Logical Methods in Computer Science
Volume 20, Issue 4, 2024, pp. 6:1–6:103
https://lmcs.episciences.org/

Submitted Oct. 07, 2022
Published Oct. 07, 2024

ASYNCHRONOUS SESSION-BASED CONCURRENCY: DEADLOCK FREEDOM IN CYCLIC PROCESS NETWORKS

BAS VAN DEN HEUVEL [a] AND JORGE A. PÉREZ [b]

[a] HKA Karlsruhe and University of Freiburg, Germany

[b] University of Groningen, The Netherlands

Abstract. We tackle the challenge of ensuring the deadlock-freedom property for message-passing processes that communicate asynchronously in cyclic process networks. Our contributions are twofold. First, we present Asynchronous Priority-based Classical Processes (APCP), a session-typed process framework that supports asynchronous communication, delegation, and recursion in cyclic process networks. Building upon the Curry-Howard correspondences between linear logic and session types, we establish essential meta-theoretical results for APCP, most notably deadlock freedom. Second, we present a new concurrent λ-calculus with asynchronous session types, dubbed LAST^n. We illustrate LAST^n by example and establish its meta-theoretical results; in particular, we show how to soundly transfer the deadlock-freedom guarantee from APCP. To this end, we develop a translation of terms in LAST^n into processes in APCP that satisfies a strong formulation of operational correspondence.

1. Introduction

Modern software systems often comprise independent components that coordinate by exchanging messages. The π-calculus [MPW92, Mil89] is a mature formalism for specifying and reasoning about message-passing processes; in particular, it offers a rigorous foundation for designing *type systems* that statically enforce communication correctness. A well-known approach in this line is *session types* [Hon93, HVK98, YV07], which specify the structure of the two-party communication protocols implemented by the channels of a process. In this paper, we are interested in session types as a governing discipline in concurrent and functional paradigms, in conjunction with two important aspects of message-passing concurrency, namely the *network topologies* formed by interacting processes and the underlying discipline of *asynchronous communication*.

The study of session types has gained considerable attention after the discovery by Caires and Pfenning [CP10] and Wadler [Wad12] of Curry-Howard correspondences between session types and linear logic [Gir87]. The present work is motivated by (and develops further) two salient features of type systems derived from these correspondences, namely (i) their remarkably effective approach to establishing the *deadlock-freedom* property for processes,

Key words and phrases: concurrency, process calculi, asynchronous communication, session types.

DOI:10.46298/LMCS-20(4:6)2024

and (ii) their clear connections with *functional calculi* with concurrency. These two aspects form the central themes of this paper, and we discuss them in order.

Deadlock Freedom. Curry-Howard approaches to session types induce a very precise form of interaction between parallel processes: they interpret the cut rule of linear logic as the interaction of two processes on *exactly one channel*. While this design elegantly rules out the insidious circular dependencies that lead to deadlocks, there is a catch: typable processes cannot be connected to form *cyclic* network topologies—only tree-shaped networks are allowed. Hence, type systems based upon Curry-Howard foundations reject whole classes of process networks that are cyclic but deadlock-free [DP15, DP22]. This includes important concurrency patterns, such as those exemplified by Milner's cyclic scheduler [Mil89].

The problem of designing type systems that rule out circular dependencies and deadlocks while allowing for cyclic topologies has received considerable attention. Works by Kobayashi and others have developed advanced solutions; see, e.g., [Kob06, Pad14, DG18]. In a nutshell, these works exploit orderings based on priority annotations on types to detect and avoid circular dependencies. Dardha and Gay [DG18] have shown how to incorporate this priority-based approach in the realm of session type systems based on linear logic; it boils down to replacing the cut rule with a cycle rule and adding priority checks in other selected typing rules. Their work thus extends the class of typable processes to cover cyclic network topologies, while retaining strong ties with linear logic.

Unfortunately, none of the methods proposed until now consider *session types with asynchronous communication*. Addressing asynchronous communication is of clear practical relevance: it is the standard in most distributed systems and web-based applications nowadays. In a process calculi setting, asynchronous communication means that output prefixes are non-blocking [HT91, HT92, Bou92], and that exchanged messages implicitly or explicitly reside in an auxiliary structure, such as a buffer or a queue [BPV08]. In the context of session types, asynchrony moreover means that the ordering of messages *within a session* should be respected, but messages from *different sessions* need not be ordered [KYH11].

To address this gap, in the first part of the paper we define a new session-typed asynchronous π-calculus, `APCP` *(Asynchronous Priority-based Classical Processes)*, for which we develop its fundamental meta-theoretical results. The design of `APCP` builds upon insights developed in several prior works:

- Advanced type systems that exploit annotations on types to enforce deadlock freedom of cyclic process networks, already mentioned;
- Dardha and Gay's `PCP` (Priority-based Classical Processes) [DG18], also already mentioned, which incorporates into Wadler's `CP` [Wad12] (Classical Processes; derived from classical linear logic) Padovani's simplification of Kobayashi's type annotations [Pad14].
- DeYoung *et al.*'s asynchronous semantics for session communication, defined in context of the correspondence between intuitionistic linear logic and session types [DCPT12].

Our calculus `APCP` combines these semantics for asynchronous communication with `PCP`'s priority-based type system. The design of `APCP` uncovers fundamental properties of type systems for asynchronous communication. A particular insight is the following: because outputs are non-blocking, `APCP` simplifies priority management while preserving deadlock freedom. Additionally, as an orthogonal feature, `APCP` increases expressivity by supporting tail recursion without compromising deadlock freedom. We motivate these features of `APCP` by discussing Milner's cyclic scheduler in Section 2.1.

Functional Calculi with Concurrency. Session types are paradigm-independent, in the sense that they can be accommodated on top of programming models and languages in different paradigms—concurrent, object-oriented, and functional. In the functional setting, a milestone is the asynchronous concurrent λ-calculus with sessions by Gay and Vasconcelos [GV10], which in the following we shall call LAST.[1] LAST is a call-by-value calculus in which collections of threads (configurations) communicate following session protocols by relying on buffered channels. The type system of LAST ensures that well-typed configurations respect their ascribed protocols (protocol fidelity) but does not guarantee deadlock freedom, i.e., it allows the typing of cyclic configurations with circular dependencies.

LAST has been brought back to the spotlight through Wadler's work on GV (Good Variation), a synchronous variant of LAST without cyclic configurations. Terms in GV are guaranteed to be deadlock-free via a (typed) translation into CP [Wad12]. Subsequently, Kokke and Dardha formulated PGV (Priority-based GV), an extension of GV that strictly augments the class of deadlock-free computations with cyclic configurations by leveraging priorities [KD21a, KD21b], following Padovani and Novara [PN15].

In the second part of the paper, we study asynchronous, deadlock-free communication with support for cyclic topologies in the setting of a prototypical functional programming language. We present LAST^n, a new call-by-name variant of LAST. Notably, session communication in LAST^n is asynchronous—a feature not accounted for by GV and PGV (see also Section 6 for extended comparisons).

We equip LAST^n with a deliberately simple type system, with functional and session types, which ensures type preservation/protocol fidelity but not deadlock freedom (just like the type system for LAST). To address this gap, we develop a way of soundly transferring the deadlock-freedom property from APCP to LAST^n. This transference of results hinges on a translation of LAST^n programs into APCP processes, in the style for Milner's seminal work [Mil90, Mil92]. The translation clarifies the role of APCP as an abstract model for asynchronous, functional concurrency; it satisfies in particular a tight form of *operational correspondence* that follows the well-known formulation by Gorla [Gor10]. This way, we can ensure that a (class of) well-typed LAST^n programs with cyclic configurations satisfies deadlock freedom. While the development of LAST^n is of interest in itself (it improves over GV and PGV, as just discussed), it is also a significant test for APCP, its expressiveness and meta-theoretical results.

Contributions. In summary, in this paper we make the following contributions:

(1) The process calculus APCP, its associated type system, and its essential meta-theoretical properties: type preservation (Theorem 3.23) and deadlock freedom (Section 3.3).
(2) The functional calculus LAST^n, its associated type system, and its meta-theoretical property of type preservation (Appendix A.2).
(3) A translation of LAST^n into APCP that enjoys operational correspondence properties (Section 5.2) and unlocks the transfer of deadlock freedom from APCP to LAST^n (Theorem 5.8).

Organization. In Section 2, we motivate APCP and LAST^n by example. Section 3 defines the language of APCP processes and its type system, and establishes its meta-theoretical properties. Section 4 recalls LAST and its type system, as proposed in [GV10], and briefly discusses the issue of devising an operationally correct translation into APCP. Building upon this background, Section 5 presents LAST^n as a call-by-name variant of LAST, develops its

[1] LAST stands for 'Linear Asynchronous Session Types'.

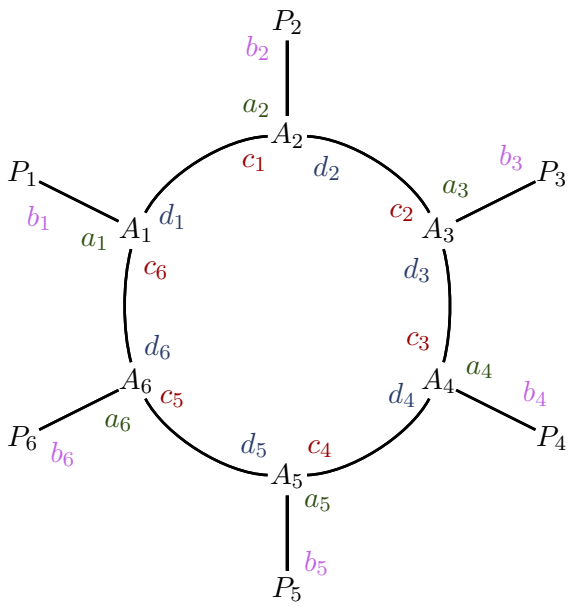


Figure 1: Milner's cyclic scheduler with 6 workers. Lines denote channels connecting processes on the indicated names.

meta-theoretical results, and gives a correct translation of $\texttt{LAST}^n$ into APCP. Section 6 discusses related work and Section 7 draws conclusions. Appendix A collects omitted definitions and proofs for $\texttt{LAST}^n$.

Origin of the results. The current paper combines, revises, and extends our papers [HP21, HP22a]. Here we provide fully detailed proofs, expanded examples, and consolidated comparisons of related works. While the paper [HP21] offered an abridged introduction to APCP, the paper [HP22a] developed CGV, a functional calculus with concurrency (also in the style of Gay and Vasconcelos' LAST), and gave a correct translation into APCP. Novelties with respect to [HP21] include a revised treatment of type preservation and recursion in APCP. With respect to [HP22a], in this presentation we have revised CGV into $\texttt{LAST}^n$, in order to showcase a more clear connection with LAST and a well-known reduction strategy (call by name). As a result, Sections 4 and 5 are entirely new to this presentation.

2. Motivating Examples

In this section, we informally describe APCP and $\texttt{LAST}^n$, the two calculi presented in this paper, using motivating examples that illustrate their distinctive features and expressiveness.

2.1. Milner's Cyclic Scheduler in APCP.

We motivate APCP by considering Milner's cyclic scheduler [Mil89], a recursive process that relies on a cyclic network to perform asynchronous communications. This example is inspired by Dardha and Gay [DG18], who use PCP to type a synchronous, non-recursive version of the scheduler.

The Scheduler, Informally. The scheduler consists of $n \geq 1$ worker processes P_i (the workers, for short), each attached to a partial scheduler A_i. The partial schedulers connect to each other in a ring structure, together forming the cyclic scheduler. Connections consist of pairs of endpoints; we further refer to these endpoints by the names that represent them.

The scheduler then lets the workers perform their tasks in rounds, each new round triggered by the leading partial scheduler A_1 (the *leader*) once each worker finishes their previous task. We refer to the non-leading partial schedulers A_{i+1} for $1 \leq i < n$ as the *followers*.

Figure 1 illustrates the process network of Milner's cyclic scheduler with 6 workers ($n = 6$). Each partial scheduler A_i has a name a_i to connect with the worker P_i's name b_i. The leader A_1 has a name c_n to connect with A_n and a name d_1 to connect with A_2 (or with A_1 if $n = 1$; we further elide this case for brevity). Each follower A_{i+1} has a name c_i to connect with A_i and a name d_{i+1} to connect with A_{i+2} (or with A_1 if $i + 1 = n$; we also elide this case).

In each round, each follower A_{i+1} awaits a start signal from A_i, and then asynchronously signals P_{i+1} and A_{i+2} to start. After awaiting acknowledgment from P_{i+1} and a next round signal from A_i, the follower then signals next round to A_{i+2}. The leader A_1, which starts each round of tasks, signals A_2 and P_1 to start, and, after awaiting acknowledgment from P_1, signals a next round to A_2. Then, the leader awaits A_n's start and next round signals. It is crucial that A_1 does not await A_n's start signal before starting P_1, as the leader would otherwise not be able to initiate rounds of tasks.

Syntax of `APCP`. Before formally specifying the scheduler, we briefly introduce the syntax of `APCP`, and how asynchronous communication works; Section 3 gives formal definitions.

Let us write $a, b, c, \ldots, x, y, z, \ldots$ to denote names. We write $x[a, b]$ and $y(w, z); P$ to denote processes for sending and receiving, respectively. The names a and w stand for the payloads, whereas b and z stand for *continuations*, i.e., the names on which the rest of the session should be performed. This *continuation-passing style* for asynchronous communication is required to ensure the correct ordering of messages within a session. A communication redex in `APCP` is thus of the form $(\boldsymbol{\nu} xy)(x[a, b] \mid y(w, z); P)$: the restriction $(\boldsymbol{\nu} xy)$ serves to declare that x and y are dual names of the same channel, and $\cdot \mid \cdot$ denotes parallel composition. This process reduces to $P\{a/w, b/z\}$: P's names w and z are substituted for by a and b, respectively. Notice how, since the send is a standalone process, it cannot block any other communications in the process.

The communication of labels $\ell, \ell', \ldots$ follows the same principle: the process $x[b] \triangleleft \ell$ denotes the output of a label ℓ on x, and the process $y(z) \triangleright \ell; P$ blocks until a label ℓ is received on y before continuing as P; here again, b and z are continuation names that are sent and received together with ℓ. Finally, process $\mu X(\tilde{x}); P$ denotes a recursive definition. Here, P has access to the names in $\tilde{x}$ and may contain recursive calls $X\langle \tilde{y} \rangle$ to indicate a repetition of P. Upon such a recursive call $X\langle \tilde{y} \rangle$, the names in $\tilde{y}$ are assigned to $\tilde{x}$ in the next round of P.

The Scheduler in `APCP`. We now formally specify the partial schedulers. Because each sent label requires a restriction to bind the selection's continuation name to the rest of the session, these processes may look rather complicated. For example, (part of) the leader is specified as follows (where d_1', a_1'' are used in the omitted remainder):

$$A_1 \triangleq \mu X(a_1, c_n, d_1); (\boldsymbol{\nu} a d_1')(d_1[a] \triangleleft \mathsf{start} \mid (\boldsymbol{\nu} b a_1')(a_1[b] \triangleleft \mathsf{start} \mid a_1'(a_1'') \triangleright \mathsf{ack}; \ldots))$$

To improve readability, we rely on the notation $\overline{x} \triangleleft \ell \cdot P$, which is syntactic sugar that elides the continuation name involved (cf. Notation 3.2); our use of the floating dot '$\cdot$' is intended to stress that the communication of ℓ along x does not block any further prefixes in P, and the overline '$\overline{\ }$' to indicate that there is a hidden restriction. Similarly, the notation $x \triangleright \ell; P$ elides the continuation name involved in receiving labels, though it remains blocking hence the '$;$'.

The leader and followers are then specified as follows:

$$\begin{aligned} A_1 &\triangleq \mu X(a_1, c_n, d_1); \overline{d_1} \triangleleft \mathsf{start} \cdot \overline{a_1} \triangleleft \mathsf{start} \cdot a_1 \triangleright \mathsf{ack}; \overline{d_1} \triangleleft \mathsf{next} \cdot \\ &\qquad c_n \triangleright \mathsf{start}; c_n \triangleright \mathsf{next}; X\langle a_1, c_n, d_1\rangle \\ A_{i+1} &\triangleq \mu X(a_{i+1}, c_i, d_{i+1}); c_i \triangleright \mathsf{start}; \overline{a_{i+1}} \triangleleft \mathsf{start} \cdot \overline{d_{i+1}} \triangleleft \mathsf{start} \cdot a_{i+1} \triangleright \mathsf{ack}; \qquad \forall 1 \leq i < n \\ &\qquad c_i \triangleright \mathsf{next}; \overline{d_{i+1}} \triangleleft \mathsf{next} \cdot X\langle a_{i+1}, c_i, d_{i+1}\rangle \end{aligned}$$

Assuming that each worker P_i is specified such that it behaves as expected on the name b_i, we formally specify the complete scheduler as a ring of partial schedulers connected to workers:

$$Sched_n \triangleq (\boldsymbol{\nu} c_1 d_1) \ldots (\boldsymbol{\nu} c_n d_n)\big((\boldsymbol{\nu} a_1 b_1)(A_1 \mid P_1) \mid \ldots \mid (\boldsymbol{\nu} a_n b_n)(A_n \mid P_n)\big)$$

We return to this example in Section 3.5, where we type check the scheduler using APCP to show that it is deadlock-free (which is not obvious from its process definition).

2.2. **A Bookshop Scenario in LAST^n.** Our new calculus LAST^n is a call-by-name variant of Gay and Vasconcelos' LAST [GV10] with linear resources. We briefly motivate the design of LAST^n by adapting the running example of [GV10], which involves a mother interacting with a bookshop to buy a book for her son.

First, we define a term representing the shop. The shop has an endpoint s on which it communicates with a client. First, the shop receives a book title and then offers a choice between buying the book or only accessing its blurb (the text on the book's back cover). If the client decides to buy, the shop receives credit card information and sends the book to the client. Otherwise, if the client requests the blurb, the shop sends its text. In LAST^n, we can define this shop as follows:

$$\begin{aligned} \mathsf{Shop}(s) \triangleq\ & \mathtt{let}\,(\mathit{title}, s_1) = \mathtt{recv}\,s\,\mathtt{in} \\ & \mathtt{case}\,s_1\,\mathtt{of}\,\{\mathsf{buy} : \lambda s_2.\mathtt{let}\,(\mathit{card}, s_3) = \mathtt{recv}\,s_2\,\mathtt{in} \\ & \qquad\qquad \mathtt{let}\,s_4 = \mathtt{send}\,\mathrm{book}(\mathit{title})\,s_3\,\mathtt{in} \\ & \qquad\qquad \mathtt{close}\,s_4; (), \\ & \qquad \mathsf{blurb} : \lambda s_2.\mathtt{let}\,s_3 = \mathtt{send}\,\mathrm{blurb}(\mathit{title})\,s_2\,\mathtt{in} \\ & \qquad\qquad \mathtt{close}\,s_3; ()\} \end{aligned}$$

where notations book(title) and blurb(title) are syntactic sugar for lookup functions, implemented, e.g., as labeled selections.

The functional behavior of LAST^n is standard, so we only explain the message-passing components in the above term:

- '$\mathtt{recv}\,s$' waits for a message to be ready on s, and returns a pair containing the message and an endpoint on which to continue the session.
- '$\mathtt{case}\,s\,\mathtt{of}\,\{\ldots\}$' waits for a label to be ready on s, determining a continuation. Continuations are defined as abstractions, which will be applied to the session's continuation once the label has been received.

- '$\texttt{send}\, M\, s$' buffers the message M on endpoint s, which can be received asynchronously on a connected endpoint.
- '$\texttt{close}\, s; M$' waits until the session on the endpoint s can be closed before running M, which acts as a continuation.

Next, we define a term abstracting the son's behavior. Term $\mathsf{Son}(s', m')$, given below, has an endpoint s' on which he communicates with the shop, and an endpoint m' for communication with his mother. The son sends a book title to the shop, and then selects to buy. Notation '$\texttt{select}\, \ell\, \mathtt{s}$' denotes the buffering of the label ℓ on endpoint s, to be received asynchronously on a connected endpoint. After this selection, to let his mother pay for him, the son proceeds to send the shop's endpoint to his mother. Finally, he receives the book from his mother, and returns it as a result of the computations.

$$\begin{aligned}\mathsf{Son}(s', m') \triangleq\ & \texttt{let}\, s'_1 = \texttt{send}\, \text{“Dune”}\, s'\, \texttt{in} \\ & \texttt{let}\, s'_2 = \texttt{select}\, \mathsf{buy}\, s'_1\, \texttt{in} \\ & \texttt{let}\, m'_1 = \texttt{send}\, s'_2\, m'\, \texttt{in} \\ & \texttt{let}\, (\mathit{book}, m'_2) = \texttt{recv}\, m'_1\, \texttt{in} \\ & \texttt{close}\, m'_2; \mathit{book}\end{aligned}$$

Finally, we define the mother, who has an endpoint m to communicate with her son. She receives the shop endpoint from her son, and then sends her credit card information to the shop. She then receives the book from the shop, and sends it to her son.

$$\begin{aligned}\mathsf{Mother}(m) \triangleq\ & \texttt{let}\, (x, m_1) = \texttt{recv}\, m\, \texttt{in} \\ & \texttt{let}\, x_1 = \texttt{send}\, \mathrm{visa}\, x\, \texttt{in} \\ & \texttt{let}\, (\mathit{book}, x_2) = \texttt{recv}\, x_1\, \texttt{in} \\ & \texttt{let}\, m_2 = \texttt{send}\, \mathit{book}\, m_1\, \texttt{in} \\ & \texttt{close}\, m_2; \texttt{close}\, x_2; ()\end{aligned}$$

Now, we only have to compose these terms together, connecting all the endpoints appropriately. This is achieved by term Sys below, which relies on two additional constructs.

- '$\texttt{new}$' creates a new channel, and returns a pair containing the channel's two endpoints; asynchronous communication is achieved by connecting the endpoints through an ordered buffer.
- '$\texttt{fork}\, M; N$' denotes splitting of M as a *thread*, which runs concurrently to the immediate continuation N.

Concretely, we have:

$$\begin{aligned}\mathsf{Sys} \triangleq\ & \texttt{let}\, (s, s') = \texttt{new}\, \texttt{in}\, \texttt{fork}\, \mathsf{Shop}(s); \\ & \texttt{let}\, (m, m') = \texttt{new}\, \texttt{in}\, \texttt{fork}\, \mathsf{Mother}(m); \\ & \mathsf{Son}(s', m')\end{aligned}$$

Note that the son cannot be forked, as he returns the result of the computation (the book). In Sections 4 and 5 we will discuss the behavior and typing of these terms.

After presenting APCP next, in Section 5 we formally define $\texttt{LAST}^n$ and illustrate the language and its runtime semantics: how new channels are created, new threads are forked, and messages are written to and read from buffers.

3. APCP: Asynchronous Priority-based Classical Processes

In this section, we define APCP, a session-typed π-calculus in which processes communicate asynchronously on connected channel endpoints. We further refer to endpoints by the names that represent them. As already discussed, the output of messages (names and labels) is non-blocking, and explicit continuations ensure the ordering of messages within a session. In our type system, names are assigned types that specify two-party protocols, in the style of binary session types [Hon93], following the Curry-Howard correspondences between linear logic and session types [CP10, Wad12].

APCP combines the salient features of Dardha and Gay's PCP [DG18] with DeYoung *et al.*'s semantics for asynchronous communication [DCPT12]. Recursion—not present in the works by Dardha and Gay and DeYoung *et al.*—is an orthogonal feature, with syntax inspired by the work of Toninho *et al.* [TCP14].

As in PCP, types in APCP rely on *priority* annotations, which enable cyclic connections while ruling out circular dependencies between sessions. A key insight of our work is that asynchrony induces different priority management than synchrony: while PCP's blocking outputs are compared to their continuation processes, APCP's non-blocking outputs are only compared to their continuation *sessions* (see Remark 3.15).

Properties of well-typed APCP processes are *type preservation* (Theorem 3.23) and *deadlock freedom* (Section 3.3). This includes cyclically connected processes, which priority-annotated types guarantee free from circular dependencies that may cause deadlock.

3.1. The Process Language.

We write $a, b, c, \ldots, x, y, z, \ldots$ to denote (channel) *names* (also known as *names*); by convention we use the early letters of the alphabet for the objects of output-like prefixes. Also, we write $\tilde{x}, \tilde{y}, \tilde{z}, \ldots$ to denote sequences of names. In APCP, communication is *asynchronous* (cf. [HT91, HT92, Bou92]) and *dyadic*: each communication involves the transmission of a pair of names, a message name and a continuation name. With a slight abuse of notation, we sometimes write $x_i \in \tilde{x}$ to refer to a specific element in the sequence $\tilde{x}$. Also, we write $i, j, k, \ldots$ to denote *labels* for choices and $I, J, K, \ldots$ to denote sets of labels. We write $X, Y, \ldots$ to denote *recursion variables*, and $P, Q, \ldots$ to denote processes.

Definition 3.1 (APCP Syntax)**.** The syntax of APCP processes is as follows:

$P, Q ::=$	$x[a, b]$	send	$\mid$	$x(y, z); P$	receive
$\mid$	$x[b] \triangleleft \ell$	selection	$\mid$	$x(z) \triangleright \{i : P\}_{i \in I}$	branch
$\mid$	$(\boldsymbol{\nu} xy)P$	restriction	$\mid$	$P \mid Q$	parallel
$\mid$	$\mathbf{0}$	inaction	$\mid$	$[x \leftrightarrow y]$	forwarder
$\mid$	$\mu X(\tilde{z}); P$	recursive definition	$\mid$	$X\langle \tilde{z} \rangle$	recursive call

Figure 2 (top) gives the syntax of processes. The send $x[a, b]$ sends along x a message name a and a continuation name b. The receive $x(y, z); P$ blocks until on x a message and continuation name are received (referred to in P as the placeholders y and z, respectively), binding y and z in P. The selection $x[b] \triangleleft i$ sends along x a label i and a continuation name b. The branch $x(z) \triangleright \{i : P_i\}_{i \in I}$ blocks until it receives on x a label $i \in I$ and a continuation name (referred to in P_i as the placeholder z), binding z in each P_i. In the rest of this paper, we refer to sends, receives, selections, and branches—including their continuations, if any—as *prefixes*. We refer to sends and selections collectively as *outputs*, and to receives and branches as *inputs*.

Process syntax:

$$
\begin{array}{rlrlrl}
P, Q ::= & x[a,b] & \text{send} & \mid & x(y,z);P & \text{receive} \\
\mid & x[b] \triangleleft \ell & \text{selection} & \mid & x(z) \triangleright \{i : P\}_{i \in I} & \text{branch} \\
\mid & (\boldsymbol{\nu} xy)P & \text{restriction} & \mid & P \mid Q & \text{parallel} \\
\mid & \mathbf{0} & \text{inaction} & \mid & [x \leftrightarrow y] & \text{forwarder} \\
\mid & \mu X(\tilde{z});P & \text{recursive definition} & \mid & X\langle \tilde{z} \rangle & \text{recursive call}
\end{array}
$$

Structural congruence:

$$\frac{P \equiv_\alpha Q}{P \equiv Q}\ [\text{CONG-ALPHA}] \qquad \frac{}{P \mid \mathbf{0} \equiv P}\ [\text{CONG-PAR-UNIT}] \qquad \frac{}{P \mid Q \equiv Q \mid P}\ [\text{CONG-PAR-COMM}] \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}\ [\text{CONG-PAR-ASSOC}]$$

$$\frac{x, y \notin \text{fn}(P)}{P \mid (\boldsymbol{\nu} xy)Q \equiv (\boldsymbol{\nu} xy)(P \mid Q)}\ [\text{CONG-SCOPE}] \qquad \frac{}{(\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P \equiv (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P}\ [\text{CONG-RES-COMM}]$$

$$\frac{}{(\boldsymbol{\nu} xy)P \equiv (\boldsymbol{\nu} yx)P}\ [\text{CONG-RES-SYMM}] \qquad \frac{}{(\boldsymbol{\nu} xy)\mathbf{0} \equiv \mathbf{0}}\ [\text{CONG-RES-INACT}] \qquad \frac{}{[x \leftrightarrow y] \equiv [y \leftrightarrow x]}\ [\text{CONG-FWD-SYMM}] \qquad \frac{}{(\boldsymbol{\nu} xy)[x \leftrightarrow y] \equiv \mathbf{0}}\ [\text{CONG-RES-FWD}]$$

$$\frac{}{\mu X(x_1, \ldots, x_n); P \equiv P\big\{\big(\mu X(y_1, \ldots, y_n); P\{y_1/x_1, \ldots, y_n/x_n\}\big)/X\langle y_1, \ldots, y_n \rangle\big\}}\ [\text{CONG-UNFOLD}]$$

Reduction:

$$\frac{}{(\boldsymbol{\nu} xy)(x[a,b] \mid y(z,y');Q) \to Q\{a/z, b/y'\}}\ [\text{RED-SEND-RECV}]$$

$$\frac{j \in I}{(\boldsymbol{\nu} xy)(x[b] \triangleleft j \mid y(y') \triangleright \{i : Q_i\}_{i \in I}) \to Q_j\{b/y'\}}\ [\text{RED-SEL-BRA}] \qquad \frac{y \neq z}{(\boldsymbol{\nu} xy)([x \leftrightarrow z] \mid P) \to P\{z/y\}}\ [\text{RED-FWD}]$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\ [\text{RED-CONG}] \qquad \frac{P \to Q}{(\boldsymbol{\nu} xy)P \to (\boldsymbol{\nu} xy)Q}\ [\text{RED-RES}] \qquad \frac{P \to Q}{P \mid R \to Q \mid R}\ [\text{RED-PAR}]$$

Figure 2: Definition of APCP's process language.

Restriction $(\boldsymbol{\nu} xy)P$ binds x and y in P, thus declaring them as the two names of a channel and enabling communication, as in [Vas12]. The process $P \mid Q$ denotes the parallel composition of P and Q. The process $\mathbf{0}$ denotes inaction. The forwarder $[x \leftrightarrow y]$ is a primitive copycat process that links together x and y. We say a forwarder $[x \leftrightarrow y]$ in P is *independent* if P does not bind x and y together through restriction (and *dependent* if it does). The process $\mu X(\tilde{z});P$ denotes a recursive definition, binding occurrences of X in P; the names $\tilde{z}$ form a context for P. Then P may contain recursive calls $X\langle \tilde{z} \rangle$ that indicate a repetition of P, providing the names $\tilde{z}$ as context. We only consider contractive recursion, disallowing processes with subexpressions of the form $\mu X_1(\tilde{z}); \ldots; \mu X_n(\tilde{z}); X_1\langle \tilde{z} \rangle$.

Names and recursion variables are free unless otherwise stated (i.e., unless they are bound somehow). We write $\mathrm{fn}(P)$ and $\mathrm{frv}(P)$ for the sets of free names and free recursion variables of P, respectively, and $\mathrm{bn}(P)$ for the set of bound names of P. Also, we write $P\{x/y\}$ to denote the capture-avoiding substitution of the free occurrences of y in P for x. Notation $P\big\{\big(\mu X(y_1,\ldots,y_n);P'\big)/X\langle y_1,\ldots,y_n\rangle\big\}$ denotes the substitution of occurrences of recursive calls $X\langle y_1,\ldots,y_n\rangle$ in P with the recursive definition $\mu X(y_1,\ldots,y_n);P'$, which we call *unfolding* recursion. We write sequences of substitutions $P\{x_1/y_1\}\ldots\{x_n/y_n\}$ as $P\{x_1/y_1,\ldots,x_n/y_n\}$.

Except for asynchrony and recursion, there are minor differences with respect to the languages of Dardha and Gay [DG18] and DeYoung *et al.* [DCPT12]. Unlike Dardha and Gay's, our syntax does not include empty send and receive prefixes that explicitly close channels; this simplifies the type system. We also do not include the operator for replicated servers, denoted $!x(y);P$, which is present in [DG18, DCPT12]. Although replication can be handled without difficulties, we omit it here; we prefer focusing on recursion, because it fits well with the examples we consider. See Section 3.6 for further discussion.

