\xrightarrow{s_2t_2}\xrightarrow{xy}\xrightarrow{s_3t_3}\xrightarrow{s_1t_1}\xrightarrow{xy}$. In this case, however, the computation is non-deterministic: the last reduction step may carry integer 3 or 5. The **select** branch which is not taken persists on the process, but is (extended) structurally congruent to $\mathbf{0}$ due to rule C-Equiv.

Figure 15 shows the encoding of mixed choices on unrestricted channels. The mixed choice process is that of Figure 14 only that the two ephemeral choices (**lin**) have been replaced by their persistent counterparts (**un**). The novelty in this case is the loops that have been created around the **case** and the **select**

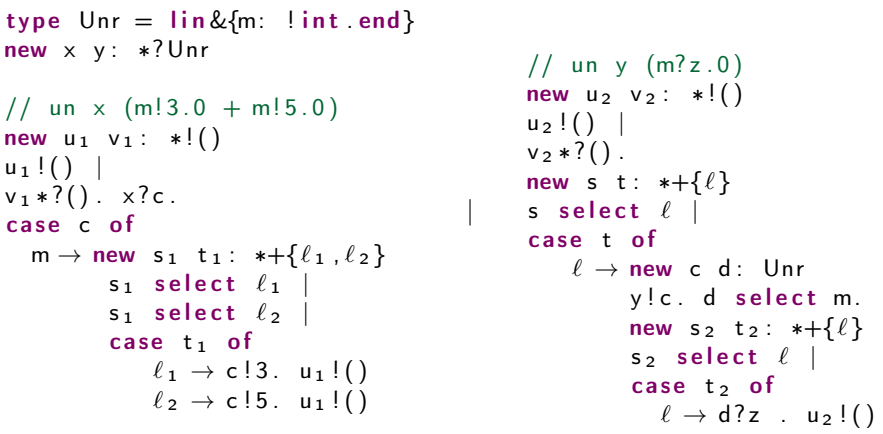


Figure 15: Translation of (new xy)(un x (m!3.0 + m!5.0) | un y (m?z.0))

$$
\begin{aligned}
(\!|\mathsf{lin}{\oplus}\{l\star_i T_i.U_i\}_{i\in I}|\!) &= \mathsf{lin}{\oplus}\{l\star_i\colon \mathsf{lin}\star_i\,(\!|T_i|\!).(\!|U_i|\!)\}_{i\in I}\\
(\!|\mathsf{lin}{\&}\{l\star_i T_i.U_i\}_{i\in I}|\!) &= \mathsf{lin}{\&}\{l\bullet_i\colon \mathsf{lin}\star_i\,(\!|T_i|\!).(\!|U_i|\!)\}_{i\in I}\\
(\!|\mu a.\mathsf{un}{\oplus}\{l\star_i T_i.a\}_{i\in I}|\!) &= \mu a.\mathsf{un}!(\mathsf{lin}{\&}\{l\star_i\colon \mathsf{lin}\star_i\,(\!|T_i|\!).\mathsf{end}\}_{i\in I}).a\\
(\!|\mu a.\mathsf{un}{\&}\{l\star_i T_i.a\}_{i\in I}|\!) &= \mu a.\mathsf{un}?(\mathsf{lin}{\&}\{l\bullet_i\colon \mathsf{lin}\star_i\,(\!|T_i|\!).\mathsf{end}\}_{i\in I}).a
\end{aligned}
$$

where $\star_i \perp \bullet_i$. Homomorphic for end, unit, bool, a, and $\mu a.T$ with $T \neq \mathsf{un}\sharp\{l\star_i T_i.U_i\}$.

Figure 16: Translation of mixed session types into classical session types

process. Loops in classical sessions can be implemented with a replicated input: a process of the form v*?x.P is a persistent process that, when invoked with a value w becomes the parallel composition P[w/x] | v*?x.P. The general form of the loops we are interested in are (new uv : *!())(u!() | v*?x.P), where *continue calls* in process P are of the form u!(). The contents of the messages that control the loop are not of interest and so we use the unit type (), so that u is of type *!(). We can see the calls u1!() and u2!() in the last lines in Figure 15, reinstating the unrestricted choice process. In this case, one step reduction in mixed sessions corresponds to a long sequence of transitions in their encodings.

7.2. The Mixed-to-classical Encoding

We now present translations for types and processes in general. The translation of mixed choice session types into classical session types is in Figure 16. The (atomic) branch-communicate nature of mixed session types, $\{l\star_i S_i\}$, is broken in its two parts, $\{l_i\colon \star S_i\}$: branch first, communicate after. In mixed sessions, choice types are labelled by label-polarity pairs ($l!$ or $l?$); in classical

session choices are labelled by labels alone. Because we want the encoding of a label $l!$ to match the encoding of $l?$, we must dualize one of them. We arbitrarily chose do dualize the labels in the & type. The typing rules for classical unrestricted processes of type $S = \mathsf{un}\,\sharp\{l_i \star S_i.T_i\}_{i\in I}$ require T_i to be equivalent ($\simeq$) to S itself. We take advantage of this restriction when translating un types. When compared to their lin counterparts, un processes feature an extra indirection inflicted by the (un) loop under which they operate.

The translation of mixed choice processes is in Figures 17 and 18. Since the translation is guided by the type of the process to be translated, we also provide the typing context to the translation function, hence the notation $(\!|\Gamma \vdash P|\!)$. Because label-polarity pairs may be duplicated in choice processes, we organize such processes in label-polarity fragments, so that a process of the form $qx \sum_{h\in H} M_h$ is written as

$$qx \sum_{i\in I} (\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik}) \ .$$

Each label-polarity fragment ($l!_i$ or $l?_i$) groups together branches with the same label and the same polarity. Such fragments may be empty for external choices, for not all label-polarity pairs in an external choice type need to be covered in the corresponding process (internal choice processes do not need to cover all choices offered by the external counterpart). The essence of the translation is discussed in the three examples above.

We distinguish six cases for choices, according to qualifiers (lin or un) and views ($\oplus$ or &) in types, but also to the qualifiers (lin or un) in processes. In all of them an NDChoice process takes care of duplicated label-polarity pairs in branches. The six cases cover all possible cases, making the translation function total. Internal choice types feature an extra occurrence of NDChoice to nondeterministically select between output and input *on the same label*. Notice that external choice must still accept both choices, so that it is not equipped with an NDChoice. Finally, unrestricted processes with unrestricted types require the encoding of a loop, accomplished by creating a new channel for the effect (νuv), installing a replicated input—$\mathsf{un}\, v?\_.P$—at one end of the channel, and invoking—$u!()$—the input once to "start" the loop and again at the end of the interaction on channel end x. The contents of the messages are of no interest and so we use the unit value $()$. However, linear processes with unrestricted types do not require the new channel (uv) to encode a loop. This case is similar to the case with linear types, but uses channel (cd) to enable communication with unrestricted processes.

Following the encoding for types, the encoding for external choice processes exchanges the polarities of choice labels: a label $l!_i$ in mixed sessions is translated into $l?_i$, and vice-versa, in the cases for $\mathsf{lin}\&$ and $\mathsf{un}\&$ choices. This allows reduction to happen in classical sessions, where we require an exact match between the label of the **select** process and that of the **case** process.

