


Fair Termination of Asynchronous Binary Sessions

Luca Padovani 

University of Bologna, Italy

Gianluigi Zavattaro 

University of Bologna, Italy

INRIA OLAS team, Sophia Antipolis, France

Abstract

We study a theory of asynchronous session types ensuring that well-typed processes terminate under a suitable fairness assumption. Fair termination entails *starvation freedom* and *orphan message freedom* namely that all messages, including those that are produced *early* taking advantage of asynchrony, are eventually consumed. The theory is based on a novel *fair asynchronous subtyping* relation for session types that is coarser than the existing ones. The type system is also the first of its kind that is firmly rooted in linear logic: fair asynchronous subtyping is incorporated as a natural generalization of the cut and axiom rules of linear logic and asynchronous communication is modeled through a suitable set of *commuting conversions* and of *deep cut reductions* in linear logic proofs.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Linear logic; Theory of computation → Program analysis

Keywords and phrases Binary sessions, fair asynchronous subtyping, fair termination, linear logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.24

Related Version *Full Version*: <https://doi.org/10.48550/arXiv.2503.07273> [43]

Funding The authors have been partially supported by the ANR project SmartCloud ANR-23-CE25-0012 and the PNRR project CN-HPC Spoke 9 Innovation Grant RTMER.

Acknowledgements We are grateful to the ECOOP reviewers for their thoughtful feedback and questions.

1 Introduction

Session type systems [32, 33, 35] have become a widespread formalism for ensuring a variety of safety and liveness properties of communicating processes through static analysis. Session types specify the *sequences* of messages that can be sent over a channel, and the type system makes sure that (1) well-typed processes comply with this protocol specification and that (2) the peer endpoints of the channel are used according to compatible protocols.

Many session type systems are defined for a synchronous calculus or language even if the actual underlying communication model is meant to be asynchronous. The point is that synchronous theories of session types are simpler and easier to work with and most if not all properties of a synchronous session type system hold even if the actual communication model is asynchronous. However, awareness of the communication model can help relaxing the type system and thus enlarging the family of well-typed processes. This observation has led to the study of *asynchronous subtyping relations* for session types [40, 39, 11] allowing processes to *anticipate* output messages with respect to the protocol specification they are expected to comply with, provided that anticipated outputs *do not depend* on incoming messages. This apparent violation of the protocol specification enabled by asynchronous subtyping is harmless precisely because output actions are non-blocking in an asynchronous setting.

The available session type systems based on asynchronous subtyping focus on the enforcement of safety properties but struggle at ensuring liveness properties of many simple communication patterns. As an illustration of such patterns consider a server that, in order



© Luca Padovani and Gianluigi Zavattaro;

licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).

Editors: Jonathan Aldrich and Alexandra Silva; Article No. 24; pp. 24:1–24:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

24:2 Fair Termination of Asynchronous Binary Sessions

to fulfill a request received from a session x , splits the request into an arbitrary number of tasks handled by a separate worker and then gathers the partial results to answer the client. We might be interested in establishing whether the client eventually receives a response.

To be more concrete, let us model the server as the term

$$x \triangleright \{\text{req} : (y)(\text{Split}\langle x, y \rangle \mid \text{Worker}\langle y \rangle)\}$$

which indicates the input of the request followed by the spawning of a *Split* process and of a *Worker* process connected by a new session y .

The *Split* process is modeled by the following definitions:

$$\text{Split}(x, y) = y \triangleleft \text{task}.\text{Split}\langle x, y \rangle \oplus y \triangleleft \text{stop}.\text{Gather}\langle x, y \rangle \quad (1)$$

$$\text{Gather}(x, y) = y \triangleright \{\text{res} : \text{Gather}\langle x, y \rangle, \text{stop} : \text{wait } y.x \triangleleft \text{resp.close } x\} \quad (2)$$

according to which the server sends to the worker a non-deterministically chosen number of **tasks** followed by a **stop** on session y , it then gathers from the worker an arbitrary number of **results** followed by a **stop**, and finally sends back to the client a **response** on session x .

One possible modeling of the worker process is according to the definition

$$\text{Worker}(y) = y \triangleright \{\text{task} : y \triangleleft \text{res}.\text{Worker}\langle y \rangle, \text{stop} : y \triangleleft \text{stop.close } y\} \quad (3)$$

so that the worker sends a **result** for each **task** it receives. The eye-catching aspect of *Worker* is that it does not interact according to the “complementary” protocol implemented on the server side, at least not in the sense that is usually intended in (synchronous) session type theories. Indeed, while the server first sends all the **tasks** and then gathers all the **results**, the worker eagerly sends one **result** after receiving each **task**.

In an asynchronous setting, the fact that *Worker* sends some messages earlier than expected is not an issue since output actions are non-blocking. However, we would like to be sure that these early **results** do not keep accumulating and are eventually consumed by *Gather*. There is nothing in the modeling of *Split*, *Gather* and *Worker* that prevents this from happening, but *we can prove the eventual consumption of every result only under the assumption that sooner or later Split will send stop to Worker*. If we broaden our viewpoint, we see that the same assumption is necessary to prove that the client interacting with the server does not starve. While the server is running, the client is awaiting for a **response** from session x . In order to *prove* that the client will eventually receive a **response**, we have to assume that *Worker* will terminate the session y , which in turn requires the assumption that *Split* will eventually send **stop** to *Worker*. In general, the proof of any non-trivial liveness property that concerns the eventual production or consumption of a message on a certain session may require the assumption that every other session eventually terminates. For this reason, ensuring the eventual termination of sessions should be a primary goal of any session type system aimed at enforcing liveness properties. As we have seen in the discussion above, proving the eventual termination of a session may require some *fairness assumptions* like the fact that *Split* will eventually stop sending **tasks**. For this reason, in the literature such eventual termination property is referred to as *fair termination* [30, 26, 2].

In this work we study a theory of asynchronous session types and an associated session type system ensuring that well-typed processes are fairly terminating under a suitable fairness assumption. The fair termination of processes entails the fair termination of the sessions they operate on, hence that all messages produced – including those sent earlier than expected – are eventually consumed and that all processes waiting for a message eventually receive one. In other words, the very same type system prevents *process starvation* and ensures the absence of *orphan messages*. Related works achieve these goals only partially:

- The theories of asynchronous session types developed by Mostrous *et al.* [40, 39], Chen *et al.* [11] and Ghilezan *et al.* [29] require that every output can be only anticipated with respect to finitely many inputs. Since there is no finite upper bound to the number of **tasks** that *Split* can produce, each early **res** produced by *Worker* anticipates an unbounded number of inputs leaving *Worker* out of reach for the aforementioned theories.
- Bravetti, Lange and Zavattaro [8] study an asynchronous subtyping relation for session types that allows early outputs to anticipate an unbounded number of inputs. However, their subtyping relation disallows any *covariance* of outputs, which is needed to account for the fact that the behavior of *Worker* is *more deterministic* than the behavior expected by *Gather*. Indeed, while *Gather* expects to receive an arbitrary number of **results**, *Worker* produces exactly as many **results** as the number of **tasks** it receives from *Split*.
- None of the aforementioned works provides guarantees on the eventual fulfillment of the client's request, either because they do not make sure that every session eventually terminates [11, 29] or because they do not take multiple sessions into account [29, 8].
- Ciccone, Dagnino and Padovani [14, 16, 13] study session type systems ensuring the fair termination of sessions, but their works are based on synchronous communication models that do not support any form of output anticipation like the one exemplified by *Worker*.

The theory of asynchronous session types that we propose in this paper addresses all these limitations. In addition, we also make the following technical contributions:

- We define a *fair asynchronous subtyping relation* for session types that is coarser than those in the literature for the family of eventually terminating session types. Unlike many existing fair/asynchronous subtyping relations [39, 14, 13, 8] our subtyping relation is *closed under duality*. This property is key for proving the type system sound.
- We give the first *fair asynchronous semantics* of session types using a labelled transition system (LTS) defined by *bounded coinduction* [1, 18]. The adoption of this semantics allows us to characterize fair asynchronous subtyping in a way that is structurally the same as the one for the well-known synchronous subtyping defined by Gay and Hole [28].
- Our theory of asynchronous session types with fair asynchronous subtyping is the first one where the process model and the type system are rooted in linear logic [45, 10, 37]. We incorporate fair asynchronous subtyping in the type system as generalized forms of the cut and linear logic axiom thanks to the aforementioned closure under duality. Also, instead of introducing explicit message buffers, we model asynchronous communications by means of suitable *commuting conversions* and *deep cut reductions* in linear logic proofs.

Structure of the paper. Section 2 describes our calculus of asynchronous processes and the properties we enforce. Section 3 introduces the semantics of asynchronous session types while Section 4 studies fair asynchronous subtyping and its properties. Section 5 describes the type system and Section 6 illustrates some common usage patterns of fair asynchronous subtyping. Section 7 shows how our LTS can be easily tailored to characterize various asynchronous subtyping relations that appear in the literature and compares them with our subtyping relation. Section 8 discusses related work in more detail and Section 9 recaps and outlines future work. Proofs and additional technical material are provided in the full paper [43].

2 A Calculus of Asynchronous Processes

In this section we present syntax and semantics of a calculus of asynchronous processes called CaP and we formulate the safety and liveness properties that our type system guarantees. At the surface level, CaP closely resembles other calculi of binary sessions based on linear logic

■ **Table 1** Syntax of CaP.

$P, Q ::=$	Process		done	termination
	$x \leftrightarrow y$		$A(\bar{x})$	invocation
	close x		wait $x.P$	signal input
	$x \triangleleft a.P$		$x \triangleright \{a_i : P_i\}_{i \in I}$	tag input
	$x(y)[P].Q$		$x(y).P$	channel input
	$P \oplus Q$		$(x)(P Q)$	composition

such as CP [45] or μCP [37]. The main differences between CaP and these calculi are that CaP supports general recursion, it includes a non-deterministic choice operator and above all it models asynchronous communication by giving a non-blocking semantics to output actions, which essentially act like message buffers.

The syntax of *processes* is shown in Table 1 and makes use of an infinite set of *channels* ranged over by x, y and z , a set **Tags** of *tags* ranged over by a, b, \dots and a set of *process names* ranged over by A, B , etc. The terminated process, which performs no actions, is denoted by **done**. The term $A(\bar{x})$, where \bar{x} is a possibly empty sequence of channels, represents the *invocation* of the process named A . We assume that for each such invocation there exists a unique global definition of the form $A(\bar{x}) = P$ that gives meaning to the name A . We also assume that all invocations occur guarded by a prefix or by a non-deterministic choice. The term $x \leftrightarrow y$ models a *link*, that is a process that forwards every message received from x to y and vice versa, effectively unifying the two channels. Links are typical of calculi based on linear logic since their typing rule correspond to the axiom of the logic. The terms **close** x and **wait** $x.P$ model processes that respectively send and receive a termination signal on x . Note that **close** x has no continuation, as is the case in most calculi based on linear logic, whereas **wait** $x.P$ continues as P . The term $x(y)[P].Q$ models a bifurcating session: it creates a new channel y , sends y on x , forks a new process P that uses y , and then continues as Q . The process $x(y).P$ waits for a channel y from x and then continues as P . The terms $x \triangleleft a.P$ and $x \triangleright \{a_i : P_i\}_{i \in I}$ model processes that respectively send and receive a tag. The sender selects one particular tag a to send. The receiver continues as P_i depending on the tag a_i that it receives. In the examples we sometimes use tags as an abstract representation of more complex messages, such as **requests** or **tasks**. The term $(x)(P | Q)$ models the parallel composition of two processes P and Q connected by the channel x . We often refer to this term as a *cut*, since its typing rule coincides with the cut rule of linear logic. Finally, the term $P \oplus Q$ models the non-deterministic choice between P and Q .

It is known that links in conjunction with bifurcating sessions can be used to encode *session delegation*, whereby processes exchange an *existing* (rather than new) channel z on x . This behavior can be modeled by a term of the form $x(y)[y \leftrightarrow z].P$. As we will see in Section 6, CaP links also act as *explicit casts* enabling useful forms of subsumption.

The notions of *free* and *bound* channels for processes are defined as expected with the proviso that a process of the form $x(y)[P].Q$ binds y in P but not in Q . We identify processes up to renaming of bound channels and we write $\text{fn}(P)$ for the set of channels occurring free in P . Also, for every global definition $A(\bar{x}) = P$ we assume $\text{fn}(P) = \{\bar{x}\}$.