A Convenient Notation. In the send $x[a,b]$ and the selection $x[b]\triangleleft\ell$, names a and b are *free*. They can be bound to a continuation process using parallel composition and restriction, as in $(\boldsymbol{\nu} ay)(\boldsymbol{\nu} bz)(x[a,b]\mid P_{y,z})$ and $(\boldsymbol{\nu} bz)(x[b]\triangleleft\ell\mid Q_z)$, respectively. We introduce useful notations that abstract away from elide these constructs and continuation names:

Notation 3.2 (Derivable Bound Communication)**.** *We use the following syntactic sugar:*

$$\overline{x}[y]\cdot P \triangleq (\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)(x[a,b]\mid P\{z/x\}) \qquad \overline{x}\triangleleft\ell\cdot P \triangleq (\boldsymbol{\nu} zb)(x[b]\triangleleft\ell\mid P\{z/x\})$$

$$x(y);P \triangleq x(y,z);P\{z/x\} \qquad x\triangleright\{i:P_i\}_{i\in I} \triangleq x(z)\triangleright\{i:P_i\{z/x\}\}_{i\in I}$$

Note our use of '$\cdot$' instead of '$;$' in sending and selection to stress that they are non-blocking operators. As we will see, these derived constructs are typable (Theorem 3.16)

Operational Semantics. We define a reduction relation for processes ($P\to Q$) that formalizes how complementary outputs/inputs on connected names may synchronize. As usual for π-calculi, reduction relies on *structural congruence* ($P\equiv Q$), which relates processes with minor syntactic differences. Structural congruence is the smallest congruence on the syntax of processes (Figure 2 (top)) satisfying the axioms in Figure 2 (center).

Definition 3.3 (Structural Congruence ($\equiv$) for `APCP`)**.** Structural congruence for `APCP`, denoted $P\equiv Q$, is the smallest congruence on the syntax of processes (Definition 3.1) satisfying the axioms in Figure 3.

Structural congruence defines the following properties for processes. Processes are equivalent up to α equivalence (Rule [cong-alpha]). Parallel composition is associative (Rule [cong-par-assoc]) and commutative (Rule [cong-par-comm]), with unit $\mathbf{0}$ (Rule [cong-par-unit]). A parallel process may be moved into or out of a restriction as long as the bound channels do not occur free in the moved process (Rule [cong-scope]): this is *scope inclusion* and *scope extrusion*, respectively. Restrictions on inactive processes may be dropped (Rule [cong-res-inact]), and the order of names in restrictions and of consecutive restrictions does not matter (Rules [cong-res-symm] and [cong-res-comm], respectively). Forwarders are symmetric (Rule [cong-fwd-symm]), and equivalent to inaction if both names are bound together through restriction (Rule [cong-res-fwd]). Finally, a recursive definition is equivalent to its unfolding (Rule [cong-unfold]), replacing any recursive calls

$$\frac{P \equiv_\alpha Q}{P \equiv Q}\ [\textsc{cong-alpha}] \qquad \frac{}{P \mid \mathbf{0} \equiv P}\ [\textsc{cong-par-unit}] \qquad \frac{}{P \mid Q \equiv Q \mid P}\ [\textsc{cong-par-comm}] \qquad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}\ [\textsc{cong-par-assoc}]$$

$$\frac{x, y \notin \mathrm{fn}(P)}{P \mid (\boldsymbol{\nu} xy)Q \equiv (\boldsymbol{\nu} xy)(P \mid Q)}\ [\textsc{cong-scope}] \qquad \frac{}{(\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P \equiv (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P}\ [\textsc{cong-res-comm}]$$

$$\frac{}{(\boldsymbol{\nu} xy)P \equiv (\boldsymbol{\nu} yx)P}\ [\textsc{cong-res-symm}] \qquad \frac{}{(\boldsymbol{\nu} xy)\mathbf{0} \equiv \mathbf{0}}\ [\textsc{cong-res-inact}] \qquad \frac{}{[x \leftrightarrow y] \equiv [y \leftrightarrow x]}\ [\textsc{cong-fwd-symm}] \qquad \frac{}{(\boldsymbol{\nu} xy)[x \leftrightarrow y] \equiv \mathbf{0}}\ [\textsc{cong-res-fwd}]$$

$$\frac{}{\mu X(x_1, \ldots, x_n); P \equiv P\big\{\big(\mu X(y_1, \ldots, y_n); P\{y_1/x_1, \ldots, y_n/x_n\}\big)/X\langle y_1, \ldots, y_n\rangle\big\}}\ [\textsc{cong-unfold}]$$

Figure 3: Structural congruence for APCP: axioms.

with copies of the recursive definition, where the recursive definition's contextual names are pairwise substituted for by the call's names.

As we will see next, the semantics of APCP is closed under structural congruence. This means that processes are *equi-recursive*; however, APCP's typing discipline (described in Section 3.2) treats recursive types as *iso-recursive* (see, e.g., Pierce [Pie02]).

We define the reduction relation $P \to Q$ by the axioms and closure rules in Figure 2 (bottom). We write $\to^*$ for the reflexive, transitive closure of $\to$. Rule [RED-SEND-RECV] synchronizes a send and a receive on connected names and substitutes the message and continuation names. Rule [RED-SEL-BRA] synchronizes a selection and a branch: the received label determines the continuation process, substituting the continuation name appropriately. Rule [RED-FWD] implements the forwarder as a substitution. Rules [RED-CONG], [RED-RES], and [RED-PAR] close reduction under structural congruence, restriction, and parallel composition, respectively.

Having communication of free names in sends and selections is different from communication in the works by Dardha and Gay [DG18] and DeYoung *et al.* [DCPT12], where, following an internal mobility discipline [Bor98], communication involves bound names only. Still, notice that free output is expressible in those works by combining bound output and forwarding.

Example 3.4. To illustrate the preservation of order *within* a session and the asynchrony *between* different sessions, we consider the following process:

$$\begin{aligned} P \triangleq (\boldsymbol{\nu} zu)\big((\boldsymbol{\nu} xy)\big((\boldsymbol{\nu} ax')(x[v_1, a] \mid x'[v_2, b]) \\ \mid (\boldsymbol{\nu} cz')(z[v_3, c] \mid y(w_1, y'); y'(w_2, y''); Q)\big) \\ \mid u(w_3, u'); R\big) \end{aligned}$$

The process P defines two consecutive sends on a session from x to y, and an asynchronous send on a session from z to u. Two reductions are possible from P:

$$P \to (\boldsymbol{\nu} zu)\big((\boldsymbol{\nu} ax')\big(x'[v_2, b] \mid (\boldsymbol{\nu} cz')(z[v_3, c] \mid a(w_2, y''); Q\{v_1/w_1\})\big) \mid u(w_3, u'); R\big) \tag{3.1}$$

$$P \to (\boldsymbol{\nu} cz')\big((\boldsymbol{\nu} xy)\big((\boldsymbol{\nu} ax')(x[v_1, a] \mid x'[v_2, b]) \mid y(w_1, y'); y'(w_2, y''); Q\big) \mid R\{v_3/w_3, c/u'\}\big) \tag{3.2}$$

The reduction (3.1) entails the synchronization of the send on x and the receive on y; afterwards, the send on x' is connected to the receive on a that prefixes Q. The reduction (3.2) entails the synchronization of the send on z and the receive on u, connecting Q and R on a new session between z' and c. Note that from P there is no reduction involving the send on x', since x' is connected to the continuation name of the send on x and is thus not (yet) paired with a dual receive.

Using the sugared syntax from Notation 3.2, we can write

$$P = (\boldsymbol{\nu} zu)((\boldsymbol{\nu} xy)(\overline{x}[v_1] \cdot \overline{x}[v_2] \cdot \mathbf{0} \mid \overline{z}[v_3] \cdot y(w_1); y(w_2); Q') \mid u(w_3); R')$$

where $Q' \triangleq Q\{y/y''\}$ and $R' \triangleq R\{u/u'\}$.

The following diagram illustrates all the possible reduction paths from P; horizontal reductions concern the session between x and y, and diagonal reductions concern the session between z and u:

$$\begin{array}{ccccc} P & \to & P_1 & \to & P_3 \\ & \searrow & P_2 & \overset{\searrow}{\to} & P_4 \overset{\searrow}{\to} P_5 \end{array}$$

where processes P_1, P_2, P_3, P_4, and P_5 are as follows:

$$P_1 \triangleq (\boldsymbol{\nu} zu)((\boldsymbol{\nu} xy)(\overline{x}[v_2] \cdot \mathbf{0} \mid \overline{z}[v_3] \cdot y(w_2); Q'\{v_1/w_1\}) \mid u(w_3); R')$$

$$P_2 \triangleq (\boldsymbol{\nu} xy)(\overline{x}[v_1] \cdot \overline{x}[v_2] \cdot \mathbf{0} \mid y(w_1); y(w_2); Q') \mid R'\{v_3/w_3\}$$

$$P_3 \triangleq (\boldsymbol{\nu} zu)(\overline{z}[v_3] \cdot Q'\{v_1/w_1, v_2/w_2\} \mid u(w_3); R')$$

$$P_4 \triangleq (\boldsymbol{\nu} xy)(\overline{x}[v_2] \cdot \mathbf{0} \mid y(w_2); Q'\{v_1/w_1\}) \mid R'\{v_3/w_3\}$$

$$P_5 \triangleq Q'\{v_1/w_1, v_2/w_2\} \mid R'\{v_3/w_3\}$$

Presentations of reduction for session-typed π-calculi derived from the Curry-Howard interpretations of linear logic often include rules that correspond to *commuting conversions* in linear logic (cf. [CP10, Wad12, DG18, DCPT12]), which allow rewriting processes in such a way that blocking prefixes on free names are "pulled out" of restrictions. Commuting conversions can be easily included for APCP (cf. [HP21]), but we do not consider them here. While Dardha and Gay [DG18] rely on commuting conversions to prove deadlock freedom, the proof of deadlock freedom for APCP takes a different approach and does not require commuting conversions—see Section 3.3 for a detailed discussion.

3.2. **The Type System.** APCP types processes by assigning binary session types to names. Following Curry-Howard interpretations, we present session types as linear logic propositions (cf., e.g., Caires *et al.* [CPT16], Wadler [Wad12], Caires and Pérez [CP17], and Dardha and Gay [DG18]). We extend these propositions with recursion and *priority* annotations on connectives. Intuitively, prefixes typed with lower priority should not be blocked by those with higher priority.

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

Definition 3.5 (Session Types for APCP)**.** The following grammar defines the syntax of *session types* A, B. Let $\circ \in \mathbb{N}$.

$$A, B ::= A \otimes^\circ B \mid A \parr^\circ B \mid \oplus^\circ\{i : A\}_{i\in I} \mid \&^\circ\{i : A\}_{i\in I} \mid \bullet \mid \mu X.A \mid X$$

A name of type $A \otimes^\circ B$ (resp. $A \parr^\circ B$) first sends (resp. receives) a name of type A and then behaves as B. A name of type $\oplus^\circ\{i : A_i\}_{i\in I}$ selects a label $i \in I$ and then behaves as A_i. A name of type $\&^\circ\{i : A_i\}_{i\in I}$ offers a choice: after receiving a label $i \in I$, the name behaves as A_i. A name of type $\bullet$ is closed; it does not require a priority, as closed names do not exhibit behavior and thus are non-blocking.

Unlike Caires and Pfenning [CP10] and Dardha and Gay [DG18], APCP does not associate any behavior with closed sessions (i.e., no closing synchronizations). Moreover, as we will see, APCP's type system allows arbitrary parallel composition. Atkey *et al.* [ALM16] have shown that, in presence of arbitrary parallel composition (i.e., linear logic's Rule [MIX] that combines sequents arbitrarily, usually interpreted as session typing rule for arbitrary parallel composition), the dual propositions $\mathbf{1}$ and $\bot$ (usually associated with complementary prefixes for closing sessions) are equivalent. Hence, since in APCP session closing is silent, we follow Caires [Cai14] in conflating the types $\mathbf{1}$ and $\bot$ to the single, self-dual type $\bullet$ for closed sessions.

Type $\mu X.A$ denotes a recursive type, in which A may contain occurrences of the recursion variable X. As customary, μ is a binder: it induces the standard notions of α equivalence, substitution (denoted $A\{B/X\}$), and free recursion variables (denoted $\mathrm{frv}(A)$). We work with tail-recursive, contractive types, disallowing types of the form $\mu X_1 \ldots \mu X_n.X_1$ and $\mu X.X \otimes^\circ A$. Recursive types are treated iso-recursively: there will be an explicit typing rule that unfolds recursive types, and recursive types are not equal to their unfolding. We postpone formalizing the unfolding of recursive types, as it requires additional definitions to ensure consistency of priorities upon unfolding.

Duality, the cornerstone notion of session types and linear logic, ensures that the two names of a channel have complementary behaviors. Furthermore, dual types must have matching priority annotations. The following inductive definition of duality suffices for our tail-recursive types (cf. Gay *et al.* [GTV20]).

Definition 3.6 (Duality)**.** The *dual* of session type A, denoted $\overline{A}$, is defined inductively as follows:

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

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

Definition 3.7 (Priorities)**.** For session type A, $\mathrm{pr}(A)$ denotes its *priority*:

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

The priority of $\bullet$ and X is the constant ω: they denote the "final", non-blocking part of protocols. Although $\otimes$ and $\oplus$ also denote non-blocking prefixes, they do block their continuation until they are received. Hence, their priority is not constant.

We now turn to formalizing the unfolding of recursive types. Recall the intuition that prefixes typed with lower priority should not be blocked by those with higher priority. Based on this rationale, we observe that the unfolding of the recursive type $\mu X.A$ should not result in $A\{(\mu X.A)/X\}$, as usual; rather, the priorities of the unfolded type should be *increased* to ensure a global ordering between actions.

Example 3.8. Consider the recursive type $\mu X.A \parr^0 X$. By unfolding this type without increasing the priority, we obtain $A \parr^0 (\mu X.A \parr^0 X)$, a type in which the priorities do not ensure a global ordering between the two receives. In contrast, increasing the priority in the unfolded type as in, e.g., $A \parr^0 (\mu X.A \parr^1 X)$, does ensure a global ordering.

We make this intuition precise by defining the *lift* of priorities in types:

Definition 3.9 (Lift)**.** For proposition A and $t \in \mathbb{N}$, we define $\uparrow^t A$ as the *lift* operation:

$$\uparrow^t(A \otimes^\circ B) \triangleq (\uparrow^t A) \otimes^{\circ+t} (\uparrow^t B) \qquad \uparrow^t(\oplus^\circ\{i : A_i\}_{i\in I}) \triangleq \oplus^{\circ+t}\{i : \uparrow^t A_i\}_{i\in I} \qquad \uparrow^t \bullet \triangleq \bullet$$
$$\uparrow^t(A \parr^\circ B) \triangleq (\uparrow^t A) \parr^{\circ+t} (\uparrow^t B) \qquad \uparrow^t(\&^\circ\{i : A_i\}_{i\in I}) \triangleq \&^{\circ+t}\{i : \uparrow^t A_i\}_{i\in I}$$
$$\uparrow^t(\mu X.A) \triangleq \mu X.(\uparrow^t A) \qquad \uparrow^t X \triangleq X$$

Henceforth, the unfolding of $\mu X.A$ is $A\{\big(\mu X.(\uparrow^t A)\big)/X\}$, denoted $\mathrm{unfold}^t(\mu X.A)$, where $t \in \mathbb{N}$ depends on the highest priority of the types occurring in a typing context. We recall that we do not consider types to be equi-recursive: recursive types are not equal to their unfolding. Recursive types can only be unfolded by typing rules, discussed next.

We now define the top priority of a type, i.e., the highest priority appearing in a type:

Definition 3.10 (Top Priority)**.** For session type A, $\mathrm{top}(A)$ denotes its *top priority*:

$$\mathrm{top}(A \otimes^\circ B) \triangleq \mathrm{top}(A \parr^\circ B) \triangleq \max(\mathrm{top}(A), \mathrm{top}(B), \circ)$$
$$\mathrm{top}(\oplus^\circ\{i : A_i\}_{i\in I}) \triangleq \mathrm{top}(\&^\circ\{i : A_i\}_{i\in I}) \triangleq \max(\max_{i\in I}(\mathrm{top}(A_i)), \circ)$$
$$\mathrm{top}(\mu X.A) \triangleq \mathrm{top}(A) \qquad \mathrm{top}(\bullet) \triangleq \mathrm{top}(X) \triangleq 0$$

Notice how the top priority of $\bullet$ and X is 0, in contrast to their priority (as given by Definition 3.7): as we will see next, they do not contribute to the increase in priority needed for unfolding recursive types.

Typing Rules. The typing rules of `APCP` ensure that prefixes with lower priority are not blocked by those with higher priority (cf. Dardha and Gay [DG18]). To this end, they enforce the following laws:

(1) Sends and selections with priority $\circ$ must have continuations/payloads with priority strictly larger than $\circ$;
(2) A prefix with priority $\circ$ must be prefixed only by receives and branches with priority strictly smaller than $\circ$;
(3) Dual prefixes leading to a synchronization must have equal priorities (cf. Definition 3.6).

Judgments are of the form $\Omega \vdash P :: \Gamma$, where:

- P is a process;
- Γ is a context that assigns types to channels $(x : A)$;
- Ω is a context that assigns tuples of types to recursion variables $(X : (A, B, \ldots))$.

A judgment $\Omega \vdash P :: \Gamma$ then means that P can be typed in accordance with the type assignments for names recorded in Γ and the recursion variables in Ω. Intuitively, the latter context ensures that types concur between the context names of recursive definitions and calls. Both contexts Γ and Ω obey *exchange*: assignments may be silently reordered. Γ is *linear*, disallowing *weakening* (i.e., all assignments must be used, except names typed $\bullet$) and *contraction* (i.e., assignments may not be duplicated). Ω allows weakening and contraction, because a recursive definition may be called *zero or more* times.

The empty context is written $\emptyset$. In writing $\Gamma, x : A$ we assume that $x \notin \mathrm{dom}(\Gamma)$ (and similarly for Ω). We write $\uparrow^t\Gamma$ to denote the component-wise extension of lift (Definition 3.9) to typing contexts. Also, we write $\mathrm{pr}(\Gamma)$ to denote the least of the priorities of all types in Γ (Definition 3.7). An assignment $\tilde{z} : \tilde{A}$ means $z_1 : A_1, \ldots, z_k : A_k$. We define the top priority of a sequence of types $\mathrm{top}(\tilde{A})$ as $\max_{A_i \in \tilde{A}}(\mathrm{top}(A_i))$.

Figure 4 (top) gives the typing rules. We describe the typing rules from a *bottom-up* perspective. Rule [TYP-SEND] types a send; this rule does not have premises to provide a continuation process, leaving the free names to be bound to a continuation process using Rules [TYP-PAR] and [TYP-RES] (discussed hereafter). Similarly, Rule [TYP-SEL] types an unbound selection. Both rules require that the priority of the subject is lower than the priorities of both objects (continuation and payload)—this enforces Law 1. Rules [TYP-RECV] and [TYP-BRA] type receives and branches, respectively. In both cases, the used name's priority must be lower than the priorities of the other types in the continuation's typing context—this enforces Law 2.

Rule [TYP-PAR] types the parallel composition of two processes that do not share assignments on the same names. Rule [TYP-RES] types a restriction, where the two restricted names must be of dual type and thus have matching priority—this enforces Law 3. Rule [TYP-END] silently removes a closed name from the typing context. Rule [TYP-INACT] types an inactive process with no names. Rule [TYP-FWD] types forwarding between names of dual type—this also enforces Law 3.

Example 3.11. To illustrate the typing rules discussed so far, we recall process P from Example 3.4:

$$\begin{aligned} P = (\boldsymbol{\nu} zu)\big((\boldsymbol{\nu} xy)\big((\boldsymbol{\nu} ax')(x[v_1, a] \mid x'[v_2, b]) \\ \qquad\qquad \mid (\boldsymbol{\nu} cz')(z[v_3, c] \mid y(w_1, y'); y'(w_2, y''); Q)\big) \\ \mid u(w_3, u'); R\big) \end{aligned}$$

We give the typing of the two consecutive sends on x (omitting the context Ω):

$$\dfrac{\dfrac{\dfrac{\circ < \mathrm{pr}(A_1), \pi}{\vdash x[v_1, a] :: x : A_1 \otimes^\circ A_2 \otimes^\pi B,\ v_1 : \overline{A_1}, a : \overline{A_2 \otimes^\pi B}}\ [\text{TYP-SEND}] \quad \dfrac{\pi < \mathrm{pr}(A_2), \mathrm{pr}(B)}{\vdash x'[v_2, b] :: x' : \overline{A_2} \otimes^\pi B,\ v_2 : \overline{A_2}, b : \overline{B}}\ [\text{TYP-SEND}]}{\vdash x[v_1, a] \mid x'[v_2, b] :: v_1 : \overline{A_1}, v_2 : \overline{A_2}, b : \overline{B}, x : A_1 \otimes^\circ A_2 \otimes^\pi B,\ a : \overline{A_2 \otimes^\pi B}, x' : A_2 \otimes^\pi B}\ [\text{TYP-PAR}]}{\vdash (\boldsymbol{\nu} ax')(x[v_1, a] \mid x'[v_2, b]) :: v_1 : \overline{A_1}, v_2 : \overline{A_2}, b : \overline{B}, x : A_1 \otimes^\circ A_2 \otimes^\pi B}\ [\text{TYP-RES}]$$

As discussed before, this typing leaves the (free) names v_1, v_2, and b to be accounted for by the context.

Now let us derive the typing of the consecutive receives on y, i.e., of the subprocess $y(w_1, y'); y'(w_2, y''); Q$. Because x and y are dual names in P, the type of y should be dual

$$\frac{\circ < \mathrm{pr}(A), \mathrm{pr}(B)}{\Omega \vdash x[y,z] :: x : A \otimes^\circ B, y : \overline{A}, z : \overline{B}}\ [\textsc{typ-send}] \qquad \frac{\Omega \vdash P :: \Gamma, y : A, z : B \qquad \circ < \mathrm{pr}(\Gamma)}{\Omega \vdash x(y,z); P :: \Gamma, x : A \parr^\circ B}\ [\textsc{typ-recv}]$$

$$\frac{j \in I \qquad \circ < \mathrm{pr}(A_j)}{\Omega \vdash x[z] \lhd j :: x : \oplus^\circ\{i : A_i\}_{i\in I}, z : \overline{A_j}}\ [\textsc{typ-sel}] \qquad \frac{\forall i \in I.\ \Omega \vdash P_i :: \Gamma, z : A_i \qquad \circ < \mathrm{pr}(\Gamma)}{\Omega \vdash x(z) \rhd \{i : P_i\}_{i\in I} :: \Gamma, x : \&^\circ\{i : A_i\}_{i\in I}}\ [\textsc{typ-bra}]$$

$$\frac{\Omega \vdash P :: \Gamma}{\Omega \vdash P :: \Gamma, x : \bullet}\ [\textsc{typ-end}] \qquad \frac{\Omega \vdash P :: \Gamma \qquad \Omega \vdash Q :: \Delta}{\Omega \vdash P \mid Q :: \Gamma, \Delta}\ [\textsc{typ-par}] \qquad \frac{\Omega \vdash P :: \Gamma, x : A, y : \overline{A}}{\Omega \vdash (\boldsymbol{\nu} xy)P :: \Gamma}\ [\textsc{typ-res}]$$

$$\frac{}{\Omega \vdash \mathbf{0} :: \emptyset}\ [\textsc{typ-inact}] \qquad \frac{}{\Omega \vdash [x \leftrightarrow y] :: x : \overline{A}, y : A}\ [\textsc{typ-fwd}]$$

$$\frac{\Omega, X : \tilde{A} \vdash P :: \tilde{z} : \tilde{U} \qquad t \in \mathbb{N} > \mathrm{top}(\tilde{A}) \qquad \forall U_i \in \tilde{U}.\ U_i = \mathrm{unfold}^t(\mu X.A_i)}{\Omega \vdash \mu X(\tilde{z}); P :: \tilde{z} : \widetilde{\mu X.A}}\ [\textsc{typ-rec}]$$

$$\frac{t \in \mathbb{N} \qquad \forall U_i \in \tilde{U}.\ U_i = \mu X.\uparrow^t A_i}{\Omega, X : \tilde{A} \vdash X\langle \tilde{z} \rangle :: \tilde{z} : \tilde{U}}\ [\textsc{typ-var}]$$

$$\frac{\Omega \vdash P :: \Gamma, y : A, x : B \qquad \circ < \mathrm{pr}(A), \mathrm{pr}(B)}{\Omega \vdash \overline{x}[y] \cdot P :: \Gamma, x : A \otimes^\circ B}\ [\textsc{typ-send}\star]$$

$$\frac{\Omega \vdash P :: \Gamma, x : A_j \qquad j \in I \qquad \circ < \mathrm{pr}(A_j)}{\Omega \vdash \overline{x} \lhd j \cdot P :: \Gamma, x : \oplus^\circ\{i : A_i\}_{i\in I}}\ [\textsc{typ-sel}\star] \qquad \frac{\Omega \vdash P :: \Gamma \qquad t \in \mathbb{N}}{\Omega \vdash P :: \uparrow^t \Gamma}\ [\textsc{typ-lift}]$$

Figure 4: The typing rules of APCP (top) and derivable rules (bottom).

to the type of x above:

$$\dfrac{\dfrac{\vdash Q :: \Gamma, w_1 : \overline{A_1}, w_2 : \overline{A_2}, y'' : \overline{B} \qquad \pi < \mathrm{pr}(\Gamma, w_1 : \overline{A_1})}{\vdash y'(w_2, y''); Q :: \Gamma, w_1 : \overline{A_1}, y' : \overline{A_2} \parr^\pi \overline{B}}\ [\textsc{typ-recv}] \qquad \circ < \mathrm{pr}(\Gamma)}{\vdash y(w_1, y'); y'(w_2, y''); Q :: \Gamma, y : \overline{A_1} \parr^\circ \overline{A_2} \parr^\pi \overline{B}}\ [\textsc{typ-recv}]$$

These two derivations tell us that $\circ < \pi < \mathrm{pr}(A_1), \mathrm{pr}(A_2), \mathrm{pr}(B), \mathrm{pr}(\Gamma)$. This way, the type system ensures that none of the sessions in Q can be connected to sessions that block the sends on x, x', which may leave the door open for a deadlock otherwise. In Section 5.3, Example 5.9 illustrates such a situation in the context of $\texttt{LAST}^n$ and how the type system catches it.

Consider the usual Rule [TYP-CUT] in type systems based on linear logic [CP10, Wad12]:

$$\frac{\Omega \vdash P :: \Gamma, x : A \qquad \Omega \vdash Q :: \Delta, y : \overline{A}}{\Omega \vdash (\boldsymbol{\nu} xy)(P \mid Q) :: \Gamma, \Delta}\ [\text{TYP-CUT}]$$

Note that a single application of Rule [TYP-PAR] followed by Rule [TYP-RES] coincides with Rule [TYP-CUT]. Without annotations and conditions related to priority, Rules [TYP-PAR] and [TYP-RES] give rise to deadlocks, as the following example shows.

Example 3.12. Consider the following process, arguably the paradigmatic example of a deadlock:

$$Q \triangleq (\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)(x(u); \overline{z}[u'] \cdot \mathbf{0} \mid w(v); \overline{y}[v'] \cdot \mathbf{0})$$

Without priorities (and priority checks), this process can be typed using Rules [TYP-PAR] and [TYP-RES] (omitting "TYP-" from rule labels):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{0} :: \emptyset}\,[\text{INACT}]}{\vdash \mathbf{0} :: z : \bullet, x : \bullet, u : \bullet, u' : \bullet}\,[\text{END}]^4}{\vdash \overline{z}[u'] \cdot \mathbf{0} :: z : \bullet \otimes \bullet, x : \bullet, u : \bullet}\,[\text{SEND}\star]}{\vdash x(u); \overline{z}[u'] \cdot \mathbf{0} :: z : \bullet \otimes \bullet, x : \bullet \parr \bullet}\,[\text{RECV}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{0} :: \emptyset}\,[\text{INACT}]}{\vdash \mathbf{0} :: w : \bullet, y : \bullet, v : \bullet, v' : \bullet}\,[\text{END}]^4}{\vdash \overline{y}[v'] \cdot \mathbf{0} :: w : \bullet, y : \bullet \otimes \bullet, v : \bullet}\,[\text{SEND}\star]}{\vdash w(v); \overline{y}[v'] \cdot \mathbf{0} :: w : \bullet \otimes \bullet, y : \bullet \parr \bullet}\,[\text{RECV}]}{\vdash Q :: \emptyset}\,[\text{PAR}]+[\text{RES}]^2$$

On the other hand, were we to restrict parallel composition and restriction using Rule [TYP-CUT], Q would not be typable: Rule [TYP-CUT] can only type one of the restrictions, not both. With priorities, Q would not be typable either, due to the requirements induced by Rule [TYP-RECV]: (i) the priority $\circ$ of the input on x is smaller than the priority π of the send on z (left-hand side above), and (ii) the priority π of the input on w is smaller than the priority $\circ$ of the send on y (right-hand side above). Clearly, these requirements combined are unsatisfiable.

Rules [TYP-REC] and [TYP-VAR] type recursive definitions and recursive calls, respectively. To justify their formulation, let us consider naive formulations for each of them:

$$\frac{\Omega \vdash P :: \tilde{z} : \tilde{A}}{\Omega, X : |\tilde{z}| \vdash \mu X(\tilde{z}); P :: \tilde{z} : \widetilde{\mu X.A}}\,[\text{TYP-REC-NAIVE}] \qquad \frac{}{\Omega, X : |\tilde{z}| \vdash X\langle \tilde{z} \rangle :: \tilde{z} : \tilde{X}}\,[\text{TYP-VAR-NAIVE}]$$

Rule [TYP-REC-NAIVE] requires a name typed $\mu X.A$ at the recursive definition to be typed simply A in the recursive body. The associated Rule [TYP-VAR-NAIVE] then requires all names to be typed X, using the recursive context to make sure that the number of names concurs between recursive definition and call. However, as the following example shows, such a combination of naive rules discards priority annotations to a point where it is possible to type processes that deadlock:

Example 3.13. Consider the processes

$$P \triangleq \mu X(x, y); \overline{x}[a] \cdot x(b); \overline{y}[c] \cdot y(d); X\langle y, x \rangle,$$
$$Q \triangleq \mu X(u, v); u(a); \overline{u}[b] \cdot v(c); \overline{v}[d] \cdot X\langle u, v \rangle.$$

Notice how the recursive call in P swaps x and y. Let us see what happens if we unfold the recursion in P and Q:

$$P \equiv \overline{x}[a] \cdot x(b); \overline{y}[c] \cdot y(d); \mu X(y,x); \overline{y}[a] \cdot y(b); \overline{x}[c] \cdot x(d); X\langle x,y\rangle$$
$$Q \equiv u(a); \overline{u}[b] \cdot v(c); \overline{v}[d] \cdot \mu X(u,v); u(a); \overline{u}[b] \cdot v(c); \overline{v}[d] \cdot X\langle u,v\rangle$$

If we connect these processes on the names x and u and on y and v, we can see that the second recursive definition of this process contains a deadlock: the second receive on y is blocking the second send on x, while the second receive on u (waiting for the second send on x) is blocking the second send on v (for which the second receive on y is waiting).

Yet, P is typable using the naive typing rules described above:

$$\dfrac{\dfrac{\dfrac{}{X : 2 \vdash X\langle y,x\rangle :: x : X, y : X}\,[\textsc{typ-var-naive}]}{X : 2 \vdash \overline{x}[a] \cdot x(b); \overline{y}[c] \cdot y(d); X\langle y,x\rangle :: x : \bullet \otimes^0 \bullet \parr^1 X, y : \bullet \otimes^2 \bullet \parr^3 X}\,\cdots}{\emptyset \vdash \mu X(x,y); \overline{x}[a] \cdot x(b); \overline{y}[c] \cdot y(d); X\langle y,x\rangle :: x : \mu X.\, \bullet \otimes^0 \bullet \parr^1 X,\; y : \mu X.\, \bullet \otimes^2 \bullet \parr^3 X}\,[\textsc{typ-rec-naive}]$$

Thus, these naive rules prevent the type system of APCP from guaranteeing deadlock freedom.

The solution is for Rule [TYP-REC] to unfold all types. While unfolding, the priorities in these types are lifted by a common value, denoted t in the rule, that must be greater than the top priority occurring in the original types (cf. Definition 3.10). This makes sure that any priority requirements that come up in the typing of the recursive body of the process remain valid. The recursive context is used to record the bodies of the original folded types. Rule [TYP-REC] then requires that the types of names are recursive on the recursion variable used for the call. It checks that the bodies of the types concur with the types recorded in the recursive context, up to a lift by a common value t (i.e., the lifter used in the application of Rule [TYP-REC]).

