A Continuation-Based Solution of the Linearity Challenge

Luca Padovani
luca.padovani2@unibo.it
University of Bologna
Claudia Raffaelli
University of Camerino

Research Article

Keywords: session types, linear logic, continuations, deadlock freedom, Agda

Posted Date: January 21st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8499890/v1

Additional Declarations: No competing interests reported.

A Continuation-Based Solution of the Linearity Challenge

Luca Padovani[1*] and Claudia Raffaelli[2]

[1]Department of Computer Science and Engineering, Università di Bologna, Mura Anteo Zamboni 7, Bologna, 40126, BO, Italy.
[2]Computer Science Division, Università di Camerino, via Madonna delle Carceri 7, Camerino, 62032, MC, Italy.

*Corresponding author(s). E-mail(s): luca.padovani2@unibo.it;
Contributing authors: claudia.raffaelli@studenti.unicam.it;

Abstract

The formalisation of session calculi is made difficult by the management of session channels, which are linear resources that cannot be discarded or duplicated and whose type changes over time, as input/output operations are performed on them. Context splitting, the channel management technique directly related to the way session type theories are usually written with pen and paper, is often considered a hindrance and a notable source of complexity, to the point where several alternative approaches have been recently proposed. In this paper we describe the Agda formalisation of a process calculus based on classical linear logic that supports the modeling of binary sessions through their encoding with explicit continuation channels. The formalisation turns out to be remarkably compact despite the adoption of context splitting. We argue that the logical nature of the calculus and the use of explicit continuations are contributing factors to the simplicity of its formalisation.

Keywords: session types, linear logic, continuations, deadlock freedom, Agda

1 Introduction

The Concurrent Calculi Formalisation Benchmark [1] is a collection of challenges concerning the mechanisation of core models of concurrent and distributed programming languages. These models often make use of distinctive features that set them apart from the models of sequential programming languages, such as the adoption of substructural (linear, affine) type systems, the dynamic scope of first-class channels in systems of communicating processes, the need for coinductive definitions and proof methods for describing and reasoning on possibly infinite behaviours. The benchmark aims at identifying effective formalisation techniques that take these features into account so as to foster the adoption of machine-checked proofs in research work concerning concurrent and distributed programming languages.

One of the challenges in the benchmark, henceforth called *linearity challenge*, concerns the formalisation of a *minimal calculus of sessions*. Sessions and session types [2–5] are established abstractions for the static analysis of distributed programs based on peer-to-peer communications. Every session type system revolves around three key ideas: (1) session endpoints are *linear resources* that cannot be discarded or duplicated without compromising some safety and liveness properties of a program; (2) the type of a session endpoint is *updated* after each use to reflect the state of the protocol it describes; (3) peer session endpoints are meant to be used in complementary ways so as to guarantee the absence of communication errors and, to some extent, progress of the interaction. The linearity challenge is based on the observation that the proper management of linear resources in a formalisation often requires a large number of auxiliary definitions and technical results that divert from the main problem under study [1]. One of the alleged culprits of such complexity is *context splitting*, namely the operation that partitions a typing context in such a way that the linear resources described therein end up in only one of the partitions. This observation has led to the exploration of various alternative techniques including leftover typing [6], the use of linearity predicates [7] and tagged contexts [8].

In this paper we approach the linearity challenge from a different angle: *instead of proposing new techniques that make it easy to formalise the calculus in the challenge, we propose a (relatively) new calculus that is easy to formalise with the existing techniques.* More specifically, we describe the Linear Calculus of Continuations (LCC) whose type system coincides with the proof system of classical linear logic and that features *linear channels* instead of sessions. While a session endpoint can be used *multiple times* (sequentially), linear channels must be used *exactly once*. LCC retains the expressiveness of other session calculi thanks to *explicit continuations*, which enable the encoding of (binary) sessions is terms of linear channels [9, 10]: each message exchanged on a linear channel may include one or two fresh channels – the *continuations* – on which the rest of the conversation takes place. Overall, LCC is nothing but a low-level version of CP [11] – Wadler's calculus of sessions based on classical linear logic – such that sessions can be encoded instead of being featured natively.

The logical foundations of LCC and the use of explicit continuations play an important role in taming the complexity of the formalisation. Working with a calculus based on linear logic prevents *by construction* the same (sequential) process to own both endpoints of a session, which is undesirable since it does not correspond to a useful pattern of interaction (every meaningful session requires its endpoints to be used by parallel processes) and is a potential source of deadlocks. From the standpoint of the formalisation, where the representation of channels is a primary design choice, it spares us the need to distinguish the two endpoints of a session, e.g. by means of polarities [6, 12] or by using different names connected by the same binder [5, 13]. Using a calculus with linear channels and explicit continuations spares us the need to *update* the type of channels in typing contexts. Once a channel has been used it is effectively consumed, therefore its type can be *removed* from the typing context and the type of the continuation channel is *added* back to the typing context. As is turns out, removing and adding types is easier than updating them. Even more so considering that the type of continuation channels must be added *at the beginning* of a typing context, since such channels are fresh by definition.

The formalisation of LCC that we obtain is both the most complete (in terms of features supported by the calculus) and the most streamlined (in terms of code size) among the known formalisations of session/linear calculi [6–8, 14–19]. It is also one of only two formalisations that prove the deadlock freedom property for a session calculus [19] and it does so with substantially less code.

The rest of the paper is organised as follows. Section 2 describes the syntax and the operational semantics of LCC and states the properties of well-typed processes that we formalise and prove, namely typing preservation, deadlock freedom and runtime safety. Section 3 illustrates the key aspects of the Agda formalisation with particular emphasis on the representation of channels and of typing contexts. We assume that the reader is somewhat familiar about Agda but we recall the lesser known definitions from Agda's standard library. Section 4 discusses related work more in detail by providing a qualitative and quantitative comparison between the known formalisations of session/linear calculi. Section 5 summarises our contributions and discusses ongoing and future work.

The formalisation has been checked with Agda 2.8.0 and the code is available on in a public repository on GitHub [20].

2 A Linear Calculus of Continuations

In this section we give a cursory presentation of LCC starting from its types (Section 2.1) then moving on to the syntax of processes (Section 2.2), their reduction semantics (Section 2.3), the typing rules (Section 2.4) and the formulation of the properties ensured by the type system (Section 2.5).

As we have anticipated in Section 1, LCC is closely related to CP [5, 11] except that LCC features linear channels instead of sessions. We refer the reader to the literature on CP [5, 11, 21] and other session calculi based on linear logic [22, 23] for a thorough introduction to these models.

2.1 Types

The types of LCC, ranged over by A, B, ..., are the linear logic propositions generated by the grammar

$$A, B ::= X \mid X^\perp \mid \top \mid \mathbf{0} \mid \bot \mid \mathbf{1} \mid A \& B \mid A \oplus B \mid A \parr B \mid A \otimes B \mid \forall X.A \mid \exists X.A \mid {!A} \mid {?A}$$

where X, Y, ... range over an infinite set of *type (or proposition) variables.*

Table 1 Syntax of LCC.

P, Q	$::=$	$x \leftrightarrow y$	link
	$\mid$	$x \triangleright \{\}$	fail
	$\mid$	$x().P$	wait
	$\mid$	$x[]$	close
	$\mid$	$x \triangleright (z)\{P, Q\}$	case
	$\mid$	$x \triangleleft \mathsf{inj}_i[z].P$	select
	$\mid$	$x(y, z).P$	join
	$\mid$	$x[y, z](P \mid Q)$	fork
	$\mid$	$x(X, z).P$	for all
	$\mid$	$x[A, z].P$	exists
	$\mid$	$!x(y).P$	server
	$\mid$	$?x[y].P$	client
	$\mid$	$?x[].P$	weakening
	$\mid$	$?x[y, z].P$	contraction
	$\mid$	$(x : A)(P \mid Q)$	cut

The interpretation of linear logic propositions as behaviours is quite standard. Constants and connectives describe *linear channels*, which must be used for a single communication, whereas the modalities ! and ? describe *shared channels*. The multiplicative constants $\bot$ and $\mathbf{1}$ describe channels used for receiving/sending an empty message (without continuations). The additive constants describe unusable channels. They can play the role of smallest/largest type in type systems that support a notion of subtyping [24], but will mostly ignore them in this work. The additive connectives $A \mathbin{\&} B$ and $A \oplus B$ describe channels used for receiving/sending either a continuation of type A or a continuation of type B. The sender selects one of the two possibilities, while the receiver offers both. The multiplicative connectives $A \parr B$ and $A \otimes B$ describe channels used for receiving/sending two continuations, one of type A and the other of type B. The quantifiers $\forall X.A$ and $\exists X.A$ describe channels used for receiving/sending a type X along with a continuation of type A. They are useful to describe parametric protocol polymorphism. Finally, the "of course" modality $!A$ and the "why not" modality $?A$ describe shared channels on which servers and clients accept and request connections of type A.

The notions of *free type variables*, of *duality* and of *type substitution* are standard [11]. In particular, we write $A^\bot$ for the dual of A and $A\{B/X\}$ for the type obtained by replacing the free occurrences of X in A with B.

2.2 Processes

The syntax of processes makes use of an infinite set of *channels*, ranged over by x, y and z, and is shown in Table 1. A link $x \leftrightarrow y$ denotes the merging of the channels x and y, so that each message sent on one of the channels is forwarded to the other. As discussed in the literature [21] and illustrated in Example 2.2, this form is useful for modeling the exchange of an existing channel on another channel. The processes $x().P$ and $x[]$ respectively model the input and output of an empty message on the channel x. The latter process terminates after the message has been sent, while the former continues as P once the message has been received. The process $x \triangleright \{\}$ can be used to denote a failure concerning the channel x. The process $x \triangleright (z)\{Q_1, Q_2\}$ offers a choice on channel x and continues as either Q_1 or Q_2 depending on which branch is selected with z bound to the received continuation channel. The process $x \triangleleft \mathsf{inj}_i[z].P$ performs a choice (represented by a label inj_i with $i = 1, 2$) and sends a fresh continuation channel z on the channel x. The processes $x(y, z).R$ and $x[y, z](P \mid Q)$ describe the input/output of two fresh continuations channels y and z on the channel x. The receiver can use y and z in whatever order. The sender forks into P and Q, each using y and z respectively. The processes $x(X, z).P$ and $x[A, z].P$ describe the input/output of a type on the channel x along with a fresh continuation z.

Next we have process forms dealing with shared (non-linear) channels. The processes $!x(y).P$ and $?x[y].P$ respectively denote *servers* and *clients* acting on the shared channel x. Each request (from a client) spawns a copy of the server's body using the continuation channel y. The process $?x[].P$ denotes an explicit *weakening*, that is a client that *does not* use x. The process $?x[y, z].P$ denotes an explicit *contraction* whereby a client uses x multiple times (once with name y and once with name z).

Finally, cuts of the form $(x : A)(P \mid Q)$ represent the parallel composition of the processes P and Q connected by a channel x, which has type A in P and type $A^\perp$ in Q. Henceforth we write $(x)(P \mid Q)$ omitting the type annotation A when it is irrelevant or clear from the context.

The notions of free and bound channels are fairly standard, bearing in mind that output prefixes bind continuation channels in (some) continuation processes. For instance, $x \triangleleft \mathsf{inj}_i[z].P$ binds z in P, while $x[y, z](P \mid Q)$ binds y in P but not in Q and binds z in Q but not in P. We write $\mathsf{fc}(P)$ for the set of channels occurring free in P and we identify processes up to renaming of bound channels.