We have anticipated that CaP adopts an asynchronous communication model. In practice this would be implemented by FIFO buffers storing messages that have been produced but not consumed. In CaP, we model asynchrony giving a non-blocking semantics to the output actions $x(y)[Q].P$ and $x \triangleleft a.P$. That is, we allow the continuation P to reduce and/or

■ **Table 2** Structural pre-congruence and reduction semantics of CaP.

[S-CALL]	$A(\bar{x}) \sqsupseteq P$	$A(\bar{x}) = P$
[S-LINK]	$x \leftrightarrow y \sqsupseteq y \leftrightarrow x$	
[S-COMM]	$(x)(P \mid Q) \sqsupseteq (x)(Q \mid P)$	
[S-WAIT]	$(x)(\mathcal{B}_x[\mathbf{wait} \ y.P] \mid Q) \sqsupseteq \mathbf{wait} \ y.(x)(\mathcal{B}_x[P] \mid Q)$	$x \neq y$
[S-CASE]	$(x)(\mathcal{B}_x[y \triangleright \{a_i : P_i\}_{i \in I}] \mid Q) \sqsupseteq y \triangleright \{a_i : (x)(\mathcal{B}_x[P_i] \mid Q)\}_{i \in I}$	$x \neq y$
[S-JOIN]	$(x)(\mathcal{B}_x[y(z).P] \mid Q) \sqsupseteq y(z).(x)(\mathcal{B}_x[P] \mid Q)$	$x \neq y, z \notin \text{fn}(\mathcal{B}_x)$
[S-PULL-0]	$(x)(\mathcal{B}_x[P] \mid \mathcal{B}'_x[x \leftrightarrow y]) \sqsupseteq \mathcal{B}_y[(x)(P \mid \mathcal{B}'_x[x \leftrightarrow y])]$	
[S-PULL-1]	$(x)(y(z)[P].Q \mid R) \sqsupseteq y(z)[(x)(P \mid R)].Q$	$x \neq y, x \in \text{fn}(P)$
[S-PULL-2]	$(x)(\mathcal{B}_y[P] \mid Q) \sqsupseteq \mathcal{B}_y[(x)(P \mid Q)]$	$x \neq y, x \notin \text{fn}(\mathcal{B}_y)$
[S-PULL-3]	$\mathcal{B}_x[y(z)[P].Q] \sqsupseteq y(z)[\mathcal{B}_x[P]].Q$	$x \neq y, x \in \text{fn}(P)$
[S-PULL-4]	$\mathcal{B}_x[\mathcal{B}'_y[P]] \sqsupseteq \mathcal{B}'_y[\mathcal{B}_x[P]]$	$x \neq y, x \in \text{fn}(P)$
[R-CHOICE]	$P_1 \oplus P_2 \rightarrow P_k$	$k \in \{1, 2\}$
[R-LINK]	$(x)(x \leftrightarrow y \mid P) \rightarrow P\{y/x\}$	
[R-CLOSE]	$(x)(\mathbf{close} \ x \mid \mathbf{wait} \ x.P) \rightarrow P$	
[R-SELECT]	$(x)(x \triangleleft a_k.P \mid \mathcal{B}_x[x \triangleright \{a_i : Q_i\}_{i \in I}]) \rightarrow (x)(P \mid \mathcal{B}_x[Q_k])$	$k \in I$
[R-FORK]	$(x)(x(y)[P].Q \mid \mathcal{B}_x[x(y).R]) \rightarrow (y)(P \mid (x)(Q \mid \mathcal{B}_x[R]))$	
[R-CUT]	$(x)(P \mid R) \rightarrow (x)(Q \mid R)$	$P \rightarrow Q$
[R-BUFFER]	$\mathcal{B}_x[P] \rightarrow \mathcal{B}_x[Q]$	$P \rightarrow Q$
[R-STR]	$P \rightarrow Q$	$P \sqsupseteq R \rightarrow Q$

interact with other sub-processes even if the prefix has not been consumed. In a sense, we consider the prefixes $x(y)[Q]$ and $x \triangleleft a$ as *floating messages* or parts of a *buffer* associated with channel x . In general, we call *buffer* any term generated by the following grammar:

Buffer $\mathcal{B}_x ::= [] \mid x \triangleleft a.\mathcal{B}_x \mid x(y)[P].\mathcal{B}_x$

A buffer is either empty, represented by a *hole* $[]$, or a tag a sent on x followed by a buffer for x , or a fresh channel y (with associated process P) sent on x and followed by a buffer for x . Note that the annotation x in the metavariable \mathcal{B}_x is meant to *bind* the channel on which the messages in the buffer have been sent. Therefore, having at our disposal a buffer \mathcal{B}_x , we can write \mathcal{B}_y for the buffer that has the same structure as \mathcal{B}_x but where x has been replaced by y . Buffers vaguely resemble *reduction contexts*, except that they allow us to place a hole *behind* output prefixes since these are meant to be non-blocking. Hereafter we write $\mathcal{B}_x[P]$ for the process obtained by replacing the hole in \mathcal{B}_x with P .

The operational semantics of CaP is given by a structural pre-congruence relation \sqsupseteq and a reduction relation \rightarrow , both defined in Table 2. In simple words, structural pre-congruence relates processes that are essentially equivalent except for the order of independent actions, while reduction describes communications and the resolution of non-deterministic choices.

We can roughly classify the rules for structural pre-congruence in four groups. The first group contains [S-CALL], [S-LINK] and [S-COMM], which capture expected properties of process invocations, links and parallel compositions: a process invocation $A(\bar{x})$ is indistinguishable from P if $A(\bar{x}) = P$; a link $x \leftrightarrow y$ is indistinguishable from $y \leftrightarrow x$; a cut $(x)(P \mid Q)$ is indistinguishable from $(x)(Q \mid P)$, that is parallel composition is commutative.

The second group of rules contains [S-WAIT], [S-CASE] and [S-JOIN]. These rules allow an input action on some channel y to be extruded from a cut on x when $x \neq y$. The purpose of these transformations is to move inputs on y close to outputs on y , so as to enable interactions

in the session y . These transformations correspond to well-known rearrangements of linear logic proofs (for example, they are sometimes referred to as *external reductions* [3, 25]), except that in **CaP** we allow the input prefix to be found *within an arbitrary buffer* \mathcal{B}_x , coherently with the intuition that (output) actions in a buffer are non-blocking. The side condition $z \notin \text{fn}(\mathcal{B}_x)$ in [S-JOIN] makes sure that free occurrences of z in \mathcal{B}_x are not accidentally captured by the binding prefix. Since we consider processes (and buffers) modulo renaming of bound names and there is an infinite supply of names, the bound z can always be renamed so as to enable this process transformation.

The third group of rules contains [S-PULL-0], [S-PULL-1] and [S-PULL-2]. These rules are similar to those of the second group, except that they allow whole *buffers* of messages on y to float through cuts on x when $x \neq y$. [S-PULL-2] is simple and speaks for itself while [S-PULL-1] deals with the case in which the name x bound by the cut occurs in the process P associated with a fresh channel z sent on $y \neq x$. In this case the cut as a whole becomes associated with z . In principle we should also specify the side condition $x \notin \text{fn}(Q)$, but this condition holds for well-typed processes when $x \in \text{fn}(P)$. The rule [S-PULL-0] covers a somewhat peculiar case of [S-PULL-2], whereby a buffer \mathcal{B}_x can be extruded from the cut on x because there is a link $x \leftrightarrow y$ that acts as a forwarder from x to y . Note that the extruded buffer \mathcal{B}_x turns into \mathcal{B}_y , so as to reflect the forwarding effect of the link.

The fourth and final group of rules contains [S-PULL-3] and [S-PULL-4]. These rules allow buffers for different channels to be permuted. Again [S-PULL-3] deals with the special case in which a channel x occurs in the process P associated with a fresh channel z .

In the definition of structural pre-congruence there are some glaring omissions (*e.g.* associativity of parallel composition) and very few rules are invertible. This is not because the missing rules would be unsound, but because they turn out to be unnecessary for proving that well-typed processes are deadlock free and fairly terminating.

Base reductions consist of [R-CHOICE], [R-CLOSE], [R-SELECT] and [R-FORK] which respectively model the reduction of a non-deterministic process, the termination of the session x and the consumption of tags and channels sent on channel x . The rules are almost standard, except that [R-SELECT] and [R-FORK] allow input actions on x to operate from within arbitrary buffers for x . The buffers represent asynchronously sent messages that do not block subsequent actions. At the logical level, these interactions correspond to *deep cut reductions* in a sense that resembles *deep inference* [31], whereby logical rules can be applied deep within a context. Unlike [R-SELECT] and [R-FORK], there is no buffer around `wait $x.P$` in [R-CLOSE]. This rule implies that a session cannot be closed unless all the messages (asynchronously) produced therein have also been consumed. Note that this property is enforced by the type system and is not meant to be checked at runtime. Rules [R-CUT], [R-BUFFER] and [R-STR] propagate reductions across cuts and buffers and close them by structural pre-congruence.

Hereafter we write \rightarrow^* for the reflexive, transitive closure of \rightarrow . We write $P \rightarrow$ if $P \rightarrow Q$ for some Q and $P \not\rightarrow$ if not $P \rightarrow$.

We can now formally define the safety and liveness properties we are interested in.

► **Definition 1** (deadlock freedom). *We say that P is deadlock free if for every Q such that $P \rightarrow^* Q \not\rightarrow$ we have $Q \sqsupseteq \text{done}$.*

Deadlock freedom is an instance of safety property. A deadlock-free process either reduces or it is (structurally pre-congruent to) `done`. For example, the process $\Omega() = \Omega\langle \rangle \oplus \Omega\langle \rangle$ is deadlock free (we have $\Omega\langle \rangle \rightarrow \Omega\langle \rangle$) whereas $(x)(\text{close } y \mid \text{wait } x.\text{done})$ is deadlocked. When a deadlock-free process stops reducing, it contains no pending actions and all of its sessions have been closed. In particular, all the messages in buffers have been consumed.

The liveness properties we are interested in are related to *termination*, of which there exist several variants. A *reduction sequence* of P is a sequence (P_0, P_1, \dots) such that $P_0 = P$ and $P_i \rightarrow P_{i+1}$ whenever $i + 1$ is not greater than the length of the sequence. A *run* is a maximal reduction sequence, in the sense that either it is infinite or the last process in the sequence (say P_n) does not reduce (that is, $P_n \not\rightarrow$). We say that P is *weakly terminating* if it has a finite run, that P is *terminating* if every run of P is finite, and that P is *diverging* if every run of P is infinite. For example, $\Omega\langle \rangle \oplus \text{done}$ is weakly terminating but not terminating, $\text{done} \oplus \text{done}$ is terminating, and $\Omega\langle \rangle$ is diverging. Note that here we call “termination” the mere inability to reduce further and not the fact that a process has become *done*. For example, done , $\text{close } x$ and $(x)(\text{close } y \mid \text{wait } x.\text{done})$ are all terminated (they do not reduce), but only done is also deadlock free. So it really is the combination of deadlock freedom (Definition 1) and some termination property that we wish to enforce with our type system.

The termination property we target in this work is called *fair termination* [30, 26, 2]. Fair termination consists of those processes such that all of their infinite runs are considered to be unrealistic or *unfair* and therefore can be ignored insofar termination is concerned. We could say that these processes may diverge in principle, but they terminate in practice. Clearly, fair termination depends on a *fairness notion* that discriminates *fair* runs from *unfair* ones. Among all fairness notions, here we consider a particular instance of *full fairness* [44].

► **Definition 2.** *A run is fair if it contains finitely many weakly terminating processes.*

Remember that a run is a *maximal* reduction sequence of a process. So, along a fair run the process only has finitely many chances to terminate. This can happen either because the run is finite (the process eventually terminates) or because the run contains a diverging process (at some point termination is no longer possible). For example, the infinite run $(\Omega\langle \rangle \oplus \text{done}, \Omega\langle \rangle, \dots)$ is fair because only the first process in it is weakly terminating. We find it useful to also look at the *negation* of the notion of fair run: an *unfair run* is necessarily infinite and contains infinitely many weakly terminating processes. In other words, an unfair run describes a computation along which termination is always reachable, but it is never reached as if the process is avoiding it on purpose. For example, if $A() = A\langle \rangle \oplus \text{done}$ then the infinite run $(A\langle \rangle, A\langle \rangle, \dots)$ is unfair because done is always reachable but never reached.

