Asynchronous Functional Sessions

van den Heuvel, Bas; Pérez, Jorge A.

Published in:
Electronic Proceedings in Theoretical Computer Science

DOI:
10.48550/arXiv.2209.06820

Publication date:
2022

Asynchronous Functional Sessions: Cyclic and Concurrent

Bas van den Heuvel Jorge A. Pérez
University of Groningen, The Netherlands
{b.van.den.heuvel, j.a.perez} @ rug.nl

We present Concurrent GV (CGV), a functional calculus with message-passing concurrency governed by session types. With respect to prior calculi, CGV has increased support for concurrent evaluation and for cyclic network topologies. The design of CGV draws on APCP, a session-typed asynchronous π-calculus developed in prior work. Technical contributions are (i) the syntax, semantics, and type system of CGV; (ii) a correct translation of CGV into APCP; (iii) a technique for establishing deadlock-free CGV programs, by resorting to APCP's priority-based type system.

1 Introduction

The goal of this paper is to introduce a new functional calculus with message-passing concurrency governed by linearity and session types. Our work contributes to a research line initiated by Gay and Vasconcelos [8], who proposed a functional calculus with sessions here referred to as λ^{sess}; this line of work has received much recent attention thanks to Wadler's GV calculus [26], which is a variation of λ^{sess}.

Our new calculus is dubbed Concurrent GV (CGV); with respect to previous work, it presents three intertwined novelties: asynchronous (buffered) communication; a highly concurrent reduction strategy; and thread configurations with cyclic topologies. The design of CGV rests upon a solid basis: an operationally correct translation into APCP (Asynchronous Priority-based Classical Processes), a session-typed π-calculus in which asynchronous processes communicate by forming cyclic networks [13].

We discuss the salient features of CGV by example, using a simplified syntax. As in λ^{sess}, communication in CGV is asynchronous: send operations place their messages in buffers, and receive operations read the messages from these buffers. Let us write $\mathsf{send}\ (u,x)$ to denote the output of message u along channel x, and $\mathsf{recv}\ y$ to denote an input on y. The following program expresses the parallel composition ($\|$) of two threads:

$$\begin{array}{l}\mathsf{let}\, x = \mathsf{send}\ (u,x)\, \mathsf{in} \\ \quad \mathsf{let}\, (v,y) = \mathsf{recv}\ y\, \mathsf{in}\, ()\end{array} \quad \Big\| \quad \mathsf{let}\, y = \underline{\mathsf{send}\ (w,y)}\, \mathsf{in}\, ()$$

In variants of λ^{sess} with *synchronous* communication, such as GV and Kokke and Dardha's PGV [17, 18], this program is stuck: the send on y (underlined, on the right) cannot synchronize with the receive on y (on the left): it is blocked by the send on x (on the left), and there is no receive on x. In contrast, in CGV the send on x can be buffered after which the communication on y can take place.

In CGV, reduction is "more concurrent" than usual call-by-value or call-by-name strategies. Consider the following program:

$$\begin{pmatrix}\lambda x\,.\,\mathsf{let}\,(u,y) = \mathsf{recv}\ y\,\mathsf{in} \\ \qquad \mathsf{let}\, x = \mathsf{send}\ (u,x)\,\mathsf{in}()\end{pmatrix} \quad \big(\mathsf{send}\ (v,z)\big)$$

In λ^{sess}, reduction is call-by-value and so the function on x can only be applied on a value. However, the function's parameter (send on z) is not a value, so it needs to be evaluated before the function on x can

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 75–94, doi:10.4204/EPTCS.368.5

be applied. Hence, this program can only be evaluated in one order: first the send on z, then the receive on y. In contrast, the semantics of CGV evaluates a function and its parameters *concurrently*: the send on z and the receive on y can be evaluated in any order. Note that asynchrony plays no role here: both buffering a message and synchronous communication entail a reduction in the function's parameter.

The third novelty is cyclic thread configurations: threads can be connected by channels to form cyclic networks. Consider the following program:

$$\begin{array}{l} \mathsf{let}\,(u,x) = \mathsf{recv}\,x\,\mathsf{in} \\ \quad \mathsf{let}\,y = \mathsf{send}\,(u,y)\,\mathsf{in}\,() \end{array} \quad \Big\| \quad \begin{array}{l} \mathsf{let}\,x = \mathsf{send}\,(v,x)\,\mathsf{in} \\ \quad \mathsf{let}\,(w,y) = \mathsf{recv}\,y\,\mathsf{in}\,() \end{array}$$

Here we have two threads connected on channels x and y, thus forming a cyclic thread configuration. Clearly, this program is deadlock-free. In λ^{sess}, the program is well-typed, but there is no deadlock-freedom guarantee: the type system of λ^{sess} admits deadlocked cyclic thread configurations. In GV and Fowler *et al.*'s EGV [7] (an extension of Fowler's AGV [6]) there is a deadlock-freedom guarantee for well-typed programs; however, their type systems only support *tree-shaped* thread configurations—this limitation is studied in [4, 5]. Hence, the program above is not well-typed in GV and EGV.

These novelties are *intertwined*, in the following sense. Asynchronous communication reduces the synchronization points in programs (as output-like operations are non-blocking), therefore increasing concurrent evaluation. In turn, reduced synchronization points can streamline verification techniques for deadlock-freedom based on *priorities* [16, 22, 23, 3], which unlock the analysis of process networks with cyclic topologies. Indeed, in an asynchronous setting only input-like operations require priorities.

We endow CGV with a type system with functional types and session types; we opted for a design in which well-typed terms enjoy subject reduction / type preservation but not deadlock-freedom. To validate our semantic design and attain the three novelties motivated above, we resort to APCP. In our developments, APCP operates as a "low-level" reference programming model. We give a typed translation of CGV into APCP, which satisfies strong correctness properties, in the sense of Gorla [10]. In particular, it enjoys operational correspondence, which provides a significant sanity check to justify our key design decisions in CGV's operational semantics. Interestingly, using our correct translation and the deadlock-freedom guarantees for well-typed processes in APCP, we obtain a technique for transferring the deadlock-freedom property to CGV programs. That is, given a CGV program C, we prove that if the APCP translation of C is typable (and hence, deadlock-free), then C itself is deadlock-free. This result thus delineates a class of deadlock-free CGV programs that includes cyclic thread configurations.

In summary, this paper presents the following technical contributions: (1) CGV, a new functional calculus with session-based asynchronous concurrency; (2) A typed translation of CGV into APCP, which is proven to satisfy well-studied encodability criteria; (3) A transference result for the deadlock-freedom property from APCP to CGV programs. An extended version contains omitted technical details [14].

2 Concurrent GV

2.1 Syntax and Semantics

The main syntactic entities in CGV are *terms*, *runtime terms*, and *configurations*. Intuitively, terms reduce to runtime terms; configurations correspond to the parallel composition of a main thread and several child threads, each executing a runtime term. Buffered messages are part of configurations. We define two reduction relations: one is on terms, which is then subsumed by reduction on configurations.

The syntax of terms (L,M,N) is given and described in Figure 1. We use $x,y,\ldots$ for *variables*; we write *endpoint* to refer to a variable used for session operations (send, receive, select, offer). Let

Terms (L, M, N):

x	(variable)	new	(create new channel)
$()$	(unit value)	$\mathsf{spawn}\ M$	(execute pair M in parallel)
$\lambda x.M$	(abstraction)	$\mathsf{send}\ (M,N)$	(send M along N)
$M\ N$	(application)	$\mathsf{recv}\ M$	(receive along M)
(M,N)	(pair construction)	$\mathsf{select}\ \ell\ M$	(select label ℓ along M)
$\mathsf{let}\,(x,y) = M\,\mathsf{in}\,N$	(pair deconstruction)	$\mathsf{case}\,M\,\mathsf{of}\,\{i : M\}_{i\in I}$	(offer labels in I along M)

Runtime terms $(\mathbb{L}, \mathbb{M}, \mathbb{N})$ and reduction contexts $(\mathscr{R})$:

$$\begin{aligned}\mathbb{L},\mathbb{M},\mathbb{N} ::= {} & \ldots \mid \mathbb{M}\{\!|\mathbb{N}/x|\!\} \mid \mathsf{send}'(\mathbb{M},\mathbb{N}) \\ \mathscr{R} ::= {} & [] \mid \mathscr{R}\ \mathbb{M} \mid \mathsf{spawn}\ \mathscr{R} \mid \mathsf{send}\ \mathscr{R} \mid \mathsf{recv}\ \mathscr{R} \mid \mathsf{let}\,(x,y) = \mathscr{R}\,\mathsf{in}\,\mathbb{M} \\ & \mid \mathsf{select}\ \ell\ \mathscr{R} \mid \mathsf{case}\,\mathscr{R}\,\mathsf{of}\,\{i : \mathbb{M}\}_{i\in I} \mid \mathscr{R}\{\!|\mathbb{M}/x|\!\} \mid \mathbb{M}\{\!|\mathscr{R}/x|\!\} \mid \mathsf{send}'(\mathbb{M},\mathscr{R})\end{aligned}$$

Structural congruence for terms $(\equiv_{\mathtt{M}})$ and term reduction $(\longrightarrow_{\mathtt{M}})$:

SC-SubExt	$x \notin \mathrm{fn}(\mathscr{R}) \Rightarrow (\mathscr{R}[\mathbb{M}])\{\!|\mathbb{N}/x|\!\} \equiv_{\mathtt{M}} \mathscr{R}[\mathbb{M}\{\!|\mathbb{N}/x|\!\}]$
E-Lam	$(\lambda x.\mathbb{M})\ \mathbb{N} \longrightarrow_{\mathtt{M}} \mathbb{M}\{\!|\mathbb{N}/x|\!\}$
E-Pair	$\mathsf{let}\,(x,y) = (\mathbb{M}_1,\mathbb{M}_2)\,\mathsf{in}\,\mathbb{N} \longrightarrow_{\mathtt{M}} \mathbb{N}\{\!|\mathbb{M}_1/x, \mathbb{M}_2/y|\!\}$
E-SubstName	$\mathbb{M}\{\!|y/x|\!\} \longrightarrow_{\mathtt{M}} \mathbb{M}\{y/x\}$
E-NameSubst	$x\{\!|\mathbb{M}/x|\!\} \longrightarrow_{\mathtt{M}} \mathbb{M}$
E-Send	$\mathsf{send}\ (\mathbb{M},\mathbb{N}) \longrightarrow_{\mathtt{M}} \mathsf{send}'(\mathbb{M},\mathbb{N})$
E-Lift	$\mathbb{M} \longrightarrow_{\mathtt{M}} \mathbb{N} \Rightarrow \mathscr{R}[\mathbb{M}] \longrightarrow_{\mathtt{M}} \mathscr{R}[\mathbb{N}]$
E-LiftSC	$\mathbb{M} \equiv_{\mathtt{M}} \mathbb{M}' \wedge \mathbb{M}' \longrightarrow_{\mathtt{M}} \mathbb{N}' \wedge \mathbb{N}' \equiv_{\mathtt{M}} \mathbb{N} \Rightarrow \mathbb{M} \longrightarrow_{\mathtt{M}} \mathbb{N}$

Figure 1: The CGV term language.