Example 2.1. We can represent the protocol of a boolean value being produced as the type $\mathbb{B} \stackrel{\mathsf{def}}{=} \mathbf{1} \oplus \mathbf{1}$ and that of a boolean value being consumed as its dual $\mathbb{B}^\perp = \perp \,\&\, \perp$. Following these types, the boolean constants can be modeled by the processes

$$\mathit{True}(x) \stackrel{\mathsf{def}}{=} x \triangleleft \mathsf{inj}_1[x'].x'[] \qquad \mathit{False}(x) \stackrel{\mathsf{def}}{=} x \triangleleft \mathsf{inj}_2[x'].x'[]$$

and the boolean negation function by the process

$$\mathit{Not}(x, y) \stackrel{\mathsf{def}}{=} x \triangleright (x')\{x'().\mathit{False}(y), x'().\mathit{True}(y)\}$$

As an example, the composition $(x : \mathbb{B})(\mathit{True}(x) \mid \mathit{Not}(x, y))$ produces false on y. Note the use of explicit continuations in these processes and the fact that each channel is used exactly once. The same processes in CP would be written as

$$\mathit{True}(x) \stackrel{\mathsf{def}}{=} x \triangleleft \mathsf{inj}_1.x[] \quad \mathit{False}(x) \stackrel{\mathsf{def}}{=} x \triangleleft \mathsf{inj}_2.x[] \quad \mathit{Not}(x, y) \stackrel{\mathsf{def}}{=} x \triangleright \begin{Bmatrix} x().\mathit{False}(y) \\ x().\mathit{True}(y) \end{Bmatrix}$$

where each channel is used multiple times to indicate the sequence of input/output actions pertaining to the same session. In general, the CP version of an LCC process can be obtained by reusing the same channel x in place of the continuation z in Table 1. ⌟

Table 2 Operational semantics of LCC. Many side conditions for the [s-*] rules are omitted (see text).

[s-link]	$x \leftrightarrow y$	$\sqsupseteq$	$y \leftrightarrow x$	
[s-comm]	$(x)(P \mid Q)$	$\sqsupseteq$	$(x)(Q \mid P)$	
[s-fail]	$(x)(y \triangleright \{\} \mid P)$	$\sqsupseteq$	$y \triangleright \{\}$	
[s-wait]	$(x)(y().P \mid Q)$	$\sqsupseteq$	$y().(x)(P \mid Q)$	
[s-case]	$(x)(y \triangleright (z)\{P, Q\} \mid R)$	$\sqsupseteq$	$y \triangleright (z)\{(x)(P \mid R), (x)(Q \mid R)\}$	
[s-select]	$(x)(y \triangleleft \mathsf{inj}_i[z].P \mid Q)$	$\sqsupseteq$	$y \triangleleft \mathsf{inj}_i[z].(x)(P \mid Q)$	
[s-join]	$(x)(y(u, z).P \mid Q)$	$\sqsupseteq$	$y(u, z).(x)(P \mid Q)$	
[s-fork-l]	$(x)(y[u, z](P \mid Q) \mid R)$	$\sqsupseteq$	$y[u, z]((x)(P \mid R) \mid Q)$	$x \in \mathsf{fc}(P)$
[s-fork-r]	$(x)(y[u, z](P \mid Q) \mid R)$	$\sqsupseteq$	$y[u, z](P \mid (x)(Q \mid R))$	$x \in \mathsf{fc}(Q)$
[s-forall]	$(x)(y(X, z).P \mid Q)$	$\sqsupseteq$	$y(X, z).(x)(P \mid Q)$	
[s-exists]	$(x)(y[A, z].P \mid Q)$	$\sqsupseteq$	$y[A, z].(x)(P \mid Q)$	
[s-server]	$(x)(!y(u).P \mid !x(v).Q)$	$\sqsupseteq$	$!y(u).(x)(P \mid !x(v).Q)$	
[s-client]	$(x)(?y[z].P \mid Q)$	$\sqsupseteq$	$?y[z].(x)(P \mid Q)$	
[s-weaken]	$(x)(?y[].P \mid Q)$	$\sqsupseteq$	$?y[].(x)(P \mid Q)$	
[s-contract]	$(x)(?y[u, v].P \mid Q)$	$\sqsupseteq$	$?y[u, v].(x)(P \mid Q)$	
[r-link]	$(x)(x \leftrightarrow y \mid P)$	$\rightsquigarrow$	$P\{y/x\}$	
[r-close]	$(x)(x[] \mid x().P)$	$\rightsquigarrow$	P	
[r-select]	$(x)(x \triangleleft \mathsf{inj}_i[z].P \mid x \triangleright (z)\{Q_1, Q_2\})$	$\rightsquigarrow$	$(z)(P \mid Q_i)$	$i \in \{1, 2\}$
[r-fork]	$(x)(x[y, z](P \mid Q) \mid x(y, z).R)$	$\rightsquigarrow$	$(y)(P \mid (z)(Q \mid R))$	
[r-exists]	$(x)(x[A, z].P \mid x(X, z).Q)$	$\rightsquigarrow$	$(z)(P \mid Q\{A/X\})$	
[r-connect]	$(x)(!x(y).P \mid ?x[y].Q)$	$\rightsquigarrow$	$(y)(P \mid Q)$	
[r-weaken]	$(x)(!x(y).P \mid ?x[].Q)$	$\rightsquigarrow$	$?\overline{z}[].Q$	$\mathsf{fc}(P) = \{y, \overline{z}\}$
[r-contract]	$(x)(!x(y).P \mid ?x[x', x''].Q)$	$\rightsquigarrow$	$?\overline{z}[\overline{z}', \overline{z}''].(x')(!x'(y).P' \mid (x'')(!x''(y).P'' \mid Q))$	*
[r-cut]	$(x)(P \mid R)$	$\rightsquigarrow$	$(x)(Q \mid R)$	$P \rightsquigarrow Q$
[r-cong]	P	$\rightsquigarrow$	Q	$P \sqsupseteq R \rightsquigarrow Q$

$^{*}\mathsf{fc}(P) = \{y, \overline{z}\}, P' = P\{\overline{z}'/\overline{z}\}, P'' = P\{\overline{z}''/\overline{z}\}$

2.3 Operational Semantics

The operational semantics of LCC is shown in Table 2 and is given by two relations: a *structural pre-congruence* relation $\sqsupseteq$, which relates essentially indistinguishable processes, and a *reduction* relation $\rightsquigarrow$, which models communications. Let us describe the two relations more in detail.

Structural pre-congruence is the least pre-congruence defined by the [s-*] rules. [s-link] and [s-comm] assert that links and parallel compositions are commutative. The remaining rules, when read from left to right, push a cut on x underneath the topmost prefix on y of one of its sub-processes when $x \neq y$. These rules are key to float input/output actions to the top-level of a process, so that they can interact with corresponding complementary actions in the surrounding context. All these rules have implicit side-conditions (not shown in Table 2) aimed at preserving the meaning of channels when binders are moved around: terms entering or exiting the scope of a binder for x must not have free occurrences of x. This holds also for the type variable X in the rule [s-forall]. We content ourselves with such informal description of these side conditions given that we are going to formalise LCC later on. Note also that there are two versions of [s-fork-l] and [s-fork-r] depending on which of the two continuations (either P or Q) contains a free occurrence of the restricted channel x.

Another rule that deserves attention is [s-server]. In this case, a cut can be pushed underneath a server prefix $!y(u)$ only provided that the other process in the cut is also a server on the channel restricted by the cut.

Reduction is defined by the [r-*] rules, most of them coinciding with principal cut reductions of linear logic. The rules [r-link], [r-close], [r-select], [r-fork] and [r-exists] erase the topmost cut and replace it with zero, one or two new cuts, depending on the number of continuation channels that are exchanged. Note that the rule [r-link] eliminates a link $x \leftrightarrow y$ by substituting y for x in the scope of x and [r-close] models the communication of an empty message (without continuations). The rule [r-exists] models the instantiation of a polymorphic variable X as the communication of a type A. We write $Q\{A/X\}$ for the process obtained by replacing every free occurrence of the type variable X with A. The rule [r-connect] models the connection between a client and a server, whereas the rules [r-weaken] and [r-contract] respectively model the disposal of an unused server and the duplication of a server. In these rules we use some slightly informal notation for denoting sequences of (pairwise distinct) channels and prefixes. In particular, $\overline{z}$ stands for a sequence $z_1, \dots, z_n$ of channels, $?\overline{z}[].Q$ stands for a sequence $?z_1[] \dots ?z_n[].Q$ of weakening prefixes and $?\overline{z}[\overline{z}', \overline{z}''].R$ stands for a sequence $?z_1[z_1', z_1''] \dots ?z_n[z_n', z_n''].R$ of contraction prefixes.

The rule [r-cut] propagates reductions through cuts and the rule [r-cong] enables reduction up to structural pre-congruence.

Example 2.2. In this example we illustrate the role of links for the communication of free channels by modeling an echo server that consumes boolean values and sends them back unchanged. The protocol of the server we want to model is described by the type $!(\mathbb{B}^\perp \parr (\mathbb{B} \otimes \mathbf{1}))$ where the modality $!$ indicates that the server is able to accept an unbounded number of requests and the type $\mathbb{B}^\perp \parr (\mathbb{B} \otimes \mathbf{1})$ describes the sequence of actions performed by the server at each connection with a client: the server first consumes a boolean value (say u, of type $\mathbb{B}^\perp$), then produces another boolean value (say v, of type $\mathbb{B}$) and finally sends an empty message. We can model the server and a possible client thus:

$$\begin{aligned} \mathit{Server}(x) &\stackrel{\mathsf{def}}{=} !x(y).y(u, y').y'[v, y''](v \leftrightarrow u \mid y''[]) \\ \mathit{Client}(x, z) &\stackrel{\mathsf{def}}{=} ?x[y].y[u, y'](\mathit{True}(u) \mid y'(v, y'').y''().v \leftrightarrow z) \end{aligned}$$

Notice the use of continuations for chaining communications together. In the server, the link $v \leftrightarrow u$ merges v and u so as to send the same channel u received from the client. In the client, the link $v \leftrightarrow z$ "assigns" the message v received from the server to the free channel z, which represents the result of the interaction. If we write $\rightsquigarrow^*$ for the reflexive, transitive closure of $\rightsquigarrow$ it is easy to verify that $(x)(\mathit{Client}(x) \mid \mathit{Server}(x)) \rightsquigarrow^* \mathit{True}(z)$. ⌟

2.4 Type System

We use typing contexts (i.e. sequents) to keep track of the type of channels in processes. Typing contexts are finite maps from channels to types written as $x_1 : A_1, \dots, x_n : A_n$ and ranged over by Γ and Δ. We write Γ, Δ for the union of Γ and Δ when they have

Table 3 Typing rules of LCC.

$$[\text{AX}]\ \frac{}{x \leftrightarrow y \vdash x : A, y : A^{\perp}} \qquad [\top]\ \frac{}{x \triangleright \{\} \vdash x : \top, \Gamma} \qquad [\perp]\ \frac{P \vdash \Gamma}{x().P \vdash x : \perp, \Gamma} \qquad [\mathbf{1}]\ \frac{}{x[] \vdash x : \mathbf{1}}$$

$$[\&]\ \frac{P \vdash y : A, \Gamma \qquad Q \vdash y : B, \Gamma}{x \triangleright (y)\{P, Q\} \vdash x : A \,\&\, B, \Gamma} \qquad [\oplus]\ \frac{P \vdash y : A_i, \Gamma}{x \triangleleft \mathsf{inj}_i[y].P \vdash x : A_1 \oplus A_2, \Gamma} \qquad [\parr]\ \frac{P \vdash y : A, z : B, \Gamma}{x(y, z).P \vdash x : A \parr B, \Gamma}$$

$$[\otimes]\ \frac{P \vdash y : A, \Gamma \qquad Q \vdash z : B, \Delta}{x[y, z](P \mid Q) \vdash x : A \otimes B, \Gamma, \Delta} \qquad [\exists]\ \frac{P \vdash y : B\{A/X\}, \Gamma}{x[A, y].P \vdash x : \exists X.B, \Gamma} \qquad [\forall]\ \frac{P \vdash y : A, \Gamma}{x(X, y).P \vdash x : \forall X.A, \Gamma}\ X \notin \mathsf{fv}(\Gamma)$$

$$[!]\ \frac{P \vdash y : A, ?\Gamma}{!x(y).P \vdash x : !A, ?\Gamma} \qquad [?]\ \frac{P \vdash y : A, \Gamma}{?x[y].P \vdash x : ?A, \Gamma} \qquad [\text{WEAKEN}]\ \frac{P \vdash \Gamma}{?x[].P \vdash x : ?A, \Gamma} \qquad [\text{CONTRACT}]\ \frac{P \vdash y : ?A, z : ?A, \Gamma}{?x[y, z].P \vdash x : ?A, \Gamma}$$

$$[\text{CUT}]\ \frac{P \vdash x : A, \Gamma \qquad Q \vdash x : A^{\perp}, \Delta}{(x : A)(P \mid Q) \vdash \Gamma, \Delta}$$

disjoint domains. We write $?\Gamma$ for some context $\Gamma = x_1 : ?A_1, \ldots, x_n : ?A_n$ where all the types in its range are prefixed by the modality ?. We call these types *unrestricted* because they are used to denote shared channels that can be (explicitly) discarded and duplicated.