Example 3.14. To see how the unfolding of types and the common lifter in Rule [TYP-REC] prevents P from Example 3.13 from being typable, let us attempt to find a typing derivation for P:

$$\dfrac{\dfrac{\dfrac{}{X : (\bullet \otimes^0 \bullet \parr^1 X, \bullet \otimes^2 \bullet \parr^3 X) \vdash X\langle y,x\rangle :: x : \mu X.\uparrow^t(\bullet \otimes^0 \bullet \parr^1 X),\; y : \mu X.\uparrow^t(\bullet \otimes^2 \bullet \parr^3 X)}\,[\textsc{typ-var}]}{X : (\bullet \otimes^0 \bullet \parr^1 X, \bullet \otimes^2 \bullet \parr^3 X) \vdash \overline{x}[a] \cdot x(b); \overline{y}[c] \cdot y(d); X\langle y,x\rangle :: x : \bullet \otimes^0 \bullet \parr^1 (\mu X.\uparrow^t(\bullet \otimes^0 \bullet \parr^1 X)),\; y : \bullet \otimes^2 \bullet \parr^3 (\mu X.\uparrow^t(\bullet \otimes^2 \bullet \parr^3 X))}\,\cdots}{\emptyset \vdash \mu X(x,y); \overline{x}[a] \cdot x(b); \overline{y}[c] \cdot y(d); X\langle y,x\rangle :: x : \mu X.\, \bullet \otimes^0 \bullet \parr^1 X,\; y : \mu X.\, \bullet \otimes^2 \bullet \parr^3 X}\,[\textsc{typ-rec}]$$

The application of Rule [TYP-VAR] at the top here is invalid: it is impossible to find a lifter t that matches the priorities in the type of x with those in the second type assigned to X, while simultaneously doing the same for the type of y and the first type assigned to X. However, if the call were $X\langle x,y\rangle$, the application of Rule [TYP-VAR] would be valid.

Remark 3.15 (Comparison to PCP)**.** Consider the typing rules for sending and selection in PCP, in which both are blocking prefixes and do not involve continuation passing. Note that

$$\dfrac{\dfrac{\vdash P :: \Gamma, y : A, x : B \qquad \circ < \mathrm{pr}(A), \mathrm{pr}(B)}{\vdash \overline{x}[y] \cdot P :: \Gamma, x : A \otimes^\circ B}\,[\text{TYP-SEND}*]}{\Rightarrow}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\circ < \mathrm{pr}(A), \mathrm{pr}(B)}{\vdash x[a,b] :: x : A \otimes^\circ B, a : \overline{A}, b : \overline{B}}\,[\text{TYP-SEND}] \qquad \vdash P\{z/x\} :: \Gamma, y : A, z : B}{\vdash x[a,b] \mid P\{z/x\} :: \Gamma, x : A \otimes^\circ B, y : A, a : \overline{A}, z : B, b : \overline{B}}\,[\text{TYP-PAR}]}{\vdash \underbrace{(\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)(x[a,b] \mid P\{z/x\})}_{\overline{x}[y] \cdot P \text{ (cf. Notation 3.2)}} :: \Gamma, x : A \otimes^\circ B}\,2\times[\text{TYP-RES}]}$$

$$\dfrac{\dfrac{\vdash P :: \Gamma, x : A_j \qquad j \in I \qquad \circ < \mathrm{pr}(A_j)}{\vdash \overline{x} \lhd j \cdot P :: \Gamma, x : \oplus^\circ \{i : A_i\}_{i \in I}}\,[\text{TYP-SEL}*]}{\Rightarrow}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{j \in I}{\vdash x[b] \lhd j :: x : \oplus^\circ \{i : A_i\}_{i \in I}, b : \overline{A_j}}\,[\text{TYP-SEL}] \qquad \vdash P\{z/x\} :: \Gamma, z : A_j}{\vdash x[b] \lhd j \mid P\{z/x\} :: \Gamma, x : \oplus^\circ \{i : A_i\}_{i \in I}, z : A_j, b : \overline{A_j}}\,[\text{TYP-PAR}]}{\vdash \underbrace{(\boldsymbol{\nu} zb)(x[b] \lhd j \mid P\{z/x\})}_{\overline{x} \lhd j \cdot P \text{ (cf. Notation 3.2)}} :: \Gamma, x : \oplus^\circ \{i : A_i\}_{i \in I}}\,[\text{TYP-RES}]}$$

Figure 5: Proof that Rules [TYP-SEND∗] and [TYP-SEL∗] are derivable (cf. Theorem 3.16).

PCP does not have recursion, and we thus omit the recursive context Ω.

$$\dfrac{\vdash P :: \Gamma, y : A, x : B \qquad \circ < \mathrm{pr}(\Gamma)}{\vdash \overline{x}[y]; P :: \Gamma, x : A \otimes^\circ B}\,[\text{TYP-SEND-PCP}] \qquad \dfrac{\vdash P :: \Gamma, x : A_j \qquad j \in I \qquad \circ < \mathrm{pr}(\Gamma)}{\vdash \overline{x} \lhd j; P :: \Gamma, x : \oplus^\circ \{i : A_i\}_{i \in I}}\,[\text{TYP-SEL-PCP}]$$

These rules are similar to the Rules [TYP-SEND∗] and [TYP-SEL∗] in Figure 4 for the sugared syntax introduced in Notation 3.2; the differences are twofold. First, the semicolons ‘;’ indicate that sends and selections are indeed blocking. Second, the rules compare the priority of the send/selection to the priorities in the context, whereas our rules compare this priority to the priorities of the continuation of the send/selection.

As anticipated, the binding of sends and selections to continuation processes (Notation 3.2) is consistent with typing in APCP. The corresponding typing rules in Figure 4 (bottom) are admissible using Rules [TYP-PAR] and [TYP-RES]. Note that rules for the sugared receive and branching in Notation 3.2 are not necessary, because the sugared input constructs rely on α renaming only. Figure 4 (bottom) also includes an admissible Rule [TYP-LIFT] that lifts a process' priorities.

Theorem 3.16. *Rules* [TYP-SEND∗] *and* [TYP-SEL∗] *in Figure 4 (bottom) are derivable, and Rule* [TYP-LIFT] *in Figure 4 (bottom) is admissible.*

Proof (Sketch). To show the derivability of Rules [TYP-SEND∗] and [TYP-SEL∗], we give their derivations in Figure 5 (omitting the recursive context Ω). Rule [TYP-LIFT] is admissible:

$\Omega \vdash P :: \Gamma$ implies $\Omega \vdash P :: \uparrow^t \Gamma$ (cf. Dardha and Gay [DG18]), simply by increasing all priorities in the derivation of P by t. □

Theorem 3.16 highlights how APCP's asynchrony uncovers a more primitive, lower-level view of message passing. In the next subsection we discuss deadlock freedom, which follows from a correspondence between reduction and the transformation of typing derivations to eliminate applications of Rule [TYP-RES]. In the case of APCP, this requires care: binding sends and selections to continuation processes leads to applications of Rule [TYP-RES] that do not immediately correspond to reductions.

Remark 3.17 (Comparison To DeYoung *et al.*)**.** Our rules for sending and selection are axiomatic, whereas DeYoung *et al.*'s are in the form of Rules [TYP-SEND$*$] and [TYP-SEL$*$], even though their sends and selections are parallel atomic prefixes as well [DCPT12]. While our type system is based on classical linear logic (with single-sided judgments), their type system is based on *intuitionistic* linear logic. As a result, their typing judgments are two sided, restriction involves a single name, and there are two rules (right and left) per connective. This way, for instance, their right rules for sending and selection are as follows (again omitting the recursive context Ω for a lack of recursion):

$$\text{[TYP-SEND-R]}\quad \frac{\Gamma \vdash P :: y : A \qquad \Delta \vdash Q :: x' : B}{\Gamma, \Delta \vdash (\boldsymbol{\nu} y)(\boldsymbol{\nu} x')(x[y, x'] \mid P \mid Q) :: x : A \otimes B}$$

$$\text{[TYP-SEL-R]}\quad \frac{\Gamma \vdash P :: x' : A_j \qquad j \in I}{\Gamma \vdash (\boldsymbol{\nu} x')(x[x'] \lhd j \mid P) :: x : \oplus\{i : A_i\}_{i \in I}}$$

□

The following result is important: it shows that the type system of APCP is complete with respect to types, i.e., every syntactical type has a well-typed process.

Proposition 3.18. *Given a type A, there exists $\Omega \vdash P :: x : A$.*

Proof (Sketch). By constructing P from the structure of A. To this end, we define characteristc processes: a function $\mathrm{char}^x(A)$ that constructs a process that performs the behavior described by A on the name x.

$$\begin{aligned}
\mathrm{char}^x(A \otimes^\circ B) &\triangleq \overline{x}[y] \cdot (\mathrm{char}^y(A) \mid \mathrm{char}^x(B)) & \mathrm{char}^x(\bullet) &\triangleq \mathbf{0} \\
\mathrm{char}^x(A \parr^\circ B) &\triangleq x(y); (\mathrm{char}^y(A) \mid \mathrm{char}^x(B)) & \mathrm{char}^x(\mu X.A) &\triangleq \mu X(x); \mathrm{char}^x(A) \\
\mathrm{char}^x(\oplus^\circ\{i : A_i\}_{i \in I}) &\triangleq \overline{x} \lhd j \cdot \mathrm{char}^x(A_j) \text{ (any } j \in I) & \mathrm{char}^x(X) &\triangleq X\langle x \rangle \\
\mathrm{char}^x(\&^\circ\{i : A_i\}_{i \in I}) &\triangleq x \rhd \{i : \mathrm{char}^x(A_i)\}_{i \in I}
\end{aligned}$$

For finite types, it is obvious that $\emptyset \vdash \mathrm{char}^x(A) :: x : A$. For simplicity, we omit details about recursive types, which require unfolding. For closed, recursive types, the thesis is obvious as well: $\emptyset \vdash \mathrm{char}^x(\mu X.A) :: x : \mu X.A$. □

3.3. **Type Preservation and Deadlock Freedom.** Well-typed processes satisfy protocol fidelity, communication safety, and deadlock freedom. The former two of these properties follow from *type preservation*, which ensures that APCP's semantics preserves typing. In contrast to Caires and Pfenning [CP10] and Wadler [Wad12], where type preservation corresponds to the elimination of (top-level) applications of Rule [TYP-CUT], in APCP it corresponds to the elimination of (top-level) applications of Rule [TYPE-RES].

APCP's semantics consists of reduction and structural congruence. Since the former relies on the latter, we first need to show type preservation for structural congruence, i.e., *subject congruence*. The structural congruence rule that unfolds recursive definitions requires care, because the types of the unfolded process are also unfolded:

Example 3.19. Consider the following typed recursive definition:

$$\emptyset \vdash \mu X(x,y); x(a); y(b); X\langle x,y\rangle :: x : \mu X.\bullet\parr^0 X, y : \mu X.\bullet\parr^1 X$$

Let us derive the typing of the unfolding of this process:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X : (\bullet\parr^2 X, \bullet\parr^3 X) \vdash X\langle x,y\rangle :: x : \mu X.\underbrace{\bullet\parr^6 X}_{\uparrow^4(\bullet\parr^2 X)}, y : \mu X.\underbrace{\bullet\parr^7 X}_{\uparrow^4(\bullet\parr^3 X)}}\,[\text{TYP-VAR}]}{X : (\bullet\parr^2 X, \bullet\parr^3 X) \vdash X\langle x,y\rangle :: x : \mu X.\bullet\parr^6 X, y : \mu X.\bullet\parr^7 X, b' : \bullet}\,[\text{TYP-END}]}{X : (\bullet\parr^2 X, \bullet\parr^3 X) \vdash y(b'); X\langle x,y\rangle :: x : \mu X.\bullet\parr^6 X, y : \bullet\parr^3 \mu X.\bullet\parr^7 X}\,[\text{TYP-RECV}]}{X : (\bullet\parr^2 X, \bullet\parr^3 X) \vdash y(b'); X\langle x,y\rangle :: x : \mu X.\bullet\parr^6 X, y : \bullet\parr^3 \mu X.\bullet\parr^7 X, a' : \bullet}\,[\text{TYP-END}]}{X : (\bullet\parr^2 X, \bullet\parr^3 X) \vdash x(a'); y(b'); X\langle x,y\rangle :: x : \underbrace{\bullet\parr^2 \mu X.\bullet\parr^6 X}_{\text{unfold}^4(\mu X.\bullet\parr^2 X)}, y : \underbrace{\bullet\parr^3 \mu X.\bullet\parr^7 X}_{\text{unfold}^4(\mu X.\bullet\parr^3 X)}}\,[\text{TYP-RECV}]}{\emptyset \vdash \mu X(x,y); x(a'); y(b'); X\langle x,y\rangle :: x : \mu X.\bullet\parr^2 X, y : \mu X.\bullet\parr^3 X}\,[\text{TYP-REC}]}{\emptyset \vdash \mu X(x,y); x(a'); y(b'); X\langle x,y\rangle :: x : \mu X.\bullet\parr^2 X, y : \mu X.\bullet\parr^3 X, b : \bullet}\,[\text{TYP-END}]}{\emptyset \vdash y(b); \mu X(x,y); x(a'); y(b'); X\langle x,y\rangle :: x : \mu X.\bullet\parr^2 X, y : \bullet\parr^1 \mu X.\bullet\parr^3 X}\,[\text{TYP-RECV}]}{\emptyset \vdash y(b); \mu X(x,y); x(a'); y(b'); X\langle x,y\rangle :: x : \mu X.\bullet\parr^2 X, y : \bullet\parr^1 \mu X.\bullet\parr^3 X, a : \bullet}\,[\text{TYP-END}]}{\emptyset \vdash x(a); y(b); \mu X(x,y); x(a'); y(b'); X\langle x,y\rangle :: x : \underbrace{\bullet\parr^0 \mu X.\bullet\parr^2 X}_{\text{unfold}^2(\mu X.\bullet\parr^0 X)}, y : \underbrace{\bullet\parr^1 \mu X.\bullet\parr^3 X}_{\text{unfold}^2(\mu X.\bullet\parr^1 X)}}\,[\text{TYP-RECV}]$$

Clearly, the typing of the unfolded process is not the same as the initial type, but they are equal up to unfolding of types. Note that the application of Rule [TYP-REC] unfolds the types with a common lifter of 4, because it needs to be larger than the top priority in the types before unfolding which is 3.

Hence, type preservation holds *up to unfolding*. To formalize this, we define the relation $(\vdash P :: \Gamma) \gtrsim \Gamma'$, which says that Γ and Γ' are equal up to (un)folding of recursive types consistent with the typing of P under Γ:

Definition 3.20. We define an asymmetric relation between a typed process $(\Omega \vdash P :: \Gamma)$ and a typing context Γ', denoted $(\Omega \vdash P :: \Gamma) \gtrsim \Gamma'$. The relation is defined by the inference rules in Figure 6, where each rule implicitly requires that $\Omega \vdash P :: \Gamma$ is a valid typing derivation by the rules in Figure 4.

We write $(\Omega \vdash P :: \Gamma) \cong (\Omega' \vdash Q :: \Gamma')$ if $(\Omega \vdash P :: \Gamma) \gtrsim \Gamma'$ and $(\Omega' \vdash Q :: \Gamma') \gtrsim \Gamma$.

$$\frac{(\Omega \vdash P\{(\mu X(\tilde{y}); P\{\tilde{y}/\tilde{z}\})/X\langle\tilde{y}\rangle\} :: \tilde{z} : \tilde{U}) \precsim \tilde{z} : \tilde{U}' \qquad \forall U_i' \in \tilde{U}';\ U_i' = \mathrm{unfold}^t(\mu X.A_i')}{(\Omega \vdash P\{(\mu X(\tilde{y}); P\{\tilde{y}/\tilde{z}\})/X\langle\tilde{y}\rangle\} :: \tilde{z} : \tilde{U}) \precsim \tilde{z} : \widetilde{\mu X.A'}}\ [\textsc{unf-fold}]$$

$$\frac{(\Omega \vdash \mu X(\tilde{z}); P :: \tilde{z} : \widetilde{\mu X.A}) \precsim \tilde{z} : \widetilde{\mu X.A'} \qquad \forall U_i' \in \tilde{U}'.\ U_i' = \mathrm{unfold}^t(\mu X.A_i')}{(\Omega \vdash \mu X(\tilde{z}); P :: \tilde{z} : \widetilde{\mu X.A}) \precsim \tilde{z} : \tilde{U}'}\ [\textsc{unf-unf}]$$

.....

$$\frac{}{(\Omega \vdash x[y,z] :: x : A \otimes^\circ B, y : \overline{A}, z : \overline{B}) \precsim x : A \otimes^\circ B, y : \overline{A}, z : \overline{B}}\ [\textsc{unf-send}]$$

$$\frac{(\Omega \vdash P :: \Gamma, y : A, z : B) \precsim \Gamma', y : A', z : B'}{(\Omega \vdash x(y,z); P :: \Gamma, x : A \parr^\circ B) \precsim \Gamma', x : A' \parr^\circ B'}\ [\textsc{unf-recv}]$$

$$\frac{}{(\Omega \vdash x[z] \lhd j :: x : \oplus^\circ\{i : A_i\}_{i\in I}, z : \overline{A_j}) \precsim x : \oplus^\circ\{i : A_i\}_{i\in I}, z : \overline{A_j}}\ [\textsc{unf-sel}]$$

$$\frac{\forall i \in I.\ (\Omega \vdash P_i :: \Gamma, z : A_i) \precsim \Gamma', z : A_i'}{(\Omega \vdash x(z) \rhd \{i : P_i\}_{i\in I} :: \Gamma, x : \&^\circ\{i : A_i\}_{i\in I}) \precsim \Gamma', x : \&^\circ\{i : A_i'\}_{i\in I}}\ [\textsc{unf-bra}]$$

$$\frac{(\Omega \vdash P :: \Gamma) \precsim \Gamma'}{(\Omega \vdash P :: \Gamma, x : \bullet) \precsim \Gamma', x : \bullet}\ [\textsc{unf-end}] \qquad \frac{(\Omega \vdash P :: \Gamma) \precsim \Gamma' \qquad (\Omega \vdash Q :: \Delta) \precsim \Delta'}{(\Omega \vdash P \mid Q :: \Gamma, \Delta) \precsim \Gamma', \Delta'}\ [\textsc{unf-par}]$$

$$\frac{(\Omega \vdash P :: \Gamma, x : A, y : \overline{A}) \precsim \Gamma', x : A', y : \overline{A'}}{(\Omega \vdash (\boldsymbol{\nu} xy)P :: \Gamma) \precsim \Gamma'}\ [\textsc{unf-res}] \qquad \frac{}{(\Omega \vdash \mathbf{0} :: \emptyset) \precsim \emptyset}\ [\textsc{unf-inact}]$$

$$\frac{}{(\Omega \vdash [x \leftrightarrow y] :: x : \overline{A}, y : A) \precsim x : \overline{A}, y : A}\ [\textsc{unf-fwd}] \qquad \frac{(\Omega, X : \tilde{A} \vdash P :: \tilde{z} : \tilde{U}) \precsim \tilde{z} : \tilde{U}'}{(\Omega \vdash \mu X(\tilde{z}); P :: \tilde{z} : \widetilde{\mu X.A}) \precsim \tilde{z} : \widetilde{\mu X.A'}}\ [\textsc{unf-rec}]$$

$$\frac{}{(\Omega, X : \tilde{A} \vdash X\langle\tilde{z}\rangle :: \tilde{z} : \tilde{U}) \precsim \tilde{z} : \tilde{U}}\ [\textsc{unf-var}]$$

Figure 6: Inference rules for Definition 3.20.

The most important rules of Figure 6 are Rules [UNF-UNF] and [UNF-FOLD] (above the line), as they relate unfolding and folding; the other rule (below the line) follow the typing

rules in Figure 4. Note that the rules in Figure 6 require no priority requirements, as they are covered by the implicit validity of the derivation of $\Omega \vdash P :: \Gamma$.

Proposition 3.21. *If (i)* $(\Omega_1 \vdash P_1 :: \Gamma_1) \cong (\Omega_2 \vdash P_2 :: \Gamma_2)$ *and (ii)* $(\Omega_2 \vdash P_2 :: \Gamma_2) \cong (\Omega_3 \vdash P_3 :: \Gamma_3)$*, then* $(\Omega_1 \vdash P_1 :: \Gamma_1) \cong (\Omega_3 \vdash P_3 :: \Gamma_3)$.

Proof (Sketch). Since $\cong$ is reflexive by definition, it suffices to show that (i) $(\Omega_1 \vdash P_1 :: \Gamma_1) \gtrsim \Gamma_2$ and (ii) $(\Omega_2 \vdash P_2 :: \Gamma_2) \gtrsim \Gamma_3$ imply $(\Omega_1 \vdash P_1 :: \Gamma_1) \gtrsim \Gamma_3$. Assumptions (i) and (ii) relate $\Gamma_1, \Gamma_2, \Gamma_3$ by applications of Rules [UNF-FOLD] and [UNF-UNF]. Clearly, the relation between Γ_1 and Γ_3 can then be derived directly by combining the applications of these rules. □

Subject congruence additionally requires the following property of typing derivations involving recursion variables:

Proposition 3.22. *Suppose given a process P and a derivation π of $\Omega, X : (A_l)_{l \in L} \vdash P :: \Gamma$. In every step in π, the assignment $X : (A_l)_{l \in L}$ in the recursive context is not modified nor removed, and no other assignments on the recursion variable X are added.*

Proof (Sketch). By induction on the height of the derivation π: no typing rule eliminates assignments from the recursive context, and no typing rule changes the types assigned to a recursion variable in the recursive context. Moreover, in our type system, any rule adding an assignment to a context implicitly assumes that the newly introduced name or recursion variable is not in the context yet. Hence, an assignment on X cannot be added by any rule. □

Now we can state and prove subject congruence, which holds up to the relation in Definition 3.20:

[Subject Congruence] If $\Omega \vdash P :: \Gamma$ and $P \equiv Q$, then there exists Γ' such that $\Omega \vdash Q :: \Gamma'$ and $(\Omega \vdash P :: \Gamma) \cong (\Omega \vdash Q :: \Gamma')$.

Proof. By induction on the derivation of $P \equiv Q$. The cases for the structural rules follow from the IH directly. All base cases are straightforward, except the cases of unfolding and folding. We detail unfolding, where $P = \mu X(\tilde{z}); P'$ and $Q = P'\{(\mu X(\tilde{y}); P'\{\tilde{y}/\tilde{z}\})/X\langle\tilde{y}\rangle\}$. The other direction, folding, is similar.

By type inversion of Rule [TYP-REC],

$$\Omega \vdash P :: \Omega; \tilde{z} : \widetilde{\mu X.A}$$
$$\Omega, X : \tilde{A} \vdash P' :: \tilde{z} : \tilde{U}$$

where there exists $t \in \mathbb{N} > \mathrm{top}(\widetilde{\mu X.A})$ such that each

$$U_i = \mathrm{unfold}^t(\mu X.A_i) = A_i\{(\mu X.\uparrow^t A_i)/X\}.$$

Let ∇ denote the typing derivation of P'.

P' may contain $m \geq 0$ recursive calls $X\langle\tilde{y}\rangle$. For each call, there is an application of Rule [TYP-VAR] in ∇. We uniquely identify each such application of Rule [TYP-VAR] as ∂_j for $0 < j \leq m$. Since `APCP` only allows tail recursion, these multiple recursive calls can only occur inside branches on names not in $\tilde{z}$, so the common lifter of each ∂_j must be t.

$$\partial_j \left\{ \quad \frac{}{\Omega'_j, X : \tilde{A} \vdash X\langle\tilde{y}_j\rangle :: \tilde{y}_j : \widetilde{\mu X.\uparrow^t A}}\,[\text{TYP-VAR}] \right.$$

$$\dfrac{\dfrac{}{\vdash x[a,b] :: x : A \otimes^\circ B, a : \overline{A}, b : \overline{B}}\,[\text{TYP-SEND}] \quad \dfrac{\vdash P :: \Gamma, z : \overline{A}, y' : \overline{B}}{\vdash y(z,y'); P :: \Gamma, y : \overline{A} \parr^\circ \overline{B}}\,[\text{TYP-RECV}]}{\vdash (\boldsymbol{\nu} xy)(x[a,b] \mid y(z,y'); P) :: \Gamma, a : \overline{A}, b : \overline{B}}\,[\text{TYP-PAR}]+[\text{TYP-RES}]$$

$\rightarrow$

$$\vdash P\{a/z, b/y'\} :: \Gamma, a : \overline{A}, b : \overline{B}$$

Figure 7: Example of subject reduction (cf. Theorem 3.23) in Rule [RED-SEND-RECV]. The well typedness of the process before reduction allows us to infer its typing derivation, also giving us the typing of P. Typing the process after reduction is then a matter of inductively substituting names in the typing derivation of P.

where $\Omega'_j \supseteq \Omega$, because rules applied in ∇ can only add assignments to the recursive context. Recall that $\tilde{y}_j : \widetilde{\mu X.\uparrow^t A}$ is notation for $y_{j_1} : \mu X.\uparrow^t A_1, \ldots, y_{j_k} : \mu X.\uparrow^t A_k$, as introduced in Section 3.2.

The unfolding of a recursive process replaces each recursive call by a copy of the recursive definition. Hence, to find a typing derivation for Q we proceed as follows, for each $0 < j \leq m$:

(1) We obtain a derivation ∇'_j of $P'\{\tilde{y}_j/\tilde{z}\}$ from ∇ by substituting $\tilde{z}$ for $\tilde{y}_j$ (while avoiding capturing names) and by lifting all priorities by t (including the priorities of X in the recursive context of ∂_j).
(2) We apply Rule [TYP-REC] on X in the conclusion of ∇'_j, resulting in a new typing derivation ∂'_j:

$$\partial'_j \left\{ \quad \dfrac{\begin{array}{c}\nabla'_j \\ \Omega'_j, X : \widetilde{\uparrow^t A} \vdash P'\{\tilde{y}_j/\tilde{z}\} :: \tilde{y}_j : \widetilde{\uparrow^t U}\end{array}}{\Omega'_j \vdash \mu X(\tilde{y}_j); P'\{\tilde{y}_j/\tilde{z}\} :: \tilde{y}_j : \widetilde{\mu X.\uparrow^t A}}\,[\text{TYP-REC}] \right.$$

For every $0 < j \leq m$, the context of the conclusion of ∂'_j coincides with the context of the conclusion of ∂_j, up to the assignment $X : \widetilde{\uparrow^t A}$ in the recursive context not present in ∂'_j. We intend to obtain from ∇ a new derivation ∇' by replacing each ∂_j with ∂'_j. By Proposition 3.22, the fact that each ∂'_j is missing the assignment to X in the recursive context does not influence the steps in ∇. Hence, we can indeed obtain such a derivation ∇':

$$\Omega \vdash \underbrace{P'\big\{\big(\mu X(\tilde{y}); P'\{\tilde{y}/\tilde{z}\}\big)/X\langle\tilde{y}\rangle\big\}}_{Q} :: \tilde{z} : \tilde{U} \qquad \nabla'$$

where $\forall U_i \in \tilde{U}.\ U_i = \text{unfold}^t(A_i)$.

Let $\Gamma' = \tilde{z} : \tilde{U}$. We have $\Omega \vdash Q :: \Gamma'$. To prove the thesis, we have to show that Γ and Γ' are equal up to unfolding. Following the typing rules applied in the derivation of $\Omega \vdash Q :: \tilde{z} : \tilde{U}$, we have $(\Omega \vdash Q :: \tilde{z} : \tilde{U}) \gtrsim \tilde{z} : \tilde{U}$ (cf. Definition 3.20). Then, by Rule [UNF-FOLD], $(\Omega \vdash Q :: \tilde{z} : \tilde{U}) \gtrsim \tilde{z} : \widetilde{\mu X.A}$. Similarly, we have $(\Omega \vdash P :: \tilde{z} : \widetilde{\mu X.A}) \gtrsim \tilde{z} : \widetilde{\mu X.A}$, so by Rule [UNF-UNF], $(\Omega \vdash P :: \tilde{z} : \widetilde{\mu X.A}) \gtrsim \tilde{z} : \tilde{U}$. Hence, $(\Omega \vdash P :: \Gamma) \cong (\Omega \vdash Q :: \Gamma')$, proving the thesis. □

Having established subject congruence, we can prove that reduction preserves typing, i.e., *subject reduction*:

Theorem 3.23 (Subject Reduction)**.** *If* $\Omega \vdash P :: \Gamma$ *and* $P \to Q$*, then there exists* Γ' *such that* $\Omega \vdash Q :: \Gamma'$ *and* $(\Omega \vdash P :: \Gamma) \cong (\Omega \vdash Q :: \Gamma')$.

Proof (Sketch). By induction on the derivation of $P \to Q$ (Figure 2 (bottom)), we find a Γ' such that $\Omega \vdash Q :: \Gamma'$ and $(\Omega \vdash P :: \Gamma) \cong (\Omega \vdash Q :: \Gamma')$.

The cases of Rules [RED-RES] and [RED-PAR] follow directly from the IH. Consider, e.g., $(\boldsymbol{\nu} xy)(P \mid R) \to (\boldsymbol{\nu} xy)(Q \mid R)$ derived from $P \to Q$. By inversion of typing on the process before reduction, e.g., $\Omega \vdash P :: \Gamma, x : A$. Then, by the IH, $(\Omega \vdash P :: \Gamma, x : A) \cong (\Omega \vdash Q :: \Gamma', x : A')$. The thesis then follows from an application of Rules [UNF-PAR] and [UNF-RES].

The case of Rule [RED-CONG] follows from the IH and subject congruence (Section 3.3). To be precise, the rule says that (i) $P \equiv P'$, (ii) $P' \to Q'$, and (iii) $Q' \equiv Q$ imply $P \to Q$. By Section 3.3 (subject congruence) on assumptions (i) and (iii), we have $(\Omega \vdash P :: \Gamma) \cong (\Omega \vdash P' :: \Gamma')$ and $(\Omega' \vdash Q' :: \Delta') \cong (\Omega' \vdash Q :: \Delta)$. Also, by the IH on assumption (ii), $(\Omega \vdash P' :: \Gamma') \cong (\Omega' \vdash Q' :: \Delta')$ where $\Omega = \Omega'$. Then, by Proposition 3.21 (transitivity of $\cong$), $(\Omega \vdash P :: \Gamma) \cong (\Omega \vdash Q :: \Delta)$, proving the thesis.

Key cases are Rules [RED-SEND-RECV], [RED-SEL-BRA], and [RED-FWD]. Figure 7 shows the representative instance of Rule [RED-SEND-RECV], and an example where Rules [RED-RES], [RED-PAR], and [RED-CONG] are used. □

Protocol fidelity ensures that processes respect their intended (session) protocols. Communication safety ensures the absence of communication errors and mismatches in processes. Correct typability gives a static guarantee that a process conforms to its ascribed session protocols; type preservation gives a dynamic guarantee. Because session types describe the intended protocols and error-free exchanges, type preservation entails both protocol fidelity and communication safety. For a detailed account, we refer the curious reader to the early work by Honda *et al.* [HVK98], which defines error processes and shows by contradiction that well-typed processes do not reduce to an error.

In what follows, we consider a process to be deadlocked if it is not the inactive process and cannot reduce. Our deadlock-freedom result for `APCP` adapts that for `PCP` [DG18]. The equivalent of our Rule [TYP-RES] in `PCP` is Rule [CYCLE]. Deadlock freedom for `PCP` involves three steps to eliminate applications of Rule [CYCLE]:

(1) First, [CYCLE] elimination states that we can remove all applications of [CYCLE] in a typing derivation without affecting the derivation's assumptions and conclusion.
(2) Only the removal of *top-level* applications of [CYCLE] captures the intended process semantics; the removal of other applications of [CYCLE] corresponds to reductions behind prefixes, which is not allowed [Wad12, DG18]. Therefore, the second step is *top-level deadlock freedom* (referred to here as *progress*), which states that a process with a top-level application of [CYCLE] reduces until there are no top-level applications of [CYCLE] left. This steps requires commuting conversions: a process with a top-level application of [CYCLE] may not have reductions ready, so commuting conversions are used to remove top-level applications of [CYCLE] by blocking them with prefixes.
(3) Third, deadlock freedom follows for processes typable under empty contexts.

Here, we adapt and address [TYP-RES] elimination and progress in one proof.