$\mathrm{fn}(M)$ denote the free variables of a term. All variables are free unless bound: $\lambda x.M$ binds x in M, and $\mathsf{let}\,(x,y) = M\,\mathsf{in}\,N$ binds x and y in N. We introduce syntactic sugar for applications of abstractions: $\mathsf{let}\,x = M\,\mathsf{in}\,N$ denotes $(\lambda x.N)\ M$. For $(\lambda x.M)\ N$, we assume $x \notin \mathrm{fn}(N)$, and for $\mathsf{let}\,(x,y) = M\,\mathsf{in}\,N$, we assume $x \neq y$ and $x, y \notin \mathrm{fn}(M)$.

Figure 1 also gives the reduction semantics of CGV terms $(\longrightarrow_{\mathtt{M}})$, which relies on runtime terms $(\mathbb{L}, \mathbb{M}, \mathbb{N})$, reduction contexts $(\mathscr{R})$, and structural congruence $(\equiv_{\mathtt{M}})$. Note that this semantics comprises the functional fragment of CGV; we define the concurrent semantics of CGV hereafter.

Runtime terms, whose syntax extends that of terms, guide the evaluation strategy of CGV; we discuss an example evaluation of a term using runtime terms after introducing the reduction rules (Example 2.1). Explicit substitution $\mathbb{M}\{\!|\mathbb{N}/x|\!\}$ enables the concurrent execution of a function and its parameters. The intermediate primitive $\mathsf{send}'(\mathbb{M},\mathbb{N})$ enables $\mathbb{N}$ to reduce to an endpoint; the send primitive takes a pair of terms as an argument, inside which reduction is not permitted (cf. [8]). Reduction contexts define the

Markers (ϕ), messages (m,n), configurations (C,D,E), thread ($\mathscr{F}$) and configuration ($\mathscr{G}$) contexts:

$$\phi ::= \blacklozenge \mid \lozenge \qquad m,n ::= \mathbb{M} \mid \ell$$
$$C,D,E ::= \phi\,\mathbb{M} \mid C \parallel D \mid (\boldsymbol{\nu} x[\vec{m}\rangle y)C \mid C\{\!|\mathbb{M}/x|\!\}$$
$$\mathscr{F} ::= \phi\,\mathscr{R} \mid C\{\!|\mathscr{R}/x|\!\} \qquad \mathscr{G} ::= [] \mid \mathscr{G} \parallel C \mid (\boldsymbol{\nu} x[\vec{m}\rangle y)\mathscr{G} \mid \mathscr{G}\{\!|\mathbb{M}/x|\!\}$$

Structural congruence for configurations ($\equiv_{\mathsf{C}}$):

SC-TermSC	$\mathbb{M} \equiv_{\mathtt{M}} \mathbb{M}' \Rightarrow \phi\,\mathbb{M} \equiv_{\mathsf{C}} \phi\,\mathbb{M}'$
SC-ResSwap	$(\boldsymbol{\nu} x[\varepsilon\rangle y)C \equiv_{\mathsf{C}} (\boldsymbol{\nu} y[\varepsilon\rangle x)C$
SC-ResComm	$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\boldsymbol{\nu} z[\vec{n}\rangle w)C \equiv_{\mathsf{C}} (\boldsymbol{\nu} z[\vec{n}\rangle w)(\boldsymbol{\nu} x[\vec{m}\rangle y)C$
SC-ResExt	$x,y \notin \mathrm{fn}(C) \Rightarrow (\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D) \equiv_{\mathsf{C}} C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y)D$
SC-ResNil	$x,y \notin \mathrm{fn}(C) \Rightarrow (\boldsymbol{\nu} x[\varepsilon\rangle y)C \equiv_{\mathsf{C}} C$
SC-Send'	$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathscr{F}}[\mathsf{send}'(\mathbb{M},x)] \parallel C) \equiv_{\mathsf{C}} (\boldsymbol{\nu} x[\mathbb{M},\vec{m}\rangle y)(\hat{\mathscr{F}}[x] \parallel C)$
SC-Select	$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathscr{F}[\mathsf{select}\,\ell\,x] \parallel C) \equiv_{\mathsf{C}} (\boldsymbol{\nu} x[\ell,\vec{m}\rangle y)(\mathscr{F}[x] \parallel C)$
SC-ParNil	$C \parallel \lozenge() \equiv_{\mathsf{C}} C$
SC-ParComm	$C \parallel D \equiv_{\mathsf{C}} D \parallel C$
SC-ParAssoc	$C \parallel (D \parallel E) \equiv_{\mathsf{C}} (C \parallel D) \parallel E$
SC-ConfSubst	$\phi\,(\mathbb{M}\{\!|\mathbb{N}/x|\!\}) \equiv_{\mathsf{C}} (\phi\,\mathbb{M})\{\!|\mathbb{N}/x|\!\}$
SC-ConfSubstExt	$x \notin \mathrm{fn}(\mathscr{G}) \Rightarrow (\mathscr{G}[C])\{\!|\mathbb{M}/x|\!\} \equiv_{\mathsf{C}} \mathscr{G}[C\{\!|\mathbb{M}/x|\!\}]$

Configuration reduction ($\longrightarrow_{\mathsf{C}}$):

E-New	$\mathscr{F}[\mathsf{new}] \longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\mathscr{F}[(x,y)])$
E-Spawn	$\hat{\mathscr{F}}[\mathsf{spawn}\ (\mathbb{M},\mathbb{N})] \longrightarrow_{\mathsf{C}} \hat{\mathscr{F}}[\mathbb{N}] \parallel \lozenge\,\mathbb{M}$
E-Recv	$(\boldsymbol{\nu} x[\vec{m},\mathbb{M}\rangle y)(\hat{\mathscr{F}}[\mathsf{recv}\ y] \parallel C) \longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathscr{F}}[(\mathbb{M},y)] \parallel C)$
E-Case	$j \in I \Rightarrow (\boldsymbol{\nu} x[\vec{m},j\rangle y)(\mathscr{F}[\mathsf{case}\,y\,\mathsf{of}\,\{i : \mathbb{M}_i\}_{i\in I}] \parallel C) \longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathscr{F}[\mathbb{M}_j\ y] \parallel C)$
E-LiftC	$C \longrightarrow_{\mathsf{C}} C' \Rightarrow \mathscr{G}[C] \longrightarrow_{\mathsf{C}} \mathscr{G}[C']$
E-LiftM	$\mathbb{M} \longrightarrow_{\mathtt{M}} \mathbb{M}' \Rightarrow \mathscr{F}[\mathbb{M}] \longrightarrow_{\mathsf{C}} \mathscr{F}[\mathbb{M}']$
E-ConfLiftSC	$C \equiv_{\mathsf{C}} C' \wedge C' \longrightarrow_{\mathsf{C}} D' \wedge D' \equiv_{\mathsf{C}} D \Rightarrow C \longrightarrow_{\mathsf{C}} D$

Figure 2: The CGV configuration language.

non-blocking parts of terms, where subterms may reduce. We write $\mathscr{R}[\mathbb{M}]$ to denote the runtime term obtained by replacing the hole $[]$ in $\mathscr{R}$ by $\mathbb{M}$, and $\mathrm{fn}(\mathscr{R})$ to denote $\mathrm{fn}(\mathscr{R}[()])$; we will use similar notation for other kinds of contexts later.

We discuss the reduction rules. The Structural congruence rule SC-SubExt allows the scope extrusion of explicit substitutions along reduction contexts. Rule E-Lam enforces application, resulting in an explicit substitution. Rule E-Pair unpacks the elements of a pair into two explicit substitutions

(arbitrarily ordered, due to the syntactical assumptions introduced above). Rules E-SUBSTNAME and E-NAMESUBST convert explicit substitutions of or on variables into standard substitutions. Rule E-SEND reduces a send into a send′ primitive. Rules E-LIFT and E-LIFTSC close term reduction under contexts and structural congruence, respectively. We write $\mathbb{M} \longrightarrow_{\mathtt{M}}^{k} \mathbb{N}$ to denote that $\mathbb{M}$ reduces to $\mathbb{N}$ in k steps.

Example 2.1. *We illustrate the evaluation of terms using runtime terms through the following example, which contains a* send *primitive and nested abstractions and applications. In each reduction step, we underline the subterm that reduces and give the applied rule:*

$$
\begin{array}{ll}
\underline{(\lambda x.\,\mathsf{send}\ ((),x))\ ((\lambda y.\,y)\ z)} & \text{(E-LAM)} \\
\longrightarrow_{\mathtt{M}} \underline{(\mathsf{send}\ ((),x))}\,\{\!|\,((\lambda y.\,y)\ z)/x\,|\!\} & \text{(E-SEND)} \\
\longrightarrow_{\mathtt{M}} (\mathsf{send}'((),x))\,\{\!|\,\underline{((\lambda y.\,y)\ z)}/x\,|\!\} & \text{(E-LAM)} \\
\longrightarrow_{\mathtt{M}} (\mathsf{send}'((),x))\,\{\!|\,(y\{\!|\,z/y\,|\!\})/x\,|\!\} & \text{(SC-SUBEXT)} \\
\equiv_{\mathtt{M}} \mathsf{send}'((),\underline{x\{\!|\,(y\{\!|\,z/y\,|\!\})/x\,|\!\}}) & \text{(E-NAMESUBST)} \\
\longrightarrow_{\mathtt{M}} \mathsf{send}'((),\underline{y\{\!|\,z/y\,|\!\}}) & \text{(E-SUBSTNAME)} \\
\longrightarrow_{\mathtt{M}} \mathsf{send}'((),z) &
\end{array}
$$

Notice how the send *primitive needs to reduce to a* send′ *runtime primitive such that the explicit substitution of x can be applied. Also, note that the concurrency of CGV allows many more paths of reduction.*

Note that the concurrent evaluation strategy of CGV may also be defined without explicit substitutions. In principle, this would require additional reduction contexts specific to applications on abstractions and pair deconstruction, as well as variants of Rules E-SUBSTNAME and E-NAMESUBST specific to these contexts. However, it is not clear how to define scope extrusion (Rule SC-SUBEXT) for such a semantics. Hence, we find that using explicit substitutions drastically simplifies the semantics of CGV.

Concurrency in CGV allows the parallel execution of terms that communicate through buffers. The syntax of configurations (C,D,E) is given in Figure 2. The configuration $\phi\,\mathbb{M}$ denotes a *thread*: a concurrently executed term. The thread marker helps to distinguish the *main* thread ($\phi = \blacklozenge$) from *child* threads ($\phi = \lozenge$). The configuration $C \,\|\, D$ denotes parallel composition. The configuration $(\boldsymbol{\nu} x[\vec{m}\rangle y)C$ denotes a *buffered restriction*: it connects the endpoints x and y through a buffer $[\vec{m}\rangle$, binding x and y in C. The buffer's content, $\vec{m}$, is a sequence of messages (terms and labels). Buffers are directed: in $x[\vec{m}\rangle y$, messages can be added to the front of the buffer on x, and they can be taken from the back of the buffer on y. We write $[\varepsilon\rangle$ for the empty buffer. The configuration $C\{\!|\,\mathbb{M}/x\,|\!\}$ lifts explicit substitution to the level of configurations: this allows spawning and sending terms under explicit substitution, such that the substitution can be moved to the context of the spawned or sent term.

The reduction semantics for configurations ($\longrightarrow_{\mathtt{C}}$, also in Figure 2) relies on thread and configuration contexts ($\mathscr{F}$ and $\mathscr{G}$, respectively) and structural congruence ($\equiv_{\mathtt{C}}$). We write $\hat{\mathscr{F}}$ to denote a thread context in which the hole does not occur under explicit substitution, i.e. the context is not constructed using the clause $\mathscr{R}\{\!|\,\mathbb{M}/x\,|\!\}$; this is used in rules for send′, spawn, and recv, effectively forcing the scope extrusion of explicit substitutions when terms are moved between contexts (cf. Example 2.4).