Take for T the type $\mu a.\mathsf{lin} \oplus \{l!a.a\}$, that is $T = \mathsf{lin}\oplus\{l!T.T\}$. We can show that $x\colon T \vdash \mathsf{lin}\, x(l!x.\mathbf{0})$. The encoding of this process includes the classical

$$(\!|\Gamma \vdash \mathsf{lin}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik})|\!) = \text{NDChoice}\{$$
$$x \lhd l!_i.\, \text{NDChoice}\{x!v_{ij}.(\!|\Gamma_4 \vdash P_{ij}|\!)\}_{j\in J_i},$$
$$x \lhd l?_i.\, \text{NDChoice}\{\mathsf{lin}\, x?y_{ik}.(\!|(\Gamma_2 + x\colon U'_i), y_{ik}\colon T'_i \vdash P'_{ik}|\!)\}_{k\in K_i}\}_{i\in I}$$

where $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1 \vdash_{\mathcal{M}} x\colon \mathsf{lin}\oplus\{l!_i T_i.U_i, l?_i T'_i.U'_i\}_{i\in I}$ and $\Gamma_2 + x\colon U_i = \Gamma_3 \circ \Gamma_4$ and $\Gamma_3 \vdash v_{ij}\colon T_i$.

$$(\!|\Gamma \vdash \mathsf{lin}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik})|\!) = x \rhd \{$$
$$l?_i\colon\ \text{NDChoice}\{x!v_{ij}.(\!|\Gamma_4 \vdash P_{ij}|\!)\}_{j\in J_i},$$
$$l!_i\colon\ \text{NDChoice}\{\mathsf{lin}\, x?y_{ik}.(\!|(\Gamma_2 + x\colon U_i), y_{ik}\colon T_i \vdash P'_{ik}|\!)\}_{k\in K_i}\}_{i\in I}$$

where $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Gamma_1 \vdash_{\mathcal{M}} x\colon \mathsf{lin}\&\{l!_i T_i.U_i, l?_i T'_i.U'_i\}_{i\in I}$ and $\Gamma_2 + x\colon U'_i = \Gamma_3 \circ \Gamma_4$ and $\Gamma_3 \vdash v_{ij}\colon T'_i$.

$$(\!|\Gamma \vdash \mathsf{lin}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik})|\!) = \text{NDChoice}\{$$
$$(\nu cd)x!c.d \lhd l!_i.\, \text{NDChoice}\{d!v_{ij}.(\!|\Gamma \vdash P_{ij}|\!)\}_{j\in J_i},$$
$$(\nu cd)x!c.d \lhd l?_i.\, \text{NDChoice}\{\mathsf{lin}\, d?y_{ik}.(\!|\Gamma, y_{ik}\colon T'_i \vdash P'_{ik}|\!)\}_{k\in K_i}\}_{i\in I}$$

where $\Gamma\,\mathsf{un}$ and $\Gamma \vdash_{\mathcal{M}} x\colon \mu a.\mathsf{un}\oplus\{l!_i T_i.a, l?_i T'_i.a\}_{i\in I}$ and $\Gamma \vdash_{\mathcal{M}} v_{ij}\colon T_i$.

$$(\!|\Gamma \vdash \mathsf{lin}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik})|\!) = \mathsf{lin}\, x?c.c \rhd \{$$
$$l?_i\colon\ \text{NDChoice}\{c!v_{ij}.(\!|\Gamma \vdash P_{ij}|\!)\}_{j\in J_i},$$
$$l!_i\colon\ \text{NDChoice}\{\mathsf{lin}\, c?y_{ik}.(\!|\Gamma, y_{ik}\colon T_i \vdash P'_{ik}|\!)\}_{k\in K_i}\}_{i\in I}$$

where $\Gamma\,\mathsf{un}$ and $\Gamma \vdash_{\mathcal{M}} x\colon \mu a.\mathsf{un}\&\{l!_i T_i.a, l?_i T'_i.a\}_{i\in I}$ and $\Gamma \vdash_{\mathcal{M}} v_{ij}\colon T'_i$.

$$(\!|\Gamma \vdash \mathsf{un}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik})|\!) = (\nu uv)(u!() \mid \mathsf{un}\, v?\_.\text{NDChoice}\{$$
$$(\nu cd)x!c.d \lhd l!_i.\, \text{NDChoice}\{d!v_{ij}.(u!() \mid (\!|\Gamma \vdash P_{ij}|\!))\}_{j\in J_i},$$
$$(\nu cd)x!c.d \lhd l?_i.\, \text{NDChoice}\{\mathsf{lin}\, d?y_{ik}.(u!() \mid (\!|\Gamma, y_{ik}\colon T'_i \vdash P'_{ik}|\!))\}_{k\in K_i}\}_{i\in I})$$

where $\Gamma\,\mathsf{un}$ and $\Gamma \vdash_{\mathcal{M}} x\colon \mu a.\mathsf{un}\oplus\{l!_i T_i.a, l?_i T'_i.a\}_{i\in I}$ and $\Gamma \vdash_{\mathcal{M}} v_{ij}\colon T_i$.

$$(\!|\Gamma \vdash \mathsf{un}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik})|\!) = (\nu uv)(u!() \mid \mathsf{un}\, v?\_.\mathsf{lin}\, x?c.c \rhd \{$$
$$l?_i\colon\ \text{NDChoice}\{c!v_{ij}.(u!() \mid (\!|\Gamma \vdash P_{ij}|\!))\}_{j\in J_i},$$
$$l!_i\colon\ \text{NDChoice}\{\mathsf{lin}\, c?y_{ik}.(u!() \mid (\!|\Gamma, y_{ik}\colon T_i \vdash P'_{ik}|\!))\}_{k\in K_i}\}_{i\in I})$$

where $\Gamma\,\mathsf{un}$ and $\Gamma \vdash_{\mathcal{M}} x\colon \mu a.\mathsf{un}\&\{l!_i T_i.a, l?_i T'_i.a\}_{i\in I}$ and $\Gamma \vdash_{\mathcal{M}} v_{ij}\colon T'_i$.

Figure 17: Translation of mixed choice processes into classical processes (1/2)

$$
\begin{aligned}
(\!|\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2|\!) &= (\!|\Gamma_1 \vdash P_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!) \\
(\!|\Gamma \vdash (\nu xy)P|\!) &= (\nu xy)(\!|\Gamma, x\colon T, y\colon U \vdash P|\!) \\
(\!|\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\ v\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2|\!) &= \mathsf{if}\ v\ \mathsf{then}\ (\!|\Gamma_2 \vdash P_1|\!)\ \mathsf{else}\ (\!|\Gamma_2 \vdash P_2|\!) \\
(\!|\Gamma \vdash \mathbf{0}|\!) &= \mathbf{0}
\end{aligned}
$$

where $T \perp U$ and $\Gamma_1 \vdash v\colon \mathsf{bool}$.

Figure 18: Translation of mixed choice processes into classical processes (2/2)

process $x!x.\mathbf{0}$ as a subterm, which cannot be typed under a linear type with the original T-Out rule [3]. The new T-OutC rule in Figure 11 allows typing such a process.

7.3. The Mixed-to-classical Encoding Is Minimal

This section covers typing and operational correspondences; we aim at a minimal encoding according to the definition in Section 5. The pair of maps $(\!|\cdot|\!)$ in Figures 16 to 18 constitutes the typed encoding that translates types and processes from $\mathcal{M}$ to $\mathcal{C}$.

Theorem 12 (Syntax Preservation)**.** *The mixed-to-classical encoding is*

1. *Homomorphic wrt parallel composition: if* $(\!|\Gamma \circ \Gamma'|\!) \vdash_{\mathcal{C}} (\!|\Gamma \circ \Gamma' \vdash_{\mathcal{M}} (P \mid P')|\!)$*, then* $(\!|\Gamma|\!) \circ (\!|\Gamma'|\!) \vdash_{\mathcal{C}} (\!|\Gamma \vdash_{\mathcal{M}} P|\!) \mid (\!|\Gamma' \vdash_{\mathcal{M}} P'|\!)$*;*
2. *Compositional wrt channel restriction: if* $(\!|\Gamma|\!) \vdash_{\mathcal{C}} (\!|\Gamma \vdash_{\mathcal{M}} (\nu xy)P|\!)$*, then* $(\!|\Gamma|\!) \vdash_{\mathcal{C}} (\nu xy)(\!|\Gamma, x\colon S, y\colon T \vdash_{\mathcal{M}} P|\!)$*, where* $S \perp T$*;*
3. *Name invariant: if* $(\!|\sigma(\Gamma)|\!) \vdash_{\mathcal{C}} (\!|\sigma(\Gamma) \vdash_{\mathcal{M}} \sigma(P)|\!)$*, then* $\sigma((\!|\Gamma|\!)) \vdash_{\mathcal{C}} \sigma((\!|\Gamma \vdash_{\mathcal{M}} P|\!))$*, for any injective renaming of variables* σ*.*

Proof. Items 1 and 2: immediate from Figures 17 and 18. Item 3: our encoding transforms each channel in itself, so is name invariant. □

We now move to type soundness, but before we need to type the NDChoice operator.