Typing judgments have the form $P \vdash \Gamma$ meaning that the process P is well typed in the context Γ. Equivalently, the judgment indicates that the sequent $\vdash \Gamma$ is derivable and P is a proof term corresponding to the derivation for $\vdash \Gamma$. The typing rules are shown in Table 3. They are in one-to-one correspondence with – and have exactly the same structure of – the proof rules of classical linear logic. The reader may refer to the standard literature on propositions as sessions [5, 11, 21] for the interpretation of the rules. The only relevant difference with our typing rules is that the premises mentioning the continuation channel z actually refer to the same channel x on which the process in the conclusion is acting. The side condition $X \notin \mathsf{fv}(\Gamma)$ in the rule $[\exists]$ checks that the type variable X does not occur free in the types of Γ and therefore can be generalised.

Example 2.3. Looking at Example 2.3 we notice that the server does not make any assumption on the type of the values it receives and sends. Therefore, we can define a polymorphic version of the echo server that works for every message type and not just for the booleans. The polymorphic version of the echo server is defined below:

$$\mathit{Server}(x) \stackrel{\mathsf{def}}{=} !x(y).y(X, y').y'(u, y'').y''[v, y'''](u \leftrightarrow v \mid y'''[])$$

The only difference with respect to the server in Example 2.3 is that now the process receives the type X of the messages to be processed and then continues as before. We establish that $\mathit{Server}(x)$ is well typed with the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u \leftrightarrow v \vdash u : X^{\perp}, v : X}\,[\textsc{ax}] \qquad \dfrac{}{y'''[] \vdash y''' : \mathbf{1}}\,[\mathbf{1}]}{y''[v, y'''](u \leftrightarrow v \mid y'''[]) \vdash u : X^{\perp}, y'' : X \otimes \mathbf{1}}\,[\otimes]}{y'(u, y'').y''[v, y'''](u \leftrightarrow v \mid y'''[]) \vdash y' : X^{\perp} \parr (X \otimes \mathbf{1})}\,[\parr]}{y(X, y').y'(u, y'').y''[v, y'''](u \leftrightarrow v \mid y'''[]) \vdash y : \forall X.X^{\perp} \parr (X \otimes \mathbf{1})}\,[\forall]}{\mathit{Server}(x) \vdash x : !(\forall X.X^{\perp} \parr (X \otimes \mathbf{1}))}\,[!]$$

where the side condition of the rule [∀] is trivially satisfied since the typing context does not contain bindings other than the one for y. ⌟

2.5 Properties of Well-Typed Processes

The linearity challenge [1] aims at formalising two essential properties of well-typed processes: (1) typing is preserved by reductions; (2) the peer endpoints of the same session are always used in complementary ways. This latter property is called *well formedness* in the challenge. In this work we also consider *deadlock freedom*, which is more general than well formedness and holds for LCC since its type system is based on linear logic. We now formulate these properties using the notation developed so far.

Concerning the preservation of typing, this corresponds to the usual subject reduction result, which is expressed thus:

Theorem 2.1 (subject reduction)**.** *If* $P \rightsquigarrow Q$ *then* $P \vdash \Gamma$ *implies* $Q \vdash \Gamma$*.*

In order to formulate deadlock freedom, we first need to introduce some terminology for referring to processes that are unable to make any progress. A simple example of deadlock is the process $(x)(x[] \mid x[])$. This process is unable to reduce (because a process $x[]$ is meant to interact with a process of the form $x().P$) and, more generally, it is unable to interact regardless of the context in which it is placed because the sub-processes it contains are blocked on the channel x that is restricted by the cut. In general, the property of being a deadlock is not simply the inability to reduce: there are irreducible processes that are not a deadlock because they would be able to reduce if put into a suitable context. For example, $x().P$ does not reduce, and yet it is not a deadlock because it would be able to make progress when composed in parallel with $x[]$. Even a process like $(x)(y().P \mid z().Q)$ where $x \neq y, z$ cannot be considered a deadlock, because the prefixes $y()$ and $z()$ could be exposed using [s-wait] and possibly [s-comm]. Let us make all this more precise.

We say that P is a *thread* if it is anything but a cut. In other words, a thread is either a link or a process that starts with an input/output action of some sort. Note that a thread may contain cuts, but these cuts must be guarded underneath the topmost action prefix of the thread. We say that P is *observable* if $P \sqsupseteq Q$ for some thread Q. An observable process is a process that exposes an action on a free channel and therefore can interact through that channel, if put into some appropriate context. We say that P is *reducible* if $P \rightsquigarrow Q$ for some Q. A reducible process may perform a reduction step. We say that a process is *alive* if it is either observable or reducible and that it is a *deadlock* if it is not alive.

Well-typed LCC processes are deadlock free:

Theorem 2.2 (deadlock freedom)**.** *If $P \vdash \Gamma$ then P is alive.*

We now shift the attention to *well formedness*. In the linearity challenge [1] this property ensures that, whenever two processes composed in parallel begin with actions concerning the very same session, then such actions complement each other, in the sense that they describe opposite forms of interaction. To define well formedness in our setting, we introduce *reduction contexts* as partial processes with a single unguarded hole $[\,]$, thus:

$$\mathcal{C} ::= [\,] \mid (x : A)(\mathcal{C} \mid P) \mid (x : A)(P \mid \mathcal{C})$$

As usual, we write $\mathcal{C}[P]$ for the process obtained by replacing the hole in $\mathcal{C}$ with P, noting that such replacement may capture channels that are bound in $\mathcal{C}$ and occur free in P. Now we observe that if Q_1 and Q_2 both act on the same channel x in complementary ways, then their parallel composition $(x)(Q_1 \mid Q_2)$ is reducible according to one of the principal cut reductions described in Table 2. Therefore, an alternative (and more general) way of formulating well formedness is simply this: we say that P is *well formed* if $P \sqsupseteq \mathcal{C}[Q]$ implies that Q is alive. In the particular case when $Q = (x)(Q_1 \mid Q_2)$ and both Q_1 and Q_2 start with an action on x, then Q is not observable (because actions on x cannot be pulled out of the cut that binds x) and therefore it must be reducible by Theorem 2.2.

Well-typed processes are well formed:

Theorem 2.3 (type safety)**.** *If $P \vdash \Gamma$ then P is well formed.*

Note that the properties expressed in Theorems 2.2 and 2.3 are invariant under reductions thanks to Theorem 2.1.

3 Agda Formalisation

In this section we describe the formalisation of LCC in Agda. Each of the following sub-sections matches one of the modules of the formalisation. We present in detail only some key parts of the code including the representation of types and processes, the definition of the operational semantics and the statement of the main results. The complete source code is available in LCC's public repository [20].

3.1 Type Representation

The representation of types is standard. We start by defining an indexed data type PreType n to represents (LCC) types in the scope of n quantifiers and we use elements of Fin n as de Bruijn indices for the quantified type variables. In this way, we make sure that pre-types are well scoped.

```
data PreType : ℕ → Set where
  ⊤ 𝟎 ⊥ 𝟏              : ∀{n} → PreType n
  var rav              : ∀{n} → Fin n → PreType n
  _&_ _⊕_ _⅋_ _⊗_      : ∀{n} → PreType n → PreType n → PreType n
  ‘∀ ‘∃                : ∀{n} → PreType (suc n) → PreType n
  ‘! ‘?                : ∀{n} → PreType n → PreType n
```

Note the constructors var and rav, which respectively represent type variables and their dual, and the quantifiers ‘∀ and ‘∃ which increase the number of quantifiers in the scoped pre-type.

The dual of a pre-type is computed by the following function:

```
dual : ∀{n} → PreType n → PreType n
dual ⊤         = 𝟎
dual 𝟎         = ⊤
dual ⊥         = 𝟏
dual 𝟏         = ⊥
dual (var x)   = rav x
dual (rav x)   = var x
dual (A & B)   = dual A ⊕ dual B
dual (A ⊕ B)   = dual A & dual B
dual (A ⅋ B)   = dual A ⊗ dual B
dual (A ⊗ B)   = dual A ⅋ dual B
dual (‘∀ A)    = ‘∃ (dual A)
dual (‘∃ A)    = ‘∀ (dual A)
dual (‘! A) = ‘? (dual A)
dual (‘? A) = ‘! (dual A)
```

It is straightforward to prove that duality is an involution.

```
dual-inv : ∀{n} {A : PreType n} → dual (dual A) ≡ A
```

This property is important in the rest of the formalisation so we define an implicit rewriting rule that Agda can autonomously apply whenever possible. This is achieved by means of the following pragma directive.[1]

```
{-# REWRITE dual-inv #-}
```

Next we define the function subst that simultaneously substitutes the type variables of a pre-type with other pre-types. In practice we will always substitute one variable at a time, but it is technically easier to define subst so that it accepts a function substituting *all* variables of a pre-type, possibly with themselves. The definition

[1]The directive is effective provided that the option `--rewriting` is enabled, either globally when invoking Agda or within an `OPTIONS` pragma directive in the module's source code.

of subst relies on some auxiliary functions for *renaming* type variables and *lifting* substitutions across quantifiers. These functions are straightforward adaptations of those described by Kokke et al. [25].

```
subst : ∀{m n} → (Fin m → PreType n) → PreType m → PreType n
```

Among all substitutions, we will use the one that substitutes the 0-indexed type variable with a pre-type. It is convenient to introduce this substitution once and for all, which we do here.

```
[_/] : ∀{n} → PreType n → Fin (suc n) → PreType n
[ A /] zero    = A
[ A /] (suc k) = var k
```

Duality and substitutions are meant to commute.

```
dual-subst : ∀{m n} {σ : Fin m → PreType n} {A : PreType m} →
             subst σ (dual A) ≡ dual (subst σ A)
```

It is worth looking at one case in the proof of dual-subst, namely when the type is a dualised type variable:

```
dual-subst {_} {_} {σ} {rav x} = refl
```

Here we are supposed to prove subst σ (dual (rav x)) $\equiv$ dual (subst σ (rav x)) which is definitionally equal to $\sigma\ x \equiv$ dual (dual ($\sigma\ x$)). We could easily prove this equivalence by invoking dual-inv, but thanks to the rewriting rule that we have added earlier a use of refl suffices. In this case the saved effort is negligible, but in later results, where it is necessary to use dual-inv for rewriting part of the *index* of some type families, having an implicit rewriting rule allows us to avoid writing some quite obscure Agda code.