We comment on some of the congruences and reduction rules. Rule SC-RESSWAP allows to swap the direction of an empty buffer; this way, the endpoint that could read from the buffer before the swap can now write to it. Rule SC-RESCOMM allows to interchange buffers, and Rule SC-RESEXT allows to extrude their scope. Rule SC-RESNIL garbage collects buffers of closed sessions. Rule SC-CONFSUBST lifts explicit substitution at the level of terms to the level of threads, and

Rule SC-ConfSubstExt allows the scope extrusion of explicit substitution along configuration contexts. Notably, putting messages in buffers is not a reduction: Rules SC-Send' and SC-Select *equate* sends and selects on an endpoint x with terms and labels in the buffer for x, as asynchronous outputs are computationally equivalent to messages in buffers.

Reduction rule E-New creates a new buffer, leaving a reference to the newly created endpoints in the thread. Rule E-Spawn spawns a child thread (the parameter pair's first element) and continues (as the pair's second element) inside the calling thread. Rule E-Recv retrieves a term from a buffer, resulting in a pair containing the term and a reference to the receiving endpoint. Rule E-Case retrieves a label from a buffer, resulting in a function application of the label's corresponding branch to a reference to the receiving endpoint. There are no reduction rules for closing sessions, as they are closed silently. We write $C \longrightarrow_{\mathsf{C}}^{k} D$ to denote that C reduces to D in k steps. Also, we write $\longrightarrow_{\mathsf{C}}^{+}$ to denote the transitive closure of $\longrightarrow_{\mathsf{C}}$ (i.e., reduction in at least one step).

We illustrate CGV's semantics by giving some examples. The following discusses a cyclic thread configuration which does not deadlock due to asynchrony:

Example 2.2. *Consider configuration C_1 below, in which two threads are spawned and cyclically connected through two channels. One thread first sends on the first channel and then receives on the second, while the other thread first sends on the second channel and then receives on the first. Under synchronous communication, this would determine a configuration that deadlocks; however, under asynchronous communication, this is not the case (cf. the third example in Sec. 1). We detail some interesting reductions:*

$$C_1 = \blacklozenge(\mathsf{let}\,(f,g) = \mathsf{new}\,\mathsf{in}\,\mathsf{let}\,(h,k) = \mathsf{new}\,\mathsf{in}\,\mathsf{spawn}\begin{pmatrix}\mathsf{let}\,f' = (\mathsf{send}\ (u,f))\,\mathsf{in}\\ \quad \mathsf{let}\,(v',h') = (\mathsf{recv}\ h)\,\mathsf{in}\,(),\\ \mathsf{let}\,k' = (\mathsf{send}\ (v,k))\,\mathsf{in}\\ \quad \mathsf{let}\,(u',g') = (\mathsf{recv}\ g)\,\mathsf{in}\,()\end{pmatrix})$$

$$\longrightarrow_{\mathsf{C}}^{8} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge(\mathsf{spawn}\begin{pmatrix}\mathsf{let}\,f' = (\mathsf{send}\ (u,x))\,\mathsf{in} & \mathsf{let}\,k' = (\mathsf{send}\ (v,z))\,\mathsf{in}\\ \quad \mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,(), & \quad \mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,()\end{pmatrix}) \tag{1}$$

$$\longrightarrow_{\mathsf{C}} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge\begin{pmatrix}\mathsf{let}\,k' = (\mathsf{send}\ (v,z))\,\mathsf{in}\\ \quad \mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,()\end{pmatrix} \parallel \Diamond \begin{pmatrix}\mathsf{let}\,f' = (\mathsf{send}\ (u,x))\,\mathsf{in}\\ \quad \mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,()\end{pmatrix}) \tag{2}$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge\begin{pmatrix}(\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}\ (v,z)/k'|\!\}\end{pmatrix} \parallel \Diamond \begin{pmatrix}(\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}\ (u,x)/f'|\!\}\end{pmatrix}) \tag{3}$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} w[\varepsilon\rangle z)(\blacklozenge\begin{pmatrix}(\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}'(v,z)/k'|\!\}\end{pmatrix} \parallel \Diamond \begin{pmatrix}(\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,())\\ \{\!|\mathsf{send}'(u,x)/f'|\!\}\end{pmatrix}) \tag{4}$$

$$\equiv (\boldsymbol{\nu} x[u\rangle y)(\boldsymbol{\nu} z[v\rangle w)(\blacklozenge((\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,())\{\!|z/k'|\!\}) \parallel \Diamond((\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,())\{\!|x/f'|\!\})) \tag{5}$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[u\rangle y)(\boldsymbol{\nu} z[v\rangle w)(\blacklozenge(\mathsf{let}\,(u',g') = (\mathsf{recv}\ y)\,\mathsf{in}\,()) \parallel \Diamond(\mathsf{let}\,(v',h') = (\mathsf{recv}\ w)\,\mathsf{in}\,()))$$

$$\longrightarrow_{\mathsf{C}}^{2} (\boldsymbol{\nu} x[\varepsilon\rangle y)(\boldsymbol{\nu} z[\varepsilon\rangle w)(\blacklozenge(\mathsf{let}\,(u',g') = (v,y)\,\mathsf{in}\,()) \parallel \Diamond(\mathsf{let}\,(v',h') = (v,w)\,\mathsf{in}\,())) \longrightarrow_{\mathsf{C}}^{4} \blacklozenge() \tag{6}$$

Intuitively, reduction (1) instantiates two buffers and assigns the endpoints through explicit substitutions. Reduction (2) spawns the left term as a child thread. Reduction (3) turns lets *into explicit substitutions. Reduction (4) turns the* sends *into* send′*s. Structural congruence (5) equates the* send′*s with messages in the buffers. Reduction (6) retrieves the messages from the buffers. Note that many of these steps represent several reductions that may happen in any order.*

The following example illustrates CGV's flexibility for communicating functions over channels:

Example 2.3. *In the following configuration, a buffer and two threads have already been set up (cf. Example 2.2 for an illustration of such an initialization). The main thread sends an interesting term to the child thread: it contains the* send *primitive from which the main thread will subsequently receive from the child thread. We give the configuration's major reductions, with the reducing parts underlined:*

$$
\begin{aligned}
&(\boldsymbol{\nu} x[\varepsilon\rangle y)(\blacklozenge\left(\frac{\mathsf{let}\, x' = \mathsf{send}\,\left(\lambda z.\,\mathsf{send}\,((),z),x\right)\,\mathsf{in}}{\mathsf{let}\,(v,x'') = \mathsf{recv}\,x'\,\mathsf{in}\,v}\right) \parallel \Diamond \begin{pmatrix}\mathsf{let}\,(w,y') = \mathsf{recv}\,y\,\mathsf{in}\\ \mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()\end{pmatrix})\\
\longrightarrow_{\mathsf{C}}^{3}\ &(\boldsymbol{\nu} x[\lambda z.\,\mathsf{send}\,((),z)\rangle y)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \begin{pmatrix}\mathsf{let}\,(w,y') = \underline{\mathsf{recv}\,y}\,\mathsf{in}\\ \mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()\end{pmatrix}\right)\\
\longrightarrow_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\frac{\mathsf{let}\,(w,y') = (\lambda z.\,\mathsf{send}\,((),z),y)\,\mathsf{in}}{\mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()}\right)\right)\\
\longrightarrow_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\underline{\left(\mathsf{let}\,y'' = (w\,y')\,\mathsf{in}\,()\right)\{\!\{\left(\lambda z.\,\mathsf{send}\,((),z)\right)/w,y/y'\}\!\}}\right)\right)\\
\longrightarrow_{\mathsf{C}}^{2}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\mathsf{let}\,y'' = \underline{\left(\left(\lambda z.\,\mathsf{send}\,((),z)\right)\,y\right)}\,\mathsf{in}\,()\right)\right)\\
\longrightarrow_{\mathsf{C}}^{2}\ &(\boldsymbol{\nu} y[\varepsilon\rangle x)\left(\blacklozenge\left(\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v\right) \parallel \Diamond \left(\underline{\mathsf{let}\,y'' = \mathsf{send}\,((),y)\,\mathsf{in}\,()}\right)\right)\\
\longrightarrow_{\mathsf{C}}^{3}\ &(\boldsymbol{\nu} y[()\rangle x)\left(\blacklozenge\left(\underline{\mathsf{let}\,(v,x'') = \mathsf{recv}\,x\,\mathsf{in}\,v}\right)\right) \longrightarrow_{\mathsf{C}}^{4} \blacklozenge\,()
\end{aligned}
$$

The following example illustrates why the restricted thread context $\hat{\mathscr{F}}$ is used:

Example 2.4. *Consider the configuration* $C = (\boldsymbol{\nu} x[\varepsilon\rangle y)\left(\blacklozenge\left(\left(\mathsf{send}'(z,x)\right)\{\!\{v/z\}\!\}\right) \parallel D\right)$. *Suppose Structural congruence rule* SC-SEND' *were defined on unrestricted thread contexts; then the rule applies under the explicit substitution of* z: $C \equiv_{\mathsf{C}} (\boldsymbol{\nu} x[z\rangle y)\left(\blacklozenge\left(x\{\!\{v/z\}\!\}\right) \parallel D\right)$. *Here,* C *and the right-hand-side are inconsistent with each other: in* C*, the variable* z *is bound by the explicit substitution, whereas* z *is free on the right-hand-side. With the restricted thread contexts we are forced to first extrude the scope of the explicit substitution before applying Rule* SC-SEND', *making sure that* z *remains bound:*

$$
C \equiv_{\mathsf{C}} \left((\boldsymbol{\nu} x[\varepsilon\rangle y)\left(\blacklozenge\left(\mathsf{send}'(z,x)\right) \parallel D\right)\right)\{\!\{v/z\}\!\} \equiv_{\mathsf{C}} \left((\boldsymbol{\nu} x[z\rangle y)(\blacklozenge x \parallel D)\right)\{\!\{v/z\}\!\}.
$$

2.2 Type System

We define a type system for CGV, with functional types for functions and pairs and session types for communication. The syntax and meaning of functional types (T, U) and session types (S) are as follows:

$$
\begin{array}{llllllllll}
T,U ::= & T \times U & \text{(pair)} & \mid T \multimap U & \text{(function)} & \mid \mathbf{1} & \text{(unit)} & \mid S & \text{(session)} \\
S ::= & !T.S & \text{(output)} & \mid ?T.S & \text{(input)} & \mid \oplus\{i : T\}_{i\in I} & \text{(select)} & \mid \&\{i : T\}_{i\in I} & \text{(case)} \quad \mid \mathsf{end}
\end{array}
$$

Session type duality ($\overline{S}$) is defined as usual; note that only the continuations, and not the messages, of output and input types are dualized.

$$
\overline{!T.S} = ?T.\overline{S} \quad \overline{?T.S} = !T.\overline{S} \quad \overline{\oplus\{i : S_i\}_{i\in I}} = \&\{i : \overline{S_i}\}_{i\in I} \quad \overline{\&\{i : S_i\}_{i\in I}} = \oplus\{i : \overline{S_i}\}_{i\in I} \quad \overline{\mathsf{end}} = \mathsf{end}
$$

Typing judgments use typing environments (Γ, Δ, Λ) consisting of types assigned to variables ($x : T$). We write $\emptyset$ to denote the empty environment; in writing 'Γ, Δ', we assume that the variables in Γ and Δ