As mentioned before, binding `APCP`'s asynchronous sends and selections to continuations involves additional, low-level uses of [TYP-RES], which we cannot eliminate through process

reduction. Therefore, we establish progress for *live processes* (Section 3.3). A process is live if it is equivalent to a restriction on *active names* used for unguarded prefixes. This way, e.g., in $x[y,z]$ the name x is active, but y and z are not. We additionally need a notion of *evaluation context*, under which reducible forwarders may occur.

Definition 3.24 (Active Names)**.** The *set of active names* of P, denoted $\mathrm{an}(P)$, contains the (free) names that are used for non-blocked prefixed:

$$\begin{array}{lll} \mathrm{an}(x[y,z]) \triangleq \{x\} & \mathrm{an}(x(y,z);P) \triangleq \{x\} & \mathrm{an}(\mathbf{0}) \triangleq \emptyset \\ \mathrm{an}(x[z] \triangleleft \ell) \triangleq \{x\} & \mathrm{an}(x(z) \triangleright \{i : P_i\}_{i\in I}) \triangleq \{x\} & \mathrm{an}([x \leftrightarrow y]) \triangleq \emptyset \\ \mathrm{an}(P \mid Q) \triangleq \mathrm{an}(P) \cup \mathrm{an}(Q) & \mathrm{an}(\mu X(\tilde{x});P) \triangleq \mathrm{an}(P) & \\ \mathrm{an}((\boldsymbol{\nu} xy)P) \triangleq \mathrm{an}(P) \setminus \{x,y\} & \mathrm{an}(X\langle\tilde{x}\rangle) \triangleq \emptyset & \end{array}$$

Definition 3.25 (Evaluation Context)**.** Evaluation contexts ($\mathcal{E}$) are defined by the following grammar:

$$\mathcal{E} ::= [\cdot] \mid \mathcal{E} \mid P \mid (\boldsymbol{\nu} xy)\mathcal{E} \mid \mu X(\tilde{x});\mathcal{E}$$

We write $\mathcal{E}[P]$ to denote the process obtained by replacing the hole $[\cdot]$ in $\mathcal{E}$ by P.

Definition 3.26 (Live Process)**.** A process P is *live*, denoted $\mathrm{live}(P)$, if

(1) there are names x, y and process P' such that $P \equiv (\boldsymbol{\nu} xy)P'$ with $x, y \in \mathrm{an}(P')$, or
(2) there are names x, y, z and process P' such that $P \equiv \mathcal{E}\big[(\boldsymbol{\nu} yz)([x \leftrightarrow y] \mid P')\big]$ and $z \neq x$ (i.e., the forwarder is independent).

We additionally need to account for recursion: as recursive definitions do not directly entail reductions, we must fully unfold them before eliminating applications of [TYP-RES]:

[Unfolding] If $\Omega \vdash P :: \Gamma$, then there is a process $P^\star$ such that $P^\star \equiv P$ and $P^\star$ is not of the form $\mu X(\tilde{z});Q$.

Proof. By induction on the number n of consecutive recursive definitions prefixing P, such that P is of the form $\mu X_1(\tilde{z});\ldots;\mu X_n(\tilde{z});Q$. If $n = 0$, the thesis follows immediately, as $P \equiv P$. Otherwise, $n \geq 1$. Then there are X, Q such that $P = \mu X(\tilde{z});Q$, where Q starts with $n-1$ consecutive recursive definitions. Let $R \triangleq Q\big\{\big(\mu X(\tilde{y});Q\{\tilde{y}/\tilde{z}\}\big)/X\langle\tilde{y}\rangle\big\}$. Clearly, $R \equiv P$. Then, because R starts with $n-1$ consecutive recursive definitions, the thesis follows by appealing to the IH. □

Dardha and Gay's progress result concerns a sequence of reduction steps that reaches a process that is not live anymore [DG18]. In our case, progress concerns a single reduction step only, because recursive processes might stay live across reductions forever. Moreover, because of our definition of liveness, we do not need commuting conversions for this step.

[Progress] If $\emptyset \vdash P :: \Gamma$ and $\mathrm{live}(P)$, then there is a process Q such that $P \to Q$.

Proof. We distinguish the two cases of $\mathrm{live}(P)$: P contains a restriction on a pair of active names, or P contains a restriction on a forwarded name under a reduction context. In both cases, we first unfold any recursive definitions preceding the involved prefixes/forwarders, resulting in $P^\star \equiv P$. By subject congruence (Section 3.3), there exists Γ' such that $\emptyset \vdash P^\star :: \Gamma'$ and $(\vdash P^\star :: \Gamma') \gtrsim \Gamma$. By Definition 3.20, the only difference between Γ' and Γ is the unfolding of recursive types.

(1) $P^\star$ contains a restriction on a pair of active names. That is, $P^\star \equiv (\boldsymbol{\nu} xy)P^{\star\prime}$ and $x, y \in \mathrm{an}(P^{\star\prime})$. The rest of the analysis depends on how x and y occur as active

names in $P^{\star\prime}$. As a representative case, we consider that x occurs as the subject of a send. By inversion of typing, then y occurs as the subject of a receive. Hence, we have $P^\star \equiv \mathcal{E}\big[(\boldsymbol{\nu} xy)(x[a,b] \mid y(v,z); P^{\star\prime\prime})\big]$. Let $Q \triangleq \mathcal{E}[P^{\star\prime\prime}\{a/v, b/z\}]$. Then, by Rule [RED-SEND-RECV], $P^\star \to Q$. Hence, by Rule [RED-CONG], $P \to Q$, proving the thesis.

(2) $P^\star$ contains a restriction on a forwarded name under a reduction context. That is, $P^\star \equiv \mathcal{E}\big[(\boldsymbol{\nu} yz)([x \leftrightarrow y] \mid P^{\star\prime})\big]$ where $z \neq x$. Let $Q = \mathcal{E}[P^{\star\prime}\{x/z\}]$. Then, by Rule [RED-FWD], $P^\star \to Q$. Hence, by Rule [RED-CONG], $P \to Q$, proving the thesis. □

Our deadlock-freedom result concerns processes typable under empty contexts (as in, e.g., Caires and Pfenning [CP10] and Dardha and Gay [DG18]). We first need a lemma which ensures that non-live processes typable under empty contexts do not contain prefixes (sends/receives/selections/branches) or *independent* forwarders (whose endpoints are not bound together using restriction). This lemma is in fact the crux of our deadlock-freedom result, as it relies on the priority checks induced by our typing system:

If $\emptyset \vdash P :: \emptyset$ and P is not live, then P contains no prefixes or independent forwarders.

Proof. W.l.o.g., assume all recursion in P is unfolded. Take $P^\star \equiv P$ such that $P^\star = (\boldsymbol{\nu} x_i x_i')_{i\in I} \prod_{j\in J} P_j$ where, for every $j \in J$, P_j is a prefix or forwarder. That is, every P_j is a *thread* which cannot be broken down further into parallel components or restrictions.

By abuse of notation, we write $\mathrm{pr}(P_j)$ to denote the priority of the type of the subject of the prefix of P_j. Also, we write $\mathrm{pr}(x)$ to denote the priority of the type of x.

Towards a contradiction, assume that there is at least one prefix or independent forwarder in P. We apply induction on the size of J.

- In the base case, $J = \emptyset$. Then there cannot be any prefixes or independent forwarders in P: a contradiction.
- In the inductive case, $J = J' \cup \{j\}$. W.l.o.g., assume that, for every $j' \in J'$, $\mathrm{pr}(P_j) \leq \mathrm{pr}(P_{j'})$ (i.e., pick P_j as one of the prefixes/forwarders with the least priority of all threads). By assumption, P_j denotes a prefix or forwarder on some endpoint x_i. By well typedness, x_i is connected through restriction to x_i'. By Rule [TYP-RES], $\mathrm{pr}(x_i) = \mathrm{pr}(x_i')$. Further analysis depends on P_j: a receive (or, analogously, a branch), a send (or, analogously, a selection), or a forwarder.
  - Suppose $P_j = x_i(y,z); P_j'$. By Rule [TYP-RECV], $x_i' \notin \mathrm{fn}(P_j')$, because the rule requires $\mathrm{pr}(x_i) < \mathrm{pr}(x_i')$: otherwise, this would contradict the fact that $\mathrm{pr}(x_i) = \mathrm{pr}(x_i')$. Hence, there exists $j' \in J'$ such that $x_i' \in \mathrm{fn}(P_{j'})$.
    If $J' = \emptyset$, the contradiction is immediate. Otherwise, the analysis depends on the prefix of $P_{j'}$. Since P is not live, this cannot be a send with subject x_i' or an independent forwarder on x_i'. The prefix can also not be a dependent forwarder, for x_i already appears in P_j. This leaves us with two possibilities: a receive (or, analogously, a branch) on another endpoint, or a send (or, analogously, a selection) with object x_i'.
    - If $P_{j'} = x_k(v,w); P_{j'}'$ where $x_i' \in \mathrm{fn}(P_{j'}')$, then, by Rule [TYP-RECV], $\mathrm{pr}(x_k) < \mathrm{pr}(x_i')$. Hence, $\mathrm{pr}(P_{j'}) = \mathrm{pr}(x_k) < \mathrm{pr}(x_i') = \mathrm{pr}(x_i) = \mathrm{pr}(P_j)$: this contradicts the assumption that $\mathrm{pr}(P_j) \leq \mathrm{pr}(P_{j'})$.
    - If $P_{j'} = x[a,b]$ where $x_i' \in \{a,b\}$, w.l.o.g., assume $x_i' = a$. Then, by Rule [TYP-SEND], $\mathrm{pr}(x_k) < \mathrm{pr}(x_i')$. The contradiction follows as in the previous case.
  - Suppose $P_j = x_i[a,b]$. By Rule [TYP-SEND], $x_i' \notin \{a,b\}$, because the rule requires $\mathrm{pr}(x_i) < \mathrm{pr}(x_i')$: otherwise, this would contradict the fact that $\mathrm{pr}(x_i) = \mathrm{pr}(x_i')$. Hence,

there exists $j' \in J'$ such that $x'_i \in \text{fn}(P_{j'})$. The contradiction follows as in the previous case.
– Suppose $P_j = [x_i \leftrightarrow x_k]$. By well typedness, x_i is connected through restriction to x'_i. Since P is not live, it must be that $x'_i = x_k$: P_j is not an independent forwarder. Then there must be a $j' \in J'$ where $P_{j'}$ fulfills the assumption that P contains at least one prefix or independent forwarder. The contradiction then follows from the IH. □

We now state our deadlock-freedom result:
[Deadlock Freedom] If $\emptyset \vdash P :: \emptyset$, then either $P \equiv \mathbf{0}$ or $P \to Q$ for some Q.

Proof. The analysis depends on whether P is live or not.

- If P is not live, then, by Section 3.3, it does not contain any prefixes or independent forwarders. Any recursive definitions in P are thus of the form $\mu X_1(); \ldots; \mu X_n(); \mathbf{0}$: contractiveness requires recursive calls to be prefixed by receives/branches or bound to parallel sends/selections/forwarders, of which there are none. Hence, we can use structural congruence to rewrite each recursive definition in P to $\mathbf{0}$ by unfolding, yielding $P' \equiv P$. Any dependent forwarders in P are of the form $(\boldsymbol{\nu} xy)[x \leftrightarrow y]$, and can be rewritten to $\mathbf{0}$ using Rule [CONG-RES-FWD] (cf. Figure 2). The remaining derivation of P' only contains applications of Rule [TYP-INACT], [TYP-PAR], [TYP-END], or [TYP-RES] on closed names. It follows straightforwardly that $P \equiv P' \equiv \mathbf{0}$.
- If P is live, by Section 3.3 there is Q s.t. $P \to Q$. □

3.4. **Reactivity.** Progress (Section 3.3) is an important liveness property, as it defines precisely the conditions under which processes can reduce (namely, liveness in Definition 3.26). As such, progress plays a key role in deadlock freedom (Section 3.3). However, in the presence of recursion, the strength of progress is limited: even if the majority of subprocesses is stuck, progress holds if at least one subprocess is live. Nonetheless, `APCP`'s type system allows us to prove a stronger result: *reactivity*, which essentially states that infinite recursion in `APCP` cannot block sessions from progressing.

Processes typable under empty contexts are not only deadlock free, they are *reactive*, in the following sense: for each name in the process, we can eventually observe a reduction involving that name. To formalize this property, we define *labeled reductions*, which expose details about synchronizations:

Definition 3.27 (Labeled Reductions)**.** Consider the labels

$$\alpha ::= [x \leftrightarrow y] \mid x\rangle y : a \mid x\rangle y : \ell \qquad \text{(forwarding, send/receive, selection/branching)}$$

where each label has subjects x and y. The *labeled reduction* $P \xrightarrow{\alpha}_{\curlywedge} Q$ is defined by the following rules:

$$\frac{}{(\boldsymbol{\nu} xy)(x[a,b] \mid y(v,z);P) \xrightarrow{x\rangle y:a}_{\curlywedge} P\{a/v,b/z\}}\ [\textsc{lred-send-recv}]$$

$$\frac{j \in I}{(\boldsymbol{\nu} xy)(x[b] \triangleleft j \mid y(z) \triangleright \{i : P_i\}_{i\in I}) \xrightarrow{x\rangle y:j}_{\curlywedge} P_j\{b/z\}}\ [\textsc{lred-sel-bra}] \qquad \frac{}{(\boldsymbol{\nu} yz)([x \leftrightarrow y] \mid P) \xrightarrow{[x\leftrightarrow y]}_{\curlywedge} P\{x/z\}}\ [\textsc{lred-fwd}]$$

$$\frac{P \equiv P' \qquad P' \xrightarrow{\alpha}_{\curlywedge} Q' \qquad Q' \equiv Q}{P \xrightarrow{\alpha}_{\curlywedge} Q}\ [\textsc{lred-cong}] \qquad \frac{P \xrightarrow{\alpha}_{\curlywedge} Q}{(\boldsymbol{\nu} xy)P \xrightarrow{\alpha}_{\curlywedge} (\boldsymbol{\nu} xy)Q}\ [\textsc{lred-res}] \qquad \frac{P \xrightarrow{\alpha}_{\curlywedge} Q}{P \mid R \xrightarrow{\alpha}_{\curlywedge} Q \mid R}\ [\textsc{lred-par}]$$

For any P and P', $P \to P'$ if and only if there exists a label α such that $P \xrightarrow{\alpha}_{\curlywedge} P'$.

Proof. Immediate by definition, because each reduction in Figure 2 (bottom) corresponds to a labeled reduction, and vice versa. □

Our reactivity result states that processes typable under empty contexts have at least one finite reduction sequence (denoted $\to^\star$) that enables a labeled reduction involving a *pending* name—a name that occurs as the subject of a prefix and is not bound by an input (see below). Clearly, the typed process may have other reduction sequences, not necessarily finite.

Definition 3.28 (Pending Names)**.** Given a process P, we define the set of *pending names* of P, denoted $\mathrm{pn}(P)$, as follows:

$$\begin{array}{rlrl}
\mathrm{pn}(x[y,z]) &\triangleq \{x\} & \mathrm{pn}(x(y,z);P) &\triangleq \{x\} \cup (\mathrm{pn}(P) \setminus \{y,z\}) \\
\mathrm{pn}(x[z] \triangleleft \ell) &\triangleq \{x\} & \mathrm{pn}(x(z) \triangleright \{i : P_i\}_{i\in I}) &\triangleq \{x\} \cup (\textstyle\bigcup_{i\in I} \mathrm{pn}(P_i) \setminus \{z\}) \\
\mathrm{pn}(P \mid Q) &\triangleq \mathrm{pn}(P) \cup \mathrm{pn}(Q) & \mathrm{pn}(\mu X(\tilde{x});P) &\triangleq \mathrm{pn}(P) \qquad \mathrm{pn}(\mathbf{0}) \triangleq \emptyset \\
\mathrm{pn}((\boldsymbol{\nu} xy)P) &\triangleq \mathrm{pn}(P) & \mathrm{pn}(X\langle\tilde{x}\rangle) &\triangleq \emptyset \qquad \mathrm{pn}([x \leftrightarrow y]) \triangleq \{x,y\}
\end{array}$$

Note that the proof of reactivity below does not rely on deadlock freedom: suppose we are observing a blocked pending prefix in a process with a parallel recursive definition; deadlock freedom ensures a reduction from the recursive definition which would not unblock the pending prefix we are observing. Instead, the proof relies on a priority analysis (similar to the one in the proof of Section 3.3) to unblock pending prefixes.

[Reactivity] Suppose given a process $\emptyset \vdash P :: \emptyset$. Then, for every $x \in \mathrm{pn}(P)$ there exists a process P' such that $P \to^\star P'$ and $P' \xrightarrow{\alpha}_{\curlywedge} Q$, for some process Q and label α with subject x.

Proof. Take any $x \in \mathrm{pn}(P)$. Because P is typable under empty contexts, x is bound to some $y \in \mathrm{pn}(P)$ by restriction. By typing, in P there is exactly one prefix on x and one prefix on y (they may also occur in forwarder processes). Following the restrictions on priorities in the typing of x and y in P, the prefixes on x and y cannot occur sequentially in P (cf. the proof of Section 3.3 for details on this reasoning). By typability, the prefix on y is dual to the prefix on x.

We apply induction on the number of receives, branches, and recursive definitions in P blocking the prefixes on x and y, denoted n and m, respectively. Because P is typable under empty contexts, the blocking receives and branches that are on names in $\mathrm{pn}(P)$ also have to be bound to pending names by restriction. The prefixes on these connected names may also be prefixed by receives, branches, and recursive definitions, so we may need to unblock those prefixes as well. Since there can only be a finite number of names in any given process, we also apply induction on the number of prefixes blocking these connected prefixes.

- If $n = 0$ and $m = 0$, then the prefixes on x and y occur at the top level; because they do not occur sequentially, the synchronization between x and y can take place immediately. Hence, $P \xrightarrow{\alpha}_{\gamma} Q$ where x and y are the subjects of α. This proves the thesis, with $P' = P$.
- If $n > 0$ or $m > 0$, the analysis depends on the foremost prefixes blocking the prefixes on x and y.

  If the either of these blocking prefixes is a recursive definition ($\mu X(\tilde{y})$), we unfold the recursion. Because a corresponding recursive call ($X\langle\tilde{z}\rangle$) cannot occur as a prefix, the effect of unfolding either (i) triggers prefixes that occur in parallel to those on x and y, or (ii) the prefixes on x or y precede the punfolded recursive call. In either case, the number of prefixes decreases, and the thesis follows from the IH.

  Otherwise, if neither foremost prefix is a recursive definition, then the foremost prefixes must be on names in $\mathrm{pn}(P)$. Consider the prefix that is typable with the least priority. W.l.o.g. assume that this is the foremost prefix of x. Suppose this prefix is on some name w connected to another name $z \in \mathrm{pn}(P)$ by restriction. By typability, the priority of w is less than that of x and all of the prefixes in between. This means that the number of prefixes blocking the prefix on z strictly decreases. Hence, by the IH, $P \to^{\star} P'' \xrightarrow{\alpha'}_{\gamma} Q'$ in a finite number of steps, where w and z are the subjects of α'. The synchronization between w and z can be performed, and n decreases. By type preservation (Theorem 3.23), $\emptyset \vdash Q' :: \emptyset$. The thesis then follows from the IH: $P \to^{\star} P'' \xrightarrow{\alpha'}_{\gamma} Q' \to^{\star} P' \xrightarrow{\alpha}_{\gamma} Q$ in finite steps, where x and y are the subjects of α. □

Note that deadlock freedom (Section 3.3) can be derived directly from reactivity: liveness implies the existence of pending names, so reduction is guaranteed by Section 3.4. In the following, deadlock freedom is strong enough, but in [HP22b] where we apply `APCP` for the analysis of distributed implementations of multiparty session types we do rely on the stronger reactivity.

3.5. **Typing Milner's Cyclic Scheduler.** Here we show that our specification of Milner's cyclic scheduler from Section 2.1 is typable in `APCP`, and thus deadlock free (cf. Section 3.3). Let us recall the process definitions of the leader and followers, omitting braces '{...}' for branches with one option:

$$A_1 \triangleq \mu X(a_1, c_n, d_1); \overline{d_1} \lhd \mathsf{start} \cdot \overline{a_1} \lhd \mathsf{start} \cdot a_1 \rhd \mathsf{ack}; \overline{d_1} \lhd \mathsf{next} \cdot c_n \rhd \mathsf{start}; c_n \rhd \mathsf{next}; X\langle a_1, c_n, d_1\rangle$$

$$A_{i+1} \triangleq \mu X(a_{i+1}, c_i, d_{i+1}); c_i \rhd \mathsf{start}; \overline{a_{i+1}} \lhd \mathsf{start} \cdot \overline{d_{i+1}} \lhd \mathsf{start} \cdot a_{i+1} \rhd \mathsf{ack}; c_i \rhd \mathsf{next}; \overline{d_{i+1}} \lhd \mathsf{next} \cdot X\langle a_{i+1}, c_i, d_{i+1}\rangle \qquad \forall 1 \leq i < n$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X : (T_{a_1}, T_{c_n}, T_{d_1}) \vdash a_1 : \mu X.(\uparrow^{t_1} T_{a_1}),\ c_n : \mu X.(\uparrow^{t_1} T_{c_n}),\ d_1 : \mu X.(\uparrow^{t_1} T_{d_1})}\ (3.9)}\ [\text{TYP-VAR}]}{X : T_X \vdash a_1 : \mu X.(\uparrow^{t_1} T_{a_1}),\ c_n : \&^{\rho_n}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{c_n})\},\ d_1 : \mu X.(\uparrow^{t_1} T_{d_1})}\ (3.8)}\ [\text{TYP-BRA}]}{X : T_X \vdash a_1 : \mu X.(\uparrow^{t_1} T_{a_1}),\ c_n : \&^{\pi_n}\{\mathsf{start} : \&^{\rho_n}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{c_n})\}\},\ d_1 : \mu X.(\uparrow^{t_1} T_{d_1})}\ (3.7)}\ [\text{TYP-BRA}]}{X : T_X \vdash a_1 : \mu X.(\uparrow^{t_1} T_{a_1}),\ c_n : \&^{\pi_n}\{\mathsf{start} : \&^{\rho_n}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{c_n})\}\},\ d_1 : \oplus^{\rho_1}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{d_1})\}}\ (3.6)}\ [\text{TYP-SEL}*]}{\begin{array}{l} X : T_X \vdash a_1 : \&^{\kappa_1}\{\mathsf{ack} : \mu X.(\uparrow^{t_1} T_{a_1})\}, \\ \qquad c_n : \&^{\pi_n}\{\mathsf{start} : \&^{\rho_n}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{c_n})\}\}, \\ \qquad d_1 : \oplus^{\rho_1}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{d_1})\} \end{array}\ (3.5)}\ [\text{TYP-BRA}]$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{l} X : \underbrace{(T_{a_1}, T_{c_n}, T_{d_1})}_{T_X} \vdash a_1 : \oplus^{\circ_1}\{\mathsf{start} : \&^{\kappa_1}\{\mathsf{ack} : \mu X.(\uparrow^{t_1} T_{a_1})\}\}, \\ \qquad c_n : \&^{\pi_n}\{\mathsf{start} : \&^{\rho_n}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{c_n})\}\}, \\ \qquad d_1 : \oplus^{\pi_1}\{\mathsf{start} : \oplus^{\rho_1}\{\mathsf{next} : \mu X.(\uparrow^{t_1} T_{d_1})\}\} \end{array}\ (3.3)}\ [\text{TYP-SEL}*]}{\begin{array}{l} \emptyset \vdash a_1 : \mu X.\underbrace{\oplus^{\circ_1}\{\mathsf{start} : \&^{\kappa_1}\{\mathsf{ack} : X\}\}}_{T_{a_1}}, c_n : \mu X.\underbrace{\&^{\pi_n}\{\mathsf{start} : \&^{\rho_n}\{\mathsf{next} : X\}\}}_{T_{c_n}}, \\ \qquad d_1 : \mu X.\underbrace{\oplus^{\pi_1}\{\mathsf{start} : \oplus^{\rho_1}\{\mathsf{next} : X\}\}}_{T_{d_1}} \end{array}}\ [\text{TYP-REC}]$$

Figure 8: Typing derivation of the leader scheduler A_1 of Milner's cyclic scheduler (processes omitted).

Figure 8 gives the typing derivation of A_1, omitting processes from judgments, with the following priority requirements:

$$t_1 > \max(\circ_1, \kappa_1, \pi_n, \rho_n, \pi_1, \rho_1) \tag{3.3}$$
$$\pi_1 < \rho_1 \tag{3.4}$$
$$\circ_1 < \kappa_1 \tag{3.5}$$
$$\kappa_1 < \pi_n, \rho_1 \tag{3.6}$$
$$\rho_1 < \pi_1 + t_1 \tag{3.7}$$
$$\pi_n < \circ_1 + t_1, \pi_1 + t_1 \tag{3.8}$$
$$\rho_n < \circ_1 + t_1, \pi_1 + t_1 \tag{3.9}$$

Each process A_{i+1} for $0 \leq i < n$—thus including the leader—is typable as follows, assuming c_i is c_n for $i = 0$:

$$\emptyset \vdash A_{i+1} :: a_{i+1} : \mu X.\oplus^{\circ_{i+1}}\{\mathsf{start} : \&^{\kappa_{i+1}}\{\mathsf{ack} : X\}\},\ c_i : \mu X.\&^{\pi_i}\{\mathsf{start} : \&^{\rho_i}\{\mathsf{next} : X\}\},\ d_{i+1} : \mu X.\oplus^{\pi_{i+1}}\{\mathsf{start} : \oplus^{\rho_{i+1}}\{\mathsf{next} : X\}\}$$

Note how, for each $1 \leq i \leq n$, the types for c_i and d_i are duals and are thus assigned equal priorities.

$$\frac{}{\Omega \vdash x[] :: x : \mathbf{1}^{\circ}}\ [\text{TYP-CLOSE}] \qquad \frac{\Omega \vdash P :: \Gamma \qquad \circ < \mathrm{pr}(\Gamma)}{\Omega \vdash x(); P :: \Gamma, x : \bot^{\circ}}\ [\text{TYP-WAIT}] \qquad \frac{\circ < \mathrm{pr}(A)}{\Omega \vdash ?x[y] :: x : ?^{\circ}A, y : \overline{A}}\ [\text{TYP-CLI}]$$

$$\frac{\Omega \vdash P :: ?\Gamma, y : A \qquad \circ < \mathrm{pr}(?\Gamma)}{\Omega \vdash !x(y); P :: ?\Gamma, x : !^{\circ}A}\ [\text{TYP-SRV}] \qquad \frac{\Omega \vdash P :: \Gamma, y : A \qquad \circ < \mathrm{pr}(A)}{\Omega \vdash ?\overline{x}[y] \cdot P :: \Gamma, x : ?^{\circ}A}\ [\text{TYP-CLI}*]$$

$$\frac{\Omega \vdash P :: \Gamma}{\Omega \vdash P :: \Gamma, x : ?^{\circ}A}\ [\text{TYP-WEAKEN}] \qquad \frac{\Omega \vdash P :: \Gamma, x : ?^{\circ}A, x' : ?^{\kappa}A \qquad \pi = \min(\circ, \kappa)}{\Omega \vdash P\{x/x'\} :: \Gamma, x : ?^{\pi}A}\ [\text{TYP-CONTRACT}]$$

$$(\boldsymbol{\nu} xy)(x[] \mid y(); P) \to P\ [\text{RED-CLOSE-WAIT}] \qquad (\boldsymbol{\nu} xy)(?x[a] \mid !y(v); P \mid Q) \to P\{a/v\} \mid (\boldsymbol{\nu} xy)(!y(v); P \mid Q)\ [\text{RED-CLI-SRV}]$$

Figure 9: Typing rules (top) and reductions (bottom) for explicit closing and replicated servers.

The priority requirements in the typing derivation of each A_i are satisfiable. The derivations of these processes have the following constraints:

- For A_1 we require the inequalities listed above;
- For each $1 \leq i < n$, for A_{i+1} we require $\pi_i < \circ_{i+1}, \pi_{i+1}$, $\circ_{i+1} < \kappa_{i+1}$, $\pi_{i+1} < \rho_{i+1}$, $\kappa_{i+1} < \rho_i, \rho_{i+1}$, $\rho_i < \circ_{i+1} + t_{i+1}, \rho_{i+1}$, and $\rho_{i+1} < \pi_{i+1} + t_{i+1}$.

We can satisfy these requirements by assigning $\pi_i \triangleq i$, $\circ_i \triangleq i + 1$, $\kappa_i \triangleq i + 2$, and $\rho_i \triangleq i + 4$ for each $1 \leq i \leq n$, except with $\pi_n \triangleq n + 3$ (to satisfy (3.6) for $1 \leq n < 4$). For the application of [TYP-REC], each derivation also requires a common lifter $t_{i+1} > \max(\circ_{i+1}, \kappa_{i+1}, \pi_i, \rho_i, \pi_{i+1}, \rho_{i+1})$ for $0 \leq i < n$. The priority requirements involving t_{i+1} always require the priority lifted by t_{i+1} to be higher than the priority not lifted by t_{i+1}, so the common lifter requirement easily satisfies these requirements.

Recall from Section 2.1:

$$\mathit{Sched}_n \triangleq (\boldsymbol{\nu} c_1 d_1) \ldots (\boldsymbol{\nu} c_n d_n)\big((\boldsymbol{\nu} a_1 b_1)(A_1 \mid P_1) \mid \ldots \mid (\boldsymbol{\nu} a_n b_n)(A_n \mid P_n)\big)$$

Assuming given workers

$$\emptyset \vdash P_i :: b_i : \mu X.\&^{\circ_i}\{\mathsf{start} : \oplus^{\kappa_i}\{\mathsf{ack} : X\}\}$$

for each $1 \leq i \leq n$, we have $\emptyset \vdash \mathit{Sched}_n :: \emptyset$. Hence, it follows from Section 3.3 that Sched_n is deadlock free for each $n \geq 1$.

3.6. **Extensions: Explicit Session Closing and Replicated Servers.** As already mentioned, our presentation of APCP does not include explicit closing and replicated servers. Here we briefly discuss what APCP would look like if we were to include these constructs.

Explicit closing is useful in programming to be sure that all resources are cleaned up correctly. There are several ways of integrating explicit closing in a calculus like APCP. Following, e.g., [CP10, Wad12], here we achieve explicit closing by adding closes (empty sends) $x[]$ and waits (empty receives) $x(); P$ to the syntax in Figure 2 (top). We also add

the Rule [RED-CLOSE-WAIT] to Figure 2 (bottom). At the level of types, we replace the conflated type $\bullet$ with $\mathbf{1}^\circ$ and $\bot^\circ$, associated to closes and waits, respectively. Note that we do need priority annotations on types for closed names now, because wait is blocking and thus requires priority checks. In the type system in Figure 4 (top), we replace Rule [TYP-END] with Rules [TYPE-CLOSE] and [TYP-WAIT] in Figure 9 (top).

For replicated servers, we add (asynchronous) client requests $?x[y]$ and servers $!x(y);P$, with types $?^\circ A$ and $!^\circ A$, respectively. We include syntactic sugar for binding client requests to continuations as in Notation 3.2: $?\overline{x}[y] \cdot P \triangleq (\boldsymbol{\nu} ya)(?x[a] \mid P)$. Rule [RED-CLI-SRV] is in Figure 9 (bottom): it connects a client and a server and forks a copy of the server. Also, we add a structural congruence rule to clean up unused servers: $(\boldsymbol{\nu} xz)(!x(y);P) \equiv \mathbf{0}$. In the type system, we add Rules [TYP-CLI], [TYP-SRV], [TYP-WEAKEN], and [TYP-CONTRACT] in Figure 9 (top); the former two are for typing client requests and servers, respectively, and the latter two are for connecting to a server without requests and for multiple requests, respectively. In Rule [TYP-SRV], notation $?\Gamma$ means that every type in Γ is of the form $?^\circ A$. Figure 9 (top) also includes a derivable Rule [TYP-CLI$*$] which types the syntactic sugar for bound client requests.

4. An Intermezzo: From APCP to LAST

As attested by its expressivity and meta-theoretical results (Theorem 3.23 and Section 3.3), APCP provides a convenient framework for analyzing asynchronous message passing between cyclically connected processes. In particular, APCP provides a firm basis for designing languages with session-typed concurrency, asynchronous communication, and cyclic structures. Ideally, we would like to faithfully compile any such language into APCP, in order to transfer its correctness guarantees.