Lemma 13. *The following is an admissible typing rule for* NDChoice*.*

$$
\frac{\Gamma \vdash_{\mathcal{M}} P_i}{\Gamma \vdash_{\mathcal{M}} \mathrm{NDChoice}\{P_i\}_{i \in I}} \qquad \text{(T-NDChoice)}
$$

Proof. The typing derivation of the expansion of NDChoice leaves open the derivations for $\Gamma \vdash_{\mathcal{M}} P_i$ alone. □

The type soundness theorem for the mixed-to-classical encoding is Item 9 of Theorem 14 below; the remaining items help in building the main result.

Theorem 14 (Type Soundness)**.**

1. *If* $T\,\mathrm{un}_{\mathcal{M}}$*, then* $(\!|T|\!)\,\mathrm{un}_{\mathcal{C}}$*.*

2. *If* $\Gamma\,\mathrm{un}_{\mathcal{M}}$, *then* $(\!|\Gamma|\!)\,\mathrm{un}_{\mathcal{C}}$.
3. *If* $T <: U$, *then* $(\!|T|\!) <: (\!|U|\!)$.
4. *If* $\Gamma \vdash_{\mathcal{M}} v\colon T$, *then* $(\!|\Gamma|\!) \vdash_{\mathcal{C}} v\colon (\!|T|\!)$.
5. *If* $\Gamma = \Gamma_1 \circ_{\mathcal{M}} \Gamma_2$, *then* $(\!|\Gamma|\!) = (\!|\Gamma_1|\!) \wr\!\!\circ (\!|\Gamma_2|\!)$.
6. *If* $\Gamma = \Gamma_1 + x\colon T$, *then* $(\!|\Gamma|\!) = (\!|\Gamma_1|\!) + x\colon (\!|T|\!)$.
7. *If* $T \simeq U$, *then* $(\!|T|\!) \simeq (\!|U|\!)$.
8. *If* $T \perp U$, *then* $(\!|T|\!) \perp (\!|U|\!)$.
9. *If* $\Gamma \vdash_{\mathcal{M}} P$, *then* $(\!|\Gamma|\!) \vdash_{\mathcal{C}} (\!|\Gamma \vdash P|\!)$.

Proof. Item 1: By rule induction on the hypothesis.

Item 2: By rule induction on the hypothesis using Item 1.

Item 3: By coinduction using the set $\mathcal{T} = \{((\!|T|\!), (\!|U|\!)) \mid T <: U\}$.

Item 4: By case analysis on the hypothesis using Items 2 and 3.

Items 5 and 6: By case analysis on the hypothesis.

Item 7: By coinduction using the set $\mathcal{E} = \{((\!|T|\!), (\!|U|\!)) \mid T \simeq U\}$.

Item 8: By coinduction using the set $\mathcal{D} = \{((\!|T|\!), (\!|U|\!)) \mid T \perp U\}$ and Item 7.

Item 9: By rule induction on the hypothesis. We sketch a few cases.

Case T-Par. The premises read $\Gamma = \Gamma_1 + \Gamma_2$ and $\Gamma_2 \vdash_{\mathcal{M}} P_1$ and $\Gamma_2 \vdash_{\mathcal{M}} P_2$. Item 5 and induction give $(\!|\Gamma|\!) = (\!|\Gamma_1|\!) \circ (\!|\Gamma_2|\!)$ and $(\!|\Gamma_2|\!) \vdash_{\mathcal{C}} (\!|P|\!)$ and $(\!|\Gamma_2|\!) \vdash_{\mathcal{M}} (\!|Q|\!)$. Conclude with the T-Par rule for classical processes.

Case T-Choice. The premises read $\Gamma = \Gamma_1 \circ \Gamma_2$ and $\Gamma\, q_1$ and $\Gamma_1 \vdash_{\mathcal{M}} x\colon q_2 \sharp \{l\star_i T_i.U_i\}_{i\in I}$ and $\Gamma_2 + x\colon U_j \vdash l\star_j v_j.P_j\colon l\star_j T_j.U_j$ and $\{l\star_j\}_{j\in J} = \{l\star_i\}_{i\in I}$. We distinguish six cases according to Figures 17 and 18.

Subcase $\mathsf{lin}\oplus$. Then we have $q_1 = q_2 = \mathsf{lin}$ and $\sharp = \oplus$. From $\Gamma_2 + x\colon U_j \vdash l\star_j v_j.P_j\colon l\star_j T_j.U_j$, the premises to rules T-Out and T-In read $\Gamma_2 + x\colon U_j = \Gamma_3 + \Gamma_4$ and $\Gamma_3 \vdash_{\mathcal{M}} v_j\colon T_j$ and $\Gamma_4 \vdash_{\mathcal{M}} P_j$ and $\Gamma_2 + x\colon U_i, y\colon T_i \vdash_{\mathcal{M}} P_i$. Item 4 gives $(\!|\Gamma_3|\!) \vdash_{\mathcal{C}} v_j\colon (\!|T_j|\!)$ and by induction we know $(\!|\Gamma_4|\!) \vdash_{\mathcal{C}} (\!|\Gamma_4 \vdash P_j|\!)$ and also $(\!|\Gamma_2 + x\colon U_i, y\colon T_i'|\!) \vdash_{\mathcal{C}} (\!|\Gamma_2 + x\colon U_i, y\colon T_i' \vdash P_i'|\!)$. We now build a derivation for the conclusion. From the induction hypotheses, using the rules T-OutC and T-NDChoice (Lemma 13), and T-Sel we know that

$$(\!|\Gamma_1 \circ \Gamma_2|\!) \vdash_{\mathcal{C}} x \lhd l!_i.\,\mathrm{NDChoice}\{x!v_{ij}.(\!|\Gamma_4 \vdash P_{ij}|\!)\}_{j\in J}\}$$

Similarly, using rules T-InC, T-NDChoice, and T-Sel we know that

$$(\!|\Gamma_1 \circ \Gamma_2|\!) \vdash_{\mathcal{C}} x \lhd l?_i.\,\mathrm{NDChoice}\{\mathsf{lin}\, x?y_{ik}.(\!|(\Gamma_2 + x\colon U_i), y_{ik}\colon T_i' \vdash P_{ik}'|\!)\}$$

and we conclude the case using rule T-NDChoice again.