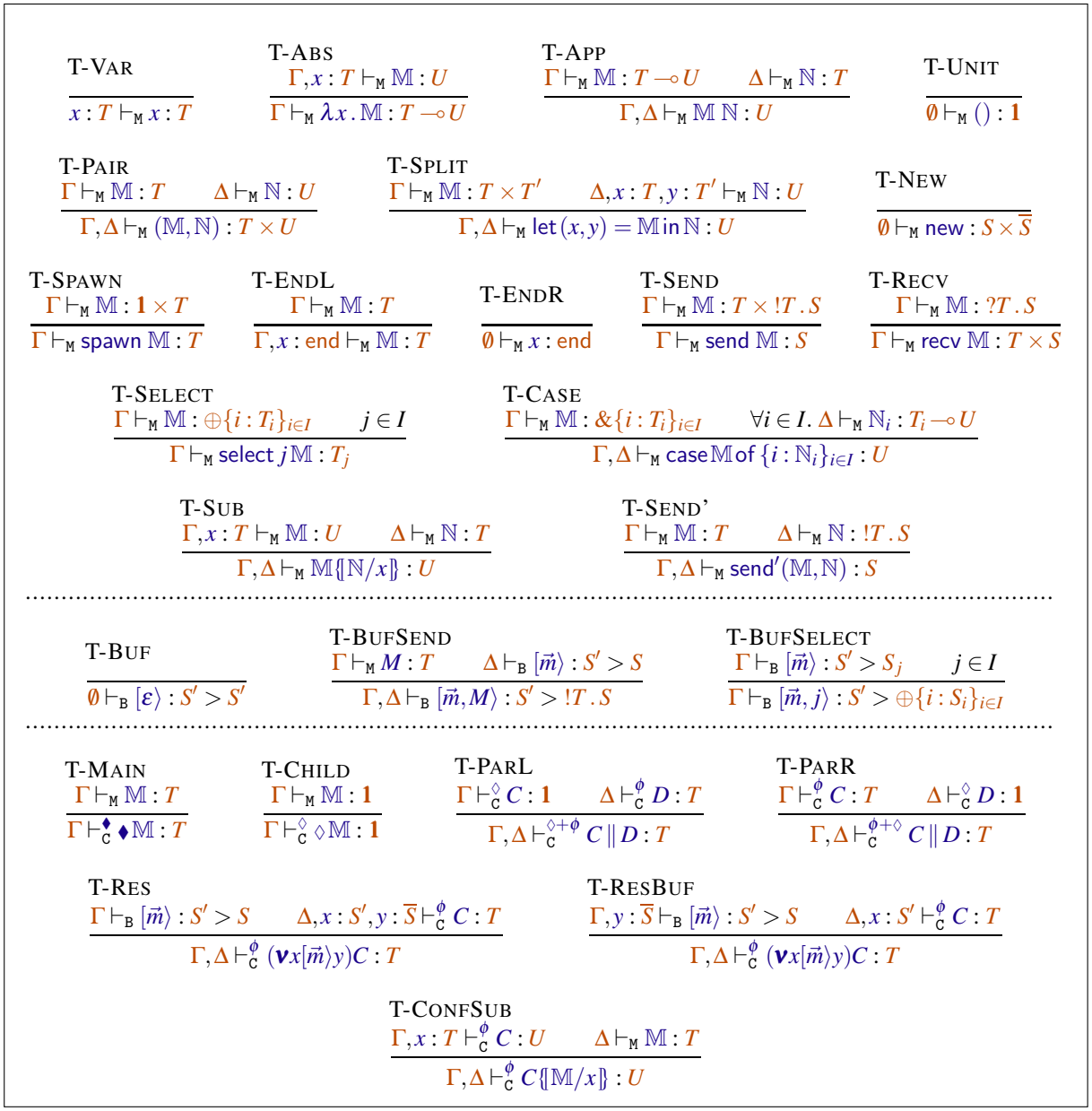

Figure 3: Typing rules for terms (top), buffers (center), and configurations (bottom).

are pairwise distinct. Figure 3 (top) gives the type system for (runtime) terms. Judgments are denoted $\Gamma \vdash_{\mathtt{M}} \mathbb{M} : T$ and have a *use-provide* reading: term $\mathbb{M}$ *uses* the variables in Γ to *provide* a behavior of type T (cf. Caires and Pfenning [1]). When a term provides type T, we often say that the term is of type T.

Typing rules T-Var, T-Abs, T-App, T-Unit, T-Pair, and T-Split are standard. Rule T-New types a pair of dual session types $S \times \overline{S}$. Rule T-Spawn types spawning a $\mathbf{1}$-typed term as a child thread, continuing as a term of type T. Rules T-EndL and T-EndR type finished sessions. Rule T-Send (resp. T-Recv) uses a term of type $!T.S$ (resp. $?T.S$) to type a send (resp. receive) of a term of type T, continuing as type S. Rule T-Select uses a term of type $\oplus\{i : T_i\}_{i \in I}$ to type selecting a label

$j \in I$, continuing as type T_j. Rule T-CASE uses a term of type $\&\{i : T_i\}_{i \in I}$ to type branching on labels $i \in I$, continuing as type U—each branch is typed $T_i \multimap U$. Rule T-SUB types an explicit substitution. Rule T-SEND' types sending directly, not requiring a pair but two separate terms.

Figure 3 (bottom) gives the typing rules for configurations. The typing judgments here are annotated with a thread marker: $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$. The thread marker serves to keep track of whether the typed configuration contains the main thread or not (i.e. $\phi = \blacklozenge$ if so, and $\phi = \lozenge$ otherwise). When typing the parallel composition of two configurations, we thus have to combine the thread markers of their judgments. This combination of thread markers ($\phi + \phi'$) is defined as follows:

$$\blacklozenge + \lozenge = \blacklozenge \qquad \lozenge + \blacklozenge = \blacklozenge \qquad \lozenge + \lozenge = \lozenge \qquad (\blacklozenge + \blacklozenge \text{ is undefined})$$

Typing rules T-MAIN and T-CHILD turn a typed term into a thread, where child threads may only be of type $\mathbf{1}$. Rules T-PARL and T-PARR compose configurations: one configuration must be of type $\mathbf{1}$ and have thread marker $\lozenge$ (i.e., it does not contain a main thread), providing the other configuration's type. Rule T-RES types buffered restriction, with output endpoint x and input endpoint y used in the configuration. It is possible to send the endpoint y on x, so there is also a Rule T-RESBUF where y is used in the buffer. Unlike with usual typing rules for restriction, the types S' of x and $\overline{S}$ of y do not necessarily have to be duals. This is because the restriction's buffer may already contain messages sent on x but not yet received on y, such that the restricted configuration only needs to use x according to a continuation of S. To ensure that S' is indeed a continuation of S in accordance with the messages in the buffer, we have additional typing rules for buffers, which we explain hereafter. Finally, Rule T-CONFSUB types an explicit substitution on the level of configurations.

For typing buffers, in Figure 3 (center), we have judgments of the form: $\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : S' > S$. The judgment denotes that S' is a continuation of S, in accordance with the messages $\vec{m}$, which use the variables in Γ. The idea of the typing rules is that, starting with an empty buffer at the top of the typing derivation (Rule T-BUF) where $S' = S$, Rules T-BUFSEND and T-BUFSELECT add messages to the end of the buffer. Rule T-BUFSEND then prefixes S with an output of the sent term's type, and Rule T-BUFSELECT prefixes S with a selection such that the sent label's continuation is S.

Example 2.5. *Figure 4 (top) shows the typing derivation of a configuration reduced from C_1 in Example 2.2 (following an alternative path after Reduction (5)). Figure 4 (bottom) shows the typing of the configuration $(\boldsymbol{\nu} x[M, \ell, ()\rangle y)C$, which has some messages in a buffer; notice how the type of x in C is a continuation of the dual of the type of y.*

In the configuration $(\boldsymbol{\nu} x[\mathsf{let}\,(z,y) = \mathsf{recv}\,y\,\mathsf{in}\,y\rangle y)C$, the endoint y is inside the buffer connecting it with x; to type it, we need Rule T-RESBUF *(omitting the derivation of the buffer):*

$$\frac{y : ?\mathsf{end}\,.\,\mathsf{end} \vdash_{\mathsf{B}} [\mathsf{let}\,(z,y) = \mathsf{recv}\,y\,\mathsf{in}\,y\rangle : \mathsf{end} > !\mathsf{end}\,.\,\mathsf{end} \qquad \Gamma, x : \mathsf{end} \vdash^{\phi}_{\mathsf{C}} C : U}{\Gamma \vdash^{\phi}_{\mathsf{C}} (\boldsymbol{\nu} x[\mathsf{let}\,(z,y) = \mathsf{recv}\,y\,\mathsf{in}\,y\rangle y)C : U}$$

Note that such buffers will always deadlock: the message in the buffer can never be received.

Type Preservation Well-typed CGV terms and configurations satisfy protocol fidelity and communication safety. These properties follow from type preservation: typing is consistent across structural congruence and reduction. In both cases the proof is by induction on the derivation of the congruence and reduction, respectively; we include full proofs in the extended version of this paper [14].

Theorem 2.6. *If $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$ and $C \equiv_{\mathsf{C}} D$ or $C \longrightarrow_{\mathsf{C}} D$, then $\Gamma \vdash^{\phi}_{\mathsf{C}} D : T$.*

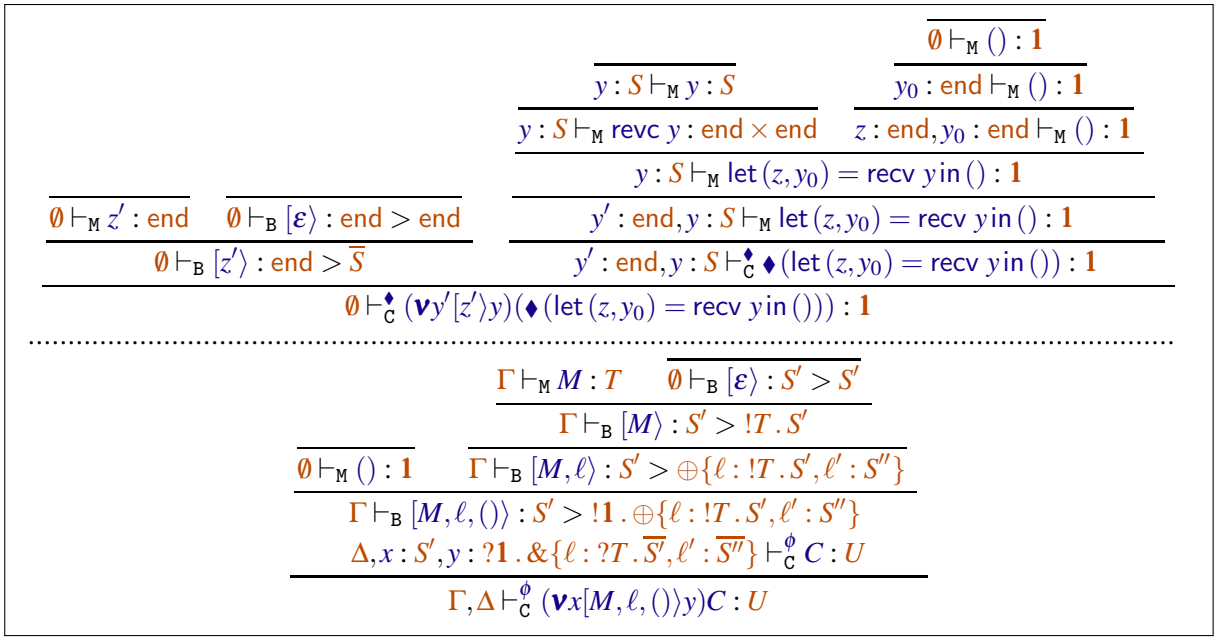

Figure 4: Derivation of configurations (cf. Example 2.5): (top) reduced from the initial one in Example 2.2 ($S = ?\mathsf{end}\,.\,\mathsf{end}$); (bottom) a buffer containing several messages.