As discussed in the Introduction, we look for answers in the realm of functional programming, in the form of variants of the λ-calculus with session-typed, message-passing concurrency. In this context, Gay and Vasconcelos's LAST [GV10] appears nicely positioned: LAST is a call-by-value language in which programs consist of threads that are cyclically connected on channels that provide asynchronous message passing (through buffers). The calculus LAST has forked several variants that connect message-passing processes with message-passing functions, as we set out to do here. Most notably, Walder [Wad12, Wad14] introduced GV, a variant of LAST with synchronous communication and non-cyclic thread connections, and gave a translation into his CP (the synchronous ancestor of APCP, without cyclic connections). Subsequently, in the same spirit, Kokke and Dardha [KD21a] presented PGV, a variant of GV with cyclic connections, and a translation from Dardha and Gay's PCP (the synchronous ancestor of APCP [DG18]).

Hence, LAST is a natural choice for a core programming language that can be studied via a translation into APCP. Notice that both GV and PGV enjoy deadlock freedom by typing, whereas LAST does not. This strengthens our motivation for designing a translation into APCP, as transferring the deadlock-freedom property from APCP to LAST would address a significant gap. To our knowledge, such a *translational* approach to ensuring deadlock freedom for LAST programs has not been achieved before.

In this section we gently recall LAST and gradually introduce the key ingredients for its translation into APCP. For presentational purposes, we find it useful to present a variant of LAST that is more convenient towards a translation, denoted LAST* (Section 4.1). The type system for LAST*, described in Section 4.2, closely follows the one for LAST in [GV10].

Terms ($M, N, \ldots$), values (v), and reduction contexts ($\mathcal{R}$):

$$
\begin{array}{rrlrl}
M, N ::= & x & \text{variable} & \mid \texttt{new} & \text{create new channel} \\
\mid & () & \text{unit value} & \mid \texttt{fork}\, M; N & \text{fork } M \text{ in parallel to } N \\
\mid & \lambda x.M & \text{abstraction} & \mid (M, N) & \text{pair construction} \\
\mid & M\ N & \text{application} & \mid \texttt{let}\,(x, y) = M\,\texttt{in}\,N & \text{pair deconstruction} \\
\mid & \texttt{send}\, M\, N & \text{send } M \text{ along } N & \mid \texttt{select}\,\ell\, M & \text{select label } \ell \text{ along } M \\
\mid & \texttt{recv}\, M & \text{receive along } M & \mid \texttt{case}\, M\, \texttt{of}\, \{i : N\}_{i \in I} & \text{offer labels in } I \text{ along } M
\end{array}
$$

$$
\begin{array}{rl}
v ::= & x \mid \lambda x.M \mid (v, v) \mid () \\
\mathcal{R} ::= & [\cdot] \mid \mathcal{R}\ M \mid v\ \mathcal{R} \mid (\mathcal{R}, M) \mid (v, \mathcal{R}) \mid \texttt{let}\,(x, y) = \mathcal{R}\,\texttt{in}\,M \\
\mid & \texttt{send}\,\mathcal{R}\,M \mid \texttt{send}\,v\,\mathcal{R} \mid \texttt{recv}\,\mathcal{R} \mid \texttt{select}\,\ell\,\mathcal{R} \mid \texttt{case}\,\mathcal{R}\,\texttt{of}\,\{i : M\}_{i \in I}
\end{array}
$$

Term reduction ($\rightarrow_{\mathtt{M}}$):

$$
\frac{}{(\lambda x.M)\ v \rightarrow_{\mathtt{M}} M\{v/x\}}\ [\textsc{red-lam}] \qquad \frac{}{\texttt{let}\,(x, y) = (v_1, v_2)\,\texttt{in}\,M \rightarrow_{\mathtt{M}} M\{v_1/x, v_2/y\}}\ [\textsc{red-pair}]
$$

$$
\frac{M \rightarrow_{\mathtt{M}} N}{\mathcal{R}[M] \rightarrow_{\mathtt{M}} \mathcal{R}[N]}\ [\textsc{red-lift}]
$$

Figure 10: The LAST* term language.

Then, in Section 4.3, we discuss the potential design of a translation from LAST* into APCP. We purposefully use the word "potential": we seek a translation that is *faithful*, i.e., that preserves and reflects behaviors in a precise sense. Given this focus, we shall argue that the call-by-value semantics of LAST* is not well suited for inducing a faithful translation. As such, this section serves as motivation for introducing LAST^n, a variant of LAST based on a call-by-name semantics with constructs for session closing and explicit substitutions, which enjoys a faithful translation into APCP and admits the translational approach to the deadlock-freedom property (Section 5).

4.1. **The Syntax and Semantics of LAST*.** In LAST*, programs consist of two layers: while *terms* ($M, N, \ldots$) define the behavior of threads, *configurations* ($C, D, \ldots$) are obtained by composing threads in parallel, so as to enable their interaction by exchanging messages on buffered channels. Before detailing these syntactic elements (and their semantics), we point out the differences between LAST and LAST*. While in LAST endpoints are created by synchronization on shared names, LAST* features a dedicated construct for endpoint creation (denoted `new`). Also, thread creation in LAST* involves an explicit continuation, not present in LAST. Moreover, in LAST buffers run next to threads, whereas they are integrated in endpoint restrictions in LAST*. Finally, for simplicity, LAST* accounts for linear sessions only.

Terms. Figure 10 (top) gives the term syntax for LAST*. The functional behavior of terms is defined by standard λ-calculus constructs for variables x, the unit value $()$, abstraction $\lambda x.M$ and application $M\ N$, and pair construction (M, N) and deconstruction $\texttt{let}\,(x, y) = M\,\texttt{in}\,N$. As usual, to improve readability, we often write $\texttt{let}\,x = M\,\texttt{in}\,N$ to denote $(\lambda x.N)\ M$.

The remaining constructs define the thread and message-passing behavior of terms; their exact semantics will be defined for configurations, so we briefly describe them here.

The construct `new` creates a new buffered channel with two endpoints (referred to with variables). The construct $\texttt{fork}\, M; N$ forks a new thread running term M and continues as N. By involving an explicit continuation N, this is slightly different than the corresponding construct in LAST. We can think of N as the continuation to be run immediately after M is spawned. This is a mild generalization, as the construct in LAST corresponds to the case in which $N = ()$; it will be convenient for the correct translation into APCP.

The constructs $\texttt{send}\, N M$ and $\texttt{recv}\, M$ denote sending and receiving messages along M once it has reduced to a variable referring to a buffer endpoint, respectively; that is, sending entails placing the message N at the end of the buffer, and receiving entails taking a message from the start of the buffer (if there). The constructs $\texttt{select}\, \ell\, M$ and $\texttt{case}\, M\, \texttt{of}\, \{i : N_i\}_{i \in I}$ denote selecting and offering labels along M once it has reduced to an endpoint variable, respectively; that is, selection entails placing the label ℓ at the end of the buffer, and offering entails taking a label $j \in I$ from the start of the buffer (if there) and continuing in the corresponding branch N_j.

Following [GV10], the term reduction semantics of LAST* employs a call-by-value (CbV) approach; Figure 10 (center) defines values v, i.e., terms that cannot further reduce on their own: variables, abstractions, pairs, and the unit value. Figure 10 (center) also defines reduction contexts $\mathcal{R}$ that define under which positions subterms may reduce. Finally, Figure 10 (bottom) defines term reduction ($\rightarrow_{\texttt{M}}$). Rule [RED-LAM] reduces an abstraction applied to a value; the substitution of a value v for a (free) variable x is denoted $\{v/x\}$, as usual. Rule [RED-PAIR] reduces the pair deconstruction of a pair of values to two substitutions. Rule [RED-LIFT] closes term reduction under reduction contexts.

Example 4.1. To illustrate the CbV semantics of LAST*, consider the following term and its behavior:

$$\big(\lambda x.x\ (\lambda y.y)\big)\ \big((\lambda w.w)\ (\lambda z.z)\big) \rightarrow_{\texttt{M}} \big(\lambda x.x\ (\lambda y.y)\big)\ (\lambda z.z) \rightarrow_{\texttt{M}} (\lambda z.z)\ (\lambda y.y) \rightarrow_{\texttt{M}} \lambda y.y$$

We will illustrate the message-passing behavior of LAST* after introducing configurations below.

Configurations. Functional calculi such as LAST* feature a clear distinction between the static and dynamic parts of their languages. That is, a LAST* program starts as a closed functions (the static part) and evolves into several threads operating in parallel and communicating through message passing (the dynamic part). We refer to the static and dynamic parts of LAST* as terms and configurations, respectively. In general, one writes a LAST* program as a single main thread containing a term that forks and connects child threads. To contrast, process calculi such as APCP blur the lines between such static and dynamic parts, as APCP "programs" are immediately written as configurations of parallel subprocesses connected on channels.

Figure 11 gives the configuration syntax for LAST*. Given a term M, the configuration $\phi\, M$ denotes a corresponding thread, where the marker ϕ is useful to distinguish the main thread $\blacklozenge$ from child threads $\lozenge$—this distinction will be useful for typing. Buffered channels are denoted $(\boldsymbol{\nu} x[\vec{m}\rangle y)C$. Here, C has access to the endpoints x and y. The buffer itself $x[\vec{m}\rangle y$ is an ordered sequence of messages (values and labels, denoted $\vec{m}$) sent on x and to be received on y. This means that C may send/select on x and receive/offer on y. Once the buffer is empty (i.e., $\vec{m} = \epsilon$), x and y may switch roles. Configuration $C \parallel D$ denotes the

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

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

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

$$[\text{SC-RES-SWAP}]\ \frac{}{(\boldsymbol{\nu} x[\epsilon\rangle y)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C} \qquad [\text{SC-RES-COMM}]\ \frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\boldsymbol{\nu} z[\vec{n}\rangle w)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} z[\vec{n}\rangle w)(\boldsymbol{\nu} x[\vec{m}\rangle y)C}$$

$$[\text{SC-RES-EXT}]\ \frac{x, y \notin \text{fv}(C)}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D) \equiv_{\mathtt{C}} C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y)D} \qquad [\text{SC-PAR-COMM}]\ \frac{}{C \parallel D \equiv_{\mathtt{C}} D \parallel C}$$

$$[\text{SC-PAR-ASSOC}]\ \frac{}{C \parallel (D \parallel E) \equiv_{\mathtt{C}} (C \parallel D) \parallel E}$$

Configuration reduction ($\rightarrow_{\mathtt{C}}$):

$$[\text{RED-NEW}]\ \frac{}{\mathcal{F}[\mathtt{new}] \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)(\mathcal{F}[(x, y)])} \qquad [\text{RED-FORK}]\ \frac{}{\mathcal{F}[\mathtt{fork}\, M; N] \rightarrow_{\mathtt{C}} \mathcal{F}[N] \parallel \lozenge\, M}$$

$$[\text{RED-SEND}]\ \frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{send}\, v\, x] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[v, \vec{m}\rangle y)(\mathcal{F}[x] \parallel C)}$$

$$[\text{RED-RECV}]\ \frac{}{(\boldsymbol{\nu} x[\vec{m}, v\rangle y)(\mathcal{F}[\mathtt{recv}\, y] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[(v, y)] \parallel C)}$$

$$[\text{RED-SELECT}]\ \frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{select}\, \ell\, x] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\ell, \vec{m}\rangle y)(\mathcal{F}[x] \parallel C)}$$

$$[\text{RED-CASE}]\ \frac{j \in I}{(\boldsymbol{\nu} x[\vec{m}, j\rangle y)(\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_i\}_{i \in I}] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[M_j\ y] \parallel C)}$$

$$[\text{RED-RES-NIL}]\ \frac{x, y \notin \text{fv}(C)}{(\boldsymbol{\nu} x[\epsilon\rangle y)C \rightarrow_{\mathtt{C}} C} \qquad [\text{RED-PAR-NIL}]\ \frac{}{C \parallel \lozenge\, () \rightarrow_{\mathtt{C}} C} \qquad [\text{RED-LIFT-C}]\ \frac{C \rightarrow_{\mathtt{C}} C'}{\mathcal{G}[C] \rightarrow_{\mathtt{C}} \mathcal{G}[C']} \qquad [\text{RED-LIFT-M}]\ \frac{M \rightarrow_{\mathtt{M}} M'}{\mathcal{F}[M] \rightarrow_{\mathtt{C}} \mathcal{F}[M']}$$

$$[\text{RED-CONF-LIFT-SC}]\ \frac{C \equiv_{\mathtt{C}} C' \qquad C' \rightarrow_{\mathtt{C}} D' \qquad D' \equiv_{\mathtt{C}} D}{C \rightarrow_{\mathtt{C}} D}$$

Figure 11: The LAST* configuration language: syntax and semantics.

parallel composition of C and D. Figure 11 (top) also defines thread contexts $\mathcal{F}$ as term reduction contexts inside threads, and configuration contexts $\mathcal{G}$.

The reduction semantics for configurations is defined on specific arrangements of buffers and threads. To ensure such arrangements, we define a structural congruence for configurations ($\equiv_{\mathsf{C}}$), the least congruence on configurations satisfying the rules in Figure 11 (center). Rule [SC-RES-SWAP] swaps the direction of buffered channels and thus the input/output roles of the channel's endpoints. Rule [SC-RES-COMM] defines commutativity of buffered channels. Rule [SC-RES-EXT] defines scope extrusion/inclusion for buffered channels. Rules [SC-PAR-COMM] and [SC-PAR-ASSOC] define commutativity and associativity for parallel composition.

Figure 11 (bottom) gives the reduction semantics for configurations ($\to_{\mathsf{C}}$). It defines how terms in threads interact with each other by exchanging messages through buffered channels. Rule [RED-NEW] reduces a `new` construct in a thread by creating a new buffered channel and returning the endpoints x and y. Rule [RED-FORK] reduces a `fork` construct in a thread by creating a new child thread. Rule [RED-SEND] reduces a `send` by placing the value at the end of the enclosing buffer and returning the endpoint. Dually, Rule [RED-RECV] reduces a `recv` by retrieving the value at the start of the enclosing buffer and returning it along with the endpoint. Rule [RED-SELECT] reduces a `select` by placing the label at the end of the enclosing buffer and returning the endpoint. Dually, Rule [RED-CASE] reduces a `case` by retrieving the label at the start of the enclosing buffer and applying the label's corresponding continuation to the endpoint. Rule [RED-RES-NIL] garbage collects buffers that are no longer used, and Rule [RED-PAR-NIL] garbage collects child threads that have reduced to unit. Rules [RED-LIFT-C] and [RED-LIFT-M] close configuration reduction under configuration contexts and enable terms in threads to reduce, respectively. Rule [RED-CONF-LIFT-SC] closes configuration reduction under structural congruence.

Example 4.2 (The Bookshop Scenario, Revisited)**.** We illustrate the message-passing behavior of `LAST`* by considering the bookshop example from Section 2.2. Note that `LAST`* does not have `close` constructs: we will motivate them in Section 5; here, for simplicity, we consider the system in Section 2.2 without these constructs.

First, let us explore how $\blacklozenge\,\mathsf{Sys}$ sets up some channels and threads:

$$
\begin{aligned}
&\blacklozenge\ \mathtt{let}\ (s, s') = \mathtt{new}\ \mathtt{in}\ \mathtt{fork}\ \mathsf{Shop}(s); \mathtt{let}\ (m, m') = \mathtt{new}\ \mathtt{in}\ \mathtt{fork}\ \mathsf{Mother}(m); \mathsf{Son}(s', m') \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big(\blacklozenge\ \mathtt{let}\ (s, s') = (y, y')\ \mathtt{in}\ \mathtt{fork}\ \mathsf{Shop}(s); \\
&\qquad\qquad \mathtt{let}\ (m, m') = \mathtt{new}\ \mathtt{in}\ \mathtt{fork}\ \mathsf{Mother}(m); \mathsf{Son}(s', m')\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big(\blacklozenge\ \mathtt{fork}\ \mathsf{Shop}(y); \mathtt{let}\ (m, m') = \mathtt{new}\ \mathtt{in}\ \mathtt{fork}\ \mathsf{Mother}(m); \mathsf{Son}(y', m')\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big(\blacklozenge\ \mathtt{let}\ (m, m') = \mathtt{new}\ \mathtt{in}\ \mathtt{fork}\ \mathsf{Mother}(m); \mathsf{Son}(y', m') \parallel \Diamond\,\mathsf{Shop}(y)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ (m, m') = (z, z')\ \mathtt{in}\ \mathtt{fork}\ \mathsf{Mother}(m); \mathsf{Son}(y', m')\big) \parallel \Diamond\,\mathsf{Shop}(y)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{fork}\ \mathsf{Mother}(z); \mathsf{Son}(y', z')\big) \parallel \Diamond\,\mathsf{Shop}(y)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\,\mathsf{Son}(y', z') \parallel \Diamond\,\mathsf{Mother}(z)\big) \parallel \Diamond\,\mathsf{Shop}(y)\big)
\end{aligned}
$$

Next, let us see how the son sends the book title and his choice to buy to the shop, and then his connection with the shop to his mother, without waiting for any of his messages to be received:

$$
\begin{aligned}
\equiv_{\mathsf{C}}\ &(\boldsymbol{\nu} y'[\epsilon\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ s_1' = \mathtt{send}\,\text{“Dune”}\,y'\ \mathtt{in}\ \ldots \parallel \Diamond\,\mathsf{Mother}(z)\big) \parallel \Diamond\,\mathsf{Shop}(y)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y'[\text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ s_1' = y'\ \mathtt{in}\ \ldots \parallel \Diamond\,\mathsf{Mother}(z)\big) \parallel \Diamond\,\mathsf{Shop}(y)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y'[\text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ s_2' = \mathtt{select}\,\mathsf{buy}\,y'\ \mathtt{in}\ \ldots \parallel \Diamond\,\mathsf{Mother}(z)\big) \parallel \Diamond\,\mathsf{Shop}(y)\big)
\end{aligned}
$$

$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ s_2' = y'\ \mathtt{in}\ldots \parallel \lozenge\,\mathsf{Mother}(z)\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$
$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ m_1' = \mathtt{send}\, y'\, z'\ \mathtt{in}\ldots \parallel \lozenge\,\mathsf{Mother}(z)\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$
$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z'[y'\rangle z)\big(\blacklozenge\ \mathtt{let}\ m_1' = z'\ \mathtt{in}\ldots \parallel \lozenge\,\mathsf{Mother}(z)\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$
$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z'[y'\rangle z)\big(\blacklozenge\ \mathtt{let}\ (book, m_2') = \ldots \parallel \lozenge\,\mathsf{Mother}(z)\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$

Now, we can see how the mother receives the shop's connection and sends her credit card information:

$$= (\boldsymbol{\nu} y'[\mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z'[y'\rangle z)\big(\blacklozenge\ \mathtt{let}\ (book, m_2') = \mathtt{recv}\, z'\ \mathtt{in}\ book \parallel \lozenge\ \mathtt{let}\ (x, m_1) = \mathtt{recv}\, z\ \mathtt{in}\ldots\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$
$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ (book, m_2') = \mathtt{recv}\, z'\ \mathtt{in}\ book \parallel \lozenge\ \mathtt{let}\ (x, m_1) = (y', z)\ \mathtt{in}\ldots\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$
$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ (book, m_2') = \mathtt{recv}\, z'\ \mathtt{in}\ book \parallel \lozenge\ \mathtt{let}\ x_1 = \mathtt{send}\,\mathsf{visa}\, y'\ \mathtt{in}\ldots\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$
$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{visa}, \mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ (book, m_2') = \mathtt{recv}\, z'\ \mathtt{in}\ book \parallel \lozenge\ \mathtt{let}\ x_1 = y'\ \mathtt{in}\ldots\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$
$$\to_{\mathsf{C}} (\boldsymbol{\nu} y'[\mathsf{visa}, \mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ (book, m_2') = \mathtt{recv}\, z'\ \mathtt{in}\ book \parallel \lozenge\ \mathtt{let}\ (book, x_2) = \mathtt{recv}\, y'\ \mathtt{in}\ldots\big) \parallel \lozenge\,\mathsf{Shop}(y)\big)$$

Finally, the sequence of reductions in Figure 12 shows how the shop reads messages and sends the book, and how the mother forwards it to her son.

4.2. **The Type System of LAST⋆.** The type system for LAST⋆ includes functional types for functions and pairs and session types for message passing. The syntax and meaning of functional types (T, U) and session types (S) are as follows:

$$\begin{array}{rrlrlrlrl} T, U & ::= & T \times U & \text{pair} \mid & T \multimap U & \text{function} \mid & \mathbf{1} & \text{unit} \mid & S & \text{session} \\ S & ::= & !T.S & \text{send} \mid & ?T.S & \text{receive} \mid & \oplus\{i : T\}_{i\in I} & \text{select} \mid & \&\{i : T\}_{i\in I} & \text{branch} \\ & \mid & \mathsf{end} & & & & & & & \end{array}$$

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

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

The type system for LAST⋆ has three layers: typing for terms, for buffers, and for configurations. Typing for terms uses judgments of the form

$$\Gamma \vdash_{\mathtt{M}} M : T$$

which decrees that M has a behavior described by T using the typing context Γ, which is defined as a list of variable-type assignments $x : T$.

Figure 13 (top) gives the typing rules for terms. Rules often combine typing contexts Γ and Δ to form Γ, Δ; this implicitly assumes that the domains of Γ and Δ are disjoint. We briefly comment on them:

- Rules [TYP-VAR], [TYP-ABS], [TYP-APP], [TYP-UNIT], [TYP-PAIR], and [TYP-SPLIT] are standard typing rules for functional terms.
- Rule [TYP-NEW] types the `new` construct as a pair of dual session types.

$$
\begin{aligned}
&(\boldsymbol{\nu} y'[\text{visa}, \mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ (\mathit{book}, m_2') = \mathtt{recv}\ z'\ \mathtt{in}\ \mathit{book} \\
&\qquad\qquad \parallel \lozenge\ \mathtt{let}\ (\mathit{book}, x_2) = \mathtt{recv}\ y'\ \mathtt{in} \ldots\big) \\
&\qquad \parallel \lozenge\ \mathtt{let}\ (\mathit{title}, s_1) = \mathtt{recv}\ y\ \mathtt{in} \ldots\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y'[\text{visa}, \mathsf{buy}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge\ \mathtt{let}\ (\mathit{title}, s_1) = (\text{“Dune”}, y)\ \mathtt{in} \ldots\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y'[\text{visa}, \mathsf{buy}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge\ \mathtt{case}\ y\ \mathtt{of}\ \{\ldots\}\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y'[\text{visa}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge\ (\lambda s_2 \ldots)\ y\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y'[\text{visa}\rangle y)\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge\ \mathtt{let}\ (\mathit{card}, s_3) = \mathtt{recv}\ y\ \mathtt{in} \ldots\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge\ \mathtt{let}\ (\mathit{card}, s_3) = (\text{visa}, y)\ \mathtt{in} \ldots\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge\ \mathtt{let}\ s_4 = \mathtt{send}\ \text{book}(\text{“Dune”})\ y\ \mathtt{in}\ ()\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\text{book}(\text{“Dune”})\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge\ \mathtt{let}\ s_4 = y\ \mathtt{in}\ ()\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\text{book}(\text{“Dune”})\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge \ldots\big) \parallel \lozenge()\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\text{book}(\text{“Dune”})\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge\ \mathtt{let}\ (\mathit{book}, x_2) = \mathtt{recv}\ y'\ \mathtt{in} \ldots\big)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge\ \mathtt{let}\ (\mathit{book}, x_2) = (\text{book}(\text{“Dune”}), y')\ \mathtt{in} \ldots\big)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge\ \mathtt{let}\ m_2 = \mathtt{send}\ \text{book}(\text{“Dune”})\ z\ \mathtt{in}\ ()\big)\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge \ldots \parallel \lozenge\ \mathtt{let}\ m_2 = \mathtt{send}\ \text{book}(\text{“Dune”})\ z\ \mathtt{in}\ ()\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} z[\text{book}(\text{“Dune”})\rangle z')\big(\blacklozenge \ldots \parallel \lozenge\ \mathtt{let}\ m_2 = z\ \mathtt{in}\ ()\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} z[\text{book}(\text{“Dune”})\rangle z')\big(\blacklozenge \ldots \parallel \lozenge()\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} z[\text{book}(\text{“Dune”})\rangle z')\big(\blacklozenge\ \mathtt{let}\ (\mathit{book}, m_2') = \mathtt{recv}\ z'\ \mathtt{in}\ \mathit{book}\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \mathtt{let}\ (\mathit{book}, m_2') = (\text{book}(\text{“Dune”}), z')\ \mathtt{in}\ \mathit{book}\big) \\
\to_{\mathsf{C}}\ &(\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\ \text{book}(\text{“Dune”})\big) \to_{\mathsf{C}} \blacklozenge\ \text{book}(\text{“Dune”})
\end{aligned}
$$

Figure 12: Reduction sequence from Example 4.2. Recall that notation book(title) is syntactic sugar for a lookup function. We abbreviate unchanged threads using "…".

- Rule [TYP-FORK] takes a term M of type $\mathbf{1}$ and a term N of type T to type the `fork` construct as T; as we will see in the typing of configurations, child threads must be typed $\mathbf{1}$, which explains the type of M.
- Rule [TYP-END] allows weakening typing contexts with end-typed variables, as closed sessions are not used.
- Rule [TYP-SEND] takes a term M of type T and a term N of type $!T.S$ to type a `send` of M along N as the continuation type S. Dually, Rule [TYP-RECV] takes a term M of type $?T.S$ to type a `recv` along M as a pair of the message's payload type and continuation $T \times S$.
- Rule [TYP-SEL] takes a term M of type $\oplus\{i : S_i\}_{i \in I}$ to type the selection of some $j \in I$ along M as the continuation S_j. Dually, Rule [TYP-CASE] takes a term M of type $\&\{i : S_i\}_{i \in I}$ and for every $i \in I$ a term N_i typed $S_i \multimap U$ (i.e., a function from the label i's continuation type S_i to some common but arbitrary type U) to type a `case` along M as U.

Example 4.3. To illustrate the typing rules, let us derive the typing of term $\mathsf{Son}(s', m')$ from Section 2.2. As in Example 4.2, we omit the `close` construct. To type the sugared

$$[\textsc{typ-var}]\ \frac{}{x : T \vdash_{\mathtt{M}} x : T} \qquad [\textsc{typ-abs}]\ \frac{\Gamma, x : T \vdash_{\mathtt{M}} M : U}{\Gamma \vdash_{\mathtt{M}} \lambda x.M : T \multimap U} \qquad [\textsc{typ-app}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : T \multimap U \quad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} M\ N : U}$$

$$[\textsc{typ-unit}]\ \frac{}{\emptyset \vdash_{\mathtt{M}} () : \mathbf{1}} \qquad [\textsc{typ-pair}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : T \quad \Delta \vdash_{\mathtt{M}} N : U}{\Gamma, \Delta \vdash_{\mathtt{M}} (M, N) : T \times U}$$

$$[\textsc{typ-split}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : T \times T' \quad \Delta, x : T, y : T' \vdash_{\mathtt{M}} N : U}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{let}\,(x, y) = M\,\mathtt{in}\,N : U} \qquad [\textsc{typ-new}]\ \frac{}{\emptyset \vdash_{\mathtt{M}} \mathtt{new} : S \times \overline{S}}$$

$$[\textsc{typ-fork}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : \mathbf{1} \quad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{fork}\,M; N : T} \qquad [\textsc{typ-end}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : T}{\Gamma, x : \mathsf{end} \vdash_{\mathtt{M}} M : T} \qquad [\textsc{typ-send}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : T \quad \Delta \vdash_{\mathtt{M}} N : !T.S}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{send}\,M\,N : S}$$

$$[\textsc{typ-recv}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : ?T.S}{\Gamma \vdash_{\mathtt{M}} \mathtt{recv}\,M : T \times S} \qquad [\textsc{typ-sel}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : \oplus\{i : S_i\}_{i \in I} \quad j \in I}{\Gamma \vdash_{\mathtt{M}} \mathtt{select}\,j\,M : S_j}$$

$$[\textsc{typ-case}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : \&\{i : S_i\}_{i \in I} \quad \forall i \in I.\ \Delta \vdash_{\mathtt{M}} N_i : S_i \multimap U}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{case}\,M\,\mathtt{of}\,\{i : N_i\}_{i \in I} : U}$$

$$[\textsc{typ-buf}]\ \frac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S'} \qquad [\textsc{typ-buf-send}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : T \quad \Delta \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S}{\Gamma, \Delta \vdash_{\mathtt{B}} [\vec{m}, M\rangle : S'\rangle !T.S} \qquad [\textsc{typ-buf-sel}]\ \frac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S_j \quad j \in I}{\Gamma \vdash_{\mathtt{B}} [\vec{m}, j\rangle : S'\rangle \oplus\{i : S_i\}_{i \in I}}$$

$$[\textsc{typ-main}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : \hat{T}}{\Gamma \vdash_{\mathtt{C}}^{\blacklozenge} \blacklozenge\, M : \hat{T}} \qquad [\textsc{typ-child}]\ \frac{\Gamma \vdash_{\mathtt{M}} M : \mathbf{1}}{\Gamma \vdash_{\mathtt{C}}^{\Diamond} \Diamond\, M : \mathbf{1}} \qquad [\textsc{typ-par}]\ \frac{\Gamma \vdash_{\mathtt{C}}^{\phi_1} C : T_1 \quad \Delta \vdash_{\mathtt{C}}^{\phi_2} D : T_2}{\Gamma, \Delta \vdash_{\mathtt{C}}^{\phi_1 + \phi_2} C \parallel D : T_1 + T_2}$$

$$[\textsc{typ-res}]\ \frac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S \quad \Delta, x : S' \vdash_{\mathtt{C}}^{\phi} C : T \quad \Gamma', y : \overline{S} = \Gamma, \Delta}{\Gamma' \vdash_{\mathtt{C}}^{\phi} (\boldsymbol{\nu} x[\vec{m}\rangle y) C : T}$$

Figure 13: LAST* typing rules for terms (top), buffers (center), and configurations (bottom).

terms $\mathtt{let}\,x = M\,\mathtt{in}\,N$ we use a sugared Rule [TYP-LET] derivable from Rules [TYP-ABS] and [TYP-APP]. We consider Str (string), B (book), and P (payment) to be primitive non-linear types that can be weakened/contracted at will and are self dual. Below, $S = !\mathsf{P}.?\mathsf{B}.\mathsf{end}$

and $S' = ?\mathsf{B}.\mathsf{end}$. We omit "TYP-" from rule labels.

$$\pi \triangleq \dfrac{\dfrac{\dfrac{}{m_1' : ?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} m_1' : ?\mathsf{B}.\mathsf{end}}\,[\textsc{var}]}{m_1' : ?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{recv}\, m_1' : \mathsf{B} \times \mathsf{end}}\,[\textsc{recv}] \quad \dfrac{\dfrac{}{book : \mathsf{B} \vdash_{\mathtt{M}} book : \mathsf{B}}\,[\textsc{var}]}{book : \mathsf{B}, m_2' : \mathsf{end} \vdash_{\mathtt{M}} book : \mathsf{B}}\,[\textsc{end}]}{m_1' : ?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{let}\,(book, m_2') = \mathtt{recv}\, m_1' \,\mathtt{in}\, book : \mathsf{B}}\,[\textsc{split}]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{s_2' : S \vdash_{\mathtt{M}} s_2' : S}\,[\textsc{var}] \quad \dfrac{}{m' : !S.?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} m' : !S.?\mathsf{B}.\mathsf{end}}\,[\textsc{var}]}{s_2' : S, m' : !S.?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{send}\, s_2'\, m' : ?\mathsf{B}.\mathsf{end}}\,[\textsc{send}] \quad \overset{\pi}{m_1' : ?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} \ldots : \mathsf{B}}}{s_2' : S, m' : !S.?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{let}\, m_1' = \mathtt{send}\, s_2'\, m' \,\mathtt{in}\, \ldots : \mathsf{B}}\,[\textsc{let}]}{s' : !\mathsf{Str}.\oplus\{\mathsf{buy} : S, \mathsf{blurb} : S'\}, m' : !S.?\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} \mathsf{Son}(s', m') : \mathsf{B}}\,\ldots$$

Similarly, the typings of the shop and the mother are as follows:

$$s : ?\mathsf{Str}.\&\{\mathsf{buy} : \overline{S}, \mathsf{blurb} : \overline{S'}\} \vdash_{\mathtt{M}} \mathsf{Shop}(s) : \mathbf{1}$$
$$m : ?S.!\mathsf{B}.\mathsf{end} \vdash_{\mathtt{M}} \mathsf{Mother}(m) : \mathbf{1}$$

As such, the types of s, s' and m, m' are pairwise dual. Hence, the typing of the entire system is simply

$$\emptyset \vdash_{\mathtt{M}} \mathsf{Sys} : \mathsf{B}.$$

The typing for buffered channels uses judgments of the form

$$\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S$$

The difference between S' and S is determined by the messages in the buffer $\vec{m}$: S denotes a sequence of sends and selections corresponding to the values and labels in $\vec{m}$, after which the type continues as S'. Thus, S denotes the type of a buffered channel's output endpoint before it sent the messages in $\vec{m}$, and S' denotes this endpoint's current type. This way, $\overline{S}$ signifies the type of the buffered channel's input endpoint, corresponding to a sequence of receives and offers corresponding to the values and labels in $\vec{m}$.