Subcase $\mathsf{lin}\&$. Here we have $q_1 = q_2 = \mathsf{un}$ and $\sharp = \&$. Similarly to the previous case, from the premises of T-Out, T-In and the induction hypothesis, we get $(\!|\Gamma_3|\!) \vdash_{\mathcal{C}} v_j\colon (\!|T_j|\!)$ and $(\!|\Gamma_4|\!) \vdash_{\mathcal{C}} (\!|\Gamma_4 \vdash P_j|\!)$ and $(\!|\Gamma_2 + x\colon U_i', y\colon T_i|\!) \vdash_{\mathcal{C}} (\!|\Gamma_2 + x\colon U_i', y\colon T_i \vdash P_i'|\!)$. We build a derivation using T-OutC, T-NDChoice, and obtaining

$$(\!|\Gamma_2|\!), x\colon \mathsf{lin}!(\!|T_i|\!).(\!|U_i|\!) \vdash_{\mathcal{C}} \mathrm{NDChoice}\{x!v_{ij}.(\!|\Gamma_4 \vdash P_{ij}|\!)\}_{j\in J}\}.$$

Similarly, using T-InC and T-NDChoice, we obtain

$$(\!|\Gamma_2|\!), x\colon \mathsf{lin}?(\!|T_i'|\!).(\!|U_i'|\!) \vdash_{\mathcal{C}} \mathrm{NDChoice}\{\mathsf{lin}\, x?y_{ik}.(\!|(\Gamma_2 + x\colon U_i'), y_{ik}\colon T_i \vdash P_{ik}'|\!)\}_{j\in J}\}.$$

We conclude using rule T-Branch.

Subcase $\mathsf{un}\oplus$, $q_1 = \mathsf{un}$. In this case $q_2 = \mathsf{un}$ and $\sharp = \oplus$. We have Γ un, hence $\Gamma_1 = \Gamma_2 = \Gamma$. Because the type for x is un, in order for $\Gamma_2 + x\colon U_j$ to be defined U_j must be equal to $\mathsf{un}\sharp\{l\star_i T_i.U_i\}_{i\in I}$, hence of the form $\mu a.\mathsf{un}\sharp\{l\star_i T_i.a\}_{i\in I}$. We now proceed as in the subcases above and build the (long) derivation for the classical process using the typing rules (Figure 11 and Lemma 13). A type for c is $\mathsf{lin}\&\{l!_i\colon \mathsf{lin}!(\!|T_i|\!).\mathsf{end}, l?_i\colon \mathsf{lin}?(\!|T_i'|\!).\mathsf{end}\}_{i\in I}$ and that of d a dual. The type of u is $\mu a.\mathsf{un}!\mathsf{unit}.a$ and that of v is dual.

Subcase $\mathsf{un}\&$, $q_1 = \mathsf{un}$. This case is analogous to the previous one, results from a long derivation using the typing rules in Figure 11 and Lemma 13, but now considering $q_2 = \mathsf{un}$ and $\sharp = \&$. The type of x is $\mu a.\mathsf{un}\&\{l\star_i T_i.a\}_{i\in I}$.

Subcases $\mathsf{un}\sharp$, $q_1 = \mathsf{lin}$. Due to the absence of uv, the derivations for these cases are similar but simpler than those for subcases $\mathsf{un}\sharp$, $q_1 = \mathsf{un}$. □

The following lemma characterizes the reductions of NDChoice processes: they reduce to one of the choices and leave an inert term G.

Lemma 15. $\mathrm{NDChoice}\{P_i\}_{i\in I} \to P_k \mid G \approx_{\mathcal{C}} P_k$, *for any* $k \in I$.

Proof. $\mathrm{NDChoice}\{P_i\}_{i\in I} \to P_k \mid G$, where $G = (\nu st)\prod_{i\in I}^{i\neq k} s \lhd l_i.\mathbf{0}$ and $G \approx_{\mathcal{C}} \mathbf{0}$. □

The following theorem fulfils the *barb preservation criterion*: if a mixed process has a barb, its translation has a weak barb on the same channel.

Theorem 16 (Barb Preservation)**.** *The typed encoding* $(\!|\cdot|\!) : \mathcal{M} \longrightarrow \mathcal{C}$ *preserves barbs, that is, if* $\Gamma \vdash_{\mathcal{M}} P \downarrow_x$, *then* $(\!|\Gamma|\!) \vdash_{\mathcal{C}} (\!|\Gamma \vdash_{\mathcal{C}} P|\!) \Downarrow_x$.

Proof. By analysis of the translation of processes with barbs. In the case that x is linear, rearranging the choice in P in fragments, we obtain that $P \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)(\mathsf{lin}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik}) \mid R)$ and so its translation is

$$
\begin{aligned}
(\!|\Gamma \vdash P|\!) = (\nu x_n y_n)\ldots(\nu x_1 y_1)(&\mathrm{NDChoice}\{\\
&x \lhd l!_i.\,\mathrm{NDChoice}\{x!v_{ij}.(\!|\Gamma_4 \vdash P_{ij}|\!)\}_{j\in J_i},\\
&x \lhd l?_i.\,\mathrm{NDChoice}\{\mathsf{lin}\, x?y_{ik}.(\!|(\Gamma_2 + x\colon U_i'), y_{ik}\colon T_i' \vdash P'_{ik}|\!)\}_{k\in K_i}\}_{i\in I}\\
&\mid (\!|\Gamma' \vdash R|\!)).
\end{aligned}
$$

This process makes internal reduction steps in the resolution of the outermost NDChoice, non-deterministically choosing one of the possible fragments, via Lemma 15. However, independently of which branch is chosen, they are all of the form $x \lhd \ell.C$, which has a barb in x. That is: $(\!|\Gamma \vdash P|\!) \Rightarrow (\nu x_n y_n)\ldots(\nu x_1 y_1)(x \lhd \ell.C \mid (\!|\Gamma' \vdash R|\!) \mid G)$, which has a barb in x. The G term is the inert remainder of the NDChoice reduction. In the unrestricted case, we have $P \equiv$

$(\nu x_n y_n)\ldots(\nu x_1 y_1)(\mathsf{un}\, x \sum_{i\in I}(\sum_{j\in J_i} l!_i v_{ij}.P_{ij} + \sum_{k\in K_i} l?_i y_{ik}.P'_{ik}) \mid R)$. So,

$$
\begin{aligned}
(\!|\Gamma \vdash P|\!) = {} & (\nu x_n y_n)\ldots(\nu x_1 y_1)((\nu uv)(u!() \mid \mathsf{un}\, v?\_.\mathrm{NDChoice}\{ \\
& (\nu cd)x!c.d \lhd l!_i.\,\mathrm{NDChoice}\{d!v_{ij}.(u!() \mid (\!|\Gamma_1 \vdash P_{ij}|\!))\}_{j\in J_i}, \\
& (\nu cd)x!c.d \lhd l?_i.\,\mathrm{NDChoice}\{\mathsf{lin}\, d?y_{ik}.(u!() \mid (\!|\Gamma_1, y_{ik}\colon T'_i \vdash P'_{ik}|\!))\}_{k\in K_i}\}_{i\in I}) \\
& \mid (\!|\Gamma_2 \vdash R|\!))
\end{aligned}
$$

The process starts by reducing via R-UnCom on the u, v channels to the process

$$
\begin{aligned}
(\!|\Gamma \vdash P|\!) \Rightarrow {} & (\nu x_n y_n)\ldots(\nu x_1 y_1)((\nu uv)(\mathrm{NDChoice}\{ \\
& (\nu cd)x!c.d \lhd l!_i.\,\mathrm{NDChoice}\{d!v_{ij}.(u!() \mid (\!|\Gamma_1 \vdash P_{ij}|\!))\}_{j\in J_i}, \\
& (\nu cd)x!c.d \lhd l?_i.\,\mathrm{NDChoice}\{\mathsf{lin}\, d?y_{ik}.(u!() \mid (\!|\Gamma_1, y_{ik}\colon T'_i \vdash P'_{ik}|\!))\}_{k\in K_i}\}_{i\in I}) \\
& \mid (\!|\Gamma_2 \vdash R|\!) \mid U)
\end{aligned}
$$

where U is the persistent part of the unrestricted process. This process, in turn, reduces via the NDChoice (Lemma 15) to one of the possible branches which are all of the form $(\nu ab)x!a.C$,

$$(\!|\Gamma \vdash P|\!) \Rightarrow (\nu x_n y_n)\ldots(\nu x_1 y_1)(\nu uv)(\nu ab)(x!a.C) \mid (\!|\Gamma_2 \vdash R|\!) \mid U \mid G).$$

Since P has a barb in x, $x \notin \{x_i, y_i\}_{i=1}^n$ and so this process also has a barb in x, concluding that $(\!|\Gamma \vdash P|\!)$ has indeed a weak barb in x. □

We now address operational completeness. Operational completeness relates the behaviour of mixed sessions against their classical sessions images: any reduction step in mixed sessions can be mimicked by a sequence of reductions steps in classical sessions, modulo extended structural congruence. The ghost reductions result from the new channels and communication inserted by the translation, namely those due to the NDChoice and to the encoding of "loops" for un mixed choices.