3 APCP (Asynchronous Priority-based Classical Processes)

APCP [13] is a linear type system for π-calculus processes that communicate asynchronously (i.e., the output of messages is non-blocking) on connected channel endpoints. The type system assigns to endpoints types that specify two-party protocols, in the style of binary session types [15]. In APCP, well-typed processes may be cyclically connected: types rely on *priority* annotations, which enable cyclic connections while ruling out circular dependencies between sessions. Properties of well-typed APCP processes are *type preservation* (Theorem 3.4) and *deadlock-freedom* (Theorem 3.5).

Syntax and Semantics We write $x, y, z, \ldots$ to denote *endpoints* (or *names*), and write $\tilde{x}, \tilde{y}, \tilde{z}, \ldots$ to denote sequences of endpoints. Also, we write $i, j, k, \ldots$ to denote *labels* and $I, J, K, \ldots$ to denote sets of labels.

Figure 5 (top) gives the syntax and meaning of processes. In APCP, all endpoints are used strictly linearly: each endpoint can be used for exactly one communication only. However, we want to assign session types to endpoints, so we have to be able to implement sequences of communications. Therefore, each communication action carries an additional *continuation endpoint* to continue the session on.

The output action $x[y, z]$ sends a message endpoint y and a continuation endpoint z along x. The input prefix $x(y, z)\,.\,P$ blocks until a message and a continuation endpoint are received on x, binding y and z in P. The selection action $x[z] \triangleleft i$ sends a label i and a continuation endpoint z along x. The branching prefix $x(z) \triangleright \{i : P_i\}_{i \in I}$ blocks until it receives a label $i \in I$ and a continuation endpoint z on x, binding z in each P_i. Restriction $(\boldsymbol{\nu} xy)P$ binds x and y in P to form a channel for communication. The process $P \mid Q$ denotes parallel composition. The process $\mathbf{0}$ denotes inaction. The forwarder process $x \leftrightarrow y$ is a primitive copycat process that links together x and y.

Endpoints are free unless they are bound somehow. We write $\mathrm{fn}(P)$ for the set of free names of P.

Process syntax:

$$\begin{array}{llllll} P,Q ::= x[y,z] & \text{(output)} & \mid\ x(y,z)\,.\,P & \text{(input)} & & \\ \mid\ x[z] \triangleleft i & \text{(selection)} & \mid\ x(z) \triangleright \{i : P\}_{i\in I} & \text{(branching)} & \mid\ (\boldsymbol{\nu} xy)P & \text{(restriction)} \\ \mid\ P \mid Q & \text{(parallel)} & \mid\ \mathbf{0} & \text{(inaction)} & \mid\ x \leftrightarrow y & \text{(forwarder)} \end{array}$$

Structural congruence:

$$\begin{array}{lll} P \equiv P' \quad (\text{if } P \equiv_\alpha P') & P \mid Q \equiv Q \mid P & x \leftrightarrow y \equiv y \leftrightarrow x \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R & P \mid \mathbf{0} \equiv P & (\boldsymbol{\nu} xy)x \leftrightarrow y \equiv \mathbf{0} \\ P \mid (\boldsymbol{\nu} xy)Q \equiv (\boldsymbol{\nu} xy)(P \mid Q) \quad (\text{if } x,y \notin \text{fn}(P)) & (\boldsymbol{\nu} xy)\mathbf{0} \equiv \mathbf{0} & \\ (\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P \equiv (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P & (\boldsymbol{\nu} xy)P \equiv (\boldsymbol{\nu} yx)P & \end{array}$$

Reduction:

$$\frac{z,y \neq x}{(\boldsymbol{\nu} yz)(x \leftrightarrow y \mid P) \longrightarrow P\{x/z\}}\ {\to_{\text{ID}}} \qquad \frac{}{(\boldsymbol{\nu} xy)(x[a,b] \mid y(v,z)\,.\,P) \longrightarrow P\{a/v,b/z\}}\ {\to_{\otimes\parr}}$$

$$\frac{j \in I}{(\boldsymbol{\nu} xy)(x[b] \triangleleft j \mid y(z) \triangleright \{i : P_i\}_{i\in I}) \longrightarrow P_j\{b/z\}}\ {\to_{\oplus\&}} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}\ {\to_{\equiv}}$$

$$\frac{P \longrightarrow Q}{(\boldsymbol{\nu} xy)P \longrightarrow (\boldsymbol{\nu} xy)Q}\ {\to_{\nu}} \qquad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}\ {\to_{\mid}}$$

Figure 5: Definition of APCP's process language.

Also, we write $P\{x/y\}$ to denote the capture-avoiding substitution of the free occurrences of y in P for x. We write sequences of substitutions $P\{x_1/y_1\}\ldots\{x_n/y_n\}$ as $P\{x_1/y_1,\ldots,x_n/y_n\}$.

The reduction relation for processes ($P \longrightarrow Q$) formalizes how complementary actions on connected endpoints may synchronize. As usual for π-calculi, reduction relies on *structural congruence* ($P \equiv Q$), which relates processes with minor syntactic differences; it is the smallest congruence on the syntax of processes (Fig. 5 (top)) satisfying the axioms in Figure 5 (center).

We define the reduction relation $P \longrightarrow Q$ by the axioms and closure rules in Figure 5 (bottom). Rule $\to_{\text{ID}}$ implements the forwarder as a substitution. Rule $\to_{\otimes\parr}$ synchronizes an output and an input on connected endpoints and substitutes the message and continuation endpoints. Rule $\to_{\oplus\&}$ synchronizes a selection and a branch: the received label determines the continuation process, substituting the continuation endpoint appropriately. Rules $\to_{\equiv}$, $\to_{\nu}$, and $\to_{\mid}$ close reduction under congruence, restriction, and parallel composition, respectively. We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$.

The Type System APCP types processes by assigning binary session types to channel endpoints. Following Curry-Howard interpretations, we present session types as linear logic propositions (cf. Caires *et al.* [2] and Wadler [26]) extended with *priority* annotations. Intuitively, actions typed with lower priority cannot be blocked by those with higher priority.

We write $\mathsf{o}, \kappa, \pi, \rho, \ldots$ to denote priorities, and ω to denote the ultimate priority that is greater than all other priorities and cannot be increased further. That is, $\forall \mathsf{o} \in \mathbb{N}.\ \omega > \mathsf{o}$ and $\forall \mathsf{o} \in \mathbb{N}.\ \omega + \mathsf{o} = \omega$.

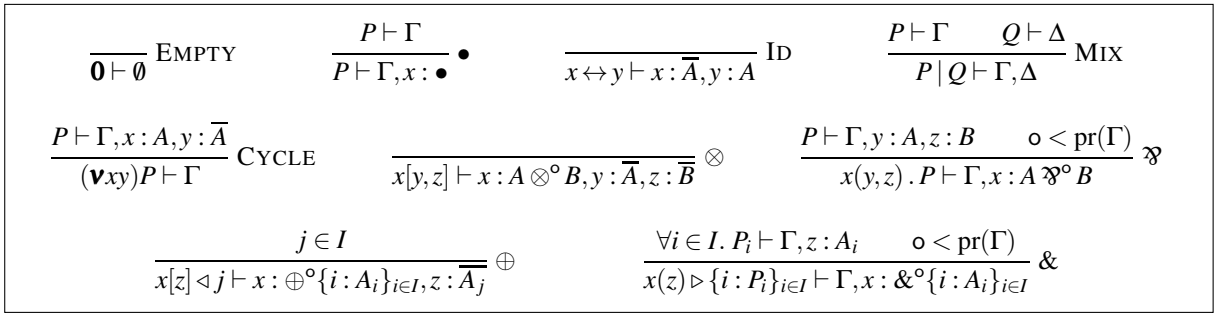

Figure 6: The typing rules of APCP.

Definition 3.1. *The following grammar defines the syntax of* session types A, B. *Let* $\circ \in \mathbb{N}$.

$$A, B ::= A \otimes^{\circ} B \quad \textit{(output)} \mid A \parr^{\circ} B \quad \textit{(input)} \mid \oplus^{\circ}\{i : A\}_{i \in I} \quad \textit{(select)} \mid \&^{\circ}\{i : A\}_{i \in I} \quad \textit{(branch)} \mid \bullet \quad \textit{(end)}$$

Note that type $\bullet$ does not require a priority.

Duality, the cornerstone of session types and linear logic, ensures that the two endpoints of a channel have matching actions. Furthermore, dual types must have matching priority annotations.

Definition 3.2. *The* dual *of session type A, denoted* $\overline{A}$*, is defined inductively as follows:*

$$\overline{A \otimes^{\circ} B} := \overline{A} \parr^{\circ} \overline{B} \qquad \overline{\oplus^{\circ}\{i : A_i\}_{i \in I}} := \&^{\circ}\{i : \overline{A_i}\}_{i \in I} \qquad \overline{\bullet} := \bullet$$
$$\overline{A \parr^{\circ} B} := \overline{A} \otimes^{\circ} \overline{B} \qquad \overline{\&^{\circ}\{i : A_i\}_{i \in I}} := \oplus^{\circ}\{i : \overline{A_i}\}_{i \in I}$$

The priority of a type is determined by the priority of the type's outermost connective:

Definition 3.3. *For session type A,* $\mathrm{pr}(A)$ *denotes its* priority*:*

$$\mathrm{pr}(A \otimes^{\circ} B) := \mathrm{pr}(A \parr^{\circ} B) := \mathrm{pr}(\oplus^{\circ}\{i : A_i\}_{i \in I}) := \mathrm{pr}(\&^{\circ}\{i : A_i\}_{i \in I}) := \circ \qquad \mathrm{pr}(\bullet) := \omega$$

The priority of $\bullet$ is ω: it denotes a "final" action of protocols without blocking behavior. Although associated with non-blocking behavior, $\otimes$ and $\oplus$ do have a non-constant priority: they are connected to $\parr$ and &, respectively, which denote blocking actions.

The typing rules of APCP ensure that actions with lower priority are not blocked by those with higher priority (cf. Dardha and Gay [3]). To this end, typing rules enforce the following laws:

1. An action with priority $\circ$ must be prefixed only by inputs and branches with priority strictly smaller than $\circ$—this law does not hold for output and selection, as they are not prefixes;
2. dual actions leading to a synchronization must have equal priorities (cf. Def. 3.2).

Judgments are of the form $P \vdash \Gamma$, where P is a process and Γ is a context that assigns types to endpoints $(x : A)$. A judgment $P \vdash \Gamma$ then means that P can be typed in accordance with the type assignments for names recorded in Γ. The context Γ obeys *exchange*: assignments may be silently reordered. Γ is *linear*, disallowing *weakening* (i.e., all assignments must be used) and *contraction* (i.e., assignments may not be duplicated). The empty context is written $\emptyset$. In writing $\Gamma, x : A$ we assume that $x \notin \mathrm{dom}(\Gamma)$. We write $\mathrm{pr}(\Gamma)$ to denote the least priority of all types in Γ (cf. Def. 3.3).