Figure 13 (center) gives the three typing rules for buffers. The typing context Γ is used in the typing of the values in the buffer. Rule [TYP-BUF] types an empty buffer; as such, $S' = S$. Rule [TYP-BUF-SEND] takes a value v of type T and a buffer typed $S'\rangle S$ to insert v at the start of the buffer now typed $S'\rangle !T.S$. Rule [TYP-BUF-SEL] takes a buffer typed $S'\rangle S_j$ (for some $j \in I$) to insert j at the start of the buffer, now typed as $S'\rangle \oplus\{i : S_i\}_{i \in I}$.

Typing for configurations uses judgments of the form

$$\Gamma \vdash_{\mathtt{C}}^{\phi} C : T$$

where the marker ϕ indicates whether C contains the main thread ($\phi = \blacklozenge$) or only child threads ($\phi = \lozenge$). When composing configurations marked ϕ_1 and ϕ_2, we compute a new marker $\phi_1 + \phi_2$, as follows:

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

Figure 13 (bottom) gives the typing rules for configurations. Rule [TYP-MAIN] takes a term of non-session type (denoted $\hat{T}$) and turns it into a main thread. Rule [TYP-CHILD] takes a term of type $\mathbf{1}$ and turns it into a child thread. Rule [TYP-PAR] composes two configurations

typed T_1 and T_2 in parallel; the rule requires one of the configurations to be typed $\mathbf{1}$ and uses the other configuration's type to type the composition $T_1 + T_2$:

$$T_1 + T_2 \triangleq \begin{cases} T_1 & \text{if } T_2 = \mathbf{1} \\ T_2 & \text{if } T_1 = \mathbf{1} \\ \text{undefined} & \text{otherwise} \end{cases}$$

As such, Rule [TYP-PAR] is not defined for configurations both not typed $\mathbf{1}$; moreover, if both configurations contain a main thread, the sum of their markers is undefined, and the rule cannot be applied. Rule [TYP-RES] types a configuration under a buffered channel with output endpoint x and input endpoint y. The rule uses typing for buffers to type the buffer $S'\rangle S$. As such, the configuration should use x of type S'. Since S is the type of x before it sent the messages already in the buffer, y should be of type $\overline{S}$. Note that y may be used in the configuration, but may also appear in a message in the buffer.

4.3. **Towards a Faithful Translation of LAST* into APCP.** As already discussed, we are interested in faithfully translating LAST* into APCP. We first discuss what we mean precisely by 'faithful'. Then, we consider some existing translations of (variants of) the λ-calculus with call-by-value semantics into (variants of) the π-calculus; we briefly discuss their status with respect to our definition of faithfulness.

Faithfulness. We shall aim for correct translations in the sense of Gorla [Gor10], a well-established set of criteria whereby faithful translations should satisfy *operational correspondence*, a criterion that is divided into *completeness* and *soundness* properties. Intuitively, completeness says that the target language *does enough* to represent the behavior of the source language, whereas soundness says that the target language *does not do too much*. Alternatively, one may see completeness and soundness as properties that ensure that reduction steps are *preserved* and *reflected* by the translation, respectively.

We formulate these requirements in the specific setting of LAST* and APCP, by considering a translation of configurations into processes, denoted '$[\![\cdot]\!]z$', where, as usual, z denotes a name on which the resulting process exhibits the behavior of the source term:

Completeness.: If $C \to_{\mathtt{C}} D$, then $[\![C]\!]z \to^* [\![D]\!]z$.
Soundness.: If $[\![C]\!]z \to^* Q$, then there exists D such that $C \to^*_{\mathtt{C}} D$ and $Q \to^* [\![D]\!]z$.

Soundness and completeness come in different flavors of strength; see, e.g., [Pet19]. Our definitions can be seen as being strictly stronger than Gorla's, in that we do not consider a behavioral relation on target processes (typically used to abstract away from 'junk processes' in comparisons with $[\![D]\!]z$).

Existing Translations are not Sound. We are not aware of translations of λ-calculi with call-by-value semantics into the π-calculus that satisfy soundness as stated above. To illustrate the problem, let us consider the well-known translation by Milner [Mil90, Mil92]. This is a translation into a π-calculus with synchronous communication, while we seek a translation into APCP, which is asynchronous. This discrepancy is not crucial: the unsound reductions we will show next are not enabled by the asynchrony of APCP (i.e., they are also enabled in synchronous processes).

It is actually sufficient to consider Milner's approach to translating variables and applications. By adapting this approach to APCP, we obtain the translations given next. Below,

we write '_' to denote a fresh name of type $\bullet$; when sending names denoted '_', we omit binders '$(\boldsymbol{\nu}\_\_)$'.

$$[\![x]\!]z \triangleq (\boldsymbol{\nu}ab)(z[\_,a] \mid b(\_,c); x[\_,c])$$
$$[\![M\ N]\!]z \triangleq (\boldsymbol{\nu}ab)(\boldsymbol{\nu}cd)\big(a(\_,e); (\boldsymbol{\nu}fg)(e[\_,f] \mid c(\_,h); g[h,z]) \mid [\![M]\!]b \mid [\![N]\!]d\big)$$

Consider the very simple term $x\ y$, which cannot reduce. Yet, its translation $[\![x\ y]\!]z$ does reduce, as shown next (each step underlines the send and receive that synchronize):

$$\begin{aligned}
[\![x\ y]\!]z = (\boldsymbol{\nu}a_1b_1)(\boldsymbol{\nu}c_1d_1)\big(&\underline{a_1(_,e_1)}; (\boldsymbol{\nu}f_1g_1)(e_1[_,f_1] \mid c_1(_,h_1); g_1[h_1,z])\\
&\mid (\boldsymbol{\nu}a_2b_2)(\underline{b_1[_,a_2]} \mid b_2(_,c_2); x[_,c_2])\\
&\mid (\boldsymbol{\nu}a_3b_3)(\underline{d_1[_,a_3]} \mid b_3(_,c_3); y[_,c_3])\big)\\
\to (\boldsymbol{\nu}c_1d_1)(\boldsymbol{\nu}a_2b_2)\big(&(\boldsymbol{\nu}f_1g_1)(\underline{a_2[_,f_1]} \mid c_1(_,h_1); g_1[h_1,z])\\
&\mid \underline{b_2(_,c_2)}; x[_,c_2]\\
&\mid (\boldsymbol{\nu}a_3b_3)(d_1[_,a_3] \mid b_3(_,c_3); y[_,c_3])\big)\\
\to (\boldsymbol{\nu}c_1d_1)(\boldsymbol{\nu}f_1g_1)\big(&\underline{c_1(_,h_1)}; g_1[h_1,z]\\
&\mid x[_,f_1]\\
&\mid (\boldsymbol{\nu}a_3b_3)(\underline{d_1[_,a_3]} \mid b_3(_,c_3); y[_,c_3])\big)\\
\to (\boldsymbol{\nu}f_1g_1)(\boldsymbol{\nu}a_3b_3)&(g_1[a_3,z] \mid x[_,f_1] \mid b_3(_,c_3); y[_,c_3]) \not\to
\end{aligned}$$

It is clear that this final term, which cannot reduce any further, cannot be reconciled with any source $\texttt{LAST}^\star$ term. Hence, the translation is unsound.

Besides Milner's, other translations of λ-calculi with call-by-value semantics into (variants of) the π-calculus have been proposed by Lindley and Morris [LM15] (a precise formalization of Walder's [Wad12, Wad14]), by Vasconcelos [Vas00], and by Fowler *et al.* [FKD+23]. One of the most significant differences between them is how they translate variables: Milner uses sends; Lindley and Morris and Fowler *et al.* use forwarders; Vasconcelos uses substitutions. None of them satisfy soundness as defined above, although Vasconcelos' and Fowler *et al.*'s enjoy a weaker form of soundness that holds up to an appropriate behavioral equivalence.

By examining these three translations, we observe that the call-by-value semantics is overly contextual, in the sense that determining whether a subterm may reduce depends on the context. This way, e.g., Rule [RED-LAM] (Figure 10) only applies to abstractions applied to values. The semantic rules for the π-calculus are much less contextual, so translations require additional machinery to prevent unwanted reductions. We are not aware of translations of call-by-value λ-calculi into π that are sound, which in our view suggests that such a sound translation may not exist.

Based on this discussion, we conclude that the call-by-value semantics of $\texttt{LAST}^\star$ does not lend itself for a faithful representation in APCP. To address this issue, in the next section we will propose a variant of LAST with a call-by-name semantics. This variant, denoted $\texttt{LAST}^n$, will admit a faithful (i.e., sound and complete) translation into APCP.

5. $\texttt{LAST}^n$ and a Faithful Translation into APCP

Here we present $\texttt{LAST}^n$, a variant of LAST with call-by-name semantics. Our presentation proceeds gradually, based on the presentation we gave for $\texttt{LAST}^\star$ (Section 4). We then give a typed translation of $\texttt{LAST}^n$ into APCP, and prove that it is faithful as discussed in Section 4.3. Finally, we show how the translation can help us identify a fragment of deadlock-free $\texttt{LAST}^n$ programs.

Terms ($M, N, \ldots$) and reduction contexts ($\mathcal{R}$):

$$
\begin{array}{rrlrl}
M, N ::= & x & \text{variable} & \mid \mathtt{new} & \text{create new channel} \\
\mid & () & \text{unit value} & \mid \mathtt{fork}\, M; N & \text{fork } M \text{ in parallel to } N \\
\mid & \lambda x.M & \text{abstraction} & \mid (M, N) & \text{pair construction} \\
\mid & M\ N & \text{application} & \mid \mathtt{let}\,(x, y) = M\ \mathtt{in}\ N & \text{pair deconstruction} \\
\mid & \mathtt{send}\, M\, N & \text{send } M \text{ along } N & \mid \mathtt{select}\, \ell\, M & \text{select label } \ell \text{ along } M \\
\mid & \mathtt{recv}\, M & \text{receive along } M & \mid \mathtt{case}\, M\, \mathtt{of}\, \{i : M\}_{i \in I} & \text{offer labels in } I \text{ along } M \\
\mid & \mathtt{close}\, M; N & \text{close } M & \mid M\{\!\![N/x]\!\!\} & \text{explicit substitution}
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{R} ::= & [\cdot] \mid \mathcal{R}\ M \mid \mathtt{send}\, M\, \mathcal{R} \mid \mathtt{recv}\, \mathcal{R} \mid \mathtt{let}\,(x, y) = \mathcal{R}\ \mathtt{in}\ M \\
\mid & \mathtt{select}\, \ell\, \mathcal{R} \mid \mathtt{case}\, \mathcal{R}\, \mathtt{of}\, \{i : M\}_{i \in I} \mid \mathtt{close}\, \mathcal{R}; M \mid \mathcal{R}\{\!\![M/x]\!\!\}
\end{array}
$$

Figure 14: Term syntax and reduction contexts for $\mathtt{LAST}^n$.

5.1. **The Language of $\mathtt{LAST}^n$.** To define $\mathtt{LAST}^n$, we start from $\mathtt{LAST}^\star$ and modify its semantics. We only need to change the definitions of reduction contexts and reduction in Figures 10 and 11. For the reader's convenience, Figure 14 gives the syntax of $\mathtt{LAST}^n$ terms and reduction contexts; Appendix A.1 contains a full, self-contained definition of the language and type system of $\mathtt{LAST}^n$.

The crux of the required changes is that call-by-name semantics applies abstractions more eagerly, and not only when arguments are values. To be precise, term reduction changes Rules [RED-LAM] and [RED-PAIR] as follows:

$$
\frac{}{(\lambda x.M)\ \underline{N} \to_{\mathtt{M}} M\{\underline{N}/x\}}\ [\text{RED-LAM}] \qquad \frac{}{\mathtt{let}\,(x, y) = (\underline{N_1}, \underline{N_2})\ \mathtt{in}\ M \to_{\mathtt{M}} M\{\underline{N_1}/x, \underline{N_2}/y\}}\ [\text{RED-PAIR}]
$$

Above, we underline modified parts: instead of requiring values, the rules allow arbitrary terms. Moreover, to enforce the eager application of these rules, we remove from the definition of reduction contexts the clauses $v\ \mathcal{R}$, $(\mathcal{R}, M)$, and $(v, \mathcal{R})$. That is, $\mathtt{LAST}^n$ disallows terms in parameter positions to reduce.

Example 5.1. We consider the term from Example 4.1, now using the call-by-name semantics of $\mathtt{LAST}^n$:

$$
\big(\lambda x.x\ (\lambda y.y)\big)\ \big((\lambda w.w)\ (\lambda z.z)\big) \to_{\mathtt{M}} \big((\lambda w.w)\ (\lambda z.z)\big)\ (\lambda y.y) \to_{\mathtt{M}} (\lambda z.z)\ (\lambda y.y) \to_{\mathtt{M}} \lambda y.y
$$

Notice how here the function on x is applied *before* its parameter is evaluated as in Example 4.1.

Following the same call-by-name spirit, message passing in $\mathtt{LAST}^n$ transmits unevaluated terms, instead of only values as in $\mathtt{LAST}^\star$. To accommodate this, we replace the two reduction-context clauses for $\mathtt{send}$ with the clause $\mathtt{send}\, M\, \mathcal{R}$, and modify Rule [RED-SEND] accordingly:

$$
\frac{}{\nu x[\vec{m}\rangle y(\mathcal{F}[\mathtt{send}\, M\, x] \parallel C) \to_{\mathtt{C}} \nu x[\vec{m}, M\rangle y(\mathcal{F}[x] \parallel C)}\ [\text{RED-SEND}]
$$

As is to be expected, there are subtle differences between the semantics of $\mathtt{LAST}^n$ and APCP. In order to deal with these discrepancies, while enabling the desired completeness and soundness results, we enrich $\mathtt{LAST}^n$ with *explicit substitutions* and *closed sessions*.

Explicit Substitutions. Term reductions [RED-LAM] and [RED-PAIR] substitute variables for terms, regardless of the position of these variables. To mimic this in APCP, we would have to be able to substitute the translations of variables for translations of terms. Although this is possible when those variables occur in reduction contexts, there is no mechanism for representing this in APCP when they occur in different contexts (where their translations are blocked by prefixes).

One way of dealing with this issue is to equate translations up to substitutions, using the so-called *substitution lifting* [TCP12]. Here, we opt for an alternative, more direct treatment based on *explicit substitutions* (see, e.g., [LM99]). Intuitively, explicit substitutions delay variable substitution until those variables occur in reduction contexts. To incorporate explicit substitutions, we proceed as follows:

- We add explicit substitutions to the syntax of terms $M\{\!\{N/x\}\!\}$ and configurations $C\{\!\{N/x\}\!\}$. These are not meant to be used when writing programs, instead appearing and disappearing as programs reduce (runtime syntax). That is, they appear when abstractions are applied and pairs deconstructed:

$$\frac{}{(\lambda x.M)\ N \to_{\mathtt{M}} M\{\!\{N/x\}\!\}}\ [\textsc{red-lam}] \qquad \frac{}{\mathtt{let}\,(x,y) = (N_1,N_2)\,\mathtt{in}\,M \to_{\mathtt{M}} M\{\!\{N_1/x, N_2/y\}\!\}}\ [\textsc{red-pair}]$$

  Explicit substitutions disappear when they meet the substituted variable, as per the following rule:

$$\frac{}{x\{\!\{N/x\}\!\} \to_{\mathtt{M}} N}\ [\textsc{red-subst}]$$

- We add reduction contexts $\mathcal{R}\{\!\{N/x\}\!\}$ and configuration contexts $\mathcal{G}\{\!\{N/x\}\!\}$. We then define a structural congruence for terms, denoted $\equiv_{\mathtt{M}}$, with a single rule that enables extruding the scope of explicit substitutions across reduction contexts:

$$\frac{x \notin \mathrm{fv}(\mathcal{R})}{(\mathcal{R}[M])\{\!\{N/x\}\!\} \equiv_{\mathtt{M}} \mathcal{R}[M\{\!\{N/x\}\!\}]}\ [\textsc{sc-sub-ext}]$$

  We close term reduction under this structural congruence:

$$\frac{M \equiv_{\mathtt{M}} M' \qquad M' \to_{\mathtt{M}} N' \qquad N' \equiv_{\mathtt{M}} N}{M \to_{\mathtt{M}} N}\ [\textsc{red-lift-sc}]$$

  We also add rules to the structural congruence for configurations that lift explicit substitutions in terms and enable scope extrusion on the level of configurations, respectively:

$$\frac{}{\phi\,(M\{\!\{N/x\}\!\}) \equiv_{\mathtt{C}} (\phi\,M)\{\!\{N/x\}\!\}}\ [\textsc{sc-conf-sub}] \qquad \frac{x \notin \mathrm{fv}(\mathcal{G})}{(\mathcal{G}[C])\{\!\{N/x\}\!\} \equiv_{\mathtt{C}} \mathcal{G}[C\{\!\{N/x\}\!\}]}\ [\textsc{sc-conf-sub-ext}]$$

- When moving terms between threads, we need to make sure that we do not affect variables that are bound by explicit substitutions. As such we define a specific form of thread context, denoted $\hat{\mathcal{F}}$, that disallows the hole to appear under explicit substitutions. Configuration reductions [RED-FORK], [RED-SEND], and [RED-RECV] (cf. Figure 11) then use these specific contexts.

- To type explicit substitutions, we add the following typing rules for terms and configurations:

$$\frac{\Gamma, x : T \vdash_{\mathtt{M}} M : U \qquad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} M\{\!\{N/x\}\!\} : U}\ [\textsc{typ-sub}] \qquad \frac{\Gamma, x : T \vdash^{\phi}_{\mathtt{C}} C : U \qquad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash^{\phi}_{\mathtt{C}} C\{\!\{N/x\}\!\} : U}\ [\textsc{typ-conf-sub}]$$

Example 5.2. We revisit Example 5.1, this time using explicit substitutions:

$$\begin{aligned}(\lambda x.x\ (\lambda y.y))\ ((\lambda w.w)\ (\lambda z.z)) &\to_{\mathtt{M}} (x\ (\lambda y.y))\{\!\{((\lambda w.w)\ (\lambda z.z))/x\}\!\}\\ &\equiv_{\mathtt{M}} (x\{\!\{((\lambda w.w)\ (\lambda z.z))/x\}\!\})\ (\lambda y.y)\\ &\to_{\mathtt{M}} ((\lambda w.w)\ (\lambda z.z))\ (\lambda y.y)\\ &\to_{\mathtt{M}} (w\{\!\{(\lambda z.z)/w\}\!\})\ (\lambda y.y) \to_{\mathtt{M}} (\lambda z.z)\ (\lambda y.y)\\ &\to_{\mathtt{M}} z\{\!\{(\lambda y.y)/z\}\!\} \to_{\mathtt{M}} \lambda y.y\end{aligned}$$

Example 5.3 (The Bookshop Scenario in $\mathtt{LAST}^n$)**.** Recall the bookshop scenario introduced in Section 2.2, already illustrated by Example 4.2 for $\mathtt{LAST}^\star$. Although the end result is the same, the CbN semantics and explicit substitutions of $\mathtt{LAST}^n$ do change the behavior of the system ($\blacklozenge\,\mathsf{Sys}$) compared to its behavior under $\mathtt{LAST}^\star$.

We start to illustrate this behavior by reconsidering how the system starts by setting up channels; as we will see, not all explicit substitutions can be immediately resolved (we sometimes abbreviate unchanged threads using "…"):

$$\begin{aligned}\blacklozenge\,\mathsf{Sys} &= \blacklozenge\ \mathtt{let}\ (s, s') = \mathtt{new}\ \mathtt{in}\ \ldots\\ &\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\blacklozenge\ \mathtt{let}\ (s, s') = (y, y')\ \mathtt{in}\ \ldots\\ &\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\blacklozenge\ \mathtt{fork}\,\mathsf{Shop}(s); \ldots\{\!\{y/s, y'/s'\}\!\}\\ &\equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\blacklozenge\ \mathtt{fork}\,\mathsf{Shop}(s); \ldots)\{\!\{y/s, y'/s'\}\!\}\big)\\ &\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\blacklozenge\ \mathtt{let}\ (m, m') = \mathtt{new}\ \mathtt{in}\ \ldots \parallel \lozenge\,\mathsf{Shop}(s))\{\!\{y/s, y'/s'\}\!\}\big)\\ &\equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\blacklozenge\ \ldots \parallel \lozenge\ \mathtt{let}\ (\mathit{title}, s_1) = \mathtt{recv}\ (s\{\!\{y/s\}\!\})\ \mathtt{in}\ \ldots)\{\!\{y'/s'\}\!\}\big)\\ &\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\blacklozenge\ \ldots \parallel \lozenge\ \mathtt{let}\ (\mathit{title}, s_1) = \mathtt{recv}\ y\ \mathtt{in}\ \ldots)\{\!\{y'/s'\}\!\}\big)\\ &\to^2_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')(\blacklozenge\ \mathtt{fork}\,\mathsf{Mother}(m); \mathsf{Son}(s', m'))\{\!\{z/m, z'/m', y'/s'\}\!\} \parallel \lozenge\,\mathsf{Shop}(y)\big)\\ &\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')(\blacklozenge\,\mathsf{Son}(s', m')\{\!\{z'/m', y'/s'\}\!\} \parallel \lozenge\,\mathsf{Mother}(m)\{\!\{z/m\}\!\}) \parallel \lozenge\,\mathsf{Shop}(y)\big)\\ &\equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')(\blacklozenge\ \ldots \parallel \lozenge\ \mathtt{let}\ (x, m_1) = \mathtt{recv}\ (m\{\!\{z/m\}\!\})\ \mathtt{in}\ \ldots) \parallel \lozenge\,\mathsf{Shop}(y)\big)\\ &\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')(\blacklozenge\ \ldots \parallel \lozenge\ \mathtt{let}\ (x, m_1) = \mathtt{recv}\ z\ \mathtt{in}\ \ldots) \parallel \lozenge\,\mathsf{Shop}(y)\big)\\ &= (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')(\blacklozenge\,\mathsf{Son}(s', m')\{\!\{z'/m', y'/s'\}\!\} \parallel \lozenge\,\mathsf{Mother}(z)) \parallel \lozenge\,\mathsf{Shop}(y)\big) \triangleq \mathsf{Sys}^1\end{aligned}$$

At this point, the explicit substitutions on the main thread cannot be resolved, as neither of m', s' appears under a reduction context in $\mathsf{Son}_{s',m'}$. We continue the example by examining

$$
\begin{aligned}
&\mathsf{Sys}^1\\
&\to_{\mathtt{C}}^{5} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\, \mathsf{Son}^1(s', z') \parallel \Diamond\, \mathsf{Mother}(z)\big) \parallel \Diamond\, \mathsf{Shop}(y)\big)\\
&\equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\, \mathtt{let}\,(\mathit{book}, m_2') = \mathtt{recv}\,(\mathtt{send}\, s_2'\, z')\,\mathtt{in}\ldots \parallel \Diamond\ldots\big)\{\!\!\{\ldots\}\!\!\} \parallel \Diamond\ldots\big)\\
&\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[s_2'\rangle z)\big(\blacklozenge\, \mathtt{let}\,(\mathit{book}, m_2') = \mathtt{recv}\, z'\,\mathtt{in}\ldots \parallel \Diamond\, \mathtt{let}\,(x, m_1) = \mathtt{recv}\, z\,\mathtt{in}\ldots\big)\\
&\qquad\qquad \{\!\!\{\ldots\}\!\!\} \parallel \Diamond\ldots\big)\\
&\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\ldots \parallel \Diamond\, \mathtt{let}\,(x, m_1) = (s_2', z)\,\mathtt{in}\ldots\{\!\!\{\ldots\}\!\!\}\big) \parallel \Diamond\ldots\big)\\
&\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\ldots \parallel \Diamond\, \mathtt{let}\, x_1 = \mathtt{send}\,\mathrm{visa}\,(x\{\!\!\{s_2'/x, \ldots\}\!\!\})\,\mathtt{in}\ldots\{\!\!\{z/m_1\}\!\!\}\big) \parallel \Diamond\ldots\big)\\
&\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\ldots \parallel \Diamond\, \mathtt{let}\, x_1 = \ldots(s_2'\{\!\!\{\mathtt{select}\,\mathsf{buy}\, s_1'/s_2', \ldots\}\!\!\})\,\mathtt{in}\ldots\big) \parallel \Diamond\ldots\big)\\
&\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\ldots \parallel \Diamond\, \mathtt{let}\, x_1 = \ldots\,(\mathtt{select}\,\mathsf{buy}\,(s_1'\{\!\!\{\mathtt{send}\,\text{“Dune”}\, s'/s_1', y'/s'\}\!\!\}))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{in}\ldots\big) \parallel \Diamond\ldots\big)\\
&\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\ldots \parallel \Diamond\, \mathtt{let}\, x_1 = \ldots(\ldots(\mathtt{send}\,\text{“Dune”}\,(s'\{\!\!\{y'/s'\}\!\!\}))\,\mathtt{in}\ldots\big) \parallel \Diamond\ldots\big)\\
&\to_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\ldots \parallel \Diamond\, \mathtt{let}\, x_1 = \ldots(\ldots(\ldots y'))\,\mathtt{in}\ldots\big) \parallel \Diamond\ldots\big) \triangleq \mathsf{Sys}^2
\end{aligned}
$$

Figure 15: The Bookshop Scenario in $\mathtt{LAST}^n$ (cf. Example 5.3). We abbreviate unchanged terms and substitutions using "…".

the behavior of the son at this point (abbreviating some substitutions using "…"):

$$
\begin{aligned}
&\mathsf{Son}(s', m')\{\!\!\{z'/m', y'/s'\}\!\!\}\\
&= \mathtt{let}\, s_1' = \mathtt{send}\,\text{“Dune”}\, s'\,\mathtt{in}\ldots\{\!\!\{z'/m', y'/s'\}\!\!\}\\
&\to_{\mathtt{M}} \mathtt{let}\, s_2' = \mathtt{select}\,\mathsf{buy}\, s_1'\,\mathtt{in}\ldots\{\!\!\{\mathtt{send}\,\text{“Dune”}\, s'/s_1', z'/m', y'/s'\}\!\!\}\\
&\to_{\mathtt{M}} \mathtt{let}\, m_1' = \mathtt{send}\, s_2'\,(m'\{\!\!\{z'/m'\}\!\!\})\,\mathtt{in}\ldots\{\!\!\{\mathtt{select}\,\mathsf{buy}\, s_1'/s_2', \mathtt{send}\,\text{“Dune”}\, s'/s_1', y'/s'\}\!\!\}\\
&\to_{\mathtt{M}} \mathtt{let}\, m_1' = \mathtt{send}\, s_2'\, z'\,\mathtt{in}\ldots\{\!\!\{\ldots\}\!\!\}\\
&\to_{\mathtt{M}} \mathtt{let}\,(\mathit{book}, m_2') = \mathtt{recv}\,(m_1'\{\!\!\{\mathtt{send}\, s_2'\, z'/m_1'\}\!\!\})\,\mathtt{in}\ldots\{\!\!\{\ldots\}\!\!\}\\
&\to_{\mathtt{M}} \mathtt{let}\,(\mathit{book}, m_2') = \mathtt{recv}\,(\mathtt{send}\, s_2'\, z')\,\mathtt{in}\ldots\{\!\!\{\ldots\}\!\!\} \triangleq \mathsf{Son}^1(s', z')
\end{aligned}
$$

This chain of explicit substitutions cannot be resolved further: s_2' does not appear under a reduction context. In fact, they will not be resolved until the son has delegated his (pending) access to z' to his mother, along with his pending messages in the form of explicit substitutions, as shown by the sequence leading to process Sys^2 in Figure 15.

At this point, the system behaves roughly as in Example 4.2: there is no further session delegation (as we just witnessed between the son and the mother), so upcoming explicit substitutions can be resolved straightforwardly.

$$
\begin{aligned}
\mathsf{Sys}^2 \to_{\mathtt{C}}^{3} (\boldsymbol{\nu} y'[\mathrm{visa}, \mathsf{buy}, \text{“Dune”}\rangle y)\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\, \mathtt{let}\,(\mathit{book}, m_2') = \mathtt{recv}\, z'\,\mathtt{in}\ldots\\
\parallel \Diamond\, \mathtt{let}\, x_1 = y'\,\mathtt{in}\ldots\{\!\!\{z/m_1\}\!\!\}\big) \parallel \Diamond\, \mathsf{Shop}(y)\big)
\end{aligned}
$$

From here, the thread representing the mother can finally perform her son's two original outputs, followed by her own (omitted for brevity).