Theorem 17 (Operational completeness)**.** *The typed encoding* $(\!|\cdot|\!) : \mathcal{M} \longrightarrow \mathcal{C}$ *is* operationally complete*, that is, if* $P \to_{\mathcal{M}} P'$*, then* $(\!|\Gamma \vdash P|\!) \Rightarrow_{\mathcal{C}} \approx_{\mathcal{C}} (\!|\Gamma \vdash P'|\!)$.

Proof. By rule induction on the derivation of $P \to_{\mathcal{M}} P'$. We detail a few cases.

Case R-Par. We have $(\!|\Gamma \vdash P_1 \mid P_2|\!) = (\!|\Gamma_1 \vdash P_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!)$ with $\Gamma = \Gamma_1 \circ \Gamma_2$. By induction we have $(\!|\Gamma_1 \vdash P_1|\!) \Rightarrow_{\mathcal{C}} Q \approx_{\mathcal{C}} (\!|\Gamma_1 \vdash P'_1|\!)$. Using rule R-Par and the fact that $\approx_{\mathcal{C}}$ is a congruence, we get $(\!|\Gamma_1 \vdash P_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!) \Rightarrow_{\mathcal{C}} Q \mid (\!|\Gamma_2 \vdash P_2|\!) \approx_{\mathcal{C}} (\!|\Gamma_1 \vdash P'_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!) = (\!|\Gamma \vdash P'_1 \mid P_2|\!)$. The cases for R-Res and R-Struct are similar.

Case R-LinLin. Considering $\Gamma, x\colon U, y\colon V = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$ with $\Gamma_1 \vdash x\colon \mathsf{lin}\&\{l!T_0.U_0, \ldots\}$ and $\Gamma_2 \vdash y\colon \mathsf{lin}\oplus\{l?T_0.V_0, \ldots\}$, and $U_0 \perp V_0$, and using the definition of the translation on $(\!|\Gamma \vdash (\nu xy)(\mathsf{lin}\, x(l!v.P + M) \mid \mathsf{lin}\, y(l?z.Q + N) \mid O)|\!)$, we get

$$(\nu xy)((\!|\Gamma_1 \vdash \mathsf{lin}\, x(l!v.P + M)|\!) \mid (\!|\Gamma_2 \vdash \mathsf{lin}\, y(l?z.Q + N)|\!)) \mid (\!|\Gamma_3 \vdash O|\!)).$$

By a sequence of applications of the rules R-Case, C-Equiv, and R-LinCom, we obtain

$$(\nu xy)(\llparenthesis \Gamma_5 \vdash P \rrparenthesis \mid \llparenthesis (\Gamma_2'' + y\colon V_0), z\colon T_0 \vdash Q \rrparenthesis [v/z] \mid \llparenthesis \Gamma_3 \vdash O \rrparenthesis),$$

where $\Gamma_1 = \Gamma_1' \circ \Gamma_1''$ and $\Gamma_1'' + x\colon U_0 = \Gamma_4 \circ \Gamma_5$ and $\Gamma_4 \vdash x\colon T_0$ and $\Gamma_2 = \Gamma_2' \circ \Gamma_2''$. Notice that $\Gamma_1' = \Delta_1, x\colon U$ where Δ_1 is un, hence Δ_1 is in Γ_1''. The same reasoning applies to Γ_2'. Since context Γ_4 is used to type v, the substitution lemma [3] reintroduces it in the context for $Q[v/z]$. Applying the definition of the translation, we conclude that translation of the original mixed process reduces to $\llparenthesis \Gamma \vdash (\nu xy)(P \mid Q[v/z] \mid O) \rrparenthesis$.

The case for R-UnUn is similar, albeit more verbose.

Case R-UnLin. Using the definition of the translation for the ν constructor, we know that the process $(\nu xy)(\mathsf{un}\, x(l!v.P + M) \mid \mathsf{lin}\, y(l?z.Q + N) \mid O)$ translates to $(\nu xy)\llparenthesis \Gamma, x\colon U, y\colon V \vdash \mathsf{un}\, x(l!v.P + M) \mid \mathsf{lin}\, y(l?z.Q + N) \mid O \rrparenthesis$, where $R \perp S$. Then, U is of the form $U = \mu a.\mathsf{un}!U_0.a$ and $V = \mu a.\mathsf{un}?V_0.a$. We are then reduced to the translation rules in Figure 17. Using the rule R-UnCom to reduce u and v in the translation of $\mathsf{un}\, x(l!v.P + M)$, and applying the rules R-Case, C-Equiv, and R-LinCom, we obtain

$$\llparenthesis \Gamma \vdash (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}\, x(l!v.P + M) \mid O) \rrparenthesis$$

The case for R-LinUn is similar.

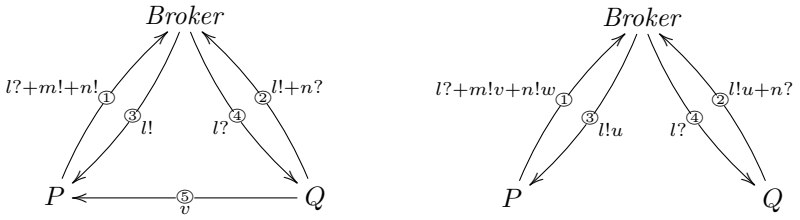
The cases for R-IfT and R-IfF are direct. □

We can show that the translation does *not* enjoy reduction soundness. Consider the classical process Q to be the encoding of process P of the form $\mathsf{un}\, y(m?z.\mathbf{0})$, described in the right part of Figure 15. Soundness requires that if $Q \to_{\mathcal{C}} Q'$, then $P \Rightarrow_{\mathcal{M}} P'$ and $Q \Rightarrow_{\mathcal{C}} \approx_{\mathcal{C}} \llparenthesis \Gamma \vdash P' \rrparenthesis$. Clearly, Q has an initial reduction step (on channel $u_2 v_2$), which cannot be mimicked by P. But this reduction is a transition internal to process Q. Equipped with a suitable notion of weak bisimulation that ignores internal transitions, we expect soundness to hold.

8. What Is in the Way of a Compiler?

This section discusses algorithmic type checking and the implementation of choice in message passing architectures.

We start with type checking and then move to the runtime system. Gay and Hole present an algorithmic subtyping system for classical sessions [20]. Algorithmic subtyping for mixed sessions can be obtained by adapting the rules in Gay and Hole. T-Sub is the only non syntax-directed rule in Figure 9.We delete this rule and distribute subtype checking among all rules that use, in their premises, sequents $\Gamma \vdash v\colon T$, as usual. Most of the rules include a non-deterministic context split operation. Take rule T-Par, for example. Rather than guessing the right split, we take the incoming context and give it all to process P, later reclaiming the unused part. This outgoing context is then



(a) Basic broker

(b) Values are piggybacked

Figure 19: Broker is an independent process

passed to process Q. The outgoing context of the parallel composition $P \mid Q$ is that of Q. See, e.g., Vasconcelos or Walker for details [3, 21]. Rule T-Res requires guessing the type of the two channel ends, so that one is dual to the other. Rather than guessing the type of channel end x, we seek the help of the programmer by working with an explicitly typed syntax—$(\nu xy : T)P$—as in Franco and Vasconcelos [3, 10], where T refers to the type of channel end x. For the type of channel end y, rather than guessing, we build it from type T; cf. [16, 17, 19, 30].