Just like dual-inv, dual-subst too is key in the formalisation that follows. Therefore, we add it to the set of implicit rewriting rules used by Agda so that we do not have to think about this property again.

```
{-# REWRITE dual-subst #-}
```

We call Type closed pre-types, those having no free type variables. From now on, we will seldom use pre-types again.

```
Type : Set
Type = PreType zero
```

3.2 Context Representation

We are going to adopt a nameless representation of channels. Hence, typing contexts are represented as lists of types, where the (polymorphic) type List and its constructors

[] and _::_ are defined in the module Data.List of Agda's standard library. We will keep using Γ, Δ and Θ to range over typing contexts, even though in the Agda formalisation they are lists and not finite maps as in Section 2.4.

```
Context : Set
Context = List Type
```

The most important operation concerning typing contexts is *splitting*. The splitting of Γ into Δ and Θ, which we denote by $\Gamma \simeq \Delta + \Theta$, represents the fact that Γ contains all the types contained in Δ and Θ, preserving both their overall multiplicity and also their relative order within Δ and Θ. A *proof* of $\Gamma \simeq \Delta + \Theta$ shows how the types in Γ are distributed in Δ and Θ from left to right.

```
data _≃_+_ : Context → Context → Context → Set where
  •  : [] ≃ [] + []
  <_ : ∀{A Γ Δ Θ} → Γ ≃ Δ + Θ → A :: Γ ≃ A :: Δ + Θ
  >_ : ∀{A Γ Δ Θ} → Γ ≃ Δ + Θ → A :: Γ ≃ Δ + A :: Θ
```

When splitting a context Γ into $\Delta + \Theta$, for each type in Γ we use one of the prefix operators < and > to indicate whether the type is meant to be placed in Δ or in Θ. Once we reach the end of the typing context, we use the constructor • to build the trivial splitting of the empty context into two empty partitions. For example, below is a proof of the splitting $[A, B, C, D] \simeq [B] + [A, C, D]$.

```
splitting-example₁ : (A :: B :: C :: D :: []) ≃ [ B ] + (A :: C :: D :: [])
splitting-example₁ = > < > > •
```

It is easy to see that splitting is commutative and that the empty context/list is both a left and right unit of splitting.

```
+-comm : ∀{Γ Δ Θ} → Γ ≃ Δ + Θ → Γ ≃ Θ + Δ
≫      : ∀{Γ} → Γ ≃ [] + Γ
≪      : ∀{Γ} → Γ ≃ Γ + []
```

Context splitting is also associative. If we write $\Delta + \Theta$ for some Γ such that $\Gamma \simeq \Delta + \Theta$, then we can prove that $\Gamma_1 + (\Gamma_2 + \Gamma_3) = (\Gamma_1 + \Gamma_2) + \Gamma_3$.

```
+-assoc-r : ∀{Γ Δ Θ Δ′ Θ′} → Γ ≃ Δ + Θ → Θ ≃ Δ′ + Θ′ →
            ∃[ Γ′ ] Γ′ ≃ Δ + Δ′ × Γ ≃ Γ′ + Θ′
+-assoc-l : ∀{Γ Δ Θ Δ′ Θ′} → Γ ≃ Δ + Θ → Δ ≃ Δ′ + Θ′ →
            ∃[ Γ′ ] Γ′ ≃ Θ′ + Θ × Γ ≃ Δ′ + Γ′
```

When proving a splitting $\Gamma \simeq [A] + \Theta$ where the left partition is a singleton $[A]$, it may be convenient to use ≫ as a shortcut for a sequence of applications of > once the A type has been reached in Γ. For instance, splitting-example₁ can be written

Notation	Definition	Meaning
Pred A ℓ	$A \to$ Set ℓ	predicate over A
∀[P]	$\forall\{x\} \to P\ x$	implicit universality
$P \Rightarrow Q$	$\lambda x \to P\ x \to Q\ x$	implication
$P \cup Q$	$\lambda x \to P\ x \uplus Q\ x$	disjunction
$P \cap Q$	$\lambda x \to P\ x \times Q\ x$	conjunction
$f \vdash P$	$\lambda x \to P\ (f\ x)$	update
U	$\lambda x \to$ Data.Unit.⊤	universal set
⋂[X : A] P	$\lambda x \to (X : A) \to P\ X\ x$	infinitary conjunction

Table 4 Useful definitions in Agda's Relation.Unary module.

equivalently and in a more compact way as shown below. More usages of ≫ will be provided in Section 3.9.

```
splitting-example₂ : (A :: B :: C :: D :: []) ≃ [ B ] + (A :: C :: D :: [])
splitting-example₂ = > < ≫
```

From now on we will make extensive use of predicates over contexts. For this reason, it is worth recalling in Table 4 a number of definitions from the module Relation.Unary of Agda's standard library. We begin using these definitions for building a few abstractions inspired to separation logic [26] that allow us to hide context splittings, at least in some cases. Following Rouvoet et al. [18], we define the *separating conjunction* $P * Q$ of two predicates P and Q over contexts:

```
data _*_ (P Q : Pred Context _) (Γ : Context) : Set where
  _⟨_⟩_ : ∀{Δ Θ} → P Δ → Γ ≃ Δ + Θ → Q Θ → (P * Q) Γ
```

If P and Q are predicates over contexts, the predicate $P*Q$ holds for those contexts Γ that can be split into Δ and Θ so that P holds for Δ and Q holds for Θ. The constructor _⟨_⟩_ has three explicit arguments witnessing the splitting $\Gamma \simeq \Delta + \Theta$ along with proofs of $P\ \Delta$ and $Q\ \Theta$. The use of metavariables P and Q for denoting predicates over contexts is appropriate: as we will see shortly, in our formalisation processes are indeed an example of predicate over typing contexts.

Along with $*$ we define the *separating implication* (also known as "magic wand")

```
_-*_ : Pred Context _ → Pred Context _ → Context → Set
(P -* Q) Δ = ∀{Θ Γ} → Γ ≃ Δ + Θ → P Θ → Q Γ
```

and prove that -* can be used to curry $*$:

```
curry* : ∀{P Q R} → ∀[ P * Q ⇒ R ] → ∀[ P ⇒ Q -* R ]
curry* F px σ qx = F (px ⟨ σ ⟩ qx)
```

To conclude the implementation of typing contexts, we define a predicate Un that holds for *unrestricted* contexts, those solely made of types of the form $?A$. We need this predicate in the definition of a server, which must comply with the typing rule [!].

```
data Un : Context → Set where
  un-[] : Un []
  un-:: : ∀{A} → ∀[ Un ⇒ (‘? A ::_) ⊢ Un ]
```

The empty context is trivially unrestricted. A non-empty context is unrestricted if its head has the form $?A$ for some A and its tail is unrestricted as well. It is easy to prove that Γ is unrestricted if so are Δ and Θ when $\Gamma \simeq \Delta + \Theta$:

```
∗-un : ∀[ Un ∗ Un ⇒ Un ]
```

3.3 Context Permutations

According to our nameless representation of channels, the *position* of a type in a typing context Γ determines the location of its binder in the structure of a process. When the binding structure of a process changes, e.g. because a structural pre-congruence rule is applied, or when a channel substitution occurs, cf. the right-hand side of the [AX] reduction in Table 2, Γ must be suitably rearranged to agree with the updated binding structure. Such rearrangement is in fact a *permutation* of the elements of Γ.

We define typing context permutations inductively, as a binary relation _↭_:

```
data _↭_ : Context → Context → Set where
  refl  : ∀{Γ} → Γ ↭ Γ
  swap  : ∀{A B Γ} → (A :: B :: Γ) ↭ (B :: A :: Γ)
  prep  : ∀{A Γ Δ} → Γ ↭ Δ → (A :: Γ) ↭ (A :: Δ)
  trans : ∀{Γ Δ Θ} → Γ ↭ Δ → Δ ↭ Θ → Γ ↭ Θ
```

Each constructor of _↭_ represents a particular kind of permutation: refl for the trivial permutation that does not change anything; swap for the permutation that swaps the first two elements of a typing context; prep for the permutation applied to the tail of a typing context; trans for the sequential composition of permutations.

The definition of the data type _↭_ is nearly the same found in the module Data.List.Relation.Binary.Permutation.Propositional of Agda's standard library. We have preferred defining our own notion of permutation for simplicity and convenience: the swap constructor does not need a sub-permutation for the tail of the typing context, which can always be performed, if needed, combining swap with prep and trans. Also, and more importantly, our data type _↭_ is monomorphic (it does not need to relate arbitrary lists) and the arguments A and B of swap and prep are implicit, which streamlines the usage of these constructors in the rest of the code.

It is easy to see that _↭_ is an equivalence relation. In the following we also use another property of permutations related to context splitting and list concatenation _++_: if $\Gamma \simeq \Delta + \Theta$, then Γ is a permutation of the concatenation of Δ and Θ.

```
↭concat : ∀{Γ Γ₁ Γ₂} → Γ ≃ Γ₁ + Γ₂ → (Γ₁ ++ Γ₂) ↭ Γ
```

3.4 Channel and Process Representation

We adopt an *intrinsically-typed* representation of processes with *nameless* channels. The intrinsically-typed representation makes sure that only well-typed processes can be constructed. This choice increases the effort in the definition of the datatypes for representing processes and their operational semantics, but pays off in the rest of the formalisation for at least three reasons:

- we need not give explicit names to channels, thus we avoid all the technicalities and pitfalls that a named representation entails;
- we conflate processes and typing rules in the same datatype, thus reducing the overall number of datatypes defined in the formalisation;
- the typing preservation results are embedded in the very definition of the operational semantics of processes and do not require separate proofs (Sections 3.5 and 3.6).

Channels are not given any name. Instead, they are represented as terms witnessing that their type is present in the typing context. This is known as *co-de Bruijn syntax* [27], whereby the typing context associated with a term (a process, a channel) only contains the types of the channels that actually occur within the term. For this reason, typing contexts are split eagerly, according to the structure of processes, to make sure that channels are appropriately (and above all linearly) distributed among sub-processes, so that each channel is used exactly once. Concretely, a channel of type A is a predicate that holds for the singleton context [A]:

```
data Ch (A : Type) : Context → Set where
  ch : Ch A [ A ]
```

A process that is well typed in a typing context Γ is a predicate that holds for Γ. Here is the datatype Proc for representing processes:

```
data Proc : Context → Set where
  link    : ∀{A} → ∀[ Ch A ∗ Ch (dual A) ⇒ Proc ]
  fail    : ∀[ Ch ⊤ ∗ U ⇒ Proc ]
  wait    : ∀[ Ch ⊥ ∗ Proc ⇒ Proc ]
  close   : ∀[ Ch 𝟏 ⇒ Proc ]
  case    : ∀{A B} →
            ∀[ Ch (A & B) ∗ ((A ::_) ⊢ Proc ∩ (B ::_) ⊢ Proc) ⇒ Proc ]
  select  : ∀{A B} →
            ∀[ Ch (A ⊕ B) ∗ ((A ::_) ⊢ Proc ∪ (B ::_) ⊢ Proc) ⇒ Proc ]
  join    : ∀{A B} → ∀[ Ch (A ⅋ B) ∗ ((A ::_) ⊢ (B ::_) ⊢ Proc) ⇒ Proc ]
  fork    : ∀{A B} →
            ∀[ Ch (A ⊗ B) ∗ ((A ::_) ⊢ Proc) ∗ ((B ::_) ⊢ Proc) ⇒ Proc ]
  all     : ∀{A} →
            ∀[ Ch ('∀ A) ∗ ⋂[ X : Type ] ((subst [ X /] A ::_) ⊢ Proc) ⇒ Proc ]
  ex      : ∀{A B} → ∀[ Ch ('∃ A) ∗ ((subst [ B /] A ::_) ⊢ Proc) ⇒ Proc ]
  server  : ∀{A} → ∀[ Ch ('! A) ∗ (Un ∩ ((A ::_) ⊢ Proc)) ⇒ Proc ]
  client  : ∀{A} → ∀[ Ch ('? A) ∗ ((A ::_) ⊢ Proc) ⇒ Proc ]
```

```
weaken   : ∀{A} → ∀[ Ch (‘? A) ∗ Proc ⇒ Proc ]
contract : ∀{A} → ∀[ Ch (‘? A) ∗ ((‘? A ::_) ⊢ (‘? A ::_) ⊢ Proc) ⇒ Proc ]
cut      : ∀{A} → ∀[ ((A ::_) ⊢ Proc) ∗ ((dual A ::_) ⊢ Proc) ⇒ Proc ]
```

The constructor link builds a link $x \leftrightarrow y$. This process is well typed in a context of the form $x : A, y : A^\perp$, namely a context satisfying the predicate Ch A ∗ Ch (dual A) which we see on the left-hand side of ⇒.

The constructor cut builds a cut $(x : A)(P \mid Q)$. This process is well typed in a context Γ if $\Gamma \simeq \Delta + \Theta$ so that P and Q are well typed in the contexts $x : A, \Delta$ and $x : A^\perp, \Theta$, which are obtained from Δ and Θ by adding the bindings $x : A$ and $x : A^\perp$, respectively. Since x is the most recently introduced channel, the types A and $A^\perp$ are added *in front* of Δ and Θ, which we do by means of the functions (A ::_) and (dual A ::_). These are partial applications of the constructor _::_ for lists to which we have supplied the left operand.

All the remaining constructors basically follow the same pattern: they possibly quantify over some types A and B and then (implicitly) over a typing context Γ through the function ∀[_] applied to a predicate of the form P ⇒ Proc. The predicate states how to build a process that is well typed in Γ, provided that Γ satisfies P. In general, P is a (separating) conjunction of sub-predicates corresponding to the channel on which the process is acting and to the premises of its typing rule.

For example, the constructor fail, which builds a process $x \triangleright \{\}$, requires the context Γ to satisfy the predicate Ch ⊤ ∗ U, meaning that Γ must contain an entry ⊤ (U is the universal predicate that holds for every context, cf. Table 4). That form of Γ matches the typing context in the conclusion of the rule [⊤].