Figure 6 gives the typing rules. Rule EMPTY types an inactive process with no endpoints. Rule $\bullet$ silently removes a closed endpoint from the typing context. Rule ID types forwarding between endpoints

$$
\begin{array}{rl|rl}
[\![T \times U]\!] = ([\![T]\!] \parr \bullet) \otimes ([\![U]\!] \parr \bullet) & & [\![!T.S]\!] = (\overline{[\![T]\!]} \otimes \bullet) \parr [\![S]\!] & [\![\oplus\{i : T_i\}_{i\in I}]\!] = \&\{i : [\![T_i]\!]\}_{i\in I} \\
[\![T \multimap U]\!] = (\overline{[\![T]\!]} \otimes \bullet) \parr [\![U]\!] & & [\![?T.S]\!] = ([\![T]\!] \parr \bullet) \otimes [\![S]\!] & [\![\&\{i : T_i\}_{i\in I}]\!] = \oplus\{i : [\![T_i]\!]\}_{i\in I} \\
[\![\mathbf{1}]\!] = \bullet & & [\![\mathsf{end}]\!] = \bullet &
\end{array}
$$

Figure 7: Translation of CGV types into session types.

of dual type. Rule MIX types the parallel composition of two processes that do not share assignments on the same endpoints. Rule CYCLE types a restriction, where the two restricted endpoints must be of dual type. Rule $\otimes$ types an output action; this rule does not have premises to provide a continuation process, leaving the free endpoints to be bound to a continuation process using MIX and CYCLE. Similarly, Rule $\oplus$ types an unbound selection action. Priority checks are confined to Rules $\parr$ and &, which type input and branching prefixes, respectively. In both cases, the used endpoint's priority must be lower than the priorities of the other types in the continuation's typing context, thus enforcing Law 1 above.

Well-typed processes satisfy protocol fidelity, communication safety, and deadlock-freedom. The first two properties follow from *type preservation*. Here we only state these results; see [13] for details.

Theorem 3.4 (Type Preservation)**.** *If $P \vdash \Gamma$ and $P \equiv Q$ or $P \longrightarrow Q$, then $Q \vdash \Gamma$.*

Theorem 3.5 (Deadlock-freedom)**.** *If $P \vdash \emptyset$, then either $P \equiv \mathbf{0}$ or $P \longrightarrow Q$ for some Q.*

4 Translating CGV into APCP

4.1 The Translation

In this section, we translate CGV into APCP. We translate entire typing derivations, following, e.g., Wadler [26]. Given the structure of CGV and its type system, the translation is defined in parts: for (run-time) terms, for configurations, and for buffers. The translation is defined on well-typed configurations which may be deadlocked, so our translation does not consider priority requirements. As we will see, typability in APCP will enable us to identify deadlock-free configurations in CGV (cf. Sec. 4.3).

The translation is informed by the semantics of CGV. It is crucial that subterms may only reduce when they occur in reduction contexts. For example, M_1 and M_2 may not reduce if they appear in a pair (M_1, M_2). The translation must thus ensure that subterms are blocked when they do not occur in reduction contexts. Translations such as Wadler's hinge on blocking outputs and inputs; for example, the pair (M_1, M_2) is translated as an output that blocks the translations of M_1 and M_2. However, outputs in APCP are non-blocking and so we use additional inputs to disable the reduction of subterms. For example, the translation of (M_1, M_2) adds extra inputs to block the translations of M_1 and M_2.

Figure 7 gives the translation of CGV types into APCP types ($[\![T]\!]$), which already captures the operation of the translation: our translation is similar to the one by Wadler, but includes the aforementioned additional inputs. It may seem odd that this translation dualizes CGV session types (e.g., an output '!' becomes an input '$\parr$'). To understand this, consider that a variable x typed $!T.S$ represents access to a session which expects the user to send a term of type T and continue as S, but not the output itself. Hence, to translate an output on x into APCP, we need to connect the translation of x to an actual output. Since this actual output would be typed with $\otimes$, this means that the translation of x would need to be dually typed, i.e., typed with $\parr$. A more technical explanation is that the translation moves from two-sided CGV judgments to one-sided APCP judgments, which requires dualization (see, e.g., [9, 12]).

Importantly, the translation preserves duality of session types (by induction on their structure):

Proposition 4.1. *Given a CGV session type S, $\overline{[\![S]\!]} = [\![\overline{S}]\!]$.*

We extend the translation of types to typing environments, defined as expected. Similarly, we extend duality to typing environments: $\overline{\Gamma}$ denotes Γ with each type dualized. In this section, we give simplified presentations of the translations, showing only the conclusions of the source and target derivations; we include the translations with full derivations in the extended version of this paper [14].

A remark on notation. Some translated terms include annotated restrictions $(\overset{\leftrightarrow}{\boldsymbol{\nu}}xy)$. These so-called *forwarder-enabled* restrictions can be ignored in this subsection, but will be useful later when proving soundness (one of the correctness properties of the translation; cf. Section 4.2).

We define the translation of (the typing rules of) terms. Since a term has a provided type, the translation takes as a parameter a name on which the translation provides this type. Figure 8 gives the translation of terms, denoted $[\![\Gamma \vdash_{\mathtt{M}} \mathbb{M} : T]\!]z$, where the type T is provided on z. By abuse of notation, we write $[\![\mathbb{M}]\!]z$ to denote the process translation of the term $\mathbb{M}$, and similary for configurations and buffers. Notice the aforementioned additional inputs to block behavior of subterms in rules such as Rule T-Pair. Before moving to buffers and configurations, we illustrate the translation of terms by an example:

Example 4.2. *Consider the following subterm from Example 2.3: $(\lambda z\,.\,\mathsf{send}\ ((),z))\ y$. We gradually discuss how this term translates to APCP, and how the translation is set up to mimick the term's behavior.*

$$[\![(\lambda z\,.\,\mathsf{send}\ ((),z))\ y]\!]q = (\boldsymbol{\nu} ab)\big([\![\lambda z\,.\,\mathsf{send}\ ((),z)]\!]a \mid (\boldsymbol{\nu} cd)(b[c,q] \mid d(e,\_)\,.\,[\![y]\!]e)\big)$$

The function application translates the function on a, which is connected to b. The output on b serves to activate the function, which will subsequently activate the functions parameter ($[\![y]\!]e = y \leftrightarrow e$) by means of an output that will be received on d.

$$[\![\lambda z\,.\,\mathsf{send}\ ((),z)]\!]a = a(f,g)\,.\,(\overset{\leftrightarrow}{\boldsymbol{\nu}} hz)((\boldsymbol{\nu}\_\_)f[h,\_] \mid [\![\mathsf{send}\ ((),z)]\!]g)$$

The translation of the function is indeed blocked until it receives on a. It then outputs on f to activate the function's parameter (which receives on d), while the function's body appears in parallel.

$$[\![\mathsf{send}\ ((),z)]\!]g = (\boldsymbol{\nu} kl)\Big([\![((),z)]\!]k \mid l(m,n)\,.\,(\boldsymbol{\nu} op)\big((\boldsymbol{\nu}\_\_)n[o,\_] \mid (\boldsymbol{\nu} rs)(p[m,r] \mid s \leftrightarrow g)\big)\Big)$$

The translation of the send *primitive connects the translation of the pair $((),z)$ on k to an input on l, receiving endpoints for the output term (m) and the output endpoint (n). Once activated by the input on l, the term representing the output endpoint is activated by means of an output on n. In parallel, the actual output (on p) sends the endpoint of the output term (m) and a fresh endpoint (r) representing the continuation channel after the message has been placed in a buffer (the forwarder $s \leftrightarrow g$).*

$$[\![((),z)]\!]k = (\boldsymbol{\nu} tu)(\boldsymbol{\nu} vw)(k[t,v] \mid u(a',\_)\,.\,[\![()]\!]a' \mid w(b',\_)\,.\,[\![z]\!]b')$$

The translation of the pair outputs on k two endpoints for the two terms it contains (to be received by whatever intends to use the pair in the context, e.g., the send *primitive on l). The translations of the two terms inside the pair ($[\![()]\!]a' = \mathbf{0}$ and $[\![z]\!]b' = z \leftrightarrow b'$) are both guarded by an input, preventing the terms from reducing until the context explicitly activates them by means of outputs.*

Analogously to the reductions from Example 2.3—$(\lambda z\,.\,\mathsf{send}\ ((),z))\ y \longrightarrow^3_{\mathtt{M}} \mathsf{send}'((),y)$—we have

$$[\![(\lambda z\,.\,\mathsf{send}\ ((),z))\ y]\!]q \longrightarrow^5 [\![\mathsf{send}'((),y)]\!]q.$$

T-VAR $[\![x : T \vdash_{\mathtt{M}} x : T]\!]z = x \leftrightarrow z \vdash x : \overline{[\![T]\!]}, z : [\![T]\!]$ T-UNIT $[\![\emptyset \vdash_{\mathtt{M}} () : \mathbf{1}]\!]z = \mathbf{0} \vdash z : \bullet$

T-ABS $[\![\Gamma \vdash_{\mathtt{M}} \lambda x.M : T \multimap U]\!]z = z(a,b).(\overleftrightarrow{\boldsymbol{\nu}}cx)((\boldsymbol{\nu}ef)a[c,e] \mid [\![M]\!]b) \vdash \overline{[\![\Gamma]\!]}, z : (\overline{[\![T]\!]} \otimes \bullet) \parr [\![U]\!]$

T-APP $[\![\Gamma,\Delta \vdash_{\mathtt{M}} M\,N : U]\!]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid (\boldsymbol{\nu}cd)(b[c,z] \mid d(e,f).[\![N]\!]e)) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]$

T-PAIR $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} (M,N)\\ : T \times U\end{array}\right]\!\!\right]z = \begin{array}{l}(\boldsymbol{\nu}ab)(\boldsymbol{\nu}cd)(z[a,c] \mid b(e,f).[\![M]\!]e \mid d(g,h).[\![N]\!]g)\\ \quad \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : ([\![T]\!] \parr \bullet) \otimes ([\![U]\!] \parr \bullet)\end{array}$

T-SPLIT $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} \mathsf{let}\,(x,y)\\ = M\,\mathsf{in}\,N : U\end{array}\right]\!\!\right]z = \begin{array}{l}(\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).(\overleftrightarrow{\boldsymbol{\nu}}ex)(\overleftrightarrow{\boldsymbol{\nu}}fy)(\\ \quad (\boldsymbol{\nu}gh)c[e,g] \mid (\boldsymbol{\nu}kl)d[f,k] \mid [\![N]\!]z)) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]\end{array}$

T-NEW $[\![\emptyset \vdash_{\mathtt{M}} \mathsf{new} : S \times \overline{S}]\!]z = \begin{array}{l}(\boldsymbol{\nu}ab)((\boldsymbol{\nu}cd)a[c,d] \mid b(e,f).(\boldsymbol{\nu}xy)[\![(x,y)]\!]z)\\ \quad \vdash z : ([\![S]\!] \parr \bullet) \otimes ([\![\overline{S}]\!] \parr \bullet)\end{array}$

T-SPAWN $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{spawn}\,M : T]\!]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).((\boldsymbol{\nu}ef)c[e,f] \mid (\boldsymbol{\nu}gh)d[z,g])) \vdash \overline{[\![\Gamma]\!]}, z : [\![T]\!]$