Running mixed sessions on a message passing architecture need not be an expensive operation. Take one of the communication axioms in Figure 2. We set up a broker process that receives the label-polarity pairs of both processes ($\{l\star_i\}_{i\in I}$ and $\{l\star_j\}_{j\in J}$), decides on a matching pair (guaranteed to exist for typed processes), and communicates the result back to the two processes. The processes then exchange the appropriate value, and proceed. If the broker is an independent process, then we exchange five messages per choice synchronisation. This *basic broker* is instantiated for two processes in Figure 19a.

$$P \triangleq \mathsf{lin}\, x(l?\_.P_1 + m!v.P_1 + n!w.P_3)$$
$$Q \triangleq \mathsf{lin}\, y(l!u.Q_1 + n?\_.Q_3)$$

We can do better by piggybacking the values in the output choices together with the label-polarities pairs. The broker passes its decision to the input side in the form of a triple label-polarity-value, yielding one less message exchanged, as showcased in Figure 19b.

Finally, we observe that the broker need not be an independent process; it can be located at one of the choice processes. This reduces the number of messages down to two messages in the general case, as described in Figures 20a and 20b where either P is the broker or Q is the broker. Even if the value was already sent by Q in the case that P is the broker, P must still let Q know which choice was taken, so that Q may proceed with the appropriate branch.

However, in particular cases one message may be enough. Take, for instance a process $P \triangleq \mathsf{un}\, x(l!u.P' + m!v.P')$. Independently of which branch is taken, the process proceeds as P'. Thus, if the broker is located in a process Q, then

(a) P is the broker

(b) Q is the broker

Figure 20: Broker is P or Q

P needs not be informed of the selected choice. The same is true for classical sessions where selection is a mixed-out choice of a single branch.

There are two other aspects that one should discuss when implementing mixed sessions on a message passing architecture other than the number of messages exchanged.

The first is related the type of broker used and to which values are revealed in a choice to the other party. In the case of the basic broker, only the chosen option value is revealed, and never to the broker itself. However, when we piggyback the values in the second type of broker, all values in the choice branches are revealed to the broker, even if they are not used in the end. This is even more striking in the case where one of the processes is the broker—the other party has access to all the possible values, independently of the choice that is taken.

The second aspect is also related to the values themselves which, in order to be presented in the choice, values must be computed *a priori*, even if they are not used in the choice. When dealing with the privacy of the values, we can choose which type of broker to use depending on how much we want to reveal to the other party.

9. Related Work

The origin of choice. Free (completely unrestricted) choice is central to process algebras, including BPA and CCS [31, 32]. Here we usually find processes of the form $P + Q$, where P and Q are arbitrary process. Free choice is also present in the very first proposal of the π-calculus [13, 14], even if Milner later uses guarded choice [33]. Sangiorgi and Walker's book builds on the pi-calculus with guarded (mixed) choice [34]. Guarded choices in all preceding proposals operate on possibly distinct channels—$x!\mathsf{true}.P + y?z.Q$— whereas choices on mixed sessions run on a common channel—$x(l!\mathsf{true}.P + m?y.Q)$.

Labelled-choices were embedded in the theory of session types by Honda et al. [2, 35, 36], where one finds primitives for value passing—$x!\mathsf{true}.P$ and $x?y.Q$—and, separately, for choice in the form of labelled selection—$x \lhd l.P$—and branching—$x \rhd \{l_i \colon P_i\}_{i \in I}$—see Section 5.1. Coalescing label selection with output and branching with input was proposed by Vasconcelos [37] (and later used by Sangiorgi [38]) as a means to describe concurrent objects. Demangeon and Honda use a similar language to study embeddings of calculi for functions and for session-based communication [39]. All these languages offer only separated (unmixed) choices and only on the input side.

Choices and labels. In process algebras choices are usually guarded by channels. For example, the interprocess communication rule for the pi-calculus, as in Sangiorgi and Walker [34] is

$$(x!y.P_1 + M_1) \mid (x?z.P_2 + M_2) \to P_1 \mid P_2[y/z]$$

where x denotes a channel and M_1 and M_2 denote sums of prefixed processes (each of the form $x!y.P$ or $x?y.P$). When one moves to algebras ruled by session types, labelled-prefixed choices are instead the norm, even if one can think of distilled choices in the form of (unlabelled) binary choice as in Caires et al. [24]. Labelled (n-ary) choices, as introduced by Honda et al. [2, 36], provide a better programming experience, in the same way that labelled sums (or variants) offer extra flexibility with respect to binary sums in functional programming.

Labels in mixed choice take a different flavour. To start with, they do not constitute keys to choices when seen as maps. Instead, labels in mixed choices are always paired with polarities and thus the obtained pairs are keys to type choices (but not to process choices), again for extra programming flexibility. Duplicated label-polarity choices in processes allow for controlled non-determinism and is much in line with the process algebra approach embodied by the above reduction rule, where channel x may prefix processes in M_1 or M_2.

As in the pi calculus, mixed choice process are not endowed with a "point of view": a choice is simply a choice. Communication safety (Theorem 3) however requires typed processes to take a view: internal or external (thus precluding unicity of types). Once classified, all options in the internal side must be included in the options of the external side. This is in clear contrast with the pi calculus, where choices such as M_1 and M_2 above may not have channels in common.

Mixed choices in the Singularity operating system. Concrete syntax apart, the language of linear mixed choices is quite similar to that of channel contracts in Sing♯ [40]. Rather than explicit recursive types, Sing♯ contracts uses named states (akin to typestates [41]), providing for more legible contracts. In Sing♯, each state in a contract corresponds to a mixed session $\mathsf{lin}\&\{l_i \star_i S_i.T_i\}$ (contracts are always written from the consumer side) where each l_i denotes a message tag, $\star$ the message direction (! or ?), S_i the type of the value in the message, and T_i the next state.

Stengel and Bultan showed that processes that follow Sing♯ contracts can engage in communication errors [42]. They further provide a realizability condition for contracts that essentially rules out mixed choices. Bono and Padovani present a calculus and a type system that models Sing♯ [17, 43]. The type system ensures that well-typed processes are exempt from communication errors, but the language of types excludes mixed-choices. So it seems that Sing♯-like languages only function properly under separated choice, yet our work survives under mixed choices. Contradiction? No! Sing♯ features asynchronous (or buffered) semantics whereas mixed sessions run under synchronous semantics. The operational semantics makes all the difference in this case.

Synchronicity, asynchronicity, and choice. Pierce and Turner identified the problem: "In an asynchronous language guarded choice should be restricted still further since an asynchronous output in a choice is sensitive to buffering" [9] and Peters et al. state that "a discussion on synchrony versus asynchrony cannot be separated from a discussion on choice" [44, 45]. Based on classical sessions, mixed sessions are naturally synchronous. The naive introduction of an asynchronous semantics would ruin the main results of the language (see Section 4). Asynchronous semantics are known to be compatible with classical sessions; see Honda et al. [46, 47] for multiparty asynchronous session types and Fowler et al. [48] and Gay and Vasconcelos [1] for two examples of functional languages with session types and asynchronous semantics. So one can ask whether a language can be designed where mixed-choices are handled synchronously and separated-choices asynchronously, where a type-guided operational semantics with by-default asynchronous semantics reverts to a synchronous semantics when in presence of mixed-choices.