The constructor wait, which builds a process $x().P$, requires Γ to (separately) satisfy Ch ⊥, that is the channel x on which the process is operating, as well as Proc, that is the continuation process P, which must be well typed in the remaining typing context.

The constructor close, which builds a process $x[]$, requires the typing context to be the singleton list [**1**].

Let us move on to the forms that produce continuation channels. As an example, the constructor case builds a process $x \triangleright (y)\{P, Q\}$, where both P and Q use the continuation channel y. In this case Γ must satisfy the predicate

$$\mathsf{Ch}\ (A \mathbin{\&} B) * ((A ::\_) \vdash \mathsf{Proc} \cap (B ::\_) \vdash \mathsf{Proc})$$

which looks intimidating at first but makes perfect sense once we recall the typing rule [&] and the definitions of ∩ and ⊢ in Table 4. Remember that we are trying to establish whether $x \triangleright (y)\{P, Q\}$ is well typed in Γ. The predicate Ch (A & B) expresses the requirement that the type of x must be of the form $A \mathbin{\&} B$ and should be found in Γ. In other words, $\Gamma \simeq [A \mathbin{\&} B] + \Delta$ for some Δ. Now P and Q must be well typed in the context Δ augmented with the association $y : A$ and $y : B$, respectively, whence the use of ∩ to verify a (non-separating) conjunction of the predicates (A ::_) ⊢ Proc and (B ::_) ⊢ Proc. The two new contexts are obtained by *adding* either A or B to Δ, which we perform using ⊢. Crucially, the types A and B are *prepended* to Δ, which is consistent with the fact that the continuation y has been freshly introduced and the channel x has been consumed. Notice how easy it is to *prepend* either A or B to Δ instead of *changing* the type of x in Γ from $A \,\&\, B$ to either A or B while preserving the position of the type, as we would have to do in a "true" session type system without explicit continuation channels.

The interpretation of the remaining constructors is analogous, so we only comment all, which builds a process $x(X, y).P$. This constructor models the continuation P using higher-order abstract syntax (HOAS): proving that a context Γ satisfies the predicate

```
⋂[ X : Type ] ((subst [ X /] A ::_) ⊢ Proc
```

means providing a function that, for every type X, produces a witness for the predicate

```
((subst [ X /] A ::_) ⊢ Proc
```

applied to Γ. Note that the side condition $X \notin \mathsf{fv}(\Gamma)$ of $[\forall]$ is trivially enforced *by definition*: A has type PreType 1, that is a pre-type with *at most one* free type variable X, whereas Γ is a typing context, that is a list of Type = PreType 0 without free type variables. Therefore, X cannot occur in Γ.

We conclude this module proving that permutations preserve process typing. Since list permutations basically correspond to channel renaming, we can read this property as the fact that typing is preserved by (bijective) name substitutions.

```
⇜⇝proc : ∀{Γ Δ} → Γ ⇜⇝ Δ → Proc Γ → Proc Δ
```

3.5 Structural Pre-Congruence

We formalise structural precongruence as a binary relation between processes that are well typed in the *same* typing context. This entails that structural precongruence preserves typing by definition.

```
data _⊒_ {Γ} : Proc Γ → Proc Γ → Set where
```

The datatype for $\sqsupseteq$ has one constructor for each structural pre-congruence rule in Table 2. Since many aspects recur repeatedly, we illustrate the implementation of just a few representative rules starting from [s-comm].

```
s-comm :
  ∀{A Γ₁ Γ₂ P Q} (p : Γ ≃ Γ₁ + Γ₂) →
  cut {A} (P ⟨ p ⟩ Q) ⊒ cut (Q ⟨ +-comm p ⟩ P)
```

The constructor s-comm models the commutativity property of parallel composition. We use +-comm to compute the proof of the splitting $\Gamma \simeq \Gamma_2 + \Gamma_1$ from p. Notice that s-comm makes key use of the implicit rewriting rule dual-inv described in Section 3.1. Indeed P and Q have type Proc $(A :: \Gamma_1)$ and Proc (dual $A :: \Gamma_2$), respectively, but the cut on the r.h.s. of $\sqsupseteq$ expects P to have type dual (dual A).

Thanks to dual-inv, Agda considers these types equivalent without requiring intricate substitutions in the index of Proc.

The constructor s-wait models the [S-WAIT] rule:

```
s-wait :
  ∀{Γ₁ Γ₂ Δ A P Q} (p : Γ ≃ Γ₁ + Γ₂) (q : Γ₁ ≃ [ ⊥ ] + Δ) →
  let _ , p′ , q′ = +-assoc-l p q in
  cut {A} (wait (ch ⟨ > q ⟩ P) ⟨ p ⟩ Q) ⊒
  wait (ch ⟨ q′ ⟩ cut (P ⟨ p′ ⟩ Q))
```

There are two non-trivial aspects worth commenting. The first one concerns the proof $> q$ used within wait. To understand the meaning of this proof, we must recall three key elements:

1. (wait (ch ⟨ $> q$ ⟩ P) is a direct sub-process of the cut, and therefore it is meant to be well typed in the context $A :: \Gamma_1$.
2. Being a wait process, such context must contain a $\bot$ type as per the typing rule $[\bot]$. That is $A :: \Gamma_1 \simeq [\bot] + A :: \Delta$ for some Δ.
3. The [S-WAIT] rule is applicable only provided that the channel restricted by the cut (say x, of type A) is different from the channel consumed by the wait process (say y, of type $\bot$). We enforce the side condition $x \neq y$ of [S-WAIT] (which we left implicit in Table 2) imposing that the type A in front of $A :: \Gamma_1$ goes to the right partition of the splitting $[\bot] + A :: \Delta$ through the use of $>$.

The other aspect that is worth commenting concerns the rearrangement of the splittings in the process after the application of structural precongruence. Overall, p and q prove the splittings $([\bot] + \Delta) + \Gamma_2$, but the precongruence rule requires this splitting to be rearranged as $[\bot] + (\Delta + \Gamma_2)$. That is, we need to apply the left-to-right associativity property of context splitting which we called +-assoc-l in Section 3.2. The nested let-in allows us to pattern match on the result of the application +-assoc-l p q and to extract the new proofs p' and q' for the rearranged splittings.

The constructors s-select-l and s-select-r model [S-SELECT] when the selected tag is respectively inj_1 and inj_2. For example, for s-select-l we have:

```
s-select-l :
  ∀{Γ₁ Γ₂ Δ A B C P Q} (p : Γ ≃ Γ₁ + Γ₂) (q : Γ₁ ≃ [ B ⊕ C ] + Δ) →
  let _ , p′ , q′ = +-assoc-l p q in
  cut {A} (select (ch ⟨ > q ⟩ inj₁ P) ⟨ p ⟩ Q) ⊒
  select (ch ⟨ q′ ⟩ inj₁ (cut (↭proc swap P ⟨ < p′ ⟩ Q)))
```

Here the process (select (ch ⟨ $> q$ ⟩) inj_1 P), that is $y \triangleleft \mathsf{inj}_1[z].P$, is found under a cut for $x : A$ and is using some channel $y : B \oplus C$ to select inj_1. The continuation process P uses a fresh continuation channel $z : B$. Therefore, P is required to be well typed in the context $B :: A :: \Delta$, where the type B of z comes *before* the type A of x since z is introduced later than x. After structural pre-congruence is applied, however, the type of the continuation channel z ends up behind that of the restricted channel x, because now z and x are introduced in the opposite order. Therefore, we need to rename the channels in P so that it is well typed in the context $A :: B :: \Delta$. Such renaming is achieved applying the function $\leftrightsquigarrow\mathsf{proc}$ to the swap permutation and to the process P.

We also discuss the modeling of the [S-FORK-L] rule, which is interesting because of its complex side conditions:

$$
\begin{aligned}
&\mathsf{s\text{-}fork\text{-}l} : \\
&\quad \forall\{\Gamma_1\ \Gamma_2\ \Delta\ \Delta_1\ \Delta_2\ A\ B\ C\ P\ Q\ R\} \\
&\quad (p : \Gamma \simeq \Gamma_1 + \Gamma_2)\ (q : \Gamma_1 \simeq [\ B \otimes C\ ] + \Delta)\ (r : \Delta \simeq \Delta_1 + \Delta_2) \to \\
&\quad \mathsf{let}\ \_\ ,\ p'\ ,\ q' = \mathsf{+\text{-}assoc\text{-}l}\ p\ q\ \mathsf{in} \\
&\quad \mathsf{let}\ \_\ ,\ p''\ ,\ r' = \mathsf{+\text{-}assoc\text{-}l}\ p'\ r\ \mathsf{in} \\
&\quad \mathsf{let}\ \_\ ,\ q''\ ,\ r'' = \mathsf{+\text{-}assoc\text{-}r}\ r'\ (\mathsf{+\text{-}comm}\ p'')\ \mathsf{in} \\
&\quad \mathsf{cut}\ \{A\}\ (\mathsf{fork}\ (\mathsf{ch}\ \langle > q \rangle\ (P\ \langle < r \rangle\ Q))\ \langle\ p\ \rangle\ R) \sqsupseteq \\
&\quad \mathsf{fork}\ (\mathsf{ch}\ \langle\ q'\ \rangle\ (\mathsf{cut}\ (\leftrightsquigarrow\mathsf{proc}\ \mathsf{swap}\ P\ \langle < q'' \rangle\ R)\ \langle\ r''\ \rangle\ Q))
\end{aligned}
$$

Recall from Table 2 that we allow using this rule on a process of the form $(x : A)(y[u, v](P \mid Q) \mid R)$ when $x \in \mathsf{fc}(P)$. We capture the condition $x \in \mathsf{fc}(P)$ by means of the splitting $< r$, implying that the type A of x ends up in the typing context for P and not in the one for Q. The symmetric rule [S-FORK-R] is modeled by another constructor s-fork-r, which is similar to s-fork-l except that $< r$ is replaced by $> r$.

Finally, in Section 2.3 we have colloquially defined $\sqsupseteq$ as a "pre-congruence", implying that it is a reflexive, transitive relation preserved by some forms of the calculus. In the formalisation we have to be precise and we introduce specific rules:

$$
\begin{aligned}
&\mathsf{s\text{-}refl} &&: \forall\{P\} \to P \sqsupseteq P \\
&\mathsf{s\text{-}tran} &&: \forall\{P\ Q\ R\} \to P \sqsupseteq Q \to Q \sqsupseteq R \to P \sqsupseteq R \\
&\mathsf{s\text{-}cong} &&: \forall\{\Gamma_1\ \Gamma_2\ A\ P\ Q\ P'\ Q'\}\ (p : \Gamma \simeq \Gamma_1 + \Gamma_2) \to \\
& &&\quad P \sqsupseteq Q \to P' \sqsupseteq Q' \to \mathsf{cut}\ \{A\}\ (P\ \langle\ p\ \rangle\ P') \sqsupseteq \mathsf{cut}\ (Q\ \langle\ p\ \rangle\ Q')
\end{aligned}
$$

Note that we define a single congruence rule s-cong that allows us to apply $\sqsupseteq$ within cuts, but not underneath prefixes. This limited form of pre-congruence turns out to be sufficient for the development that follows.

We concede that the implementation of the pre-congruence rules (including those not discussed here) can be difficult to decipher. In part, this is due to the fact that splitting proofs are manifest and no longer hidden by separating conjunctions (as in Section 3.4) because we need them to enforce the side conditions of the rules in Table 2. We should also bear in mind that, using an intrinsically-typed representation of processes, we have already taken care of the proof that structural pre-congruence preserves typing.

3.6 Reduction

Just like structural pre-congruence, reduction is formalised as a binary relation between processes that are well typed in the same typing context. Thus, the definition of reduction embeds subject reduction (Theorem 2.1).

```
data _⇝_ {Γ} : Proc Γ → Proc Γ → Set where
```

There is a constructor for each of the reduction rules in Table 2. Let us comment a few representative cases.