T-ENDL $[\![\Gamma, x : \mathsf{end} \vdash_{\mathtt{M}} M : T]\!]z = [\![M]\!]z \vdash \overline{[\![\Gamma]\!]}, x : \bullet, z : [\![T]\!]$ T-ENDR $[\![\emptyset \vdash_{\mathtt{M}} x : \mathsf{end}]\!]z = \mathbf{0} \vdash z : \bullet$

T-SEND $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{send}\,M : S]\!]z = \begin{array}{l}(\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).(\boldsymbol{\nu}ef)((\boldsymbol{\nu}gh)d[e,g]\\ \quad \mid (\boldsymbol{\nu}kl)(f[c,k] \mid l \leftrightarrow z))) \vdash \overline{[\![\Gamma]\!]}, z : [\![S]\!]\end{array}$

T-RECV $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{recv}\,M : T \times S]\!]z = \begin{array}{l}(\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c,d).(\boldsymbol{\nu}ef)(z[c,e] \mid f(g,h).d \leftrightarrow g))\\ \quad \vdash \overline{[\![\Gamma]\!]}, z : ([\![T]\!] \parr \bullet) \otimes ([\![S]\!] \parr \bullet)\end{array}$

T-SELECT $[\![\Gamma \vdash_{\mathtt{M}} \mathsf{select}\,j\,M : T_j]\!]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid (\boldsymbol{\nu}cd)(b[c] \triangleleft j \mid d \leftrightarrow z)) \vdash \overline{[\![\Gamma]\!]}, z : [\![T_j]\!]$

T-CASE $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} \mathsf{case}\,M\\ \mathsf{of}\,\{i : N_i\}_{i\in I} : U\end{array}\right]\!\!\right]z = (\boldsymbol{\nu}ab)([\![M]\!]a \mid b(c) \triangleright \{i : [\![N_i\ c]\!]z\}_{i\in I}) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]$

T-SUB $[\![\Gamma,\Delta \vdash_{\mathtt{M}} \mathbb{M}\{\!|\mathbb{N}/x|\!\} : U]\!]z = (\overleftrightarrow{\boldsymbol{\nu}}xa)([\![\mathbb{M}]\!]z \mid [\![\mathbb{N}]\!]a) \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![U]\!]$

T-SEND' $\left[\!\!\left[\begin{array}{r}\Gamma,\Delta \vdash_{\mathtt{M}} \mathsf{send}'\\ (M,\mathbb{N}) : S\end{array}\right]\!\!\right]z = \begin{array}{l}(\boldsymbol{\nu}ab)(a(c,d).[\![M]\!]c \mid (\boldsymbol{\nu}ef)([\![\mathbb{N}]\!]e \mid (\boldsymbol{\nu}gh)(f[b,g] \mid h \leftrightarrow z)))\\ \quad \vdash \overline{[\![\Gamma]\!]}, \overline{[\![\Delta]\!]}, z : [\![S]\!]\end{array}$

Figure 8: Translation of (runtime) term typing rules. See [14] for typing derivations.

Figure 9 (top) gives the translation of configurations, denoted $[\![\Gamma \vdash^{\phi}_{\mathtt{C}} C : T]\!]z$. We omit the translation of Rule T-PARR. Noteworthy are the translations of buffered restrictions: the translation of $(\boldsymbol{\nu}x[\vec{m}\rangle y)C$ relies on the translation of $[\vec{m}\rangle$, which is given the translation of C as its continuation.

The translation of buffers requires care: each message in the buffer is translated as an output in APCP, where the output of the following messages is on the former output's continuation endpoint. Once there are no more messages in the buffer, the translation uses a typed APCP process—a parameter of the translation—to provide the behavior of the continuation of the lastmost output. The translation has no requirements for the continuation process and its typing, except for the type of the buffer's endpoint. With this in mind, Figure 9 (bottom) gives the translation of the typing rules of buffers, denoted $[\![\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S' > S]\!]_x^{P \vdash^* \Lambda, x:\overline{[\![S']\!]}}$, where x is the endpoint on which the buffer outputs, and P is the continuation of the buffer's last message. Note that we never use the typing rules for buffers by themselves: they always accompany the typing of endpoint restriction, of which the translation properly instantiates the continuation process.

Because CGV configurations may deadlock, the type preservation result of our translation holds up

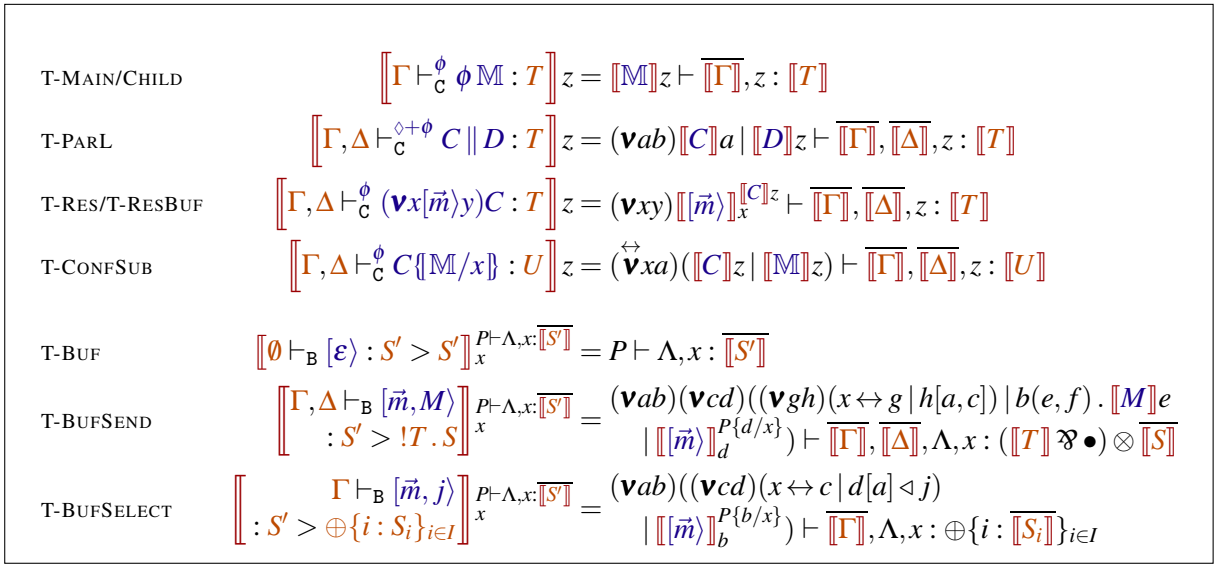

Figure 9: Translation of configuration and buffer typing rules. See [14] for typing derivations.

to priority requirements. To formalize this, we have the following definition:

Definition 4.3. *Let P be a process. We write $P \vdash^* \Gamma$ to denote that P is well-typed according to the typing rules in Figure 6 where Rules $\parr$ and & are modified by erasing priority checks.*

Hence, if $P \vdash \Gamma$ then $P \vdash^* \Gamma$ but the converse does not hold. Our translation correctly preserves the typing of terms, configurations, and buffers:

Theorem 4.4 (Type Preservation for the Translation)**.**

- $\llbracket \Gamma \vdash_{\mathtt{M}} \mathbb{M} : T \rrbracket z = \llbracket \mathbb{M} \rrbracket z \vdash^* \overline{\llbracket \Gamma \rrbracket}, z : \llbracket T \rrbracket$
- $\llbracket \Gamma \vdash^{\phi}_{\mathtt{C}} C : T \rrbracket z = \llbracket C \rrbracket z \vdash^* \overline{\llbracket \Gamma \rrbracket}, z : \llbracket T \rrbracket$
- $\llbracket \Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S' > S \rrbracket_x^{P \vdash^* \Lambda, x : \overline{\llbracket S' \rrbracket}} = \llbracket [\vec{m}\rangle \rrbracket_x^P \vdash^* \overline{\llbracket \Gamma \rrbracket}, \Lambda, x : \llbracket S \rrbracket$

Example 4.5. *Consider again the configuration $(\boldsymbol{\nu} x[M, \ell, ()\rangle y)C$. We illustrate the translation of buffers into APCP by giving the translation of this configuration (writing $\langle x\rangle[a,b]$ to denote the forwarded output $(\boldsymbol{\nu} cd)(x \leftrightarrow c \mid d[a,b])$):*

$$\llbracket (\boldsymbol{\nu} x[M, \ell, ()\rangle y)C \rrbracket z = (\boldsymbol{\nu} xy)\llbracket [M, \ell, ()\rangle \rrbracket_x^{\llbracket C \rrbracket z} = (\boldsymbol{\nu} xy)\ (\boldsymbol{\nu} ab)(\boldsymbol{\nu} cx')(\langle x\rangle[a,c] \mid b(d,\_)\,.\,\mathbf{0} \mid (\boldsymbol{\nu} ex'')(\langle x'\rangle[e] \triangleleft \ell \mid (\boldsymbol{\nu} fg)(\boldsymbol{\nu} hx''')(\langle x''\rangle[f,h] \mid g(k,\_)\,.\,\llbracket M \rrbracket k \mid \llbracket C \rrbracket z\{x'''/x\})))$$

Notice how the (forwarded) outputs are sequenced by continuation endpoints, and how the translation of C uses the last continuation endpoint x''' to interact with the buffer.

4.2 Operational Correctness

Following Gorla [10], we focus on *operational correspondence*: a translated configuration can reproduce all of the source configuration's reductions (completeness; Theorem 4.6), and any of the translated configuration's reductions can be traced back to reductions of the source configuration (soundness; Theorem 4.7). With the soundness result, our translation is stronger than related prior translations [20, 24, 19].

Our completeness result states that the reductions of a well-typed configuration can be mimicked by its translation in zero or more steps.

Theorem 4.6 (Completeness). *Given* $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, *if* $C \longrightarrow_{\mathsf{C}} D$, *then* $[\![C]\!]z \longrightarrow^{*} [\![D]\!]z$.

Proof (Sketch). By induction on the derivation of the configuration's reduction. In each case, we infer the shape of the configuration from the reduction and well-typedness. We then consider the translation of the configuration, and show that the resulting process reduces in zero or more steps to the translation of the reduced configuration. See the extended version of this paper [14] for a full proof. □

Soundness states that any sequence of reductions from the translation of a well-typed configuration eventually leads to the translation of another configuration, which the initial configuration also reduces to. Asynchrony in APCP requires us to be careful, specifically concerning the semantics of variables in CGV. Variables can only cause reductions under specific circumstances. On the other hand, variables translate to forwarders in APCP, which reduce as soon as they are bound by restriction. This semantics for forwarders turns out to be too eager for soundness. As a result, soundness only holds for an alternative, so-called *lazy semantics* for APCP, denoted $\longrightarrow_{\mathsf{L}}$, in which forwarders may only cause reductions under specific circumstances. It is here that the forwarder-enabled restrictions $(\overleftrightarrow{\boldsymbol{\nu}}xy)$ anticipated in Section 4.1 come into play. As we will see in Section 4.3, this alternative semantics does not prevent us from identifying a class of deadlock-free CGV configurations through the translation into APCP. Due to space limitations, the definitions of the lazy semantics only appears in the extended version of this paper [14].

Theorem 4.7 (Soundness). *Given* $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, *if* $[\![C]\!]z \longrightarrow^{*}_{\mathsf{L}} Q$, *then* $C \longrightarrow^{*}_{\mathsf{C}} D$ *and* $Q \longrightarrow^{*}_{\mathsf{L}} [\![D]\!]z$ *for some* D.