► **Definition 3.** *We say that P is fairly terminating if every fair run of P is finite.*

For example, $A() = A\langle \rangle \oplus \text{done}$ is fairly terminating whereas $\Omega\langle \rangle \oplus \text{done}$ is not because it has an infinite fair run. There are two reasons why full fairness is a suitable fairness assumption in our setting. First, full fairness has been shown to be the *strongest* conceivable fairness assumption [44], which means that it allows us to target the *largest* family of fairly terminating processes. Second, it has been observed [12] that this family admits the following alternative characterization which does not mention fair runs at all.

► **Theorem 4.** *P is fairly terminating iff each Q such that $P \rightarrow^* Q$ is weakly terminating.*

The relevance of this characterization rests in the fact that it provides the key proof method for the soundness of our type system. Indeed, suppose that the type system ensures that well-typed processes *weakly* terminate. We expect the type system to also enjoy subject reduction, namely the property that well-typed processes always reduce to well-typed processes. But then, using the right-to-left implication in Theorem 4, the very same type system also ensures that well-typed processes fairly terminate.

► **Example 5.** Consider the *Split* process defined in (1). We derive

$$\text{Split}\langle x, y \rangle \rightarrow y \triangleleft \text{task}.\text{Split}\langle x, y \rangle \rightarrow \dots \rightarrow (y \triangleleft \text{task})^n.\text{Split}\langle x, y \rangle$$

where $(y \triangleleft \text{task})^n$ denotes n subsequent $y \triangleleft \text{task}$ prefixes, using repeated applications of [R-CHOICE] and [R-BUFFER]. Notice how the messages pile up as the process reduces and also that, at any time, the process may reduce to $(y \triangleleft \text{task})^n.y \triangleleft \text{stop}. \text{Gather}\langle x, y \rangle$ which is weakly terminating. That is, $\text{Split}\langle x, y \rangle$ is fairly terminating by Theorem 4. We also have

$$(y)(\text{Split}\langle x, y \rangle \mid \text{Worker}\langle y \rangle) \rightarrow^* (y)((y \triangleleft \text{task})^n.\text{Split}\langle x, y \rangle \mid (y \triangleleft \text{res})^m.\text{Worker}\langle y \rangle)$$

for every n and m , indicating that Split has produced $m + n$ tasks and Worker has consumed only m of them. Since Split can always send `stop` and consume all the `res` messages produced by Worker , we also have

$$(y)((y \triangleleft \text{task})^n.\text{Split}\langle x, y \rangle \mid (y \triangleleft \text{res})^m.\text{Worker}\langle y \rangle) \rightarrow^* x \triangleleft \text{resp.done} \rightarrow$$

indicating that $(y)(\text{Split}\langle x, y \rangle \mid \text{Worker}\langle y \rangle)$ too is fairly terminating by Theorem 4. \square

3 Asynchronous Session Types

3.1 Syntax

Session types are generated by the productions below

$$\text{Session type} \quad S, T ::= \mathbf{1} \mid \perp \mid \oplus\{\mathbf{a}_i : S_i\}_{i \in I} \mid \&\{\mathbf{a}_i : S_i\}_{i \in I} \mid S \otimes T \mid S \wp T$$

and adhere to the usual interpretation given to propositions of multiplicative additive linear logic (MALL) [10, 45]: the constants $\mathbf{1}$ and \perp describe processes respectively sending and receiving a termination signal; the additive connectives $\oplus\{\mathbf{a}_i : S_i\}_{i \in I}$ and $\&\{\mathbf{a}_i : S_i\}_{i \in I}$ describe processes sending and receiving a tag \mathbf{a}_i and then behaving according to S_i ; the multiplicative connectives $S \otimes T$ and $S \wp T$ describe processes exchanging a channel of type S and then behaving according to T .

Compared to the usual linear logic propositions, we observe the following differences:

- The additive connectives are n -ary instead of binary and make use of explicit tags for improved generality and readability. In each additive connective $\oplus\{\mathbf{a}_i : S_i\}_{i \in I}$ and $\&\{\mathbf{a}_i : S_i\}_{i \in I}$ we assume that the set I is finite and that the tags \mathbf{a}_i are pairwise disjoint.
- The additive constants $\mathbf{0} \stackrel{\text{def}}{=} \oplus\{\}$ and $\top \stackrel{\text{def}}{=} \&\{\}$ are defined as degenerate (empty) versions of the additive connectives instead of being built-in.
- The productions shown above are meant to be interpreted *coinductively*. That is, we consider session types the possibly infinite *regular* trees built using the above productions. We define possibly infinite session types as solutions of equations of the form $S = \dots$ where the metavariable S may occur (guarded) on the right hand side of “=”. It is a known fact that every such finite system of equations admits a unique regular solution [17].

The *dual* of a session type S , denoted by S^\perp , describes the mirrored protocol of S and is corecursively defined by following equations:

$$\begin{aligned} \mathbf{1}^\perp &= \perp & \oplus\{\mathbf{a}_i : S_i\}_{i \in I}^\perp &= \&\{\mathbf{a}_i : S_i^\perp\}_{i \in I} & (S \otimes T)^\perp &= S^\perp \wp T^\perp \\ \perp^\perp &= \mathbf{1} & \&\{\mathbf{a}_i : S_i\}_{i \in I}^\perp &= \oplus\{\mathbf{a}_i : S_i^\perp\}_{i \in I} & (S \wp T)^\perp &= S^\perp \otimes T^\perp \end{aligned}$$

We say that the session types of the form $\mathbf{1}$, $\oplus\{\mathbf{a}_i : S_i\}_{i \in I}$ and $S \otimes T$ are *positive*, whereas the session types of the form \perp , $\&\{\mathbf{a}_i : S_i\}_{i \in I}$ and $S \wp T$ are *negative*. Positive session types describe protocols that begin with an *output action*, whereas negative session types describe protocols that begin with an *input action*. We write $\text{pos}(S)$ and $\text{neg}(S)$ to state that S is positive and negative, respectively.

■ **Table 3** Labelled transition system for asynchronous session types.

$\frac{}{\mathbf{1} \xrightarrow{!*} \mathbf{1}}$	$\frac{}{S \otimes T \xrightarrow{!S} T}$	$\frac{T \xrightarrow{? \sigma} T'}{S \otimes T \xrightarrow{? \sigma} S \otimes T'}$
$\frac{}{\perp \xrightarrow{? *} \perp}$	$\frac{}{S \wp T \xrightarrow{? S} T}$	$\frac{T \xrightarrow{! \sigma} T'}{S \wp T \xrightarrow{! \sigma} S \wp T'}$
$\frac{}{\oplus\{\mathbf{a}_i : S_i\}_{i \in I} \xrightarrow{! \mathbf{a}_k} S_k}$	$\frac{\forall i \in I : S_i \xrightarrow{? \sigma} T_i}{\oplus\{\mathbf{a}_i : S_i\}_{i \in I} \xrightarrow{? \sigma} \oplus\{\mathbf{a}_i : T_i\}_{i \in I}}$	$\frac{\exists k \in I : S_k \xrightarrow{? \sigma} T_k}{\oplus\{\mathbf{a}_i : S_i\}_{i \in I} \xrightarrow{? \sigma} \oplus\{\mathbf{a}_i : T_i\}_{i \in I}}$
$\frac{}{\&\{\mathbf{a}_i : S_i\}_{i \in I} \xrightarrow{? \mathbf{a}_k} S_k}$	$\frac{\forall i \in I : S_i \xrightarrow{! \sigma} T_i}{\&\{\mathbf{a}_i : S_i\}_{i \in I} \xrightarrow{! \sigma} \&\{\mathbf{a}_i : T_i\}_{i \in I}}$	$\frac{\exists k \in I : S_k \xrightarrow{! \sigma} T_k}{\&\{\mathbf{a}_i : S_i\}_{i \in I} \xrightarrow{! \sigma} \&\{\mathbf{a}_i : T_i\}_{i \in I}}$

3.2 Fair Asynchronous Semantics

We define the labelled transition system (LTS) for session types using the rules in Table 3. Labels of the transition system can be of the form $? \sigma$ (input of a message of type σ) or $! \sigma$ (output of a message of type σ) where σ is a *message type* of the form $*$ (the type of a termination signal), \mathbf{a} (the singleton type of the tag \mathbf{a}) or S (the session type of a channel). Hereafter we use σ and τ to range over message types and α and β to range over labels.

Before we describe the rules in detail, we must point out two unusual but important aspects of the LTS. First of all, the LTS is specified as a *Generalized Inference System* (GIS for short [1, 18]). A GIS consists of two sets of rules, those that are meant to be interpreted coinductively (the singly-lined rules in Table 3) and those that are meant to be interpreted inductively (the singly-lined rules plus the doubly-lined rules in Table 3). If we call $\xrightarrow{\alpha}_{\text{ind}}$ the relation that is defined by the inductive part of the GIS, then the actual relation $\xrightarrow{\alpha}$ being defined is the largest one included in $\xrightarrow{\alpha}_{\text{ind}}$ that satisfies the singly-lined rules in Table 3. The interested reader may refer to the literature for a thorough presentation of GIS [1, 18], but the examples we are about to discuss should suffice to clarify the nature of transitions.

The other unusual aspect of the LTS is that a transition $S \xrightarrow{\alpha} T$ is not an indication of what a process complying with S necessarily does, but rather of what the process is allowed or able to do. In particular, a transition $S \xrightarrow{! \sigma} T$ means that a process complying with S is allowed to output a message of type σ , even though S may be negative. We say that this is an *early output transition* because it describes an output that may occur ahead of time. Dually, a transition $S \xrightarrow{? \sigma} T$ means that a process complying with S is eventually able to input a message of type σ , even though S may be positive. We say that this is a *late input transition* because it describes the consumption of a message that may occur later on.

The axioms [MUST- $*$] are used to derive what we call *immediate transitions*. These are the expected transitions of session types, whereby no input is late and no output is early. For technical reasons it is convenient to have transitions also for $\mathbf{1}$ and \perp . In this way we do not have to distinguish $\mathbf{1}$ and \perp from other non-terminated protocols when defining our notions of generalized duality (Definition 8) and subtyping (Definition 12).

24:10 Fair Termination of Asynchronous Binary Sessions

The rules [MAY-*] deal with late inputs and early outputs. As an example, consider the session type $S = \&\{a : \oplus\{c : T\}\}$ for which we may derive the transition sequences

$$S \xrightarrow{?a} \oplus\{c : T\} \xrightarrow{!c} T \quad \text{and} \quad S \xrightarrow{!c} \&\{a : T\} \xrightarrow{?a} T$$

The sequence on the left, obtained by [MUST-&] followed by [MUST- \oplus], is an ordinary one: actions are performed according to the syntactic structure of the type. The sequence on the right, obtained using [MUST- \oplus], [MAY- \oplus] and [MUST-&], is peculiar to the asynchronous setting: it describes a situation in which the output $!c$ may be performed earlier than the input $?a$. Since communication is asynchronous and c is going to be sent anyway, a process complying with S might choose to send it *early*, before waiting for a . Note that this is just a possibility: a process strictly adhering to the transition sequence on the left would still comply with S .

A dual reasoning applies to late inputs. If we consider the type $S' = \oplus\{a : \&\{c : T'\}\}$ we may derive the transition sequences

$$S' \xrightarrow{!a} \&\{c : T'\} \xrightarrow{?c} T' \quad \text{and} \quad S' \xrightarrow{?c} \oplus\{a : T'\} \xrightarrow{!a} T'$$

Again, the sequence on the left is standard. The sequence on the right, obtained using [MUST-&], [MAY-&] and [MUST- \oplus], says that a process complying with S' is able to consume a c message, even though this will happen only after the process has sent a . The fact that the input action is late cannot cause issues, since both the outgoing a and the incoming c are sent asynchronously with a non-blocking operation.