Separation results. Palamidessi shows that the π-calculus with mixed choice is more expressive than its subset with separated choice [49]. Gorla provides a simpler proof [50] of the same result and Peters and Nestmann analyse the problem from the perspective of breaking initial symmetries in separated-choice processes [51]. Unlike the choices in both π-calculus with mixed and separated choice, choices in mixed sessions operate on the same channel and are guided by types. In particular, there is a communication safety property enforced by the type system in which all the options of the internal side must be included into the options of the external side. Another difference to these separation results is that our translation from mixed to classical sessions is guided by the type of the process. Due to the simple syntax of mixed sessions, we can write a process which translates to different classical sessions processes, depending on its type. A process of the form $(\nu xy)(P \mid \sigma(P))$ where P is

$$\mathsf{lin}\, x(l?\_.\mathsf{lin}\, w(m!worker.\mathbf{0}) + l!().\mathsf{lin}\, w(m!leader.\mathbf{0}))$$

and σ a substitution of x by y, would in principle be a symmetric network which elects a leader as in [51]. However, its translation will not be symmetric as one of P or $\sigma(P)$ will be the external choice and the other the internal. One may ask if we could make the translation to classical sessions independent of the type of the process, for instance, by annotating the type information at the syntax level (e.g., writing $\oplus$ and & explicitly in processes). From the viewpoint of usability and software-engineering, processes would be harder to write and code reuse made difficult. On the other hand, translation from such a syntax would be closer to that of Peters and Nestmann [51]. This is a direction to be explored in the future.

The origin of mixed sessions. Mixed sessions dawned on us when looking into an algorithm to decide the equivalence of context-free session types [52, 53]. The algorithm translates types into (simple) context-free grammars. The decision procedure runs on arbitrary simple grammars: the right-hand sides of grammar

productions may start with a label-output or a label-input pair for the same non-terminal symbol at the left of the production. We then decided to explore mixed sessions and picked the simplest possible language for the effect: the π-calculus. It would be interesting to look into mixed context-free session types, given that decidability of type equivalence is guaranteed.

10. Conclusion

We introduce mixed sessions: session types with mixed choice. Classical session types feature separated choices; in fact all the proposals in the literature we are aware of provide for choice on the input side only, even if we can easily think of choice on the output side. In contrast to separated choices, where output operations are inherently associated with the endpoint that initiates the communication, mixed choices allow to unfasten this association. Mixed sessions increase flexibility in programming and are easily realisable in conventional message passing architectures.

Mixed choices come with a type system featuring subtyping. Typability is preserved by reduction. Furthermore well-typed programs are exempt from runtime errors. We provide suggestions on how to derive a type checking procedure, even if we do not formalise one.

Classical session types can be embedded in mixed sessions: we provide for an encoding and show typing and operational correspondences. Conversely, mixed sessions can be encoded into classical session types: we present an encoding and show that it is minimal, according to the criteria of Kouzapas et al. [6].

An interesting avenue for further development is looking for a suitable notion of weak bisimulation for mixed choices in such a way that the mixed-to-classical encoding may enjoy reduction soundness. Another looks for a hybrid type-guided semantics, asynchronous by default, that reverts to synchronous when in presence of an output choice.

The generalisation of mixed choices to the multiparty setting is also an interesting topic to be explored. The multiparty session types programme has global types projected onto diverse binary session types, each describing the behaviour of one particular participant. So, one could think of developing mixed-session global types to be projected into the types proposed in this paper.

Mixed sessions are not exempt from deadlocks (see Section 4). There are different approaches to ensure the absence of deadlocks in session-based systems. One is to start from global types, an avenue discussed in the previous paragraph. Another approach is to look into extensions of the linear logic interpretations of session types [24, 27] but that may raise difficulties at the level of the logic. A third possibility is to look into type systems that capture some form dependency between actions (see [54, Chapter 7]).

Acknowledgements. We thank Simon Gay, Uwe Nestmann, Kirstin Peters, Peter Thiemann and Nobuko Yoshida for comments and discussions. We would also like to thank the anonymous reviewers for the valuable suggestions. This

work was supported by FCT through the project SafeSessions, ref. PTDC/CCI-COM/6453/2020 and the LASIGE Research Unit, refs. UIDB/00408/2020 and UIDP/00408/2020, and by Cost Action CA15123 EUTypes.

References

[1] S. J. Gay, V. T. Vasconcelos, Linear type theory for asynchronous session types, J. Funct. Program. 20 (1) (2010) 19–50. doi:10.1017/S0956796809990268.

[2] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: Programming Languages and Systems, Vol. 1381 of Lecture Notes in Computer Science, Springer, 1998, pp. 122–138. doi:10.1007/BFb0053567.

[3] V. T. Vasconcelos, Fundamentals of session types, Inf. Comput. 217 (2012) 52–70. doi:10.1016/j.ic.2012.05.002.

[4] V. T. Vasconcelos, Fundamentals of session types, in: Formal Methods for Web Services, Vol. 5569 of Lecture Notes in Computer Science, Springer, 2009, pp. 158–186. doi:10.1007/978-3-642-01918-0_4.

[5] N. Yoshida, V. T. Vasconcelos, Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication, Electr. Notes Theor. Comput. Sci. 171 (4) (2007) 73–93. doi:10.1016/j.entcs.2007.02.056.

[6] D. Kouzapas, J. A. Pérez, N. Yoshida, On the relative expressiveness of higher-order session processes, Inf. Comput. 268. doi:10.1016/j.ic.2019.06.002.

[7] V. T. Vasconcelos, F. Casal, B. Almeida, A. Mordido, Mixed sessions, in: Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Vol. 12075 of Lecture Notes in Computer Science, Springer, 2020, pp. 715–742. doi:10.1007/978-3-030-44914-8_26.

[8] F. Casal, A. Mordido, V. T. Vasconcelos, Mixed sessions: the other side of the tape, in: Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS [55], pp. 46–60. doi:10.4204/EPTCS.314.5.

[9] B. C. Pierce, D. N. Turner, Pict: a programming language based on the pi-calculus, in: Proof, Language, and Interaction, Essays in Honour of Robin Milner, The MIT Press, 2000, pp. 455–494.

[10] J. Franco, V. T. Vasconcelos, A concurrent programming language with refined session types, in: Software Engineering and Formal Methods, Vol. 8368 of Lecture Notes in Computer Science, Springer, 2013, pp. 15–28. doi:10.1007/978-3-319-05032-4_2.

[11] R. Milner, Functions as processes, in: Automata, Languages and Programming, Vol. 443 of Lecture Notes in Computer Science, Springer, 1990, pp. 167–180. doi:10.1007/BFb0032030.

[12] R. Milner, Functions as processes, Mathematical Structures in Computer Science 2 (2) (1992) 119–141. doi:10.1017/S0960129500001407.

[13] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I, Inf. Comput. 100 (1) (1992) 1–40. doi:10.1016/0890-5401(92)90008-4.

[14] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, II, Inf. Comput. 100 (1) (1992) 41–77. doi:10.1016/0890-5401(92)90009-5.

[15] H. P. Barendregt, The lambda calculus - its syntax and semantics, Vol. 103 of Studies in logic and the foundations of mathematics, North-Holland, 1985.

[16] G. Bernardi, M. Hennessy, Using higher-order contracts to model session types, Logical Methods in Computer Science 12 (2). doi:10.2168/LMCS-12(2:10)2016.

[17] V. Bono, L. Padovani, Typing copyless message passing, Logical Methods in Computer Science 8 (1). doi:10.2168/LMCS-8(1:17)2012.

[18] S. J. Gay, P. Thiemann, V. T. Vasconcelos, Duality of session types: The final cut, in: Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency [55], pp. 23–33. doi:10.4204/EPTCS.314.3.

[19] S. Lindley, J. G. Morris, Talking bananas: structural recursion for session types, in: Garrigue et al. [56], pp. 434–447. doi:10.1145/2951913.2951921.