Proof (Sketch). By induction on the structure of C. In each case, we additionally apply induction on the number k of steps $[\![C]\!]z \longrightarrow^{k}_{\mathsf{L}} Q$. We then consider which reductions might occur from $[\![C]\!]z$ to Q. Considering the structure of C, we then isolate a sequence of k' possible steps, such that $[\![C]\!]z \longrightarrow^{k'}_{\mathsf{L}} [\![D']\!]z$ for some D' where $C \longrightarrow_{\mathsf{C}} D'$. Since $[\![D']\!]z \longrightarrow^{k-k'}_{\mathsf{L}} Q$, it then follows from the induction hypothesis that there exists D such that $D' \longrightarrow^{*}_{\mathsf{C}} D$ and $[\![D']\!]z \longrightarrow^{*}_{\mathsf{L}} [\![D]\!]z$.

Key here is the independence of reductions in APCP: if two or more reductions are enabled from a (well-typed) process, they must originate from independent parts of the process, and so they do not interfere with each other. This essentially means that the order in which independent reductions occur does not affect the resulting process. Hence, we can pick "desirable" sequences of reductions, postponing other possible reductions. See the extended version of this paper [14] for a full proof of soundness. □

From the proof above we can deduce that if the translation takes at least one step, then so does the source:

Corollary 4.8. *Given* $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, *if* $[\![C]\!]z \longrightarrow^{+}_{\mathsf{L}} Q$, *then* $C \longrightarrow^{+}_{\mathsf{C}} D$ *and* $Q \longrightarrow^{*}_{\mathsf{L}} [\![D]\!]z$ *for some* D.

4.3 Transferring Deadlock-freedom from APCP to CGV

In APCP, well-typed processes typable under empty contexts ($P \vdash \emptyset$) are deadlock-free. By appealing to the operational correctness of our translation, we transfer this result to CGV configurations. Each deadlock-free configuration in CGV obtained via transference satisfies two requirements:

- The configuration is typable $\emptyset \vdash^{\blacklozenge}_{\mathsf{C}} C : \mathbf{1}$: it needs no external resources and has no external behavior.
- The typed translation of the configuration satisfies APCP's priority requirements: it is well-typed under '$\vdash$', not only under '$\vdash^{*}$' (cf. Def. 4.3).

We rely on soundness (Theorem 4.7) to transfer deadlock-freedom to configurations. However, APCP's deadlock-freedom (Theorem 3.5) considers standard semantics ($\longrightarrow$), whereas soundness considers the lazy semantics ($\longrightarrow_{\mathsf{L}}$). Therefore, we first must show that if the translation of a configuration

	λ^{sess} [8]	GV [26]	EGV [7]	PGV [17, 18]	**CGV (this paper)**
Communication	Asynch.	Synch.	Asynch.	Synch.	Asynch.
Cyclic Topologies	Yes	No	No	Yes	Yes
Deadlock-Freedom	No	Yes (typing)	Yes (typing)	Yes (typing)	Yes (via APCP)

Table 1: The features of CGV compared to its predecessors.

satisfying the requirements above reduces under $\longrightarrow$, it also reduces under $\longrightarrow_{\mathtt{L}}$; this is Theorem 4.9 below. The deadlock-freedom of these configurations (Theorem 4.10) then follows from Theorems 3.5 and 4.9. See the extended version of this paper [14] for detailed proofs of these results.

Theorem 4.9. *Given* $\emptyset \vdash^{\blacklozenge}_{\mathtt{C}} C : \mathbf{1}$, *if* $[\![C]\!]z \vdash \Gamma$ *for some* Γ *and* $[\![C]\!]z \longrightarrow Q$, *then* $[\![C]\!]z \longrightarrow_{\mathtt{L}} Q'$, *for some* Q'.

Proof (Sketch). By inspecting the derivation of $[\![C]\!]z \longrightarrow Q$. If the reduction is not derived from $\rightarrow_{\mathrm{ID}}$, it can be directly replicated under $\longrightarrow_{\mathtt{L}}$. Otherwise, we analyze the possible shapes of C and show that a different reduction under $\longrightarrow_{\mathtt{L}}$ is possible. □

Theorem 4.10 (Deadlock-freedom for CGV)**.** *Given* $\emptyset \vdash^{\blacklozenge}_{\mathtt{C}} C : \mathbf{1}$, *if* $[\![C]\!]z \vdash \Gamma$ *for some* Γ, *then* $C \equiv \blacklozenge()$ *or* $C \longrightarrow_{\mathtt{C}} D$ *for some* D.

Proof (Sketch). By assumption and Theorem 4.4, $[\![C]\!]z \vdash z : \bullet$. Then $(\boldsymbol{\nu} z\_)[\![C]\!]z \vdash \emptyset$. By Theorem 3.5, (i) $(\boldsymbol{\nu} z\_)[\![C]\!]z \equiv \mathbf{0}$ or (ii) $(\boldsymbol{\nu} z\_)[\![C]\!]z \longrightarrow Q$ for some Q. In case (i) it follows from the well-typedness and translation of C that $C \equiv_{\mathtt{C}} \blacklozenge()$. In case (ii) we deduce that the reduction of $(\boldsymbol{\nu} z\_)[\![C]\!]z$ cannot involve the endpoint z. Hence, $[\![C]\!]z \longrightarrow Q_0$ for some Q_0. By Theorem 4.9, then $[\![C]\!]z \longrightarrow_{\mathtt{L}} Q'$ for some Q'. Then, by Corollary 4.8, there exists D' such that $C \longrightarrow^{+}_{\mathtt{C}} D'$. Hence, $C \longrightarrow_{\mathtt{C}} D$ for some D, proving the thesis. □

As an example, using Theorem 4.10 we can show that C_1 from Example 2.2 is deadlock-free; see [14].

5 Conclusion

We have presented CGV, a new functional language with asynchronous session-typed communication. As illustrated in Section 1, CGV is strictly more expressive than its predecessors, thanks to a highly asynchronous semantics (compared to GV and PGV), its support for cyclic thread configurations (compared to EGV), and the ability to send whole terms and not just values (compared to all the mentioned calculi). Table 1 summarizes the features of CGV compared to its predecessors.

An operationally correct translation into APCP solidifies the design of CGV, and enables identifying a class of deadlock-free CGV programs. Interestingly, the asynchronous semantics of CGV is reminiscent of *future/promise* programming paradigms (see, e.g., [11, 21, 25]), which have been little studied in the context of session-typed communication.

The alternative to establishing deadlock-freedom in CGV via translation into APCP would be to enhance CGV's type system with priorities (in the spirit of, e.g., work by Padovani and Novara [23]). Another useful addition concerns recursion / recursive types. We leave these extensions to future work.

Acknowledgments Thanks to Simon Fowler and the anonymous reviewers for their helpful feedback. We gratefully acknowledge the support of the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

References

[1] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[2] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear Logic Propositions as Session Types*. *Mathematical Structures in Computer Science* 26(3), pp. 367–423, doi:10.1017/S0960129514000218.

[3] Ornela Dardha & Simon J. Gay (2018): *A New Linear Logic for Deadlock-Free Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, Springer International Publishing, pp. 91–109, doi:10.1007/978-3-319-89366-2_5.

[4] Ornela Dardha & Jorge A. Pérez (2015): *Comparing Deadlock-Free Session Typed Processes*. *Electronic Proceedings in Theoretical Computer Science* 190, pp. 1–15, doi:10.4204/EPTCS.190.1. arXiv:1508.06707.

[5] Ornela Dardha & Jorge A. Pérez (2022): *Comparing Type Systems for Deadlock Freedom*. *Journal of Logical and Algebraic Methods in Programming* 124, p. 100717, doi:10.1016/j.jlamp.2021.100717.

[6] Simon Fowler (2019): *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. Ph.D. thesis, University of Edinburgh.

[7] Simon Fowler, Sam Lindley, J. Garrett Morris & Sára Decova (2019): *Exceptional Asynchronous Session Types: Session Types without Tiers*. *Proceedings of the ACM on Programming Languages*, doi:10.1145/3290341.

[8] Simon J. Gay & Vasco T. Vasconcelos (2010): *Linear Type Theory for Asynchronous Session Types*. *Journal of Functional Programming* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[9] Jean-Yves Girard (1993): *On the Unity of Logic*. *Annals of Pure and Applied Logic* 59(3), pp. 201–217, doi:10.1016/0168-0072(93)90093-S.

[10] Daniele Gorla (2010): *Towards a Unified Approach to Encodability and Separation Results for Process Calculi*. *Information and Computation* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[11] Robert H. Halstead (1985): *MULTILISP: A Language for Concurrent Symbolic Computation*. *ACM Transactions on Programming Languages and Systems* 7(4), pp. 501–538, doi:10.1145/4472.4478.

[12] Bas van den Heuvel & Jorge A. Pérez (2020): *Session Type Systems Based on Linear Logic: Classical versus Intuitionistic*. *Electronic Proceedings in Theoretical Computer Science* 314, pp. 1–11, doi:10.4204/EPTCS.314.1. arXiv:2004.01320.

[13] Bas van den Heuvel & Jorge A. Pérez (2021): *Deadlock Freedom for Asynchronous and Cyclic Process Networks (Extended Version)*. *arXiv:2111.13091 [cs]*. arXiv:2111.13091. A short version appears in the Proceedings of ICE'21: arXiv:2110.00146.

[14] Bas van den Heuvel & Jorge A. Pérez (2022): *Asynchronous Functional Sessions: Cyclic and Concurrent (Extended Version)*, doi:10.48550/arXiv.2208.07644. arXiv:2208.07644.

[15] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR'93*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[16] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In Christel Baier & Holger Hermanns, editors: *CONCUR 2006 – Concurrency Theory*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 233–247, doi:10.1007/11817949_16.

[17] Wen Kokke & Ornela Dardha (2021): *Prioritise the Best Variation*. In Kirstin Peters & Tim A. C. Willemse, editors: *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 100–119, doi:10.1007/978-3-030-78089-0_6.

[18] Wen Kokke & Ornela Dardha (2021): *Prioritise the Best Variation*, doi:10.48550/arXiv.2103.14466. arXiv:2103.14466.

[19] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In Jan Vitek, editor: *Programming Languages and Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[20] Robin Milner (1989): *Communication and Concurrency*. Prentice Hall International Series in Computer Science, Prentice Hall, New York, USA.

[21] Gerald K. Ostheimer & Antony J. T. Davie (1993): *Pi-Calculus Characterizations of Some Practical Lambda-Calculus Reduction Strategies*. Technical Report CS/93/14, Department of Computing Sciences, University of St Andrews.

[22] Luca Padovani (2014): *Deadlock and Lock Freedom in the Linear π-Calculus*. In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, ACM, New York, NY, USA, pp. 72:1–72:10, doi:10.1145/2603088.2603116.

[23] Luca Padovani & Luca Novara (2015): *Types for Deadlock-Free Higher-Order Programs*. In Susanne Graf & Mahesh Viswanathan, editors: *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 3–18, doi:10.1007/978-3-319-19195-9_1.

[24] Davide Sangiorgi & David Walker (2003): *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

[25] G. Tremblay & B. Malenfant (2000): *Lenient Evaluation and Parallelism*. *Computer Languages* 26(1), pp. 27–41, doi:10.1016/S0096-0551(01)00007-8.

[26] Philip Wadler (2012): *Propositions As Sessions*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, ACM, New York, NY, USA, pp. 273–286, doi:10.1145/2364527.2364568.