The constructor r-link models the reduction $(x : A)(x \leftrightarrow y \mid P) \rightsquigarrow P\{y/x\}$ called [R-LINK] in Table 2:

```
r-link : ∀{Δ A P} (p : Γ ≃ [ dual A ] + Δ) →
         cut {A} (link (ch ⟨ < > • ⟩ ch) ⟨ p ⟩ P) ⇝ ↭proc (↭concat p) P
```

The splitting p indicates that y (of type $A^\perp$) occurs in the left sub-process of the cut (that is the link $x \leftrightarrow y$) and the splitting < > • to which link is applied is structured consistently with the syntax of the link being reduced, which is oriented so that the restricted channel x is on the left. The process P has type Proc (dual A :: Δ) and turns into $P\{y/x\}$ after the reduction. The type dual A of x in P is the first in the typing context, indicating that it is the newest channel that P is aware of; however, after the reduction, x is replaced by y which is found *somewhere* within Γ. The exact location of y in Γ is encoded in the splitting p, thus we "rename" x into y within P using the permutation ↭concat p.

The constructor r-close models [R-CLOSE]:

```
r-close : ∀{P} (p₀ q₀ : Γ ≃ [] + Γ) →
          cut (close ch ⟨ p₀ ⟩ wait (ch ⟨ < q₀ ⟩ P)) ⇝ P
```

While the process constructor close implicitly refers to the only free channel occurring in a process of the form $x[]$, the constructor wait uses a splitting proof of the form $< q_0$ to make sure that the referenced channel is also the restricted one, and therefore matches the one of the close process.

Note that r-close (and several other reduction constructors) quantifies over p_0 and q_0 which both prove the splitting $\Gamma \simeq [] + \Gamma$. Since the left partition is empty, these splittings must be equal and made of a sequence of $>$ applications followed by •. In general, Agda will not be able to "see" that they are definitionally equal, hence it is easier to quantify them separately so that we do not have to prove their equality whenever we wish to apply this reduction.

The constructor r-select-l models [R-SELECT] when the selected tag is inj_1:

```
r-select-l : ∀{Γ₁ Γ₂ A B P Q R}
             (p : Γ ≃ Γ₁ + Γ₂) (p₀ : Γ₁ ≃ [] + Γ₁) (q₀ : Γ₂ ≃ [] + Γ₂) →
             cut {A ⊕ B} (select (ch ⟨ < p₀ ⟩ inj₁ P) ⟨ p ⟩
                          case (ch ⟨ < q₀ ⟩ (Q , R))) ⇝ cut (P ⟨ p ⟩ Q)
```

There is not much to note here except again for the multiple quantifications over the trivial splittings $\Gamma_i \simeq [] + \Gamma_i$ and the use of $<$ to make sure that the channel referred to by select and case is indeed the one restricted by the cut.

The remaining constructors that describe the base reductions follow a similar pattern, except for the implementation of [R-WEAKEN] and [R-CONTRACT] which require auxiliary functions to respectively weaken and contract the typing context of the resulting process as shown in Table 2. It is worth glancing at the implementation of [R-EXISTS] since it involves a non-trivial rewriting of types:

```
r-exists : ∀{A B Γ₁ Γ₂ P F}
           (p : Γ ≃ Γ₁ + Γ₂) (p₀ : Γ₁ ≃ [] + Γ₁) (q₀ : Γ₂ ≃ [] + Γ₂) →
           cut {‘∃ A} (ex {A} {B} (ch ⟨ < p₀ ⟩ P) ⟨ p ⟩ all (ch ⟨ < q₀ ⟩ F)) ⇝
           cut (P ⟨ p ⟩ F B)
```

Recalling the definitions of ex and all from Section 3.4, we note that P has type Proc (subst [B /] A :: Γ_1) and $F\ B$ is a process of type Proc (subst [B /] (dual A) :: Γ_2). In order for these two processes to be composable in a cut, it must be the case that subst [B /] (dual A) $\equiv$ dual (subst [B /] A), which was proved in Section 3.1 under the name dual-subst. Thanks to the implicit rewriting rule, we do not have to rewrite the index in the type of $F\ B$, which is silently accepted as is.

Reduction is closed under cuts and by structural pre-congruence as per [R-CUT] and [R-CONG]. The corresponding constructors that model these features are shown below:

```
r-cut  : ∀{Γ₁ Γ₂ A P Q R} (q : Γ ≃ Γ₁ + Γ₂) →
         P ⇝ Q → cut {A} (P ⟨ q ⟩ R) ⇝ cut (Q ⟨ q ⟩ R)
r-cong : ∀{P R Q} → P ⊒ R → R ⇝ Q → P ⇝ Q
```

3.7 Deadlock Freedom

As we have seen in Section 2, the deadlock freedom property and Theorem 2.2 rest on some notions and predicates about processes which must be formalised in Agda. First of all we need to define the notion of *thread*, that is any process other than a cut. It is convenient to provide a more fine-grained classification of threads, distinguishing between links and input/output actions and sometimes also on whether such actions operate on free or bound channels. We define predicates for each of these classes:

```
data Link          : ∀{Γ} → Proc Γ → Set
data Input         : ∀{Γ} → Proc Γ → Set
data Output        : ∀{Γ} → Proc Γ → Set
data Delayed       : ∀{Γ} → Proc Γ → Set
data Server        : ∀{Γ} → Proc Γ → Set
data DelayedServer : ∀{Γ} → Proc Γ → Set
```

The implementation of these predicates is not interesting since it is essentially isomorphic to the relevant fragments of the Proc datatype. The only aspect that is worth pointing out here is that in Input, Output and Server the channel being acted upon by the thread is the *first* in Γ, hence it is the *most recently introduced* channel, whereas in Delayed and DelayedServer the channel is not the first. This allows us to distinguish those threads that, in the context of a cut, operate on the channel bound by the cut or on a free channel. To clarify this aspect, let us look at the implementation of the constructor wait in Input and in Delayed. In the former predicate we have

```
wait : ∀{Γ Δ P} (p : Γ ≃ [] + Δ) → Input (wait (ch ⟨ < p ⟩ P))
```

where the use of the constructor < indicates that the thread operates on the most recent channel. In the latter predicate we have

```
wait : ∀{C Γ Δ P} (p : Γ ≃ [ ⊥ ] + Δ) → Delayed (wait (ch ⟨ >_ {C} p ⟩ P))
```

where the use of the constructor > indicates that the thread operates on a channel other than the most recent one. Note that here we have to specify the type C in front of Γ or else Agda is unable to automatically resolve some metavariables.

The predicate Thread is simply the disjoint union of all the previous ones.

```
data Thread {Γ} (P : Proc Γ) : Set where
  link    : Link P → Thread P
  delayed : Delayed P → Thread P
  output  : Output P → Thread P
  input   : Input P → Thread P
  server  : Server P → Thread P
  dserver : DelayedServer P → Thread P
```

Observability, reducibility and aliveness are defined in the expected way:

```
Observable : ∀{Γ} → Proc Γ → Set
Observable P = ∃[ Q ] P ⊒ Q × Thread Q

Reducible : ∀{Γ} → Proc Γ → Set
Reducible P = ∃[ Q ] P ⇝ Q

Alive : ∀{Γ} → Proc Γ → Set
Alive P = Observable P ⊎ Reducible P
```

In order to prove that every (well-typed) process is alive, it is convenient to define a "canonical" form for cuts, that is a form that matches at least one of the l.h.s of one of the rules for structural pre-congruence or reduction in Table 2. This is the notion where the fine-grained classification of threads introduced earlier comes into play.

```
data CanonicalCut {Γ} : Proc Γ → Set where
  cc-link    : ∀{Γ₁ Γ₂ A P Q} (p : Γ ≃ Γ₁ + Γ₂) →
               Link P → CanonicalCut (cut {A} (P ⟨ p ⟩ Q))
  cc-redex   : ∀{Γ₁ Γ₂ A P Q} (p : Γ ≃ Γ₁ + Γ₂) →
               Output P → (Input ∪ Server) Q →
               CanonicalCut (cut {A} (P ⟨ p ⟩ Q))
  cc-delayed : ∀{Γ₁ Γ₂ A P Q} (p : Γ ≃ Γ₁ + Γ₂) →
```

```
                    Delayed P → CanonicalCut (cut {A} (P ⟨ p ⟩ Q))
    cc-servers : ∀{Γ₁ Γ₂ A P Q} (p : Γ ≃ Γ₁ + Γ₂) →
                    DelayedServer P → Server Q →
                    CanonicalCut (cut {A} (P ⟨ p ⟩ Q))
```

A *canonical cut* $(x : A)(P \mid Q)$ has one of these forms:

- P is a link (cc-link), or
- P performs an output on x and Q performs an input on x (cc-redex), or
- P operates on a channel other than x and is not a server (cc-delayed), or
- both P and Q are servers and P operates on a channel other than x (cc-servers).

We have to distinguish servers from the other input operations because the structural pre-congruence rule [S-SERVER] can only be applied when the two sub-processes of a cut are both servers.

Every cut $(x : A)(P \mid Q)$ where both P and Q are threads can be rewritten into a canonical cut using structural pre-congruence:

```
canonical-cut : ∀{A Γ Γ₁ Γ₂ P Q} (p : Γ ≃ Γ₁ + Γ₂) →
                Thread P → Thread Q →
                ∃[ R ] CanonicalCut R × cut {A} (P ⟨ p ⟩ Q) ⊒ R
```

It is easy to prove that every canonical cut is alive, either reducing it or applying structural pre-congruence to rewrite it into a thread.

```
canonical-cut-alive : ∀{Γ} {C : Proc Γ} → CanonicalCut C → Alive C
```

Now deadlock freedom for P can be proved by induction on P.

```
deadlock-freedom : ∀{Γ} (P : Proc Γ) → Alive P
```

When P is a thread, then it is obviously observable and hence alive. When P is a cut $(x : A)(P \mid Q)$, deadlock-freedom is applied recursively to P and to Q, in turn. If either of these applications yields a reduction, then the whole cut is reducible and therefore alive. If both applications yield a thread, then we conclude that the cut is alive first rewriting it into a canonical cut with canonical-cut and then using canonical-cut-alive.

3.8 Type Safety

We have seen that type safety is a simple instance of deadlock freedom, which is made even simpler to formalise in our development where processes are intrinsically typed. We start by defining reduction contexts as processes with a single hole. In our intrinsically-typed formalisation, reduction contexts are parameterised by the typing context Δ of the hole, which is invariant, and indexed by the typing context Γ of the whole reduction context:

```
data ReductionContext (Δ : Context) : Context → Set where
  hole  : ReductionContext Δ Δ
  cut-l : ∀{A} → ∀[ ((A :: _) ⊢ ReductionContext Δ) ∗ ((dual A :: _) ⊢ Proc) ⇒
                    ReductionContext Δ ]
  cut-r : ∀{A} → ∀[ ((A :: _) ⊢ Proc) ∗ ((dual A :: _) ⊢ ReductionContext Δ) ⇒
                    ReductionContext Δ ]
```

The constructor hole builds a hole, as the name implies. The constructors cut-l and cut-r build reduction contexts where the hole is found in the left (respectively, right) sub-term of a cut, as per the grammar of reduction contexts given in Section 2.5.

Substitution inside a reduction context $\mathcal{C}$ is a straightforward function _⟦_⟧ that operates recursively on the structure of $\mathcal{C}$:

```
_⟦_⟧ : ∀{Γ Δ} → ReductionContext Δ Γ → Proc Δ → Proc Γ
hole            ⟦ P ⟧ = P
cut-l (𝒞 ⟨ p ⟩ Q) ⟦ P ⟧ = cut ((𝒞 ⟦ P ⟧) ⟨ p ⟩ Q)
cut-r (Q ⟨ p ⟩ 𝒞) ⟦ P ⟧ = cut (Q ⟨ p ⟩ (𝒞 ⟦ P ⟧))
```

This notion of process substitution preserves typing by construction thanks to the fact that both processes and reduction contexts are intrinsically typed.