As it is clear looking at the rules [MAY- \oplus] and [MAY-&], late inputs and early outputs concerning a branching session type must be derivable for every branch: a message may be sent early (before an input) only if it is independent of the input; a message may be received late (after an output) only if it is independent of the output. For example, if we take $S = \&\{a : \oplus\{c : S_1\}, b : \oplus\{c : S_2, d : S_3\}\}$ then we can derive

$$S \xrightarrow{!c} \&\{a : S_1, b : S_2\} \quad \text{and also} \quad S \xrightarrow{?b} \oplus\{c : S_2, d : S_3\} \xrightarrow{!d} S_3 \quad \text{but not} \quad S \xrightarrow{!d} S'$$

no matter what S' could be. A process complying with S may send c early, before receiving either a or b , since the output of c is allowed regardless of the input. On the contrary, the output d is allowed only if the input is b and so it cannot be anticipated before the input. Symmetrically, if we consider the session type $T = \oplus\{a : \&\{c : T_1\}, b : \&\{c : T_2, d : T_3\}\}$, we may derive

$$T \xrightarrow{?c} \oplus\{a : T_1, b : T_2\} \quad \text{and also} \quad T \xrightarrow{!b} \&\{c : T_2, d : T_3\} \xrightarrow{?d} T_3 \quad \text{but not} \quad T \xrightarrow{?d} T'$$

The (late) input transition c is enabled because the process is able to receive c regardless of the tag a or b that it sends. On the contrary, the input transition on d is enabled only if the process sends b .

A subtler case of late input is illustrated by the session type $S_1 = \oplus\{a : S_1, b : \&\{c : S_2\}\}$, which describes the behavior of a process that sends an arbitrary number of a 's or a b and then waits for a c . Since the singly-lined rules in Table 3 are interpreted inductively, we can derive $S_1 \xrightarrow{?c} S'_1$ where $S'_1 = \oplus\{a : S'_1, b : S_2\}$ by means of the following *infinite* derivation:

$$\frac{\begin{array}{c} \vdots \\ \frac{S_1 \xrightarrow{?c} S'_1}{S_1 \xrightarrow{?c} S'_1} \text{ [MAY-}\oplus\text{]} \quad \frac{\quad}{\&\{c : S_2\} \xrightarrow{?c} S_2} \text{ [MUST-}\&\text{]} \end{array}}{S_1 \xrightarrow{?c} S'_1} \text{ [MAY-}\oplus\text{]} \quad (4)$$

The transition $S_1 \xrightarrow{?c} S'_1$ says that a process complying with S_1 performs a late input of c . Compared to the other examples of late inputs, this one looks more questionable and for a good reason: it may be the case that a process sending an *infinite* sequence of a 's complies with S_1 . So, claiming that such process is eventually able to input a c may lead to a message remaining orphan. In this work, however, we *assume* that a well-typed process complying with S_1 will also be *fair*, in the sense that it will eventually stop sending a 's and will send the b ensuring that the c message is consumed. The same assumption is made, in dual form, for early outputs. For example, the session type $T_1 = \&\{a : T_1, b : \oplus\{c : T_2\}\}$ performs the early output transition $T_1 \xrightarrow{!c} T'_1$ where $T'_1 = \&\{a : T'_1, b : T_2\}$. That is, a process complying with T_1 is allowed to send c early because it is guaranteed to eventually receive a b message.

The infinite derivation that we have just shown in (4) reminds us that we must be careful in the use of coinduction for defining the LTS. Indeed, an unconstrained use of coinduction would allow us to derive early/late transitions that do not correspond to any “real” action of a session type, solely using [MAY-*] rules. Consider for example the infinite session type $S = \oplus\{a : S\}$, which allows sending a neverending sequence of a 's. It is easy to derive $S \xrightarrow{?a} S$ for every a by an infinite derivation consisting of [MAY- \oplus] rules only, despite the protocol described by S does *not* enable any input transition! Of course we want to make sure that, whenever we derive a late/early transition, this is justified by the use of at least one [MUST-*] rule somewhere in the derivation. This is the reason why the LTS is defined through a GIS and not simply by the (coinductively interpreted) singly-lined rules in Table 3. On the one hand, we want to make sure that that a late/early transition is enabled along *every* branch of a session type. This is an *invariant property* enforced by the [MAY-*] rules. On the other hand, we want to make sure that there exists at least one branch along which the transition eventually originates for real. This is a *well-founded property* enforced by the [MUST-*] rules. As it has already been observed elsewhere [15], GIS are a convenient way of defining relations like $\xrightarrow{\alpha}$ that mix invariant and well-founded properties at the same time. The effect of defining the LTS as a GIS is that, whenever we build a (possibly infinite) derivation for $S \xrightarrow{\alpha} T$ using the singly-lined rules, we must also be able to find, *for each* judgment $S_i \xrightarrow{\alpha_i} T_i$ in this derivation, a *finite* derivation for $S_i \xrightarrow{\alpha_i}_{\text{ind}} T_i$. In (4) this is achieved easily, as shown below:

$$\frac{\frac{}{\&\{c : S_2\} \xrightarrow{?c}_{\text{ind}} S_2} \text{[MUST-}\&]}{S_1 \xrightarrow{?c}_{\text{ind}} S'_1} \text{[FAIR-}\oplus]$$

Notice the key role of [FAIR- \oplus] in building this *finite* derivation for $S_1 \xrightarrow{?c}_{\text{ind}} S'_1$. Since S_1 is an *infinite* session type, we would not be able to find a finite derivation for $S_1 \xrightarrow{?c}_{\text{ind}} S'_1$ if we insisted on using [MAY- \oplus] only, since this rule requires us to derive the late input transition in *every* branch of the session type. Instead, according to [FAIR- \oplus] it suffices to find *one* branch along which we can eventually derive the late input transition directly. In the literature on GIS the doubly-lined rules are called *corules*. Here, we have called them [FAIR- \oplus] and [FAIR- $\&$] because they somehow capture the fairness assumption of the LTS: whenever we derive late/early transitions in a looping session type like S_1 and T_1 , the fairness assumption makes sure that the conversation eventually follows a branch leading out of the loop.

There is one exception to what we have just said about transitions being eventually derived by [MUST-*] rules. Recalling that $\mathbf{0} = \oplus\{\}$ and $\top = \&\{\}$, by [MAY- \oplus] and [MAY- $\&$] we can easily derive $\mathbf{0} \xrightarrow{?a} \mathbf{0}$ and $\top \xrightarrow{!a} \top$ for every a . These derivations are trivially valid for the GIS since they are finite. We might be tempted to flag these cases as pathological. After

all, the protocol $\mathbf{0}$ describes an unrealistic process that is able to *input anything* and the protocol \top describes a uncontrollable process that may *output anything*. While it is true that such behaviors are practically useless, we will see in Section 4 that the derivability of these transitions for $\mathbf{0}$ and \top has an important impact in the resulting subtyping relation, for which $\mathbf{0}$ and \top will play the role of least and the greatest element.

Hereafter, we let φ and ψ range over finite sequences of labels, we write $\xrightarrow{\alpha_1 \cdots \alpha_n}$ for the composition $\xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$, we write $S \xrightarrow{\alpha}$ if $S \xrightarrow{\alpha} T$ for some T and $S \not\xrightarrow{\alpha}$ if not $S \xrightarrow{\alpha}$.

3.3 Properties of Asynchronous Session Types

To substantiate the claim that each transition derivable by the GIS is “real” (*i.e.* it originates from the syntax of the session type), we prove that every input/output transition can be derived by the application of an axiom in Table 3 after every maximal, strongly fair sequence of immediate outputs/inputs. Strong fairness is a weaker assumption implied by the full fairness of Definition 2 [44]. Formally, a (possibly infinite) sequence of transitions $S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \cdots$ is *strongly fair* [26, 44] if, whenever some S' occurs infinitely often in the sequence $S_0 S_1 \cdots$ and $S' \xrightarrow{\alpha} T'$, then also T' occurs infinitely often in the same sequence. Intuitively, a strongly fair sequence of transitions does not discriminate those transitions that are enabled infinitely often. For example, if $S = \oplus\{\mathbf{a} : S, \mathbf{b} : T\}$, the infinite sequence $S \xrightarrow{!a} S \xrightarrow{!a} \cdots$ of transitions is strongly unfair, because the transition $S \xrightarrow{!b} T$ is infinitely often enabled but never performed. On the contrary, if $S = \oplus\{\mathbf{a} : S\}$, then the infinite sequence $S \xrightarrow{!a} S \xrightarrow{!a} \cdots$ of transitions is strongly fair.

► **Theorem 6.** *Let α be the label of an input/output transition. Then $S \xrightarrow{\alpha}$ if and only if every maximal, strongly fair sequence of immediate output/input transitions $S \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \cdots$ is finite and ends in some T such that $T \xrightarrow{\alpha}$ is derivable by an axiom in Table 3.*

Note that, in the statement of Theorem 6, we cannot require the transition $T \xrightarrow{\alpha}$ to be immediate because of the phony early/late transitions enabled by $\mathbf{0}$ and \top .

We are accustomed to think that session types are used to describe race-free interactions, but we have seen examples of session types that simultaneously enable both input and output transitions. When this happens, such transitions are independent and do not interfere with each other. We formalize this fact by establishing a diamond property for session types.

► **Proposition 7.** *If $S \xrightarrow{? \sigma} S'$ and $S \xrightarrow{! \tau} S''$, then $S' \xrightarrow{! \tau} T$ and $S'' \xrightarrow{? \sigma} T$ for some T .*

4 Fair Asynchronous Subtyping

We define the subtyping relation for asynchronous session types in three steps. First of all, we formalize what we mean by *correct asynchronous composition* between the session types describing the protocols implemented by two processes using the two endpoints of a session. In many session type systems, this notion coincides with session type duality. Since we have to take asynchrony into account, duality alone is too strict so we need a notion of correct composition using the LTS defined in Section 3. Once this notion is in place, asynchronous subtyping can be defined as the relation that preserves correct asynchronous composition. This relation is “sound” by definition, but its properties can be intrinsically difficult to grasp. The final step will be to provide a *precise* (*i.e.* sound and complete) alternative characterization of asynchronous subtyping that sheds light on its properties.

The definition of correct asynchronous composition is given below.

► **Definition 8.** We say that \mathcal{R} is a correct asynchronous composition if $(S, T) \in \mathcal{R}$ implies:

1. either $\text{pos}(S)$ or $\text{pos}(T)$;
2. if $S \xrightarrow{! \sigma} S'$ and $\sigma \in \{*\} \cup \text{Tags}$, then $T \xrightarrow{? \sigma} T'$ and $(S', T') \in \mathcal{R}$;
3. if $T \xrightarrow{! \sigma} T'$ and $\sigma \in \{*\} \cup \text{Tags}$, then $S \xrightarrow{? \sigma} S'$ and $(S', T') \in \mathcal{R}$;
4. if $S \xrightarrow{! S_1} S_2$, then $T \xrightarrow{? T_1} T_2$ and $(S_1, T_1) \in \mathcal{R}$ and $(S_2, T_2) \in \mathcal{R}$;
5. if $T \xrightarrow{! T_1} T_2$, then $S \xrightarrow{? S_1} S_2$ and $(S_1, T_1) \in \mathcal{R}$ and $(S_2, T_2) \in \mathcal{R}$.

We write \bowtie for the largest correct asynchronous composition.

In words, Items 2–5 state that (S, T) forms a correct composition if, whenever one of the two types performs a (possibly early) output transition $! \sigma$, the other type is able to respond with a (possibly late) compatible input transition $? \tau$ and the continuations remain correct. In general this is not enough to guarantee progress because early outputs are only allowed but not mandatory. For example, the types $\&\{\mathbf{a} : \oplus\{\mathbf{b} : \mathbf{1}\}\}$ and $\&\{\mathbf{b} : \oplus\{\mathbf{a} : \perp\}\}$ satisfy Items 2 and 3, but two processes strictly adhering to these protocols (*i.e.* without performing early outputs) would starve. Item 1 requires that at least one of the two types is positive, namely that at least on one side of the session the outputs are guaranteed to be immediate. This is enough to ensure progress. For example, we have $\&\{\mathbf{a} : \oplus\{\mathbf{b} : \mathbf{1}\}\} \bowtie \oplus\{\mathbf{a} : \&\{\mathbf{b} : \perp\}\}$ as well as $\oplus\{\mathbf{b} : \&\{\mathbf{a} : \mathbf{1}\}\} \bowtie \oplus\{\mathbf{a} : \&\{\mathbf{b} : \perp\}\}$.

It is easy to see that \bowtie is symmetric and that duality implies correctness:

► **Proposition 9.** $S^\perp \bowtie S$ holds for every session type S .