Closed Sessions. Both in $\texttt{LAST}^n$ and APCP, closed sessions have no associated behavior. This entails some management on both sides to garbage collect unused variables with type end. As a result, translating terms with variables requires two forms: one for regular variables, and another for variables typed end. Because such a translation can get burdensome, we opt to add a form of explicit closing to $\texttt{LAST}^n$. Once both endpoints of a channel have been closed, the channel and its buffer can be garbage collected. This treatment relieves us from translating unused variables, in such a way that the translation of terms with variables comes in one intuitive form only. To formalize this idea, we proceed as follows:

- We add terms for endpoint closing $\texttt{close}\, M; N$ and a reduction context $\texttt{close}\, \mathcal{R}; N$. Similar to fork, the term N is an explicit continuation for when the endpoint is closed. We define behavior on the level of configurations by adding a new reduction rule:

$$\frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\texttt{close}\, x; M] \parallel C) \rightarrow_{\mathsf{C}} (\boldsymbol{\nu} \Box[\vec{m}\rangle y)(\mathcal{F}[M] \parallel C)} \text{[RED-CLOSE]}$$

  Here, '$\Box$' is a special runtime endpoint variable: it indicates that the endpoint x has been closed and so the variable is not used anywhere inside the enclosed configuration. Then, buffered channels can be garbage collected when both endpoints have been closed, as enabled by the following reduction:

$$\frac{}{(\boldsymbol{\nu} \Box[\epsilon\rangle \Box) C \rightarrow_{\mathsf{C}} C} \text{[RED-RES-NIL]}$$

- We type the session closing construct by replacing Rule [TYP-END] with the following:

$$\frac{\Gamma \vdash_{\mathtt{M}} M : \mathsf{end} \qquad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} \texttt{close}\, M; N : T} \text{[TYP-CLOSE]}$$

- We add a new session type for endpoints that have been already closed. With a slight abuse of notation, we denote it as $\Box$. We disallow using this type in the typing of terms, leaving it only to appear in the typing of buffered channels. To ensure that closed endpoint variables are not used in configurations, we add side conditions to Rule [TYP-RES]: $x = \Box$ if and only if $S' = \Box$ and similarly for y. The following two new rules enable typing partially closed buffered channels:

$$\frac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \Box} \text{[TYP-BUF-END-L]} \qquad \frac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \Box\rangle \mathsf{end}} \text{[TYP-BUF-END-R]}$$

Example 5.4. We illustrate the explicit closing of sessions in $\texttt{LAST}^n$ in the context of the bookstore scenario introduced in Section 2.2. Consider the system where all session interactions have taken place, and all three threads are ready to close their sessions:

$$\mathsf{Sys} \rightarrow_{\mathsf{C}}^{*} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle z')\big(\blacklozenge\, \texttt{close}\, z'; \mathrm{book}(\text{“Dune”}) \parallel \lozenge\, \texttt{close}\, z; \texttt{close}\, y'\big) \parallel \lozenge\, \texttt{close}\, y\big)$$
$$\triangleq \mathsf{Sys}^{c}$$

Once again, notation book(*title*) is syntactic sugar for a lookup function. The order in which endpoints are closed is unimportant; here, we execute the `close` primitives from left to right.

$$
\begin{aligned}
\mathsf{Sys}^c &\equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z'[\epsilon\rangle z)\big(\blacklozenge\ \mathtt{close}\, z'; \text{book}(\text{“Dune”}) \parallel \lozenge\ \mathtt{close}\, z; \mathtt{close}\, y'\big) \parallel \lozenge\, \mathtt{close}\, y\big) \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu}\Box[\epsilon\rangle z)\lozenge\ \mathtt{close}\, z; \mathtt{close}\, y' \parallel \lozenge\, \mathtt{close}\, y\big) \\
&\equiv_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu} z[\epsilon\rangle\Box)\lozenge\ \mathtt{close}\, z; \mathtt{close}\, y' \parallel \lozenge\, \mathtt{close}\, y\big) \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu} y[\epsilon\rangle y')\big((\boldsymbol{\nu}\Box[\epsilon\rangle\Box)\lozenge\ \mathtt{close}\, y' \parallel \lozenge\, \mathtt{close}\, y\big) \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu} y[\epsilon\rangle y')\big(\lozenge\ \mathtt{close}\, y' \parallel \lozenge\, \mathtt{close}\, y\big) \\
&\equiv_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu} y'[\epsilon\rangle y)\big(\lozenge\ \mathtt{close}\, y' \parallel \lozenge\, \mathtt{close}\, y\big) \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel \lozenge\,() \parallel (\boldsymbol{\nu}\Box[\epsilon\rangle y)\lozenge\, \mathtt{close}\, y \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu}\Box[\epsilon\rangle y)\lozenge\, \mathtt{close}\, y \\
&\equiv_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu} y[\epsilon\rangle\Box)\lozenge\, \mathtt{close}\, y \qquad (*) \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu}\Box[\epsilon\rangle\Box)\lozenge() \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel \lozenge() \\
&\to_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”})
\end{aligned}
$$

Additionally, we illustrate the typing of half-closed sessions on the configuration marked $(*)$. Recall from Example 4.3 that we consider B (book) a primitive non-linear type that can be weakened/contracted at will and is self dual. We have (omitting "TYP-" from rule labels):

$$
\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{M}} \text{book}(\text{“Dune”}) : \mathsf{B}}}{\emptyset \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) : \mathsf{B}}\ [\textsc{main}] \quad \dfrac{\dfrac{\dfrac{}{y : \mathsf{end} \vdash_{\mathtt{M}} y : \mathsf{end}}\ [\textsc{var}] \quad \dfrac{}{\emptyset \vdash_{\mathtt{M}} () : \mathbf{1}}\ [\textsc{unit}]}{y : \mathsf{end} \vdash_{\mathtt{M}} \mathtt{close}\, y; () : \mathbf{1}}\ [\textsc{close}]}{y : \mathsf{end} \vdash^{\lozenge}_{\mathtt{C}} \lozenge\ \mathtt{close}\, y; () : \mathbf{1}}\ [\textsc{child}] \quad \dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle\Box}\ [\textsc{buf-end-L}]}{\emptyset \vdash^{\lozenge}_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle\Box)\lozenge\ \mathtt{close}\, y; () : \mathbf{1}}\ [\textsc{res}]}{\emptyset \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\, \text{book}(\text{“Dune”}) \parallel (\boldsymbol{\nu} y[\epsilon\rangle\Box)\lozenge\ \mathtt{close}\, y; () : \mathsf{B}}\ [\textsc{par}]
$$

Type Preservation. `LAST`^{n} satisfies the expected correctness properties for session-typed languages: protocol fidelity and communication safety. Both properties follow directly from *type preservation*.

[Type Preservation for `LAST`^{n}] Given $\Gamma \vdash^{\phi}_{\mathtt{C}} C : T$, if $C \equiv_{\mathtt{C}} D$ or $C \to_{\mathtt{C}} D$, then $\Gamma \vdash^{\phi}_{\mathtt{C}} D : T$.

Proof. By proving subject congruence ($\equiv_{\mathtt{C}}$ preserves typing) and subject reduction ($\to_{\mathtt{C}}$ preserves typing) separately. In both cases, we first prove separately subject congruence and subject reduction on the term level ($\equiv_{\mathtt{M}}$ and $\to_{\mathtt{M}}$, respectively). These results follow by induction on the derivation of the structural congruence or reduction. Below we give a few interesting cases; Appendix A.2 contains detailed proofs.

- For $\equiv_{\mathtt{M}}$, the only interesting case is the only base case, i.e., Rule [SC-SUB-EXT]:

$$x \notin \text{fv}(\mathcal{R}) \implies (\mathcal{R}[M])\{\!\{N/x\}\!\} \equiv_{\mathtt{M}} \mathcal{R}[M\{\!\{N/x\}\!\}]$$

  We apply induction on the structure of the reduction context $\mathcal{R}$. As an interesting, representative case, consider $\mathcal{R} = L\{\!\{\mathcal{R}'/y\}\!\}$. Assuming $x \notin \text{fv}(\mathcal{R})$, we have $x \notin \text{fv}(L) \cup$

$\mathrm{fv}(\mathcal{R}')$. We apply inversion of typing:

$$\dfrac{\dfrac{\Gamma, y : U \vdash_{\mathtt{M}} L : T \qquad \Delta, x : U' \vdash_{\mathtt{M}} \mathcal{R}'[M] : U}{\Gamma, \Delta, x : U' \vdash_{\mathtt{M}} L\{\!\{(\mathcal{R}'[M])/y\}\!\} : T}\,[\textsc{typ-sub}] \qquad \Delta' \vdash_{\mathtt{M}} N : U'}{\Gamma, \Delta, \Delta' \vdash_{\mathtt{M}} (L\{\!\{(\mathcal{R}'[M])/y\}\!\})\{\!\{N/x\}\!\} : T}\,[\textsc{typ-sub}]$$

We can derive:

$$\dfrac{\Delta, x : U' \vdash_{\mathtt{M}} \mathcal{R}'[M] : U \qquad \Delta' \vdash_{\mathtt{M}} N : U'}{\Delta, \Delta' \vdash_{\mathtt{M}} (\mathcal{R}'[M])\{\!\{N/x\}\!\} : U}\,[\textsc{typ-sub}]$$

Since $x \notin \mathrm{fv}(\mathcal{R}')$, by Rule [SC-SUB-EXT], $(\mathcal{R}'[M])\{\!\{N/x\}\!\} \equiv_{\mathtt{M}} \mathcal{R}'[M\{\!\{N/x\}\!\}]$. Then, by the IH, $\Delta, \Delta' \vdash_{\mathtt{M}} \mathcal{R}'[M\{\!\{N/x\}\!\}] : U$. Hence, we can conclude the following:

$$\dfrac{\Gamma, y : U \vdash_{\mathtt{M}} L : T \qquad \Delta, \Delta' \vdash_{\mathtt{M}} \mathcal{R}'[M\{\!\{N/x\}\!\}] : U}{\Gamma, \Delta, \Delta' \vdash_{\mathtt{M}} L\{\!\{(\mathcal{R}'[M\{\!\{N/x\}\!\}])/y\}\!\} : T}\,[\textsc{typ-sub}]$$

It is straightforward to see that this reasoning works in opposite direction as well.

- For $\rightarrow_{\mathtt{M}}$, all cases are straightforward.
- For $\equiv_{\mathtt{C}}$, we detail the base case Rule [SC-RES-SWAP]: $(\boldsymbol{\nu} x[\epsilon\rangle y)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C$.

  The analysis depends on whether exactly one of x, y is $\Box$ or not. We discuss both cases.

  – Exactly one of x, y is $\Box$; w.l.o.g., assume $x = \Box$. We have the following:

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \Box\rangle \mathsf{end}}\,[\textsc{typ-buf-end-R}] \qquad \Gamma, x : \mathsf{end} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu}\Box[\epsilon\rangle y)C : T}\,[\textsc{typ-res}]$$

$$\equiv_{\mathtt{C}}$$

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \Box}\,[\textsc{typ-buf-end-L}] \qquad \Gamma, x : \mathsf{end} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle \Box)C : T}\,[\textsc{typ-res}]$$

  – Neither or both of x, y are $\Box$. We have the following:

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S'}\,[\textsc{typ-buf}] \qquad \Gamma, x : S', y : \overline{S'} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)C : T}\,[\textsc{typ-res}]$$

$$\equiv_{\mathtt{C}}$$

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S'}\rangle \overline{S'}}\,[\textsc{typ-buf}] \qquad \Gamma, x : S', y : \overline{S'} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C : T}\,[\textsc{typ-res}]$$

- For $\rightarrow_{\mathtt{C}}$, we detail the base case Rule [RED-SEND]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[\mathtt{send}\, M\, x] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\hat{\mathcal{F}}[x] \parallel C)$.

  This case follows by induction on the structure of $\hat{\mathcal{F}}$. The inductive cases follow from the IH straightforwardly. The fact that the hole in $\hat{\mathcal{F}}$ does not occur under an explicit substitution guarantees that we can move M out of the context of $\hat{\mathcal{F}}$ and into the buffer. We consider the base case ($\hat{\mathcal{F}} = \phi\, \mathcal{R}$). By well typedness, it must be that $\mathcal{R} = \mathcal{R}_1[\mathtt{close}\, \mathcal{R}_2; M]$. We apply induction on the structures of $\mathcal{R}_1, \mathcal{R}_2$ and consider the base cases: $\mathcal{R}_1 = \mathcal{R}_2 = [\cdot]$. We apply inversion of typing, w.l.o.g. assuming that $\phi = \blacklozenge$ and $y \in \mathrm{fv}(C)$ (omitting "TYP-"

from rule labels):

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta_1 \vdash_{\mathtt{M}} M : T \quad \overline{x : !T.\mathsf{end} \vdash_{\mathtt{M}} x : !T.\mathsf{end}}\ [\textsc{var}]}{\Delta_1, x : !T.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{send}\, M\, x : \mathsf{end}}\ [\textsc{send}] \quad \Delta_2 \vdash_{\mathtt{M}} N : U}{\Delta_1, \Delta_2, x : !T.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{close}\,(\mathtt{send}\, M\, x); N : U}\ [\textsc{close}]}{\Delta_1, \Delta_2, x : !T.\mathsf{end} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{close}\,(\mathtt{send}\, M\, x); N) : U}\ [\textsc{main}] \quad \Lambda, y : \overline{S} \vdash^{\Diamond}_{\mathtt{C}} C : \mathbf{1}}{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : !T.\mathsf{end}\rangle S \quad \Delta_1, \Delta_2, \Lambda, x : !T.\mathsf{end}, y : \overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{close}\,(\mathtt{send}\, M\, x); N) \parallel C : U}\ [\textsc{par}]}{\Gamma, \Delta_1, \Delta_2, \Lambda \vdash^{\blacklozenge}_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\blacklozenge\,(\mathtt{close}\,(\mathtt{send}\, M\, x); N) \parallel C) : U}\ [\textsc{res}]$$

Note that the derivation of $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : !T.\mathsf{end}\rangle S$ depends on the size of $\vec{m}$. By induction on the size of $\vec{m}$ (IH_2), we derive $\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}\rangle : \mathsf{end}\rangle S$:

– If $\vec{m}$ is empty, it follows by inversion of typing that $\Gamma = \emptyset$ and $S = !T.\mathsf{end}$:

$$\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : !T.\mathsf{end}\rangle !T.\mathsf{end}}\ [\textsc{typ-buf}]$$

Then, we derive the following:

$$\dfrac{\Delta_1 \vdash_{\mathtt{M}} M : T \qquad \dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \mathsf{end}}\ [\textsc{typ-buf}]}{\Delta_1 \vdash_{\mathtt{B}} [M\rangle : \mathsf{end}\rangle !T.\mathsf{end}}\ [\textsc{typ-buf-send}]$$

– If $\vec{m} = \vec{m}', L$, it follows by inversion of typing that $\Gamma = \Gamma', \Gamma''$ and $S = !T'.\mathsf{end}'$:

$$\dfrac{\Gamma' \vdash_{\mathtt{M}} L : T' \qquad \Gamma'' \vdash_{\mathtt{B}} [\vec{m}'\rangle : !T.\mathsf{end}\rangle \mathsf{end}'}{\Gamma', \Gamma'' \vdash_{\mathtt{B}} [\vec{m}', L\rangle : !T.\mathsf{end}\rangle !T'.\mathsf{end}'}\ [\textsc{typ-buf-send}]$$

By IH_2, $\Gamma'', \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle \mathsf{end}'$, allowing us to derive the following:

$$\dfrac{\Gamma' \vdash_{\mathtt{M}} L : T' \qquad \Gamma'', \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle \mathsf{end}'}{\Gamma', \Gamma'', \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}', L\rangle : \mathsf{end}\rangle !T'.\mathsf{end}'}\ [\textsc{typ-buf-send}]$$

– If $\vec{m} = \vec{m}', j$, it follows by inversion of typing that there exist types S_i for each i in a set of labels I, where $j \in I$, such that $S = \oplus\{i : S_i\}_{i \in I}$:

$$\dfrac{\Gamma \vdash_{\mathtt{B}} [\vec{m}'\rangle : !T.\mathsf{end}\rangle S_j}{\Gamma \vdash_{\mathtt{B}} [\vec{m}, j\rangle : !T.\mathsf{end}\rangle \oplus\{i : S_i\}_{i \in I}}\ [\textsc{typ-buf-sel}]$$

By IH_2, $\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle S_j$, so we derive the following:

$$\dfrac{\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle S_j}{\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}', j\rangle : \mathsf{end}\rangle \oplus\{i : S_i\}_{i \in I}}\ [\textsc{typ-buf-sel}]$$

Now, we can derive the typing of the structurally congruent configuration (omitting "TYP-" from rule labels):

$$\dfrac{\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}\rangle : \mathsf{end}\rangle S \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{end} \vdash_{\mathtt{M}} x : \mathsf{end}}\ [\textsc{var}] \quad \Delta_2 \vdash_{\mathtt{M}} N : U}{\Delta_2, x : \mathsf{end} \vdash_{\mathtt{M}} \mathtt{close}\, x; N : U}\ [\textsc{close}]}{\Delta_2, x : \mathsf{end} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{close}\, x; N) : U}\ [\textsc{main}] \qquad \Lambda, y : \overline{S} \vdash^{\Diamond}_{\mathtt{C}} C : \mathbf{1}}{\Delta_2, \Lambda, x : \mathsf{end}, y : \overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{close}\, x; N) \parallel C : U}\ [\textsc{par-r}]}{\Gamma, \Delta_1, \Delta_2, \Lambda \vdash^{\blacklozenge}_{\mathtt{C}} (\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\blacklozenge\,(\mathtt{close}\, x; N) \parallel C) : U}\ [\textsc{res}]$$

□

5.2. **Faithfully Translating $\texttt{LAST}^n$ into $\texttt{APCP}$.** We now define a typed translation from $\texttt{LAST}^n$ into $\texttt{APCP}$. Then, we show that this translation is operationally complete and sound, in the sense of Section 4.3.

The Translation: Key Ideas. Before we define the translation, let us discuss its key idea, inspired by Milner's translation of the *lazy* λ-calculus [Mil92]. The most important design decision is how to translate variables. Variables serve two purposes: (i) as placeholders for future substitutions and (ii) as an access point to buffered channels. Accordingly, our translation uses variables as sends that enable (i) an explicit substitution or (ii) interactions with a buffer. Another important design decision is the translation of buffers: the buffer's types guide the translation to form a sequence of inputs/outputs that are explicitly forwarded (i.e., not using the forwarder process) between the buffer endpoints.

The Translation: Types. Equipped with these ideas about translating variables and buffers, let us give a deeper insight into our translation by taking a look at how it turns functional and session types in $\texttt{LAST}^n$ into session types in $\texttt{APCP}$.

Before moving on, note that $\texttt{LAST}^n$ does not guarantee deadlock freedom by typing. As such, there is no point in annotating types with priorities, as the priority requirements induced by typing translated processes may not always be satisfiable. We therefore use a special $\texttt{APCP}$ typing judgment

$$\vdash^* P :: \Gamma$$

which indicates well typedness of P modulo priority annotations and requirements in Γ (we omit Ω, as $\texttt{LAST}^n$ is a finite language). We will recover priorities and deadlock freedom in Section 5.3.

Definition 5.5 below introduces two forms of type translation: while we use $[\![-]\!]$ to translate the types of terms/configurations, we use $(\!|-|\!)$ to translate the types of variables. We write $(\!|\Gamma|\!)$ to denote a component-wise translation, where each assignment $x : T$ translates to an assignment $x : (\!|T|\!)$. As such, our *typed* translation takes, e.g., a typed term $\Gamma \vdash_{\texttt{M}} M : T$ and returns a typed process

$$\vdash^* [\![M]\!]z :: (\!|\Gamma|\!), z : [\![T]\!].$$

Above, $[\![M]\!]z$ is the process that models the behavior of M on a fresh name z (defined hereafter).

Let us now define and explain the two forms of translation of types (that are mutually recursive):

Definition 5.5 (Translation of Types)**.**

$$(\!|T|\!) \triangleq \bullet \otimes \overline{[\![T]\!]} \quad (\text{if } T \neq \Box)$$

$$[\![T \times U]\!] \triangleq \overline{(\!|T|\!)} \otimes \overline{(\!|U|\!)} \qquad [\![T \multimap U]\!] \triangleq (\!|T|\!) \parr [\![U]\!] \qquad [\![\mathbf{1}]\!] \triangleq \bullet$$

$$[\![!T.S]\!] \triangleq \bullet \otimes (\!|T|\!) \parr \overline{(\!|S|\!)} \qquad [\![\oplus\{i : S_i\}_{i\in I}]\!] \triangleq \bullet \otimes \&\{i : \overline{(\!|S_i|\!)}\}_{i\in I} \qquad [\![\mathsf{end}]\!] \triangleq \bullet \otimes \bullet$$

$$[\![?T.S]\!] \triangleq \overline{(\!|T|\!)} \otimes \overline{(\!|S|\!)} \qquad [\![\&\{i : S_i\}_{i\in I}]\!] \triangleq \oplus\{i : \overline{(\!|S_i|\!)}\}_{i\in I} \qquad [\![\Box]\!] \triangleq (\!|\Box|\!) \triangleq \bullet$$

Some intuitions follow:

- Following the intuitions above, the translation $(\!|-|\!)$ (row 1) codifies a variable as a send action that enables further behaviors.

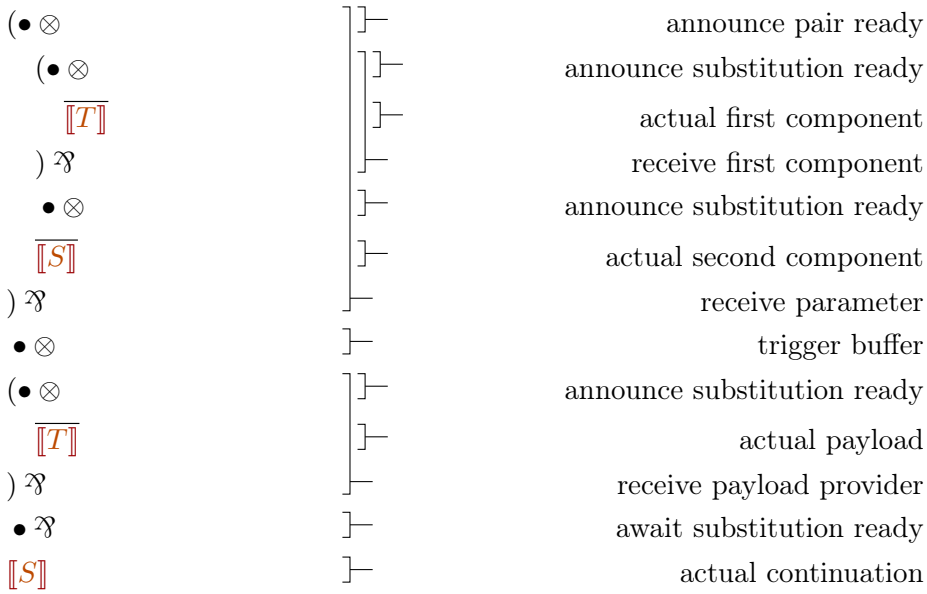


Figure 16: Example 5.6: the translation for $(T \times S) \multimap !T.S$, in detail.

- The translation $[\![-]\!]$ translates functional types straightforwardly (row 2): pairs send access to their components as variables, abstractions receive their parameter as a variable and then provide the return type, and units have no behavior.
- The translation of session types (rows 3 and 4) is more interesting: at a first glance they seem to be translated dually. This is because these are types that belong to buffered channel endpoints. As such, when a variable is typed $!T.S$ and we send a term on this variable, the translation sends something along the variable and thus the variable receives. Additionally, the translation of sends and selections have an extra send: this signifies a handshake between the variable and the buffer, indicating that both are ready to perform the actual send/selection.

Example 5.6. We illustrate the translation of types by means of an example. After giving the translation of the type $(T \times S) \multimap !T.S$, we break the resulting APCP type down and explain it in terms of the associated behavior of a process translated from a term implementing the type.

$$\begin{aligned}
[\![(T \times S) \multimap !T.S]\!] &= (\!|T \times S|\!) \parr [\![!T.S]\!] \\
&= (\bullet \otimes \overline{[\![T \times S]\!]}) \parr \bullet \otimes (\!|T|\!) \parr \overline{(\!|S|\!)} \\
&= (\bullet \otimes \overline{\overline{(\!|T|\!)} \otimes \overline{(\!|S|\!)}}) \parr \bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \overline{\bullet \otimes \overline{[\![S]\!]}} \\
&= (\bullet \otimes (\!|T|\!) \parr (\!|S|\!)) \parr \bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \bullet \parr [\![S]\!] \\
&= (\bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \bullet \otimes \overline{[\![S]\!]}) \parr \bullet \otimes (\bullet \otimes \overline{[\![T]\!]}) \parr \bullet \parr [\![S]\!]
\end{aligned}$$

Figure 16 gives a detailed explanation for this translated type.

The Translation: Terms, Configurations, and Buffers. Now we complete the definition of our translation. We proceed inductively on the typing derivations of terms, configurations, and buffers, obtaining typing derivations of APCP processes. Hence, the definition considers the typing rules in Figure 13, including the changes discussed in Section 5.1.

- The translation of a well-typed term, denoted $[\![\Gamma \vdash_{\mathtt{M}} M : T]\!]z$, corresponds to a judgment $\vdash^* P :: \Delta, z : A$ in APCP, where z is a fresh name that executes the behavior of M, as usual, and the shape of the process P, context Δ, and session type A will become precise shortly. Similarly, the translation of a well-typed configuration, denoted $[\![\Gamma \vdash^{\phi}_{\mathtt{C}} C : T]\!]z$, corresponds to the judgment $\vdash^* P :: \Delta, z : A$, for some P, Δ, and A.
- Also, $[\![\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S]\!]a\rangle b$, the translation of a well-typed buffer holding messages $\vec{m}$, corresponds to the judgment $\vdash^* P :: \Delta, a : A, b : B$, for some P, Δ, A, and B. The translated buffer stores the translations of the messages in $\vec{m}$ and forwards them on the fresh name b; the buffer receives new messages on the fresh name a. Note that, if the buffer is empty, the roles of a and b may be reversed (depending on S), and the buffer may receive new messages on b instead.

Figures 17 to 19 give the translation of term, configuration and buffer typing rules. In the figures (and in the rest of the paper), for the sake of readability, we often omit typing information: we write $[\![M]\!]z$ to refer to the typed process associated to $[\![\Gamma \vdash_{\mathtt{M}} M : T]\!]z$, and similarly for the translations of typed configurations and buffers. The omitted typing information/derivations is made precise by Section 5.2, given below.

Instead of explaining the translation separately, the figures include line-by-line explanations. Recall that we often write '_' to denote a fresh name of type $\bullet$; when sending names denoted '_', we omit binders '$(\boldsymbol{\nu}\_\_)$'.

Importantly, the translation is *type preserving* using the translations of types introduced in Definition 5.5. Appendix A.3 contains a detailed proof.

[Type Preservation for the Translation]

- If $\Gamma \vdash_{\mathtt{M}} M : T$, then $\vdash^* [\![M]\!]z :: (\!|\Gamma|\!), z : [\![T]\!]$;
- If $\Gamma \vdash^{\phi}_{\mathtt{C}} C : T$, then $\vdash^* [\![C]\!]z :: (\!|\Gamma|\!), z : [\![T]\!]$;
- If $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S$, then $\vdash^* [\![[\vec{m}\rangle]\!]a\rangle b :: (\!|\Gamma|\!), a : \overline{(\!|S'|\!)}, b : \overline{(\!|S|\!)}$.

[TYP-VAR]	$[\![x]\!]z \triangleq x[\_, z]$		
[TYP-ABS]	$[\![\lambda x.M]\!]z \triangleq z(x, a); [\![M]\!]a$		receive x, then run body
[TYP-APP]	$[\![M\ N]\!]z \triangleq (\boldsymbol{\nu} ab)(\boldsymbol{\nu} cd)([\![M]\!]a$		run abstraction
	$\mid b[c, z]$		trigger function body
	$\mid d(\_, e); [\![N]\!]e)$		parameter as future substitution
[TYP-UNIT]	$[\![()]\!]z \triangleq \mathbf{0}$		
[TYP-PAIR]	$[\![(M, N)]\!]z \triangleq (\boldsymbol{\nu} ab)(\boldsymbol{\nu} cd)(z[a, c]$		announce pair is ready
	$\mid b(\_, e); [\![M]\!]e \mid d(\_, f); [\![N]\!]f)$		components as future substitutions
[TYP-SPLIT]	$[\![\texttt{let}\ (x, y) = M\ \texttt{in}\ N]\!]z \triangleq (\boldsymbol{\nu} ab)(a(x, y); [\![N]\!]z$		block body until pair ready
	$\mid [\![M]\!]b)$		run pair
[TYP-NEW]	$[\![\texttt{new}]\!]z \triangleq (\boldsymbol{\nu} ab)(a[\_, z]$		activate buffer
	$\mid b(\_, c); (\boldsymbol{\nu} dx)(\boldsymbol{\nu} ey)($		block until activated
	$[\![\langle \epsilon \rangle]\!]d\rangle e$		prepare buffer
	$\mid [\![(x, y)]\!]c))$		return pair of endpoints
[TYP-FORK]	$[\![\texttt{fork}\ M; N]\!]z \triangleq (\boldsymbol{\nu} ab)(a[\_, z]$		activate bodies
	$\mid b(\_, c); ($		block until activated
	$(\boldsymbol{\nu} \_\_)[\![M]\!]\_ \mid [\![N]\!]c))$		run bodies
[TYP-CLOSE]	$[\![\texttt{close}\ M; N]\!]z \triangleq (\boldsymbol{\nu} ab)([\![M]\!]a$		run argument to activate buffer
	$\mid b(\_, \_); [\![N]\!]z)$		wait for buffer to close
[TYP-SEND]	$[\![\texttt{send}\ M\ N]\!]z \triangleq (\boldsymbol{\nu} ab)(\boldsymbol{\nu} cd)(a(\_, e); [\![M]\!]e$		block payload until received
	$\mid [\![N]\!]c$		run channel term to activate buffer
	$\mid d(\_, f); (\boldsymbol{\nu} gh)($		wait for buffer to activate
	$f[b, g]$		send to buffer
	$\mid h[\_, z]))$		prepare returned endpoint variable
[TYP-RECV]	$[\![\texttt{recv}\ M]\!]z \triangleq (\boldsymbol{\nu} ab)([\![M]\!]a$		run channel term to activate buffer
	$\mid b(c, d);$		receive from buffer
	$(\boldsymbol{\nu} ef)(z[c, e] \mid f(\_, g); d[\_, g]))$		returned pair
[TYP-SEL]	$[\![\texttt{select}\ j\ M]\!]z \triangleq (\boldsymbol{\nu} ab)([\![M]\!]a$		run channel term to activate buffer
	$\mid b(\_, c); (\boldsymbol{\nu} de)($		wait for buffer to activate
	$c[d] \triangleleft j$		select with buffer
	$\mid e[\_, z]))$		prepare returned endpoint variable

Figure 17: Translating $\texttt{LAST}^n$ into $\texttt{APCP}$, Part 1/3.

[TYP-CASE] $[\![\texttt{case}\ M\ \texttt{of}\ \{i : N_i\}_{i\in I}]\!]z \triangleq (\boldsymbol{\nu} ab)([\![M]\!]a$
run channel term to activate buffer
$\quad \mid b(c) \triangleright \{i :$ branch on buffer
$\quad [\![N_i\ c]\!]z\}_{i\in I})$
apply continuation to endpoint

[TYP-SUB] $[\![M\{\!\{N/x\}\!\}]\!]z \triangleq (\boldsymbol{\nu} xa)([\![M]\!]z$ run body
$\quad a(\_, b); [\![N]\!]b)$
block until body is variable

[TYP-MAIN] $[\![\blacklozenge\, M]\!]z \triangleq [\![M]\!]z$

[TYP-CHILD] $[\![\lozenge\, M]\!]z \triangleq [\![M]\!]z$

[TYP-PAR] $(T_1 = \mathbf{1})$ $[\![C \parallel D]\!]z \triangleq (\boldsymbol{\nu} \_\_)[\![C]\!]\_ \mid [\![D]\!]z$

[TYP-PAR] $(T_2 = \mathbf{1})$ $[\![C \parallel D]\!]z \triangleq [\![C]\!]z \mid (\boldsymbol{\nu} \_\_)[\![D]\!]\_$

[TYP-RES] $[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)C]\!]z \triangleq (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!]a\rangle b$ run buffer
$\quad \mid [\![C]\!]z)$ run configuration

[TYP-CONF-SUB] $[\![C\{\!\{N/x\}\!\}]\!]z \triangleq (\boldsymbol{\nu} xa)([\![C]\!]z$ run configuration
$\quad \mid a(\_, b); [\![N]\!]b$
block until configuration is variable

[TYP-BUF] $(S' = !T.S)$ $[\![[\epsilon\rangle]\!]a\rangle b \triangleq a(\_, c); (\boldsymbol{\nu} de)($
wait for activation
$\quad c[\_, d]$ activate send
$\quad \mid e(f, g); (\boldsymbol{\nu} hk)($
receive from send
$\quad b(\_, l);$ wait for activation
$\quad l[f, h]$ send to receive
$\quad \mid [\![[\epsilon\rangle : S\rangle S]\!]g\rangle k))$

[TYP-BUF] $(S' = \oplus\{i : S_i\}_{i\in I})$ $[\![[\epsilon\rangle]\!]a\rangle b \triangleq a(\_, c); (\boldsymbol{\nu} de)($
wait for activation
$\quad c[\_, d]$ activate select
$\quad \mid e(f) \triangleright \{i : (\boldsymbol{\nu} gh)($
receive selection
$\quad b(\_, k);$ wait for activation
$\quad k[g] \triangleleft i$ make selection
$\quad \mid [\![[\epsilon\rangle : S_i\rangle S_i]\!]f\rangle h)\})$

Figure 18: Translating $\texttt{LAST}^n$ into APCP, Part 2/3.