A process P is well formed if every unguarded sub-process Q in it is alive.

```
WellFormed : ∀{Γ} → Proc Γ → Set
WellFormed {Γ} P = ∀{Δ} {𝒞 : ReductionContext Δ Γ} {Q : Proc Δ} →
                   P ⊒ (𝒞 ⟦ Q ⟧) → Alive Q
```

The proof of type safety ends up being a trivial application of deadlock-freedom. No work is needed to deduce that the process Q in the hole of a reduction context is well typed because structural pre-congruence preserves typing by definition.

```
type-safety : ∀{Γ} (P : Proc Γ) → WellFormed P
type-safety P {_} {_} {Q} _ = deadlock-freedom Q
```

3.9 Examples

In this section we revisit and expand the processes discussed in Examples 2.1 and 2.3 and show their encoding in our formalisation. The encoding of $\mathbb{B}$ is straightforward

```
𝔹 : Type
𝔹 = 𝟏 ⊕ 𝟏
```

and the boolean constants are encoded thus:

```
True : Proc [ 𝔹 ]
True = select (ch ⟨ < ≫ ⟩ inj₁ (close ch))
```

```
False : Proc [ 𝔹 ]
False = select (ch ⟨ < ≫ ⟩ inj₂ (close ch))
```

We take advantage of the host language for programming higher-order processes. For example, we can define a conditional process thus:

```
If_Else : ∀[ Proc ⇒ Proc ⇒ (dual 𝔹 ::_) ⊢ Proc ]
If P Else Q = curry∗ case ch (< ≫) ( wait (ch ⟨ < ≫ ⟩ P)
                                   , wait (ch ⟨ < ≫ ⟩ Q))
```

A term If P Else Q is a process that waits for a boolean value (cf. the dual 𝔹 type at the front of its typing context) and continues as either P or Q depending on whether it receives true or false. We use curry∗ (defined in Section 3.2) to curry the constructor case so that we can supply its arguments one by one saving a few parentheses and reducing clutter (more on this in Remark 3.1 at the end of this section).

Next we define a process Drop P that consumes a boolean and continues as P regardless of its value.

```
Drop : ∀[ Proc ⇒ (dual 𝔹 ::_) ⊢ Proc ]
Drop P = If P Else P
```

Using these higher-order forms, it is easy to define the usual boolean connectives.

```
!! : Proc [ 𝔹 ] → Proc [ 𝔹 ]
!! B = curry∗ cut B ≫ (If False Else True)

_&&_ _||_ : Proc [ 𝔹 ] → Proc [ 𝔹 ] → Proc [ 𝔹 ]
A && B = curry∗ cut A ≫ $
         curry∗ cut B ≫ $
         If curry∗ link ch (< ≫) ch Else (Drop False)
A || B   = !! (!! A && !! B)
```

The function $ (defined in Agda's standard library) is just a low-precedence, visible function application operator. We use it as a separator to flatten deeply nested expressions and save a bunch of parentheses. For the sake of illustration, we have chosen to define the disjunction || from the conjunction && and negation !! using De Morgan's laws.

To test our definitions, we implement a simple evaluator using the deadlock freedom property. We have not proved a termination result, but since linear logic enjoys cut elimination we can safely annotate the evaluator as terminating.

```
{-# TERMINATING #-}
eval : ∀[ Proc ⇒ Proc ]
eval P with deadlock-freedom P
```

```
... | inj₁ (Q , _ , _) = Q
... | inj₂ (Q , _)     = eval Q
```

Now if we ask Agda to normalise the goal eval (False || False) we obtain select (ch ⟨ < • ⟩ inj₂ (close ch)), that is the definition of False, as expected.

For the encoding of the polymorphic echo server (Example 2.3), we start by encoding its type $!(\forall X.X^{\perp} \parr (X \otimes \mathbf{1}))$:

```
ServerT : Type
ServerT = ‘! (‘∀ (rav (# 0) ⅋ (var (# 0) ⊗ 𝟏)))
```

The notation # n (defined in Agda's standard library) creates an element of Fin from the natural number n. Here it is used to create the de Bruijn index of the type variable X. We now encode the server

```
Server : Proc [ ServerT ]
Server = curry (curry∗ server ch (< ≫)) un-[] $
           curry∗ all ch (< ≫) λ X →
           curry∗ join ch (< ≫) $
           curry∗ (curry∗ fork ch (< ≫)) (curry∗ link ch (< > •) ch) (< ≫) $
           close ch
```

and the client that sends true to it

```
Client : Proc (dual ServerT :: 𝔹 :: [])
Client = curry∗ client ch (< ≫) $
         curry∗ (ex {_} {𝔹}) ch (< ≫) $
         curry∗ (curry∗ fork ch (< ≫)) True ≫ $
         curry∗ join ch (< ≫) $
         curry∗ wait ch (< ≫) $
         curry∗ link ch (< > •) ch
```

To test our definitions, we compose client and server in parallel

```
Main : Proc [ 𝔹 ]
Main = curry∗ cut Client (< •) Server
```

and then ask Agda to normalize Main, which yields True as expected.

Remark 3.1. Writing processes in Agda would be more pleasant if the constructors of the data type Proc were naturally curried, instead of currying them on demand with curry∗ as we do here. Below is the naturally curried constructor fork of a hypothetical data type Proc', obtained by expanding the definition of separating conjunction:

```
fork : ∀{A B Γ Δ Θ Θ₁ Θ₂} → Ch (A ⊗ B) Δ → Γ ≃ Δ + Θ →
         Proc' (A :: Θ₁) → Θ ≃ Θ₁ + Θ₂ → Proc' (B :: Θ₂) → Proc' (A ⊗ B :: Γ)
```

This version of fork is fully curried, but also less readable than the one we gave in Section 3.4 because of the (now visible) context splittings. We can recover some clarity and still obtain a curried version of fork using (literally) the magic wand:

```
fork : ∀{A B} →
        ∀[ Ch (A ⊗ B) ⇒ (A ::_) ⊢ Proc" −∗ (B ::_) ⊢ Proc" −∗ Proc" ]
```

However, the first arrow must be a plain implication ⇒ and not a magic wand to account for the appropriate amount of context splittings. We found this formulation of the constructors harder to explain and motivate in Section 3.4. Since none of the alternative definitions of Proc was fully satisfactory, we preferred the most elegant version of the data type at the expense of additional clutter in this section. ⌟

4 Related Work

We have compared our formalisation with others of typed calculi that support binary sessions either natively or through their encoding using continuations [9, 28].

Goto et al. [14] describe the formalisation of a session-based variant of the π-calculus which supports channel polymorphism. This is the oldest formalisation of a session-based calculus for which we were able to retrieve the source code (the work of Gay [29] predates this one, but its source code is not publicly available any more).

Thiemann [15] formalises a subset of GV [30], a functional language extended with session communication primitives, along with an interpreter. Ciccone and Padovani [17] have taken inspiration from his work to formalise a variant of the linear π-calculus [28] that supports dependent types, so as to enable the description of communication protocols whose structure may depend on the content of messages.

Castro-Perez et al. [16] describe EMTST, a library for the formalisation of session type systems that includes as case studies the session calculus of Honda et al. [3] (called "original system") and a revised version of it that is more amenable to be formalised using a locally nameless representation of channels.

Rouvoet et al. [18] present a library of abstractions inspired to separation logic aiding the formalisation of interpreters for languages with linear resources. One of the presented case studies is the formalisation of a fragment of GV [11, 30]. Unlike the other formalisations we are discussing, Rouvoet et al. [18] do not define a small-step semantics for GV but their formalisation is intrinsically typed, hence the interpreter preserves proves a form of subject reduction property. The separating conjunction defined in Section 3.2 and the typing of the constructors for the representation of processes in Section 3.4 have been adapted from this work of Rouvoet et al. [18].

Jacobs et al. [19] formalise a library of *connectivity graphs* for reasoning on and enforcing deadlock freedom in a variant of GV [11, 30]. This is the first formalisation of deadlock freedom for a calculus of sessions.

All the formalisations mentioned so far make use of context splitting. In contrast, Zalakain and Dardha [6] formalise a generalisation of the linear π-calculus which is parametric in a *usage algebra* (to account for channel sharing/linearity) and that is based on *leftover typing* [31]. Typing judgments have the form $\Gamma \vdash P \triangleright \Delta$ so that a process P is typed with respect to an input context Γ, which describes all the available channels, and a context of leftovers Δ, which describes the residual channels not consumed by the process. In this way, it is possible to "concatenate" typing judgments by matching the leftovers in one judgment with the input context of the subsequent one, with no need for splitting. As Zalakain and Dardha [6] nicely summarise, context splittings are not necessary because they "contain usage information that is already present in processes." This is true provided that channels are named (Zalakain and Dardha [6] use de Bruijn indices to this aim). In fact, the co-de Bruijn representation of processes [27], whereby channels are nameless and context splitting is performed eagerly, can be seen as the "dual approach" of leftover typing: channel names provide information that is already present in their (singleton) typing context, hence they can be omitted from contexts and processes.

Motivated by the technical difficulties arising from context splittings, Sano et al. [7] define a structural version of CP using an approach based on *linearity predicates*. The key idea is to treat typing contexts structurally and to enforce the linear usage of channels by checking their syntactical occurrence in processes. Interestingly, this approach relies on the *explicit naming of continuations* so as to precisely account for the number of times a channel is actually used. Sano et al. [7] do not connect their technique with the continuation-passing encoding of binary sessions [9, 10], but the analogies are evident even though the role of continuations differs.

Zackon et al. [8] describe a typing context management technique where channels are associated not just with a type but also with a *tag*, that is an element of a given resource algebra that summarises the number of allowed usages of a particular channel, including the possibility that the channel is not available. This approach streamlines context splitting since contexts can be treated in an essentially structural way, except for tags which are conveninently combined using operations from the resource algebra.

Table 5 shows an overview of the aforementioned formalisations (sorted by publication date) including our own. The first five columns identify the calculus being formalised. We provide its reference paper, the prover in which it is formalised and an acronym that gives an idea of the flavour of the calculus. We also specify whether the calculus features *cuts* (that is, the combination of restriction and parallel composition corresponding to the cut of linear logic) and continuations. CP [5, 11] and GV [11, 30] are well-known acronyms in the literature on session types. Sπ refers to (variants of) the session-based π-calculus presented by Honda et al. [3] while Lπ refers to (variants of) the linear π-calculus [28]. Finally, SCP is the structural version of CP introduced by Sano et al. [7] and LCC is our calculus. We emphasize that the actual calculus being formalised usually differs from (typically, is a strict subset of) the one identified by the acronym and that the same acronym may sometimes refer to different versions of the same calculus. In particular, GV in a logical setting is described by Wadler [11] but its first (non-logical) version is due to Gay and Vasconcelos [30].

Concerning the use of continuations, the approaches based directly on the linear π-calculus (into which sessions can be encoded) are marked with $+$ and those based on a calculus with native sessions are marked with $-$. The calculus SCP is marked with $\pm$ because, while not directly inspired to the linear π-calculus, it makes use of explicit continuations for defining the predicates that check the linear usage of channels. Finally, all the approaches based on GV are also marked with $\pm$. Officially,

Table 5 Overview of different formalisations of binary session calculi (sizes in kb).

Reference paper	Prover	Calculus	Cuts	Continuations	Linearity	Channels	Intrinsically typed	Subject reduction	Library	Deadlock freedom	Library	**Total size**
Goto et al. [14]	Coq	Sπ	–	–	splits	loc. nameless	–	543	–	–	–	**543** [a]
Thiemann [15]	Agda	GV	–	±	splits	co-de Bruijn	+	177	–	–	–	**177** [b]
Rouvoet et al. [18]	Agda	GV	–	±	splits	co-de Bruijn	+	27	55	–	–	**82** [c]
Castro-Perez et al. [16]	Coq	Sπ	–	–	splits	loc. nameless	–	204	–	–	–	**204** [d]
Ciccone and Padovani [17]	Agda	Lπ	–	+	splits	co-de Bruijn	+	77	–	–	–	**77** [e]
Zalakain and Dardha [6]	Agda	Lπ	–	+	leftovers	de Bruijn	–	82	8	–	–	**90** [f]
Jacobs et al. [19]	Coq	GV	+	±	splits	named	–	68	–	25	171	**264** [g]
Sano et al. [7]	Beluga	SCP	+	±	predicates	HOAS	–	35	–	–	–	**35**
Zackon et al. [8]	Beluga	CP	+	–	tags	HOAS	–	56	73	–	–	**129** [h]
this	Agda	LCC	+	+	splits	co-de Bruijn	+	21	–	15	–	**36** [i]

[a] Includes shared and polymorphic channels. Excluded safety results.
[b] Includes shared channels, recursive types, subtyping and the interpreter.
[c] Includes type-preserving evaluator and library for proof-relevant separation algebra.
[d] Includes shared channels. Excluded original syntax.
[e] Includes shared channels, recursive and dependent session types.
[f] Includes shared channels and the library for the algebra of types.
[g] Includes deadlock freedom and the library for connectivity graphs.
[h] Excluded correspondence between CP and SCP.
[i] Includes shared, polymorphic channels and deadlock freedom. Excluded safety results.

none of these calculi makes use of continuations, but GV is designed in such a way that each operation acting on a channel s is a function that returns the result of the operation (if present) *along with the same channel* s. In this way, the type of s can be conveniently "updated" to take into account the effect of the operation. As observed by Padovani [32], this semantics of the communication primitives is virtually indistinguishable from one making use of explicit continuation channels.