Other properties of \bowtie are more surprising. For example, if $S = \&\{\mathbf{a} : S, \mathbf{b} : \oplus\{\mathbf{c} : \mathbf{1}\}\}$ and $R = \oplus\{\mathbf{a} : R\}$, we have that $S \bowtie R$ does *not* hold because of the early output transition $S \xrightarrow{! \mathbf{c}}$ to which R is unable to respond. The lack of compatibility between S and R is due to the fact that (the process behaving as) S makes a *fairness assumption* on the behavior of the process it is interacting with. More precisely, a process complying with S assumes that sooner or later a \mathbf{b} message will be received, finally enabling the output of \mathbf{c} . In anticipation of this, the process may decide to perform an early output of \mathbf{c} , but in doing so it would generate an orphan message when interacting with another process adhering to R , which is not honoring this fairness assumption. It is also easy to see that $\mathbf{0} \bowtie S$ holds for every S and that $\top \bowtie S$ implies $S = \mathbf{0}$. These properties of $\mathbf{0}$ and \top follow directly from the $[\text{MAY-}\oplus]$ and $[\text{MAY-}\&]$ rules that we have commented in Section 3.

► **Example 10.** Let $! \mathbf{a}.S$ stand for $\oplus\{\mathbf{a} : S\}$ and $(! \mathbf{a})^n.S$ stand for $! \mathbf{a} \dots ! \mathbf{a}.S$ with n $! \mathbf{a}$ prefixes. Consider the session types $S = \oplus\{\text{task} : S, \text{stop} : T\}$ and $T = \&\{\text{res} : T, \text{stop} : \perp\}$ and $U = \&\{\text{task} : \oplus\{\text{res} : U\}, \text{stop} : \oplus\{\text{stop} : \mathbf{1}\}\}$ which respectively describe the behaviors of *Split*, *Gather* and *Worker* of Section 1 on the channel y . It is easy to establish that

$$S \stackrel{\text{def}}{=} \{(S, (! \text{res})^n.U) \mid n \in \mathbb{N}\} \cup \{(T, (! \text{res})^n.!\text{stop}.\mathbf{1}) \mid n \in \mathbb{N}\} \cup \{(\perp, \mathbf{1})\}$$

is a correct asynchronous composition hence $S \bowtie U$. ┘

We can now define fair asynchronous subtyping semantically using *Liskov's substitution principle* [38] where the property being preserved is session correctness (Definition 8).

► **Definition 11** (fair asynchronous subtyping). We say that S is a fair asynchronous subtype (or just subtype) of T , notation $S \leq T$, if $R \bowtie T$ implies $R \bowtie S$ for every R .

Paraphrasing, this definition says that a process using a channel x according to T can be safely replaced by a process using x according to S when S is a subtype of T . Indeed, the peer process, which is assumed to use the same channel x according to some session type R such that $R \bowtie T$, will still interact correctly after the substitution has taken place.

A few subtyping relations are easy to figure out. For example, we have $\mathbf{0} \leq S$ and $S \leq \top$ for every S because of the properties of $\mathbf{0}$ and \top that we have pointed out above. It is also easy to see that $\oplus\{\mathbf{a} : \&\{\mathbf{b} : S\}\} \leq \&\{\mathbf{b} : \oplus\{\mathbf{a} : S\}\}$ holds in an asynchronous setting. After all, the process that behaves according to $\oplus\{\mathbf{a} : \&\{\mathbf{b} : S\}\}$ is sending a tag \mathbf{a} that also the process that behaves according to $\&\{\mathbf{b} : \oplus\{\mathbf{a} : S\}\}$ *may* anticipate. Note that the inverse relation $\&\{\mathbf{b} : \oplus\{\mathbf{a} : S\}\} \leq \oplus\{\mathbf{a} : \&\{\mathbf{b} : S\}\}$ does not hold, despite the fact that $\oplus\{\mathbf{a} : \&\{\mathbf{b} : S\}\}$ and $\&\{\mathbf{b} : \oplus\{\mathbf{a} : S\}\}$ perform exactly the same transitions, because $\&\{\mathbf{a} : \oplus\{\mathbf{b} : S^\perp\}\}$ forms a correct asynchronous composition with $\oplus\{\mathbf{a} : \&\{\mathbf{b} : S\}\}$ but not with $\&\{\mathbf{b} : \oplus\{\mathbf{a} : S\}\}$ (Item 1 of Definition 8 is violated).

The above notion of subtyping is sound “by definition”, but provides little information concerning the shape of related session types. To compensate for this problem, we also give a sound and complete coinductive characterization of \leq , which relates directly to Definition 8.

► **Definition 12** (coinductive asynchronous subtyping). *We say that \mathcal{S} is a coinductive asynchronous subtyping if $(S, T) \in \mathcal{S}$ implies:*

1. *either $\text{pos}(S)$ or $\text{neg}(T)$;*
2. *if $T \xrightarrow{? \sigma} T'$ and $\sigma \in \{*\} \cup \text{Tags}$, then $S \xrightarrow{? \sigma} S'$ and $(S', T') \in \mathcal{S}$;*
3. *if $S \xrightarrow{! \sigma} S'$ and $\sigma \in \{*\} \cup \text{Tags}$, then $T \xrightarrow{! \sigma} T'$ and $(S', T') \in \mathcal{S}$;*
4. *if $T \xrightarrow{? T_1} T_2$, then $S \xrightarrow{? S_1} S_2$ and $(S_1, T_1) \in \mathcal{S}$ and $(S_2, T_2) \in \mathcal{S}$;*
5. *if $S \xrightarrow{! S_1} S_2$, then $T \xrightarrow{! T_1} T_2$ and $(S_1, T_1) \in \mathcal{S}$ and $(S_2, T_2) \in \mathcal{S}$.*

Items 2–5 of Definition 12 specify the expected requirements for a session subtyping relation: every input transition of the supertype T must be matched by an input transition of the subtype S and the corresponding continuations should still be related by subtyping; dually, every output transition of the subtype S must be matched by an output transition of the supertype T and the corresponding continuations should still be related by subtyping. Interestingly, these items are essentially the same found in analogous characterizations of *synchronous subtyping for session types* [28], modulo the different orientation of \leq due to our viewpoint based on the substitution of processes rather than on the substitution of channels.¹ However, the clauses of Definition 12 are not mutually exclusive, because the same session type may perform both input and output transitions. Also, session types related by subtyping need not start with the same type constructor. In this respect, Item 1 makes sure that the smaller session type can only anticipate (and not postpone) outputs, as argued above.

Definition 12 is a sound and complete characterization of \leq .

► **Theorem 13.** *\leq is the largest coinductive asynchronous subtyping.*

Using Definition 12 and Theorem 13 we can prove some significant properties of \leq :

- (input contravariance) $\&\{\mathbf{a}_i : S_i\}_{i \in I} \leq \&\{\mathbf{a}_i : S_i\}_{i \in J}$ if $J \subseteq I$ by Item 2;
- (output covariance) $\oplus\{\mathbf{a}_i : S_i\}_{i \in I} \leq \oplus\{\mathbf{a}_i : S_i\}_{i \in J}$ if $I \subseteq J$ by Item 3;
- (output anticipation) $\oplus\{\mathbf{a}_j : \&\{\mathbf{b}_i : S_{ij}\}_{i \in I}\}_{j \in J} \leq \&\{\mathbf{b}_i : \oplus\{\mathbf{a}_j : S_{ij}\}_{j \in J}\}_{i \in I}$. Note that the inverse relation does not hold, despite these two session types have exactly the same transitions, because of Item 1.

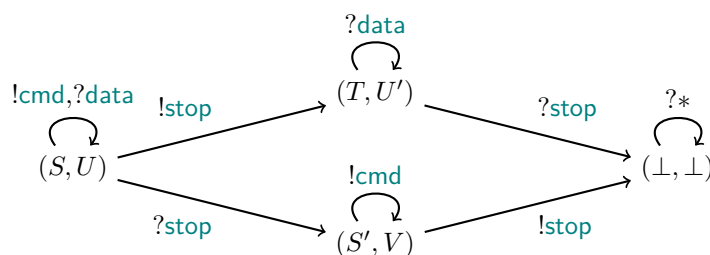
There are exceptions to output covariance and input contravariance when one of the two types specifies a non-terminating protocol. The next example illustrates one of such cases.

¹ The interested reader may refer to Gay [27] for a comparison of the two viewpoints.

► **Example 14.** Consider $S = \&\{a : S, b : \oplus\{c : 1\}\}$ and $T = \&\{a : T\}$. Despite S is a subtype of T for other session subtypings [28, 39, 11, 8], we have $S \not\leq T$ because $S \xrightarrow{!c}$ and $T \not\xrightarrow{!c}$. To see the reason why admitting this relation could cause a problem, consider a process that complies with the protocol $R = \oplus\{a : R\}$. Such a process would output infinitely many a 's and would not expect to input anything. Still, if we consider $S' = \&\{a : S', b : 1\}$ we have that $\oplus\{c : S'\} \leq S$ holds. That is, a process complying with $\oplus\{c : S'\}$ performs immediately the early output in S . By transitivity of \leq (whose validity is implied by Theorem 13), we would also have $\oplus\{c : S'\} \leq T$. Now the problem is clear: $R \bowtie T$ holds, but a process complying with R would not be able to handle the incoming c message if we composed it with a process complying with $\oplus\{c : S'\}$. \perp

The cases in which co/contra variance does not hold have no impact on the typeability of CaP processes: since our type system ensures the fair termination of well-typed processes, protocols like T in Example 14 are not inhabited. We will see in Section 7 that \leq includes other subtyping relations supporting full co/contra variance for the family of fairly terminating session types, those that always allow the protocol to end.

► **Example 15.** We borrow a scenario from Bravetti, Lange and Zavattaro [8] to showcase an interesting example of fair asynchronous subtyping. Imagine a system made of a ground station and a satellite such that, at each flyby, the satellite sends data from the previous orbit and receives commands to execute in the next one. In principle, the ground station should follow the protocol $U = \&\{data : U, stop : V\}$ where $V = \oplus\{cmd : V, stop : \perp\}$. However, since the flyby window may be short, it makes sense to implement the ground station so that it communicates with the satellite in full duplex by anticipating the output of the commands. In this case, the ground station follows the protocol $S = \oplus\{cmd : S, stop : T\}$ where $T = \&\{data : T, stop : \perp\}$. Is this implementation correct? We can answer in the affirmative by proving $S \leq U$ with the diagram below, which represents a coinductive asynchronous subtyping containing the pair (S, U) . In the diagram we also use the types $S' = \oplus\{task : S', stop : \perp\}$ and $U' = \&\{res : U', stop : \perp\}$.



Note that S allows for the anticipation of an unbounded number of outputs before an unbounded number of inputs. \perp

We now list a few properties of \leq . First of all, we establish that \leq is closed by duality.

► **Proposition 16.** *If $S \leq T$, then $T^\perp \leq S^\perp$.*

Proof. By Theorem 13 it suffices to show that if \mathcal{S} is a coinductive asynchronous subtyping, then so is $\mathcal{S}^\perp \stackrel{\text{def}}{=} \{(T^\perp, S^\perp) \mid (S, T) \in \mathcal{S}\}$. This follows immediately from Definition 12. \blacktriangleleft

Then, we show how to characterize \bowtie solely in terms of \leq .

► **Theorem 17.** *$S \bowtie T$ if and only if $S \leq T^\perp$*

Finally, we observe that similarly to the other asynchronous session type theories correct composition is undecidable. The source of undecidability follows from the possibility to use communicating buffers to model unbounded memories like tapes of Turing Machines [36] or queues of Queue Machines [6].

► **Theorem 18.** *Given two types S and T , the problem of checking $S \bowtie T$ is undecidable.*

As a direct consequence of Theorems 17 and 18, we have that checking $S \leq T$ is also an undecidable problem. Despite these negative results, the coinductive characterizations of correctness (Definition 8) and subtyping (Definition 12) are useful to prove that these relations hold in specific cases, as done in Examples 10 and 15.

5 Type System

In this section we describe the type system for CaP ensuring that well-typed processes weakly terminate, besides being free from communication errors and deadlocks. Theorem 4 then allows us to conclude that the very same type system also ensures fair termination when we make the fairness assumption stated in Definition 2. The type system is based on the proof rules of MALL but also includes typing rules for the non-logical process forms, namely termination, process invocation and non-deterministic choice.