[20] S. J. Gay, M. Hole, Subtyping for session types in the pi calculus, Acta Inf. 42 (2-3) (2005) 191–225. doi:10.1007/s00236-005-0177-z.

[21] D. Walker, Advanced Topics in Types and Programming Languages, The MIT Press, 2005, Ch. Substructural Type Systems, pp. 3–44.

[22] N. Kobayashi, B. C. Pierce, D. N. Turner, Linearity and the pi-calculus, in: Conference Record of POPL'96, ACM Press, 1996, pp. 358–371. doi:10.1145/237721.237804.

[23] N. Kobayashi, B. C. Pierce, D. N. Turner, Linearity and the pi-calculus, ACM Trans. Program. Lang. Syst. 21 (5) (1999) 914–947. doi:10.1145/330249.330251.

[24] L. Caires, F. Pfenning, B. Toninho, Linear logic propositions as session types, Mathematical Structures in Computer Science 26 (3) (2016) 367–423. doi:10.1017/S0960129514000218.

[25] L. Caires, F. Pfenning, Session types as intuitionistic linear propositions, in: P. Gastin, F. Laroussinie (Eds.), CONCUR 2010 - Concurrency Theory, Vol. 6269 of Lecture Notes in Computer Science, Springer, 2010, pp. 222–236. doi:10.1007/978-3-642-15375-4_16.

[26] P. Wadler, Propositions as sessions, in: ACM SIGPLAN International Conference on Functional Programming, ACM, 2012, pp. 273–286. doi:10.1145/2364527.2364568.

[27] P. Wadler, Propositions as sessions, J. Funct. Program. 24 (2-3) (2014) 384–418. doi:10.1017/S095679681400001X.

[28] R. Milner, D. Sangiorgi, Barbed bisimulation, in: W. Kuich (Ed.), Automata, Languages and Programming, 19th International Colloquium, Vol. 623 of Lecture Notes in Computer Science, Springer, 1992, pp. 685–695. doi:10.1007/3-540-55719-9_114.

[29] K. Honda, M. Tokoro, On asynchronous communication semantics, in: Object-Based Concurrent Computing, Vol. 612 of Lecture Notes in Computer Science, Springer, 1991, pp. 21–51. doi:10.1007/3-540-55613-3_2.

[30] G. Bernardi, O. Dardha, S. J. Gay, D. Kouzapas, On duality relations for session types, in: Trustworthy Global Computing, Vol. 8902 of Lecture Notes in Computer Science, Springer, 2014, pp. 51–66. doi:10.1007/978-3-662-45917-1_4.

[31] J. A. Bergstra, J. W. Klop, Process theory based on bisimulation semantics, in: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Vol. 354 of Lecture Notes in Computer Science, Springer, 1988, pp. 50–122. doi:10.1007/BFb0013021.

[32] R. Milner, A Calculus of Communicating Systems, Vol. 92 of Lecture Notes in Computer Science, Springer, 1980. doi:10.1007/3-540-10235-3.

[33] R. Milner, The polyadic pi-calculus: A tutorial, ECS-LFCS 91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, this report was published in F. L. Hamer, W. Brauer and H. Schwichtenberg, editors, Logic and Algebra of Specification. Springer-Verlag, 1993 (1991).

[34] D. Sangiorgi, D. Walker, The Pi-Calculus - a theory of mobile processes, Cambridge University Press, 2001.

[35] K. Honda, Types for dyadic interaction, in: CONCUR '93, 4th International Conference on Concurrency Theory, Vol. 715 of Lecture Notes in Computer Science, Springer, 1993, pp. 509–523. doi:10.1007/3-540-57208-2_35.

[36] K. Takeuchi, K. Honda, M. Kubo, An interaction-based language and its typing system, in: PARLE '94: Parallel Architectures and Languages Europe, Vol. 817 of Lecture Notes in Computer Science, Springer, 1994, pp. 398–413. doi:10.1007/3-540-58184-7_118.

[37] V. T. Vasconcelos, Typed concurrent objects, in: Object-Oriented Programming, Vol. 821 of Lecture Notes in Computer Science, Springer, 1994, pp. 100–117. doi:10.1007/BFb0052178.

[38] D. Sangiorgi, An interpretation of typed objects into typed pi-calculus, Inf. Comput. 143 (1) (1998) 34–73. doi:10.1006/inco.1998.2711.

[39] R. Demangeon, K. Honda, Full abstraction in a subtyped pi-calculus with linear types, in: CONCUR 2011 - Concurrency Theory, Vol. 6901 of Lecture Notes in Computer Science, Springer, 2011, pp. 280–296. doi:10.1007/978-3-642-23217-6_19.

[40] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, S. Levi, Language support for fast and reliable message-based communication in singularity OS, in: Proceedings of the 2006 EuroSys Conference, ACM, 2006, pp. 177–190. doi:10.1145/1217935.1217953.

[41] R. E. Strom, S. Yemini, Typestate: A programming language concept for enhancing software reliability, IEEE Trans. Software Eng. 12 (1) (1986) 157–171. doi:10.1109/TSE.1986.6312929.

[42] Z. Stengel, T. Bultan, Analyzing singularity channel contracts, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ACM, 2009, pp. 13–24. doi:10.1145/1572272.1572275.

[43] V. Bono, C. Messa, L. Padovani, Typing copyless message passing, in: Programming Languages and Systems, Vol. 6602 of Lecture Notes in Computer Science, Springer, 2011, pp. 57–76. doi:10.1007/978-3-642-19718-5_4.

[44] K. Peters, J. Schicke, U. Nestmann, Synchrony vs causality in the asynchronous pi-calculus, in: Proceedings 18th International Workshop on Expressiveness in Concurrency, Vol. 64 of EPTCS, 2011, pp. 89–103. doi:10.4204/EPTCS.64.7.

[45] K. Peters, J. Schicke-Uffmann, U. Goltz, U. Nestmann, Synchrony versus causality in distributed systems, Mathematical Structures in Computer Science 26 (8) (2016) 1459–1498. doi:10.1017/S0960129514000644.

[46] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2008, pp. 273–284. doi:10.1145/1328438.1328472.

[47] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, J. ACM 63 (1) (2016) 9:1–9:67. doi:10.1145/2827695.

[48] S. Fowler, S. Lindley, J. G. Morris, S. Decova, Exceptional asynchronous session types: session types without tiers, PACMPL 3 (POPL) (2019) 28:1–28:29. doi:10.1145/3290341.

[49] C. Palamidessi, Comparing the expressive power of the synchronous and asynchronous pi-calculi, Mathematical Structures in Computer Science 13 (5) (2003) 685–719. doi:10.1017/S0960129503004043.

[50] D. Gorla, Towards a unified approach to encodability and separation results for process calculi, Inf. Comput. 208 (9) (2010) 1031–1053. doi:10.1016/j.ic.2010.05.002.

[51] K. Peters, U. Nestmann, Breaking symmetries, Mathematical Structures in Computer Science 26 (6) (2016) 1054–1106. doi:10.1017/S0960129514000346.

[52] B. Almeida, A. Mordido, V. T. Vasconcelos, Deciding the bisimilarity of context-free session types, in: Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Vol. 12079 of Lecture Notes in Computer Science, Springer, 2020, pp. 39–56. doi:10.1007/978-3-030-45237-7_3.

[53] P. Thiemann, V. T. Vasconcelos, Context-free session types, in: Garrigue et al. [56], pp. 462–475. doi:10.1145/2951913.2951926.

[54] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, N. Yoshida, Behavioral types in programming languages, Found. Trends Program. Lang. 3 (2-3) (2016) 95–230. doi:10.1561/2500000031. URL https://doi.org/10.1561/2500000031

[55] Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020, Vol. 314 of EPTCS. doi:10.4204/EPTCS.314.

[56] J. Garrigue, G. Keller, E. Sumii (Eds.), Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ACM, 2016. doi:10.1145/2951913.