The three middle columns of Table 5 report the relevant qualitative aspects of the formalisations, namely the management of typing contexts, the representation of channels and whether processes are intrinsically or extrinsically typed.

The rightmost columns report the size (in kilobytes) of the formalisations as rough (and possibly questionable) estimates of their complexity. Papers describing formalisations typically report the "lines of code" as a measure of development effort, but the number of lines may be affected by code indentation styles and syntactical constraints of the proof assistant being used. For this reason, we have preferred to count the total number of characters after comments have been removed and spaces have been squeezed.[2] The reported sizes account for the source code of the formalisations excluding examples and any safety result, if present. We have excluded safety results because their meaning varies widely across the formalisations and, except for our

[2] Sequences of two or more consecutive space-like characters are collapsed into a single space. The squeezing is obtained by running the command `tr -s [:space:] file` on Unix-like systems.

own, they all differ from the one stated in the linearity challenge [1]. Some formalisations [6, 8, 18, 19] define *libraries* which can be reused in different contexts. In these cases, the size of the library is reported separately next to the size of the part of the development that uses it.

In general, it is difficult to draw firm conclusions on the effectiveness of the various approaches in addressing the linearity challenge because the formalisations differ widely for a variety of entangled factors. Looking at the available data, we can make the following observations. The adoption of context splitting, which is very well represented, does not seem to be a good indicator of the complexity of the formalisation. Indeed, the formalisations based on context splitting span the whole range of sizes, from the largest by Goto et al. [14] (543kb) to our own (21kb, without the proof of deadlock freedom) which is also the only one supporting all the features of CP.

The two largest formalisations [14, 16] are also the ones that adopt a locally nameless representation of channels. In these formalisations channels are represented in two different ways, depending on whether they are free or bound. This entails some duplication of effort as well as some transformation machinery between the two representations. Other channel representations are not strong complexity indicators. Note that the adoption of co-de Bruijn syntax implies the use of context splitting, hence the two aspects are not completely independent.

There is no strong evidence that the intrinsically typed representation of processes reduces the size of the formalisation. As observed in Section 3.4, this choice helps reducing the overall number of datatypes to be defined and makes some results trivial (e.g. Theorem 2.3 formalised by type-safety), but the definitions are also more involved because they incorporate invariants and bits of the proofs of typing preservation. We speculate that the effort for representing processes, types and typing rules is not substantially impacted overall, but the data types for representing syntax and semantics of untyped processes in extrinsically-typed representations are certainly more readable.

Using the cut in the style of linear logic instead of separate restriction and parallel composition simplifies the representation of channels (or session endpoints). All the formalisations of calculi that adopt the cut tend to be small (if we exclude the libraries), but this is not a general rule.

Finally, it appears that the use of (explicit) continuations is related to the complexity of the formalisation more than anything else. Indeed, the six smallest formalisations (excluding the deadlock freedom results) – with an average size of around 62kb – are all based on continuations, no matter if they are explicit (Lπ, SCP, LCC) or "virtual" (GV), while the remaining ones are 263kb on average. At the very least, the use of continuations enables a cleaner management of typing contexts since linear channels are true "use-once" resources and there is no need to update their type.

5 Concluding Remarks

We have presented a formalisation of LCC, a linear calculus of continuations closely related to the linear π-calculus [28] and supported by the same type system of CP [11].

Binary sessions can be modeled in LCC using the continuation-passing encoding described by Kobayashi [9, extended version] and Dardha et al. [10].

The linear calculus of continuations and the calculus described in the linearity challenge [1] are incomparable in terms of expressiveness. On the one hand, the challenge only considers a minimal calculus of first-order, monomorphic sessions while LCC supports linear, shared, higher-order, polymorphic channels; on the other hand, the calculus of the challenge allows the modeling of sequential processes owning both endpoints of a session and in general of cyclic network topologies, none of which can be modeled in LCC because of its tight correspondence with linear logic. We think that LCC deserves its own space in the context of the linearity challenge alongside with (but not in substitution of) more traditional session calculi.

Considering the richness of LCC in terms of features and proved properties, the simple formalisation of LCC casts some doubts on the actual role of context splitting as a source of complexity. We perceive more tangible benefits from the adoption of a calculus with explicit continuations where channels are linear in a literal sense. In this respect, we find it intriguing that, among the alternative approaches that have been proposed to overcome the difficulties of context splitting, the one by Sano et al. [7] makes key use of explicit continuations.

The compact formalisation of LCC is a good starting point for further developments. We have already extended LCC with support for coinductive (i.e. possibly infinite) types and recursive processes (this extension is in LCC's public repository [20]). In the future, it would be interesting to formalise the strong normalisation property of LCC as a consequence of cut elimination of classical linear logic.

In this work we have focused on models of *binary sessions* (those connecting exactly two processes), but there are also formalisations of *multiparty sessions*, notably those by Jacobs et al. [33] and Tirore et al. [34], which can be significantly more complex than those of binary sessions. The formalisation by Jacobs et al. [33] amounts to 173kb and the one by Tirore et al. [34] to more than 1Mb of Coq code. Also in these cases, the formalisation based on (virtual) continuations [33] happens to be substantially smaller. Whether this is a coincidence or further evidence of the effectiveness of the continuation-based approaches is left for future investigations.

Declarations

Funding

No funding was received for conducting this study.

Author Contributions

C.R. developed the initial Agda formalisation and reviewed the existing related work. L.P. refined and extended the formalisation and wrote the main manuscript text. All authors reviewed the manuscript.

References

[1] Carbone, M., Castro-Perez, D., Ferreira, F., Gheri, L., Jacobsen, F.K., Momigliano, A., Padovani, L., Scalas, A., Tirore, D.L., Vassor, M., Yoshida, N., Zackon, D.: The concurrent calculi formalisation benchmark. In: Castellani, I., Tiezzi, F. (eds.) Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14676, pp. 149–158. Springer, Germany (2024). https://doi.org/10.1007/978-3-031-62697-5_9

[2] Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer, Germany (1993). https://doi.org/10.1007/3-540-57208-2_35

[3] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer, Germany (1998). https://doi.org/10.1007/BFB0053567

[4] Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3–1336 (2016) https://doi.org/10.1145/2873052

[5] Gay, S.J., Vasconcelos, V.T.: Session Types. Cambridge University Press, Cambridge, UK (2025). https://doi.org/10.1017/9781009000062

[6] Zalakain, U., Dardha, O.: π with leftovers: A mechanisation in agda. In: Peters, K., Willemse, T.A.C. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 41st IFIP WG 6.1 International Conference, FORTE 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12719, pp. 157–174. Springer, Germany (2021). https://doi.org/10.1007/978-3-030-78089-0_9

[7] Sano, C., Kavanagh, R., Pientka, B.: Mechanizing session-types using a structural view: Enforcing linearity without linearity. Proc. ACM Program. Lang. **7**(OOPSLA2), 374–399 (2023) https://doi.org/10.1145/3622810

[8] Zackon, D., Sano, C., Momigliano, A., Pientka, B.: Split decisions: Explicit contexts for substructural languages. In: Stark, K., Timany, A., Blazy, S., Tabareau,

N. (eds.) Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025, pp. 257–271. ACM, USA (2025). https://doi.org/10.1145/3703595.3705888

[9] Kobayashi, N.: Type systems for concurrent programs. In: 10th Anniversary Colloquium of UNU/IIST. LNCS 2757, pp. 439–453. Springer, Germany (2002). Extended version available at http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf

[10] Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017) https://doi.org/10.1016/j.ic.2017.06.002

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

[12] Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (2005) https://doi.org/10.1007/S00236-005-0177-Z

[13] Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012) https://doi.org/10.1016/J.IC.2012.05.002

[14] Goto, M.A., Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: An extensible approach to session polymorphism. Math. Struct. Comput. Sci. **26**(3), 465–509 (2016) https://doi.org/10.1017/S0960129514000231

[15] Thiemann, P.: Intrinsically-typed mechanized semantics for session types. In: Komendantskaya, E. (ed.) Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019, pp. 19–11915. ACM, USA (2019). https://doi.org/10.1145/3354166.3354184

[16] Castro-Perez, D., Ferreira, F., Yoshida, N.: EMTST: engineering the meta-theory of session types. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 278–285. Springer, Germany (2020). https://doi.org/10.1007/978-3-030-45237-7_17

[17] Ciccone, L., Padovani, L.: A dependently typed linear π-calculus in agda. In: PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020, pp. 8–1814. ACM, USA (2020). https://doi.org/10.1145/3414080.3414109

[18] Rouvoet, A., Poulsen, C.B., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on

Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020, pp. 284–298. ACM, USA (2020). https://doi.org/10.1145/3372885.3373818

[19] Jacobs, J., Balzer, S., Krebbers, R.: Connectivity graphs: a method for proving deadlock freedom based on separation logic. Proc. ACM Program. Lang. **6**(POPL), 1–33 (2022) https://doi.org/10.1145/3498662

[20] Padovani, L., Raffaelli, C.: Agda Formalisation of the Linear Calculus of Continuations. Last access 2025-12-29 (2025). https://github.com/boystrange/LinearityChallenge

[21] Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Vitek, J. (ed.) Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9032, pp. 560–584. Springer, Germany (2015). https://doi.org/10.1007/978-3-662-46669-8_23

[22] Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6269, pp. 222–236. Springer, Germany (2010). https://doi.org/10.1007/978-3-642-15375-4_16

[23] Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Math. Struct. Comput. Sci. **26**(3), 367–423 (2016) https://doi.org/10.1017/S0960129514000218

[24] Horne, R., Padovani, L.: A logical account of subtyping for session types. J. Log. Algebraic Methods Program. **141**, 100986 (2024) https://doi.org/10.1016/J.JLAMP.2024.100986

[25] Kokke, W., Siek, J.G., Wadler, P.: Programming language foundations in agda. Sci. Comput. Program. **194**, 102440 (2020) https://doi.org/10.1016/J.SCICO.2020.102440

[26] O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symb. Log. **5**(2), 215–244 (1999) https://doi.org/10.2307/421090

[27] McBride, C.: Everybody's got to be somewhere. In: Atkey, R., Lindley, S. (eds.) Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018. EPTCS, vol. 275, pp. 53–69 (2018). https://doi.org/10.4204/EPTCS.275.6

[28] Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems **21**(5), 914–947 (1999)

[29] Gay, S.J.: A framework for the formalisation of pi calculus type systems in isabelle/hol. In: Boulton, R.J., Jackson, P.B. (eds.) Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2152, pp. 217–232. Springer, Germany (2001). https://doi.org/10.1007/3-540-44755-5_16

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

[31] Allais, G.: Typing with leftovers - A mechanization of intuitionistic multiplicative-additive linear logic. In: Abel, A., Forsberg, F.N., Kaposi, A. (eds.) 23rd International Conference on Types for Proofs and Programs, TYPES 2017, Budapest, Hungary, May 29 - June 1, 2017. LIPIcs, vol. 104, pp. 1–1122. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2017). https://doi.org/10.4230/LIPICS.TYPES.2017.1

[32] Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. **27**, 4 (2017) https://doi.org/10.1017/S0956796816000289

[33] Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: functional multiparty session types with certified deadlock freedom. Proc. ACM Program. Lang. **6**(ICFP), 466–495 (2022) https://doi.org/10.1145/3547638

[34] Tirore, D.L., Bengtson, J., Carbone, M.: Multiparty asynchronous session types: A mechanised proof of subject reduction. In: Aldrich, J., Silva, A. (eds.) 39th European Conference on Object-Oriented Programming, ECOOP 2025, Bergen, Norway, June 30 - July 2, 2025. LIPIcs, vol. 333, pp. 31–13130. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2025). https://doi.org/10.4230/LIPICS.ECOOP.2025.31