5.1 Measuring Processes and Type Annotations

It is a known fact that infinitary proof systems for linear logic (*e.g.* μ MALL [3, 25]) require validity conditions to enjoy the cut elimination property. As a consequence, the infinitary type system for CaP requires validity conditions to ensure the (weak) termination property. We implement these validity conditions using the technique of Dagnino and Padovani [19] and decorate types and typing judgments with quantitative information that estimates the amount of effort required to terminate a process. It is natural to measure such effort in terms of *number of reductions* of that process (Table 2). Since most reductions arise from session interactions and each session interaction involves a process outputting a message on a channel (with a *positive* type) and a process inputting a message from the same channel (with a *negative* type) we count the number of process forms representing outputs to establish the number of reductions that are necessary to normalize a process. In the presence of branching processes (those whose behavior depends on a tag received from a channel), we can tentatively compute an upper bound for terminating *each* branch.

Let us use the definitions in Section 1 to walk through this approach on increasingly complex examples. To start, consider the process *Gather*, which we repeat here for convenience:

$$Gather(x, y) = y \triangleright \{ \text{res} : Gather\langle x, y \rangle, \text{stop} : \text{wait } y.x \triangleleft \text{resp.close } x \}$$

Now suppose that n is the measure associated with *Gather* and note that *Gather* is a branching and recursive process. The **res** branch does not contain any output actions, whereas the **stop** branch performs two outputs on x . Therefore, n should satisfy the relations $n \geq n$ (for the **res** branch) and $n \geq 2$ (for the **stop** branch) whose least solution is $n = 2$. That is, *Gather* weakly terminates by performing at most two outputs. Note that the actual number of interactions performed by *Gather* depends on the number of received **res** messages and may be larger than 2. However, these received messages are accounted for in the measure of the sender, while the measure 2 we give to *Gather* only accounts for the messages sent by *Gather*.

If we now consider the process

$$\mathit{Split}(x, y) = y \triangleleft \mathit{task}.\mathit{Split}\langle x, y \rangle \oplus y \triangleleft \mathit{stop}.\mathit{Gather}\langle x, y \rangle$$

we have to measure a non-deterministic choice. Since non-deterministic choices are performed autonomously by a process and we are interested in proving that processes *weakly* terminate, we can measure a non-deterministic choice by considering the branch with the least measure and the additional reduction due to [R-CHOICE]. So, in this case, the measure of Split – say m – must satisfy the equation $m \geq 1 + \min\{1 + m, 3\}$ where $1 + m$ is the measure of the left branch and 3 is the (least) measure of the right branch, having already established that Gather can be given measure 2. The least least solution of this constraint is $m = 4$.

A more challenging process to measure is

$$\mathit{Worker}(y) = y \triangleright \{\mathit{task} : y \triangleleft \mathit{res}.\mathit{Worker}\langle y \rangle, \mathit{stop} : y \triangleleft \mathit{stop.close } y\}$$

in which each task input is immediately followed by a res output. If we try to measure Worker following the same reasoning applied to Gather , we end up looking for some n satisfying the system of inequations $n \geq 1 + n$ (for the task branch) and $n \geq 2$ (for the stop branch). Since no n satisfies these inequations, we have to refine our measuring strategy or else Worker would end up being ill typed. This is precisely what happens in some type systems ensuring the fair termination of sessions [14, 12] which is unfortunate because Worker is not ill behaved after all: the fairness assumption we are making ensures that the number of tasks will be finite, albeit unbounded. Likewise, the number of results will be finite but unbounded as well. The problem is that we are unable to find a *single* measure for Worker that accounts for all possibilities.

We refine our measuring technique allowing the measure of Worker to *depend* on the tags it receives. We realize this dependency by annotating tags with measures that are charged to senders and discharged from receivers. To see this mechanism at work in the case of Split and Worker , consider the following annotated session types:

$$S = \oplus\{\mathit{task}^1 : S, \mathit{stop}^0 : T\} \tag{5}$$

$$T = \&\{\mathit{res}^0 : T, \mathit{stop}^0 : \perp\} \tag{6}$$

$$U = \&\{\mathit{task}^1 : \oplus\{\mathit{res}^0 : U\}, \mathit{stop}^0 : \oplus\{\mathit{stop}^0 : \mathbf{1}\}\} \tag{7}$$

The annotations in task^1 and stop^0 mean that a process like Split , which complies with S , is *charged* by 1 unit of measure whenever it sends task and by 0 when it sends stop . This measure is charged *in addition* to the cost of the output it is performing and accounts for the cost of sending the result in Worker . Since this cost is already charged on Split , it can be *discharged* from the task branch of Worker when we compute its measure, as shown by U . This way we end up solving for Worker the equations $n \geq n$ (for the task branch) and $n \geq 2$ (for the stop branch) whose least solution is $n = 2$. In principle we have to reconsider the measure we gave to Split to account for the 1 unit that is now charged for each output of task , but since the measure of Split was determined by the stop branch (which has a null annotation) it remains unchanged.

From now on we consider a refinement of the session types presented in Section 3 where tags are annotated with natural numbers. These annotations do not interfere in any way with the properties and characterizations of session types we have presented there and in Section 4, with the proviso that wherever we have written matching message types in Sections 3 and 4 we mean that measures also match. In particular, Definition 8 entails that the amount of measure charged on one side of the session matches the amount of measure discharged from

24:18 Fair Termination of Asynchronous Binary Sessions

■ **Table 4** Typing rules for CaP.

$\frac{}{\text{done} \vdash^n \emptyset} \quad [\text{DONE}]$	$\frac{P \vdash^n x : \bar{S}}{A(\bar{x}) \vdash^{m+n} x : \bar{S}} \quad A(\bar{x}) = P \quad [\text{CALL}]$	$\frac{}{\text{close } x \vdash^{1+n} x : \mathbf{1}} \quad [\mathbf{1}]$	$\frac{P \vdash^n \Gamma}{\text{wait } x.P \vdash^n \Gamma, x : \perp} \quad [\perp]$
$\frac{P \vdash^{n_1} \Gamma \quad Q \vdash^{n_2} \Gamma}{P \oplus Q \vdash^{1+n_k} \Gamma} \quad [\text{CHOICE}]$	$\frac{[\otimes] \quad P \vdash^m \Gamma, y : S \quad Q \vdash^n \Delta, x : T}{x(y)[P].Q \vdash^{1+m+n} \Gamma, \Delta, x : S \otimes T}$	$\frac{[\wp] \quad P \vdash^n \Gamma, y : S, x : T}{x(y).P \vdash^n \Gamma, x : S \wp T}$	
$\frac{[\oplus] \quad P \vdash^n \Gamma, x : S_k}{x \triangleleft \mathbf{a}_k.P \vdash^{1+n+m_k} \Gamma, x : \oplus \{ \mathbf{a}_i^{m_i} : S_i \}_{i \in I}}$	$\frac{[\&] \quad \forall i \in I : P_i \vdash^{n+m_i} \Gamma, x : S_i}{x \triangleright \{ \mathbf{a}_i : P_i \}_{i \in I \cup J} \vdash^n \Gamma, x : \& \{ \mathbf{a}_i^{m_i} : S_i \}_{i \in I}}$		
$\frac{[\text{LINK}] \quad S^\perp \leq T}{x \leftrightarrow y \vdash^{1+n} x : S, y : T}$	$\frac{[\text{CUT}] \quad P \vdash^m \Gamma, x : S \quad Q \vdash^n \Delta, x : T}{(x)(P \mid Q) \vdash^{m+n} \Gamma, \Delta} S \bowtie T$		

the other side of the session (like for S and U above) and the clauses of Definition 12 entail that matching actions in session types related by subtyping carry the same measure. In principle, it would be possible to relax these constraints and allow some variance of measure annotations. Since we do not have concrete examples that take advantage of this further refinement, we spare the additional complexity.

5.2 Typing Rules

The typing rules for CaP are shown in Table 4. Judgments have the form $P \vdash^n \Gamma$ where P is the process being typed, n is its measure and Γ is a *typing context*, namely a partial function that maps channels to types. We let Γ and Δ range over typing contexts, we write \emptyset for the empty context, $x : S$ for the singleton context that maps x to S , and Γ, Δ for the union of Γ and Δ when they have disjoint domains. The typing rules shown in Table 4 are meant to be interpreted coinductively, hence a process is well typed provided there is a (possibly infinite) derivation built using those rules. The structure of the rules is essentially the same of other session type systems based on (classical) linear logic [45, 37, 19], so we will mainly focus on those aspects of the rules that are new or different in our setting.

The rule [DONE] states that the terminated process can have any measure and is well typed only in the empty context.

The rule [CALL] states that a process invocation is well typed in a context $\bar{x} : \bar{S}$ if so is its definition in the same context. Recall that a typing derivation can be infinite in the case of recursive processes and note that the measure of the invocation may be larger than that of its definition, allowing some measure to be discarded from one invocation to the next. This is not strictly necessary for the soundness of type system, but it is sometimes convenient to obtain simpler typing derivations.

The rules [1] and [\perp] deal with session termination in the expected way. The measure of $\text{close } x$ is strictly positive whereas the measure of $\text{wait } x.P$ coincides with that of P since the measure n in a typing judgment $\Gamma \vdash^n P$ only accounts for the outputs performed by P .

The rules $[\otimes]$ and $[\wp]$ deal with the communication of channels. The measure of a channel output $x(y)[P].Q$ accounts for both the measure of P (the process being spawned that uses one end of the fresh session y) and the measure of Q , while the additional unit of measure accounts for the output being performed. The measure of a channel input $x(y).P$ simply coincides with that of the continuation process.

The rules $[\oplus]$ and $[\&]$ deal with the communication of tags. The measure of a tag output $x \triangleleft \mathbf{a}.P$ accounts for the measure of the continuation P and of the measure annotation m associated with \mathbf{a} in the type of x , while the additional unit of measure accounts for the output being performed. The measure of a tag input $x \triangleright \{\mathbf{a}_i : P_i\}_{i \in I}$ is the residual measure of each branch after the measure annotation m_i associated with \mathbf{a}_i in the type of x has been discharged. Note that a tag input may in general provide more branches than those actually occurring in the type of x . As a special case, $[\&]$ also captures the introduction rule for \top in MALL. In particular, we have $x \triangleright \{\} \vdash^n \Gamma, x : \top$ for every n and every Γ .

The rule $[\text{CHOICE}]$ deals with non-deterministic choices in the expected way. The rule does not specify which branch of a non-deterministic choice determines the measure of the whole choice. Since the typeability of a process rests on its measurability, the obvious strategy is to pick the continuation with the smallest measure (*cf.* Section 5.1).

The rule $[\text{LINK}]$ is a generalization of the axiom in MALL. It allows the unification of x and y provided that $S^\perp \leq T$ (or equivalently $T^\perp \leq S$, by Proposition 16) where S is the type of x and T is the type of y . As we will see in Section 6, this formulation of the link typing rule allows us to capture some recurring usage patterns of subtyping. A link has a strictly positive measure since it accounts for one $[\text{R-LINK}]$ reduction.

Finally, the rule $[\text{CUT}]$ rule ensures that the processes P and Q connected by the new session x use the channel correctly requiring $S \bowtie T$ to hold where S is the type of x as used by P and T is the type of x as used by Q . Note that this correctness condition can be equivalently expressed in terms of subtyping with the relation $S \leq T^\perp$ by Theorem 17. The measure of the composition is the combination of the measures of P and Q .

► **Example 19.** Let us build a typing derivation for the server $x \triangleright \{\text{req} : (y)(\text{Split}\langle x, y \rangle \mid \text{Worker}\langle y \rangle)\}$ using the types S, T and U defined in Equations (5)–(7). It is convenient to also use the type $V = \oplus\{\text{resp}^0 : \mathbf{1}\}$. Table 5 shows the typing derivations for the *Split*, *Gather* and *Worker* processes. Note that the measures obtained for these processes coincide with those previously inferred in Section 5.1. From $S \bowtie U$ (Example 10) we derive

$$\frac{\frac{\frac{\vdots}{\text{Split}\langle x, y \rangle \vdash^4 x : V, y : S} \quad \frac{\vdots}{\text{Worker}\langle y \rangle \vdash^2 y : U}}{\frac{(y)(\text{Split}\langle x, y \rangle \mid \text{Worker}\langle y \rangle) \vdash^6 x : V} \text{ [CUT]}}{x \triangleright \{\text{req} : (y)(\text{Split}\langle x, y \rangle \mid \text{Worker}\langle y \rangle)\} \vdash^6 x : \&\{\text{req}^0 : V\}} \text{ [}\&\text{]}$$

confirming that the server is well typed. \lrcorner

5.3 Properties of Well-Typed Processes

We conclude this section presenting a series of results showing the main properties of well-typed processes, starting from the preservation of typing under reductions (*i.e.* subject reduction) which underlies most of the subsequent results. Subject reduction is formulated in a slightly non-standard way because the typing context may become “smaller” – *i.e.* “more precise” – according to the subtyping relation \leq after each reduction. This is a consequence of the formulation of the rule $[\text{LINK}]$, where the types of x and y need not be dual to each other, but can be related by subtyping. In the statement of Theorem 20, we write \leq for the pointwise extension of subtyping to typing contexts.

“unguarded” top-level structure of the process which is guaranteed to be finite because of the guardedness restriction we have assumed on processes in Section 2. This component is needed to cope with the typing rules [CALL] and [CUT] for which the measure of the premise(s) is not necessarily smaller than the measure in the conclusion of the rules.

Theorem 22 can be strengthened to a termination result for *deterministic* processes, those that do not contain non-deterministic choices.

► **Theorem 23** (termination). *If P is deterministic and $P \vdash^n \Gamma$, then P is terminating.*

The combination of Theorem 22 with Theorem 21 also guarantees the absence of *orphan messages*. Indeed, suppose that P is a well-typed closed process and $P \rightarrow^* Q$ where Q contains non-empty buffers. By Theorems 21 and 22 we know that $Q \rightarrow^* \sqsupseteq$ **done**. Since in a closed process messages cannot simply disappear and the only way of consuming them is by means of the reductions [R-CLOSE], [R-FORK] and [R-SELECT], we know that all the messages in Q have been received.

Finally, using the proof principle stated in Theorem 4 we obtain, as a straightforward corollary of Theorem 22, that well-typed processes are *fairly terminating* (Definition 3).

► **Corollary 24**. *If $P \vdash^n \Gamma$, then P is fairly terminating.*

6 Subtyping Usage Patterns

A typical usage of subtyping is to make sure that processes occurring in different branches of a non-deterministic choice $P_1 \oplus P_2$ or of a tag input $x \triangleright \{\mathbf{a}_i : P_i\}_{i \in I}$ can be typed in the same typing context. Indeed, the typing rules [CHOICE] and [&] require each P_i to be typed in the same typing context (with the exception that in [&], the type of x can be different in each branch), but there are cases in which this requirement is overly restrictive.

As an example, consider a variation of the scenario depicted in Section 1 in which the worker behaves either as $Worker\langle y \rangle$ (as defined as in Equation (3)) or as $BatchWorker\langle y \rangle$ that first gathers all tasks and then sends back all the results. The $BatchWorker\langle y \rangle$ version can be modeled by the equations

$$\begin{aligned} BatchWorker(y) &= y \triangleright \{\mathbf{task} : BatchWorker\langle y \rangle, \mathbf{stop} : SendResults\langle y \rangle\} \\ SendResults(y) &= y \triangleleft \mathbf{res}.SendResults\langle y \rangle \oplus y \triangleleft \mathbf{stop.close} \ y \end{aligned}$$

for which it is easy to obtain a derivation $BatchWorker\langle y \rangle \vdash y : S^\perp$ where S is defined as in Equation (5). Ideally, we could express the server using a non-deterministic choice, thus:

$$x \triangleright \{\mathbf{req} : (y)(Split\langle x, y \rangle \mid (Worker\langle y \rangle \oplus BatchWorker\langle y \rangle))\}$$

however, this process turns out to be ill typed because $Worker\langle y \rangle$ and $BatchWorker\langle y \rangle$ use the channel y according to different types. Moreover, it should be clear that there is no one-size-fits-all session type that can be associated with y .

We can exploit the relation $U \leq S^\perp$ to make $Worker\langle y \rangle$ (which complies with U defined in Equation (7)) look like a process that complies with S^\perp . In many type systems with subtyping, this is simply achieved with an application of the *subsumption rule*. The type system we have presented for CaP does not have a subsumption rule, but this rule is derivable thanks to the formulation of [LINK]. In our specific example, we can derive

$$\frac{\frac{z \leftrightarrow y \vdash z : U^\perp, y : S^\perp \quad [LINK] \quad \frac{\vdots}{Worker\langle z \rangle \vdash z : U} [CUT]}{(z)(z \leftrightarrow y \mid Worker\langle z \rangle) \vdash y : S^\perp} [CUT]}$$

24:22 Fair Termination of Asynchronous Binary Sessions

to obtain an implementation of the streaming worker that complies with S^\perp . Now we derive

$$(z)(z \leftrightarrow y \mid \text{Worker}\langle z \rangle) \oplus \text{BatchWorker}\langle y \rangle \vdash y : S^\perp$$

to obtain a well-typed worker. Incidentally, this example also illustrates the reason why Theorem 20 is formulated so that, when $P \rightarrow Q$, the typing context of Q may be smaller (according to \leq) than the typing context of P . Indeed we have $(z)(z \leftrightarrow y \mid \text{Worker}\langle z \rangle) \oplus \text{BatchWorker}\langle y \rangle \rightarrow (z)(z \leftrightarrow y \mid \text{Worker}\langle z \rangle) \rightarrow \text{Worker}\langle y \rangle$ and we would not be able to obtain a typing derivation for $\text{Worker}\langle y \rangle \vdash y : S^\perp$.

Another typical usage pattern for links is the implementation of *delegation*, whereby the endpoint of an existing session is exchanged through another session. Because of our formulation of [LINK], in our type system this mechanism also enables a form of substitution principle for channels so that a channel of type T can be safely used where a channel of type S is expected if $S \leq T$ (the reason why this formulation of the substitution principle may seem to go in the wrong direction is again due to the fact that we adopt the viewpoint whereby session types describe processes rather than channels).

To illustrate this usage pattern, consider the typing derivation

$$\frac{\frac{\frac{}{y \leftrightarrow z \vdash^1 y : U, z : T} \text{[LINK]} \quad \frac{\vdots}{P \vdash^n \Gamma, x : V} \quad \frac{\vdots}{Q \vdash^m \Delta, x : W, y : S}}{x(y)[y \leftrightarrow z].P \vdash^{n+1} \Gamma, x : U \otimes V, z : T} \text{[}\otimes\text{]} \quad \frac{}{x(y).Q \vdash^m \Delta, x : S \wp W} \text{[}\wp\text{]}}{(x)(x(y)[y \leftrightarrow z].P \mid x(y).Q) \vdash^{n+m+1} \Gamma, \Delta, z : T} \text{[CUT]}$$

which concerns the composition of a process $x(y)[y \leftrightarrow z].P$ that delegates z to another process $x(y).Q$. Note that the sender delegates a channel of type T while the receiver expects to receive a channel of type S . In order for the cut to be well typed we must have $U \otimes V \wp S \wp W$, that is $U \wp S$ and $V \wp W$. The application of [LINK] requires $U^\perp \leq T$, that is $T^\perp \leq U$ by Proposition 16. From this relation and $U \wp S$ we deduce $T^\perp \wp S$ by definition of \leq . Using Theorem 17 we obtain $T^\perp \leq S^\perp$, that is $S \leq T$.

7 Comparison With Other Subtyping Relations

In this section we provide a more detailed comparison between \leq and some connected subtyping relations appeared in the literature. We show that \leq is coarser than these relations when we focus on the family of *First-order Fairly-terminating Session Types* (FFST). Fairly terminating session types are those describing protocols that can always eventually terminate. The reason why this family is relevant for us is that *fairly terminating session types are the only inhabited types in our type system*: since well-typed processes are fairly terminating (Corollary 24), they only use sessions that can always eventually terminate. For simplicity we also focus on *first-order* session types (without the multiplicative connectives \otimes and \wp) because the handling of higher-order communications varies substantially depending on whether the type system is based on linear logic or not.

To carry out the comparison between \leq and the other subtyping relations we define two restricted versions of the transition relation $\xrightarrow{\alpha}$. We write $\xrightarrow{\alpha}$ for the relation inductively defined by the [MUST-*] rules and we write $\xRightarrow{\alpha}$ for the relation inductively defined by the [MUST-*] and [MAY-*] rules. So we have $\xrightarrow{\alpha} \subset \xRightarrow{\alpha} \subset \xrightarrow{\alpha}$ but the relation $\xrightarrow{\alpha}$ only allows *immediate transitions* and $\xRightarrow{\alpha}$ differs from $\xrightarrow{\alpha}$ because early outputs and late inputs can only be preceded by a bounded number of inputs and outputs, respectively. For example, if $S = \&\{a : S, b : \oplus\{c : 1\}\}$ and $T = \&\{a : T, b : 1\}$, we have $S \xrightarrow{!c} T$ and $S \not\xRightarrow{!c} T$.

We define fairly terminating session types thus:

► **Definition 25.** We say that S is fairly terminating if, for every maximal strongly fair sequence $S = S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} S_2 \xrightarrow{\alpha_3} \dots$ starting from S , there exists k such that $S_k \in \{\mathbf{1}, \perp\}$.

Gay and Hole [28] introduced the first *synchronous subtyping relation* for session types. We qualify this relation as “synchronous” because it does not allow any form of output anticipation. However, it supports unconstrained output covariance and input contravariance as shown in its characterization below, which uses immediate transitions only.

► **Definition 26.** We write \leq_s for the largest relation such that $S \leq_s T$ implies:

1. either $\text{pos}(S)$ or $\text{neg}(T)$;
2. if $T \xrightarrow{? \sigma} T'$ then $S \xrightarrow{? \sigma} S'$ and $S' \leq_s T'$;
3. if $S \xrightarrow{! \sigma} S'$ then $T \xrightarrow{! \sigma} T'$ and $S' \leq_s T'$.

Mostrous *et al.* [40, 39] studied the first *asynchronous subtypings* for session types. Chen *et al.* [11] subsequently refined the relation of Mostrous and Yoshida [39] so as to avoid orphan messages. The resulting relation has been shown to be the largest one included in the original asynchronous subtyping that is closed under duality [7]. Using our LTS we can characterize the asynchronous subtyping relations in this line of research by the following definition.

► **Definition 27.** We write \leq_a for the largest relation such that $S \leq_a T$ implies:

1. either $\text{pos}(S)$ or $\text{neg}(T)$;
2. if $T \xrightarrow{? \sigma} T'$ then $S \xRightarrow{? \sigma} S'$ and $S' \leq_a T'$;
3. if $S \xrightarrow{! \sigma} S'$ then $T \xRightarrow{! \sigma} T'$ and $S' \leq_a T'$.

Like \leq_s , also \leq_a supports unconstrained contravariance of inputs and covariance of outputs but the clauses (2) and (3) are more generous because early/late transitions enable the anticipation of outputs. For example, $\oplus\{a : \&\{b : S, c : \&\{d : T\}\}\} \leq_a \&\{b : \oplus\{a : S\}, c : \&\{d : \oplus\{a : T\}\}\}$. However, the use of $\xRightarrow{\alpha}$ (instead of $\xrightarrow{\alpha}$) in Definition 27 means that anticipated outputs can only be preceded by a bounded number of inputs. If we consider $S = \&\{b : S, c : \mathbf{1}\}$ and $T = \&\{b : T, c : \oplus\{a : \mathbf{1}\}\}$ we have $\oplus\{a : S\} \not\leq_a T$ because $S \xrightarrow{! a}$ and $T \not\xRightarrow{! a}$. Note that clause (2) incorporates the condition of Chen *et al.* [11] that guarantees orphan-message freedom: if the subtype S anticipates an output and T starts with an input, *i.e.* $S \xrightarrow{! \sigma}$ and $T \xrightarrow{? \tau}$ for some σ and τ , then clause (2) guarantees that S performs a corresponding late input transition.

The use of $\xRightarrow{\alpha}$ is both a strength and a limitation of \leq_a . It is a strength because, just like \leq_s , the *property that \leq_a prevents orphan messages holds regardless of any fairness assumption*. It is a limitation in the sense that, as we have seen since Section 1, there are contexts in which it is desirable to work with a coarser asynchronous subtyping relation. To overcome this limitation, Bravetti *et al.* [8] have characterized a *fair asynchronous subtyping relation* which, in our setting, translates to the use of $\xrightarrow{\alpha}$ instead of $\xRightarrow{\alpha}$, as shown below.

► **Definition 28.** We write \leq_{fa} for the largest relation such that $S \leq_{fa} T$ implies:

1. either $\text{pos}(S)$ or $\text{neg}(T)$;
2. if $T \xrightarrow{? \sigma} T'$ then $S \xrightarrow{? \sigma} S'$ and $S' \leq_{fa} T'$;
3. if $S \xrightarrow{! \sigma} S'$ then $T \xrightarrow{! \sigma} T'$ and $S' \leq_{fa} T'$ and $T \xrightarrow{\varphi ! \tau}$ implies $S \xrightarrow{! \tau}$ for every sequence φ of input labels.

Note that the clauses (2–3) of Definition 28 are not symmetric, implying that \leq_{fa} is not closed under duality. While clause (2) supports input contravariance, differently from all the other considered subtypings clause (3) *disallows output covariance altogether*. If we take $S = \oplus\{\text{task} : S, \text{stop} : T\}$, $T = \&\{\text{res} : T, \text{stop} : \perp\}$ and $U = \&\{\text{task} : \oplus\{\text{res} : U\}, \text{stop} : \oplus\{\text{stop} : \mathbf{1}\}\}$ from Example 10, we have $U \not\leq_{\text{fa}} S^\perp$ because $S^\perp \xrightarrow{?stop} \xrightarrow{!res}$ but $U \xrightarrow{?stop} \not\xrightarrow{!res}$. This formulation of clause (3) is due to the fact that \leq_{fa} , unlike \leq_s , \leq_a and \leq as well, is meant to preserve fair session termination (at the type level) which requires a controlled form of output covariance that Bravetti *et al.* [8] have conservatively approximated in this way. The next example shows that removing the additional conditions in clause (3) may compromise fair termination.

► **Example 29.** Consider the session types $S = \&\{\text{play} : \oplus\{\text{win} : S, \text{lose} : S\}, \text{quit} : \mathbf{1}\}$ and $U = \oplus\{\text{play} : \&\{\text{win} : \oplus\{\text{quit} : \perp\}, \text{lose} : U\}\}$ where S specifies the behavior of a slot machine that allows players to **play** an unbounded number of games and U specifies the behavior of a player who plays relentlessly until he **wins** a game. Not only do we have $S \bowtie U$, but also the composition of S and U is fairly terminating in the sense that, at each stage of the interaction between player and slot machine, it is always possible to extend the interaction so that the player wins. If we now consider an *unfair* implementation of the slot machine such that its actual behavior is described by the session type $T = \&\{\text{play} : \oplus\{\text{lose} : T\}, \text{quit} : \perp\}$, we have that $T \leq S$ and therefore $T \bowtie U$ still holds, but the composition of T and U is no longer fairly terminating. Therefore $T \not\leq_{\text{fa}} S$ and rightly so. In our setting, preserving fair session termination at the type level is not so important because the type system is able to enforce fair termination at the process level anyway thanks to its logical foundation. \dashv

► **Theorem 30.** *For FFST session types we have $\leq_s \subset \leq_a$ and $\leq_a \subset \leq$ and $\leq_{\text{fa}} \subset \leq$.*

8 Related Work

Asynchronous subtyping. The asynchronous subtyping relations for binary session types defined in the literature [39, 11, 8] are formulated syntactically using *input contexts* to identify the output actions in the larger session type that have been anticipated in the smaller session type. The work of Ghilezan *et al.* [29] considers multiparty sessions: in their setting a smaller session type can anticipate outputs also w.r.t. outputs sent to a different partner. This is achieved by considering also appropriate *output contexts*. We follow a different approach: once we have defined a LTS for session types that captures their asynchronous semantics (Definitions 11 and 12), we are able to provide a characterization of \leq that is essentially the same as the one for synchronous subtyping (Definition 26).

It is worth pointing out that there is no contradiction in the fact that our subtyping relation turns out to be coarser than others that have been proved *complete* [9] or *precise* [11, 29]. The point is that the completeness of a subtyping relation for session types is always relative to a notion of correct session composition. Depending on this notion, the induced subtyping relation may vary. Our subtyping relation is coarser than \leq_a [39, 11] because it relies on a fairness assumption that enables a larger degree of output anticipation and it is coarser than \leq_{fa} [8] because it is not meant to preserve fair session termination and therefore it allows (almost) unconstrained output covariance.

Asynchronous subtyping relations for session types are known to be undecidable [36, 6, 7] and \leq is no exception (Theorems 17 and 18). Despite the proof technique adopted to prove our undecidability results is similar to those used in other papers [6, 8], we could not directly derive our results from the undecidability of other relations because \leq is coarser.

A synchronous version of fair subtyping for session types has been studied by Padovani [41]. This relation is stricter than \leq because it does not allow early outputs and it is meant to preserve fair session termination. Unlike the other fair/asynchronous subtyping relations for session types (with the exception of the orphan message free subtyping of Chen *et al.* [11]), \leq is closed under duality (Proposition 16).

Fair session termination. Type systems ensuring the fair termination of binary sessions have been studied by Ciccone, Dagnino and Padovani [14, 16, 42, 13]. These works are all based on a synchronous communication model. The technique adopted in this paper for measuring processes is essentially the same presented by Dagnino and Padovani [19] which in turn is an elaboration of resource-aware session type systems [22, 21, 23]. Fair termination is a weaker form of termination [30, 2, 26]. Session type systems enforcing the termination of well-typed processes are typically based on linear logic [45, 37] where termination is a direct consequence of the cut elimination property of the logic.

Logical approaches to asynchrony and subtyping. The literature on asynchronous sessions is mostly based on calculi that are not connected to linear logic and where buffers are modeled explicitly. A notable exception is the work of DeYoung *et al.* [24] which presents a type system based on linear logic for an asynchronous calculus of sessions. In that work, asynchrony is intended as the fact that outputs are non-blocking, but no anticipation is allowed. The basic idea is the same used in the encoding of binary session into the linear π -calculus [20]: session communications make use of explicit continuation channels and outputs can be spawned as soon as they are produced leaving the sender process free to reduce further.

The deep cut reductions of CaP bear some similarities with *deep inference* [31], whereby logical rules (including the cut) may operate on “deep” fragments of a proof derivation. At this stage we are unable to say whether there is a more meaningful connection between our modeling of asynchrony and deep inference.

The use of links to incorporate subtyping in the type system echoes the approach studied by Horne and Padovani [34] who give a *coercive semantics* to subtyping in a logical setting: a subtyping relation $S \leq T$ corresponds to a forwarder process (*i.e.* a link) that consumes a channel of type S^\perp and operates on a channel of type T . In CaP, the availability of the native link is fundamental since buffers may grow arbitrarily and the forwarder process is neither finitely representable nor finitely measurable in general.

9 Concluding Remarks

Sessions are meant to enable the compositional enforcement of safety and liveness properties of communicating processes. However, most liveness properties concerning one particular session – like the fact that a given message is eventually produced or consumed – can only be ensured if one makes the assumption that every other session eventually terminates. For this reason, the eventual termination of sessions is of primary importance. From the property that a session eventually terminates, other properties of interest usually follow “for free” as a straightforward consequence of session typing.

In this paper we have introduced a new theory of asynchronous session types that ensures the eventual termination of every session under a full fairness assumption. As a consequence, every produced message is guaranteed to be eventually consumed and every process waiting for a message is guaranteed to eventually receive one. This theory improves previous ones in a number of ways: we define a novel fair asynchronous subtyping relation that supports the

anticipation of outputs before an unbounded number of inputs, output covariance and that is closed under duality; we extend the soundness properties to multiple, possibly interleaved and dynamically created sessions; we base our theory on a calculus of asynchronous processes whose features, types and typing rules are rooted in linear logic.

Clearly, the undecidability results about the correctness of session composition and of fair asynchronous subtyping are an obstacle to the applicability of the presented theory. While decidable approximations of these notions have been presented in the literature [5, 8, 4], they cover relatively small families of session types. In future work we envision the possibility to define sound (but necessarily incomplete) algorithms for checking correct composition and subtyping inspired by the coinductive characterizations of correct session composition and of fair asynchronous subtyping introduced in this paper.

We also conjecture that our semantic approach to subtyping based on an LTS with early/late transitions scales smoothly to the multiparty setting where it may lead to simpler characterizations of (fair) asynchronous subtyping relations. It may be interesting to verify the validity of this conjecture in future work.

References

- 1 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.
- 2 Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 189–198. ACM Press, 1987. doi:10.1145/41625.41642.
- 3 David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPICs*, pages 42:1–42:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.CSL.2016.42.
- 4 Laura Bocchi, Andy King, and Maurizio Murgia. Asynchronous subtyping by trace relaxation. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14570 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2024. doi:10.1007/978-3-031-57246-3_12.
- 5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: <https://lmcs.episciences.org/7238>.
- 6 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017. doi:10.1016/J.IC.2017.07.010.
- 7 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. doi:10.1016/J.TCS.2018.02.010.
- 8 Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. Fair asynchronous session subtyping. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/LMCS-20(4:5)2024.

- 9 Mario Bravetti and Gianluigi Zavattaro. Asynchronous session subtyping as communicating automata refinement. *Softw. Syst. Model.*, 20(2):311–333, 2021. doi:10.1007/S10270-020-00838-X.
- 10 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.
- 11 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.
- 12 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPICs*, pages 26:1–26:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ECOOP.2022.26.
- 13 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair termination of multiparty sessions. *J. Log. Algebraic Methods Program.*, 139:100964, 2024. doi:10.1016/J.JLAMP.2024.100964.
- 14 Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi:10.1145/3498666.
- 15 Luca Ciccone and Luca Padovani. Inference systems with corules for combined safety and liveness properties of binary session types. *Log. Methods Comput. Sci.*, 18(3), 2022. doi:10.46298/LMCS-18(3:27)2022.
- 16 Luca Ciccone and Luca Padovani. An infinitary proof theory of linear logic ensuring fair termination in the linear π -calculus. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPICs*, pages 36:1–36:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CONCUR.2022.36.
- 17 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.
- 18 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Log. Methods Comput. Sci.*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.
- 19 Francesco Dagnino and Luca Padovani. small caps: An infinitary linear logic for a calculus of pure sessions. In Alessandro Bruni, Alberto Momigliano, Matteo Pradella, Matteo Rossi, and James Cheney, editors, *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, Milano, Italy, September 9-11, 2024*, pages 4:1–4:13. ACM, 2024. doi:10.1145/3678232.3678234.
- 20 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.
- 21 Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021. doi:10.1109/CSF51468.2021.00004.
- 22 Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 305–314. ACM, 2018. doi:10.1145/3209108.3209146.
- 23 Ankush Das and Frank Pfenning. Rast: A language for resource-aware session types. *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/LMCS-18(1:9)2022.
- 24 Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPICs*, pages 228–242. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICs.CSL.2012.228.

- 25 Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. PhD thesis, Paris Diderot University, France, 2017. URL: <https://tel.archives-ouvertes.fr/tel-01676953>.
- 26 Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986. doi:10.1007/978-1-4612-4886-6.
- 27 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.
- 28 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/S00236-005-0177-Z.
- 29 Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for asynchronous multiparty sessions. *ACM Trans. Comput. Log.*, 24(2):14:1–14:73, 2023. doi:10.1145/3568422.
- 30 Orna Grumberg, Nissim Francez, and Shmuel Katz. Fair termination of communicating processes. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 254–265, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800222.806752.
- 31 Alessio Guglielmi and Lutz Straßburger. Non-commutativity and MELL in the calculus of structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2001. doi:10.1007/3-540-44802-0_5.
- 32 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.
- 33 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *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*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFB0053567.
- 34 Ross Horne and Luca Padovani. A logical account of subtyping for session types. *J. Log. Algebraic Methods Program.*, 141:100986, 2024. doi:10.1016/J.JLAMP.2024.100986.
- 35 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostros, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.
- 36 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.
- 37 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 434–447. ACM, 2016. doi:10.1145/2951913.2951921.
- 38 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.

- 39 Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π -calculus. *Inf. Comput.*, 241:227–263, 2015. doi:10.1016/J.IC.2015.02.002.
- 40 Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009. doi:10.1007/978-3-642-00590-9_23.
- 41 Luca Padovani. Fair subtyping for multi-party session types. *Math. Struct. Comput. Sci.*, 26(3):424–464, 2016. doi:10.1017/S096012951400022X.
- 42 Luca Padovani. On the fair termination of client-server sessions. In Delia Kesner and Pierre-Marie Pédro, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20–25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPICs*, pages 5:1–5:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.TYPES.2022.5.
- 43 Luca Padovani and Gianluigi Zavattaro. Fair termination of asynchronous binary sessions. Technical report, University of Bologna, 2025. doi:10.48550/arXiv.2503.07273.
- 44 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.
- 45 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.