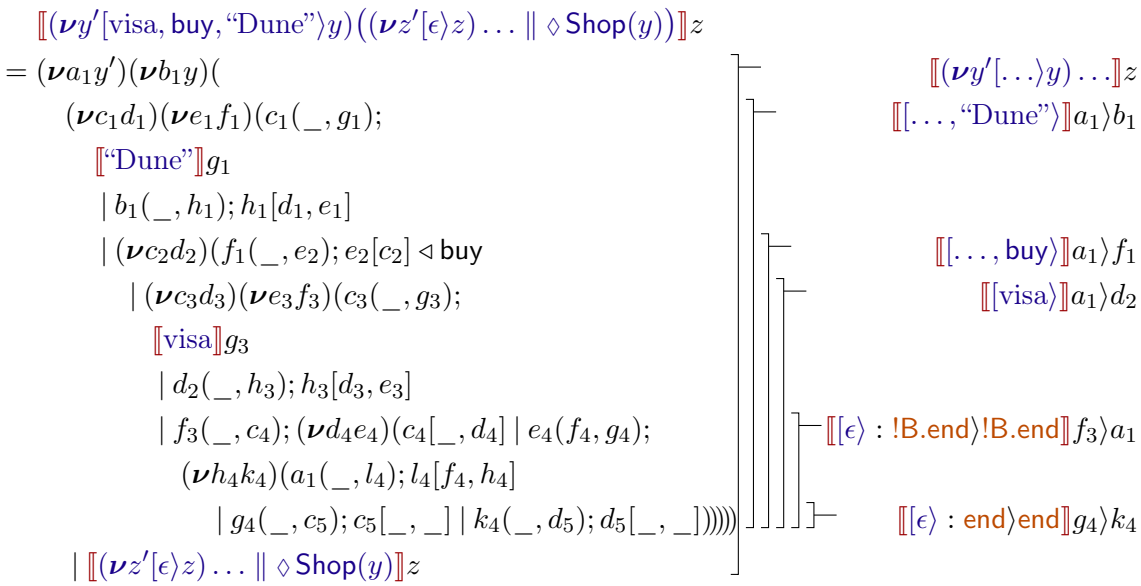
[TYP-BUF] $(S' \in \{?T.S, \&\{i : S_i\}_{i\in I}\})$	$[\![[\epsilon\rangle]\!]a\rangle b \triangleq [\![[\epsilon\rangle : \overline{S'}\rangle\overline{S'}]\!] b\rangle a$
[TYP-BUF] $(S' = \mathsf{end})$	$[\![[\epsilon\rangle]\!]a\rangle b \triangleq a(\_, c); c[\_, \_] \mid b(\_, d); d[\_, \_]$ close concurrently
[TYP-BUF] $(S' = \square)$	$[\![[\epsilon\rangle]\!]a\rangle b \triangleq \mathbf{0}$
[TYP-BUF-SEND]	$[\![[\vec{m}, M\rangle]\!]a\rangle b \triangleq (\boldsymbol{\nu} cd)(\boldsymbol{\nu} ef)(c(\_, g);$ block payload until received $\quad \mid b(\_, h);$ wait for activation $\quad h[d, e]$ send to receive $\quad \mid [\![[\vec{m}\rangle]\!]a\rangle f)$ run buffer
[TYP-BUF-SEL]	$[\![[\vec{m}, j\rangle]\!]a\rangle b \triangleq (\boldsymbol{\nu} cd)(b(\_, e);$ wait for activation $\quad e[c] \triangleleft j$ make selection $\quad \mid [\![[\vec{m}\rangle]\!]a\rangle d)$ run buffer
[TYP-BUF-END-L]	$[\![[\epsilon\rangle]\!]a\rangle b \triangleq a(\_, c); c[\_, \_]$
[TYP-BUF-END-R]	$[\![[\epsilon\rangle]\!]a\rangle b \triangleq b(\_, c); c[\_, \_]$

Figure 19: Translating $\texttt{LAST}^n$ into APCP, Part 3/3.

Example 5.7. We illustrate our translation by translating a subterm of the shop defined in Section 2.2 (leaving the translation of the syntactic sugar book(*title*) unspecified):

$$
\begin{aligned}
&[\![\lambda s_2.\, \texttt{let}\,(\mathit{card}, s_3) = \texttt{recv}\, s_2\, \texttt{in}\, \texttt{let}\, s_4 = \texttt{send}\,\mathrm{book}(\mathit{title})\, s_3\, \texttt{in}\, \texttt{close}\, s_4; ()]\!]z \\
&= z(s_2, a_0); && \vdash [\![\lambda s_2 \ldots]\!]z \\
&\quad (\boldsymbol{\nu} a_1 b_1)(a_1(\mathit{card}, s_3); && \vdash [\![\texttt{let}\,(\mathit{card}, s_3) = \ldots]\!]a_0 \\
&\quad\quad (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)(&& \vdash [\![\texttt{let}\, s_4 = \ldots]\!]a_0 \\
&\quad\quad\quad a_2(s_4, a_3); && \vdash [\![\lambda s_4 \ldots]\!]a_2 \\
&\quad\quad\quad\quad (\boldsymbol{\nu} a_4 b_4)(&& \vdash [\![\texttt{close}\ldots]\!]a_3 \\
&\quad\quad\quad\quad\quad s_4[\_, a_4] && \vdash [\![s_4]\!]a_4 \\
&\quad\quad\quad\quad\quad \mid b_4(\_, \_); \\
&\quad\quad\quad\quad\quad\quad \mathbf{0}) && \vdash [\![()]\!]a_3 \\
&\quad\quad\quad b_2[c_2, a_0] \mid d_2(\_, e_2); \\
&\quad\quad\quad\quad (\boldsymbol{\nu} a_5 b_5)(\boldsymbol{\nu} c_5 d_5)(a_5(\_, e_5); && \vdash [\![\texttt{send}\ldots]\!]e_2 \\
&\quad\quad\quad\quad\quad [\![\mathrm{book}(\mathit{title})]\!]e_5 \\
&\quad\quad\quad\quad\quad \mid s_3[\_, c_5] && \vdash [\![s_3]\!]c_5 \\
&\quad\quad\quad\quad\quad \mid d_5(\_, f_5); (\boldsymbol{\nu} g_5 h_5)(f_5[b_5, g_5] \mid h_5[\_, e_2]))) \\
&\quad\quad \mid (\boldsymbol{\nu} a_6 b_6)(&& \vdash [\![\texttt{recv}\ldots]\!]b_1 \\
&\quad\quad\quad s_2[\_, a_6] && \vdash [\![s_2]\!]a_6 \\
&\quad\quad\quad \mid b_6(c_6, d_6); (\boldsymbol{\nu} e_6 f_6)(b_1[c_6, e_6] \mid f_6(\_, g_6); d_6[\_, g_6])))
\end{aligned}
$$

We also give the translation of the outer buffer in the final state of the system in Example 5.3:



Design Decisions: Explicit Substitutions and Closing. Having presented our typed translation, we reflect on our design decision to enrich $\texttt{LAST}^n$ with explicit substitutions and closing.

As we will see next, adopting explicit substitutions leads to a direct operational correctness result. To see why, suppose we were to apply substitutions immediately. As an example, consider function application (Figure 17), which entails substituting a variable in the body of the function. The translation would need to encode the substitution of the translation of this variable. However, if the variable does not occur under an evaluation context, there is no way in APCP to perform such a substitution immediately. Hence, the translation would still need to encode such *implicit* substitutions in $\texttt{LAST}^n$ *explicitly* in APCP. This discrepancy would then have to be handled by means of the aforementioned substitution lifting in our operational correspondence results. Although this is a perfectly viable approach, we prefer to have a more direct operational correspondence, which entails more direct proofs that are not affected by an asymmetric treatment of substitutions.

Our choice for explicit closing of sessions is more pragmatic: it leads to a more compact translation. Suppose we were to treat closed sessions by silently weakening them. Consider, e.g., Rule [RED-SEND] (Figure 11): the `send` primitive is replaced by a variable pointing to the buffered channel, even if the session ends after the send. Hence, the translation would need a separate case for translating variables for closed sessions. Consequently, the translation would need similar such separate cases anywhere variables/closed sessions may occur. This would lead to a translation where the key ideas are unnecessarily obfuscated. We find it then preferable to treat closed sessions explicitly in favor of a more streamlined translation.

Faithfulness: Operational Correspondence. Following the discussion in Section 4.3, here we finally show that the translation presented above satisfies the operational correctness criterion (completeness and soundness) as proposed by Gorla [Gor10]. We first state both results, and then their proofs.

[Completeness] Given $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, if $C \to_{\mathsf{C}} D$, then $[\![C]\!]z \to^* [\![D]\!]z$.

[Soundness] Given $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, if $[\![C]\!]z \to^* Q$, then there exists D such that $C \to^*_{\mathsf{C}} D$ and $Q \to^* [\![D]\!]z$.

Both results rely on the following lemma, which decomposes translations of terms/configurations under contexts into some evaluation context containing the translation of the respective term/configuration, and transfers free variables/names and substitutions in and out of the translation (cf. Appendix A.4):

- $[\![\mathcal{R}[M]]\!]z = \mathcal{E}[[\![M]\!]z']$ for some $\mathcal{E}, z'$;
- $[\![\mathcal{F}[M]]\!]z = \mathcal{E}[[\![M]\!]z']$ for some $\mathcal{E}, z'$;
- $[\![\mathcal{G}[C]]\!]z = \mathcal{E}[[\![C]\!]z']$ for some $\mathcal{E}, z'$;
- $x \notin \mathrm{fv}(C)$ implies $x \notin \mathrm{fn}([\![C]\!]z)$;
- $x \in \mathrm{fv}(C)$ implies $[\![C\{y/x\}]\!]z = [\![C]\!]z\{y/x\}$.

Proof (Sketch). The first three items follow by induction on the structure of the contexts. In the base case, the context is a hole and the thesis follows immediately. The inductive cases follow by construction of the translation and IH. The last two items follow by induction on the structure of the configuration and construction of the translation. □

Completeness additionally relies on the following lemma, that decomposes the translation of buffers of several shapes into an evaluation context containing the translation of a continuation of the respective buffer:

- $S' \neq \Box$ implies $[\![\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : S'\rangle S]\!]a\rangle b = \mathcal{E}\big[[\![\emptyset \vdash_{\mathsf{B}} [\epsilon\rangle : S'\rangle S']\!]a\rangle c\big]$ for some $\mathcal{E}, c$;
- $S' \neq \Box$ implies $[\![\Gamma, \Delta \vdash_{\mathsf{B}} [\mathbb{M}, \vec{m}\rangle : S'\rangle S]\!]a\rangle b = \mathcal{E}\big[[\![\Delta \vdash_{\mathsf{B}} [\mathbb{M}\rangle : S'\rangle !T.S']\!]a\rangle c\big]$ for some $\mathcal{E}, c$;
- $S' \neq \Box$ implies $[\![\Gamma \vdash_{\mathsf{B}} [j, \vec{m}\rangle : S'\rangle S]\!]a\rangle b = \mathcal{E}\big[[\![\emptyset \vdash_{\mathsf{B}} [j\rangle : S'\rangle \oplus\{i : S_i\}_{i\in I} \cup \{j : S'\}]\!]a\rangle c\big]$ for some $\mathcal{E}, c$;
- $S \neq \Box$ implies $[\![\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : \mathsf{end}\rangle S]\!]a\rangle b = \mathcal{E}\big[[\![\emptyset \vdash_{\mathsf{B}} [\epsilon\rangle : \mathsf{end}\rangle \mathsf{end}]\!]a\rangle c\big]$ for some $\mathcal{E}, c$;
- $S \neq \Box$ implies $[\![\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : \Box\rangle S]\!]a\rangle b = \mathcal{E}\big[[\![\emptyset \vdash_{\mathsf{B}} [\epsilon\rangle : \Box\rangle \mathsf{end}]\!]a\rangle c\big]$ for some $\mathcal{E}, c$.

Proof (Sketch). Each item follows by induction on the size of $\vec{m}$, and a case analysis on the shape of S. In the base case, $\vec{m} = \epsilon$ and the thesis follows immediately. In the inductive cases, the thesis follows by IH and the construction of the translation. □

With these prerequisites in place, we move on to discuss the proofs of Section 5.2.

Proof of completeness (Section 5.2). By induction on the derivation of $C \to_{\mathsf{C}} D$. We discuss one representative rule for message passing, as well as the structural rules; other rules are detailed in Appendix A.4.1.

- Rule [RED-SEND]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[\mathsf{send}\, M\, x] \parallel C) \to_{\mathsf{C}} (\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\hat{\mathcal{F}}[x] \parallel C)$. W.l.o.g., assume C is a child thread. By Section 5.2, for any L, $[\![\mathcal{F}[L]]\!]z = \mathcal{E}_1\big[[\![L]\!]z'\big]$ for some $\mathcal{E}_1, z'$ $(*_1)$. Moreover, since $\hat{\mathcal{F}}$ does not have its hole under an explicit substitution, it does not capture any free variables of M; hence, by Section 5.2, $\mathcal{E}_1$ does not capture any free names of $[\![M]\!]u$ for any u $(*_2)$. By inversion of typing, $\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : S_1\rangle S$ where

$S_1 = !T.S_2$ $(*_3)$. By Section 5.2, $[\![\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S_1\rangle S]\!]a\rangle b = \mathcal{E}_2[\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S_1\rangle S_1]\!]a\rangle c]$ $(*_4)$ and $[\![\Gamma, \Delta \vdash_{\mathtt{B}} [M, \vec{m}\rangle : S_1\rangle S]\!]a\rangle b = \mathcal{E}_2[\![\Delta \vdash_{\mathtt{B}} [M\rangle : S_1\rangle S_1]\!]a\rangle c]$ $(*_5)$ for some $\mathcal{E}_2, c$. Below, we omit types from translations of buffers. The thesis holds as follows:

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[\mathtt{send}\, M\, x] \parallel C)]\!]z \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!]a\rangle b \mid [\![\hat{\mathcal{F}}[\mathtt{send}\, M\, x]]\!]z \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\mathcal{E}_2[[\![[\epsilon\rangle]\!]a\rangle c] \mid \mathcal{E}_1[[\![\mathtt{send}\, M\, x]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_1,*_4) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[a(\_, c_1); (\boldsymbol{\nu} d_1 e_1)(c_1[\_, d_1] \mid e_1(f_1, g_1); (\boldsymbol{\nu} h_1 k_1)(c(\_, l_1); l_1[f_1, h_1] \mid [\![[\epsilon\rangle]\!]g_1\rangle k_1))] \\
&\quad \mid \mathcal{E}_1[(\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)(a_2(\_, e_2); [\![M]\!]e_2 \mid x[\_, c_2] \mid d_2(\_, f_2); (\boldsymbol{\nu} g_2 h_2)(f_2[b_2, g_2] \mid h_2[\_, z']))] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_3) \\
&\to (\boldsymbol{\nu} c_2 d_2)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[(\boldsymbol{\nu} d_1 e_1)(c_2[\_, d_1] \mid e_1(f_1, g_1); (\boldsymbol{\nu} h_1 k_1)(c(\_, l_1); l_1[f_1, h_1] \mid [\![[\epsilon\rangle]\!]g_1\rangle k_1))] \\
&\quad \mid \mathcal{E}_1[(\boldsymbol{\nu} a_2 b_2)(a_2(\_, e_2); [\![M]\!]e_2 \mid d_2(\_, f_2); (\boldsymbol{\nu} g_2 h_2)(f_2[b_2, g_2] \mid h_2[\_, z']))] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\to (\boldsymbol{\nu} e_1 d_1)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[e_1(f_1, g_1); (\boldsymbol{\nu} h_1 k_1)(c(\_, l_1); l_1[f_1, h_1] \mid [\![[\epsilon\rangle]\!]g_1\rangle k_1)] \\
&\quad \mid \mathcal{E}_1[(\boldsymbol{\nu} a_2 b_2)(a_2(\_, e_2); [\![M]\!]e_2 \mid (\boldsymbol{\nu} g_2 h_2)(d_1[b_2, g_2] \mid h_2[\_, z']))] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\to (\boldsymbol{\nu} b_2 a_2)(\boldsymbol{\nu} g_2 h_2)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[(\boldsymbol{\nu} h_1 k_1)(c(\_, l_1); l_1[b_2, h_1] \mid [\![[\epsilon\rangle]\!]g_2\rangle k_1)] \\
&\quad \mid \mathcal{E}_1[a_2(\_, e_2); [\![M]\!]e_2 \mid h_2[\_, z']] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\equiv (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(&& (*_2) \\
&\quad \mathcal{E}_2[(\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} h_1 k_1)(a_2(\_, e_2); [\![M]\!]e_2 \mid c(\_, l_1); l_1[b_2, h_1] \mid [\![[\epsilon\rangle]\!]a\rangle k_1)] \\
&\quad \mid \mathcal{E}_1[x[\_, z']] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\mathcal{E}_2[[\![[M\rangle]\!]a\rangle c] \mid \mathcal{E}_1[[\![x]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[M, \vec{m}\rangle]\!]a\rangle b \mid [\![\hat{\mathcal{F}}[x]]\!]z \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_1,*_5) \\
&= [\![(\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\hat{\mathcal{F}}[x] \parallel C)]\!]z
\end{aligned}
$$

- Rule [RED-LIFT-C]: $C \to_{\mathtt{C}} C'$ implies $\mathcal{G}[C] \to_{\mathtt{C}} \mathcal{G}[C']$. This case follows from Section 5.2 and the IH.
- Rule [RED-LIFT-M]: $M \to_{\mathtt{M}} M'$ implies $\mathcal{F}[M] \to_{\mathtt{M}} \mathcal{F}[M']$. By Section 5.2, it suffices to show completeness on the level of terms. Hence, we apply induction on the derivation of $M \to_{\mathtt{M}} M'$ (IH_2). In Appendix A.4.1, this property is proven separately as Appendix A.4.1. Here, we discuss one representative reduction for computation, as well as the structural rules:

– Rule [RED-LAM]: $(\lambda x.M)\ N \to_{\mathtt{M}} M\{\!\{N/x\}\!\}$. The thesis holds as follows:

$$\begin{aligned}[\![(\lambda x.M)\ N]\!]z &= (\boldsymbol{\nu} a_1 b_1)(\boldsymbol{\nu} c_1 d_1)(a_1(x,a_2); [\![M]\!]a_2 \mid b_1[c_1,z] \mid d_1(\_,e_1); [\![N]\!]e_1)\\ &\to (\boldsymbol{\nu} c_1 d_1)([\![M]\!]z\{c_1/x\} \mid d_1(\_,e_1); [\![N]\!]e_1)\\ &\equiv (\boldsymbol{\nu} x d_1)([\![M]\!]z \mid d_1(\_,e_1); [\![N]\!]e_1)\\ &= [\![M\{\!\{N/x\}\!\}]\!]z\end{aligned}$$

– Rule [RED-LIFT]: $M \to_{\mathtt{M}} N$ implies $\mathcal{R}[M] \to_{\mathtt{M}} \mathcal{R}[N]$. This case follows from Section 5.2 and IH_2.

– Rule [RED-LIFT-SC]: $M \equiv_{\mathtt{M}} M'$, $M' \to_{\mathtt{M}} N'$, and $N' \equiv_{\mathtt{M}} N$ imply $M \to_{\mathtt{M}} N$. By IH_2, it suffices to show that the translation preserves structural congruence of terms as structural congruence of processes. Hence, we apply induction on the derivation of $M \equiv_{\mathtt{M}} M'$ (and similarly for $N' \equiv_{\mathtt{M}} N$; IH_3). In Appendix A.4.1, this property is proven separately as Appendix A.4.1. The inductive cases follow from IH_3 and Section 5.2 directly. We detail the (only) base case of Rule [SC-SUB-EXT]: $x \notin \mathrm{fn}(\mathcal{R})$ implies $(\mathcal{R}[M])\{\!\{N/x\}\!\} \equiv_{\mathtt{M}} \mathcal{R}[M\{\!\{N/x\}\!\}]$.
The analysis is by induction on the structure of $\mathcal{R}$ (IH_4), assuming $x \notin \mathrm{fn}(\mathcal{R})$. The base case where $\mathcal{R} = [\cdot]$ is immediate. We detail one representative inductive case: $\mathcal{R} = \mathcal{R}'\ M'$. The thesis holds as follows:

$$\begin{aligned}&[\![(\mathcal{R}'[M]\ M')\{\!\{N/x\}\!\}]\!]z\\ &= (\boldsymbol{\nu} x a_1)\big((\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)([\![\mathcal{R}'[M]]\!]a_2 \mid b_2[c_2,z] \mid d_2(\_,e_2); [\![M']\!]e_2) \mid a_1(\_,b_1); [\![N]\!]b_1\big)\\ &\equiv (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)\big((\boldsymbol{\nu} x a_1)([\![\mathcal{R}'[M]]\!]a_2 \mid a_1(\_,b_1); [\![N]\!]b_1) \mid b_2[c_2,z] \mid d_2(\_,e_2); [\![M']\!]e_2\big)\\ &= (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)\big([\![(\mathcal{R}'[M])\{\!\{N/x\}\!\}]\!]a_2 \mid b_2[c_2,z] \mid d_2(\_,e_2); [\![M']\!]e_2\big)\\ &\equiv (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)\big([\![\mathcal{R}'[M\{\!\{N/x\}\!\}]]\!]a_2 \mid b_2[c_2,z] \mid d_2(\_,e_2); [\![M']\!]e_2\big) \qquad (\mathrm{IH}_4)\\ &= [\![(\mathcal{R}'[M\{\!\{N/x\}\!\}])\ M']\!]z = [\![\mathcal{R}[M\{\!\{N/x\}\!\}]]\!]z \qquad 3\end{aligned}$$

- Rule [RED-CONF-LIFT-SC]: $C \equiv_{\mathtt{C}} C'$, $C' \to_{\mathtt{C}} D'$, and $D' \equiv_{\mathtt{C}} D$ imply $C \to_{\mathtt{C}} D$. By the IH, it suffices to show that the translation preserves structural congruence of configurations as structural congruence of processes. Hence, we apply induction on the derivation of $C \equiv_{\mathtt{C}} C'$ (and similarly for $D' \equiv_{\mathtt{C}} D$; IH_2). In Appendix A.4.1, this property is proven separately as Appendix A.4.1. The inductive cases follow from IH_2 and Section 5.2 straightforwardly. We detail the interesting base case of Rule [SC-RES-SWAP]: $(\boldsymbol{\nu} x[\epsilon\rangle y)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C$. Both directions are analogous; we detail the left to right direction. We first infer the typing of the left configuration:

$$\frac{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S \qquad \Gamma \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma' \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)C : T}\ [\text{TYP-RES/-BUF}]$$

Here, $\Gamma' = \Gamma \setminus x : S', y : \overline{S}$. The analysis depends on whether $x = \square$ and/or $y = \square$. In each case, we show that

$$[\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S]\!]a\rangle b = [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S}\rangle \overline{S'}]\!]b\rangle a : \qquad (5.1)$$

– Case $x = y = \square$, or $x \neq \square$ and $x \neq \square$. Either way, $S' = S$. If $S' = \square$, both translations are $\mathbf{0}$, from which the thesis follows immediately. Otherwise, the thesis follows by induction on the structure of S'; clearly, the resulting translations are exactly the same.

– Case $x = \square$ and $y \neq \square$, or $x \neq \square$ and $y = \square$. W.l.o.g., assume the former. Then $S' = \square$ and $S = \mathsf{end}$. The thesis then holds as follows:

$$\begin{aligned}[\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \square\rangle\mathsf{end}]\!]a\rangle b &= b(\_, c); c[\_, \_] \\ &= [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle\square]\!]b\rangle c\end{aligned}$$

The thesis then holds as follows:

$$\begin{aligned}[\![(\boldsymbol{\nu} x[\epsilon\rangle y)C]\!]z &= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S]\!]a\rangle b \mid [\![C]\!]z) \\ &= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S}\rangle\overline{S'}]\!]b\rangle a \mid [\![C]\!]z) \qquad (5.1) \\ &\equiv (\boldsymbol{\nu} by)(\boldsymbol{\nu} ax)([\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S}\rangle\overline{S'}]\!]b\rangle a \mid [\![C]\!]z) \\ &= [\![(\boldsymbol{\nu} y[\epsilon\rangle x)C]\!]z \qquad \square\end{aligned}$$

Proof of soundness (Section 5.2). By induction on the number k of steps $[\![C]\!]z \to^k Q$ (IH_1). We distinguish cases on all possible initial reductions $[\![C]\!]z \to Q_0$ and discuss all possible following reductions. Here, we rely on APCP's confluence of independent reductions, allowing us to focus on a specific sequence of reductions, postponing other possibilities that eventually lead to the same result.

We then use induction on the structure of C (IH_2). The goal is to identify some D_0 such that we can isolate $k_0 \geq 0$ reductions such that $C \to_{\mathtt{C}} D_0$ and $[\![C]\!]z \to Q_0 \to^{k_0} [\![D_0]\!]z$ (where k_0 may be different in each case). We then have $[\![D_0]\!]z \to^{k-k_0} Q$, so it follows from IH_1 that there exists D such that $D_0 \to^*_{\mathtt{C}} D$ and $[\![D_0]\!]z \to^* [\![D]\!]z$. Here, we detail only the interesting case of an interaction between the translations of a buffer and a contained configuration; Appendix A.4.2 details the whole proof. That is, $C = (\boldsymbol{\nu} x[\vec{m}\rangle y)C_1$. We have

$$[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)C_1]\!]z = (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)([\![[\vec{m}\rangle]\!]a_1\rangle b_1 \mid [\![C_1]\!]z).$$

The reduction may originate from (i) $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$, (ii) $[\![C_1]\!]z$, (iii) a synchronization between a_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and x in $[\![C_1]\!]z$, or (iv) a synchronization between b_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and y in $[\![C_1]\!]z$. We detail each case:

(i) The reduction originates from $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$. No matter what $[\vec{m}\rangle$ is, no reduction is possible.

(ii) The reduction originates from $[\![C_1]\!]z$. We thus have $[\![C_1]\!]z \to Q_1$. By IH_2, there are $D_1, k_1 \geq 0$ such that $C_1 \to_{\mathtt{C}} D_1$ and $[\![C_1]\!]z \to Q_1 \to^{k_1} [\![D_1]\!]z$. Let $D_0 \triangleq (\boldsymbol{\nu} x[\vec{m}\rangle y)D_1$. We have $C \to_{\mathtt{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)([\![[\vec{m}\rangle]\!]a_1\rangle b_1 \mid [\![C_1]\!]z) \\ &\to^{k_1+1} (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)([\![[\vec{m}\rangle]\!]a_1\rangle b_1 \mid [\![D_1]\!]z) \\ &= [\![D_0]\!]z\end{aligned}$$

(iii) The reduction originates from a synchronization between a_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and x in $[\![C_1]\!]z$. By well typedness, $\Delta \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S$. Note first that, by Section 5.2, $S' \neq \square$ implies that there are $\mathcal{E}_2, c_1$ such that $[\![[\vec{m}\rangle : S'\rangle S]\!]a_1\rangle b_1 = \mathcal{E}_2[\![[\epsilon\rangle : S'\rangle S']\!]a_1\rangle c_1]$. The analysis depends on S', so we consider all possibilities. In each case, if the reduction is indeed possible, we show that the reduction is the first step in the execution of some rule such that $C \to_{\mathtt{C}} D_0$. The corresponding reduction $[\![C]\!]z \to Q_0 \to^{k_0} [\![D_0]\!]z$ follows the corresponding case in the proof of Section 5.2 (Faithfulness: Operational Correspondence).

- Case $S' = \square$. Then $x = \square$ is not free in C_1, and thus x is not free in $[\![C_1]\!]z$: the reduction is not possible.
- Case $S' = \mathsf{end}$. The analysis depends on whether $S = \square$ or not; w.l.o.g., assume not. We have

$$[\![[\epsilon] : \mathsf{end} \rangle \mathsf{end}]\!] a_1 \rangle c_1 = a_1(\_, c_2); \ldots \mid \ldots$$

  Thus, the synchronization is between the receive on a_1 and a send on x in $[\![C_1]\!]z$. A send on a variable x can only occur in the translation of that variable directly, under some reduction context. Since x is of type end and its translation appears under a reduction context, the only well-typed way for x to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}[\mathtt{close}\, x; M_1]\big]$. We then have

$$C \equiv_{\mathsf{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{close}\, x; M_1] \mid C_2)\big].$$

  Hence, the observed reduction is the first step of executing Rule [RED-CLOSE].
- Case $S' = !T_2.S_2'$. We have

$$[\![[\epsilon] : !T_2.S_2' \rangle !T_2.S_2']\!] a_1 \rangle c_1 = a_1(\_, c_2); \ldots$$

  Thus, the synchronization is between the receive on a_1 and a send on x in $[\![C_1]\!]z$. A send on a variable x can only occur in the translation of that variable directly, under some reduction context. Since x is of type $!T_2.S_2'$ and its translation appears under a reduction context, the only well-typed way for x to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}[\mathtt{send}\, M_1\, x]\big]$. We then have

$$C \equiv_{\mathsf{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}'[\mathtt{send}\, M_1\, x] \mid C_2)\big].$$

  Hence, the observed reduction is the first step of executing Rule [RED-SEND].
- Case $S' = ?T_2.S_2'$. We have

$$[\![[\epsilon] : ?T_2.S_2' \rangle ?T_2.S_2']\!] a_1 \rangle c_1 = \Big[\!\Big[[\epsilon] : !T_2.\overline{S_2'} \rangle !T_2.\overline{S_2'}\Big]\!\Big] c_1 \rangle a_1 = c_1(\_, c_2); \ldots$$

  Thus, the reduction is not possible.
- Case $S' = \oplus\{i : S_2^i\}_{i \in I}$. We have

$$[\![[\epsilon] : \oplus\{i : S_2^i\}_{i \in I} \rangle \oplus\{i : S_2^i\}_{i \in I}]\!] a_1 \rangle c_1 = a_1(\_, c_2); \ldots$$

  Thus, the synchronization is between the receive on a_1 and a send on x in $[\![C_1]\!]z$. A send on a variable x can only occur in the translation of that variable directly, under some reduction context. Since x is of type $\oplus\{i : S_2^i\}_{i \in I}$ and its translation appears under a reduction context, the only well-typed way for x to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}[\mathtt{select}\, j\, x]\big]$ where $j \in I$. We then have

$$C \equiv_{\mathsf{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{select}\, j\, x] \mid C_2)\big].$$

  Hence, the observed reduction is the first step of executing Rule [RED-SELECT].
- Case $S' = \&\{i : S_2^i\}_{i \in I}$. We have

$$\begin{aligned}[\![[\epsilon] : \&\{i : S_2^i\}_{i \in I} \rangle \&\{i : S_2^i\}_{i \in I}]\!] a_1 \rangle c_1 &= \Big[\!\Big[[\epsilon] : \oplus\{i : \overline{S_2^i}\}_{i \in I} \rangle \oplus\{i : \overline{S_2^i}\}_{i \in I}\Big]\!\Big] c_1 \rangle a_1 \\ &= c_1(\_, c_2); \ldots\end{aligned}$$

  Thus, the reduction is not possible.

(iv) The reduction originates from a synchronization between b_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and y in $[\![C_1]\!]z$. By well typedness, $\Delta \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S$. The analysis depends on S, so we consider all possibilities. In each case, if the reduction is indeed possible, we show that the reduction is the first step in the execution of some rule such that $C \to_{\mathtt{C}} D_0$. The corresponding reduction $[\![C]\!]z \to Q_0 \to^{k_0} [\![D_0]\!]z$ follows the corresponding case in the proof of Section 5.2 (Faithfulness: Operational Correspondence).

- Case $S = \Box$. Then $y = \Box$ is not free in C_1, and thus y is not free in $[\![C_1]\!]z$: the reduction is not possible.
- Case $S = \mathsf{end}$. By well typedness, then $\vec{m} = \epsilon$. Let $C' \triangleq (\boldsymbol{\nu} y[\epsilon\rangle x)C_1$; we have $C \equiv_{\mathtt{C}} C'$ and $[\![C]\!]z \equiv [\![C']\!]z$ (by Appendix A.4.1). The thesis then follows as in the analogous case under Subcase (iii) above.
- Case $S = !T_2.S_2$. By well typedness, then $\vec{m} = \vec{m}', M_1$. We have

$$[\![[\vec{m}', M_1\rangle : S'\rangle !T_2.S_2]\!]a_1\rangle b_1 = (\boldsymbol{\nu}\ldots)(\boldsymbol{\nu}\ldots)(\ldots \mid b_1(\_, h_2); \ldots \mid \ldots).$$

  Thus, the synchronization is between the receive on b_1 and a send on y in $[\![C_1]\!]z$. A send on a variable y can only occur in the translation of that variable directly, under some reduction context. Since y is of type $\overline{S} = ?T_2.\overline{S_2}$ and its translation appears under a reduction context, the only well-typed way for y to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}[\mathtt{recv}\, y]\big]$. We then have

$$C \equiv_{\mathtt{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}', M_1\rangle y)(\hat{\mathcal{F}}'[\mathtt{recv}\, y] \mid C_2)\big].$$

  Hence, the observed reduction is the first step of executing Rule [RED-RECV].
- Case $S = ?T_2.S_2$. By well typedness, then $\vec{m} = \epsilon$; this case is analogous to Case $S = \mathsf{end}$ above.
- Case $S = \oplus\{i : S_2^i\}_{i\in I}$. By well typedness, then $\vec{m} = \vec{m}', j$ where $j \in I$. We have

$$[\![[\vec{m}', j\rangle : S'\rangle \oplus\{i : S_2^i\}_{i\in I}]\!]a_1\rangle b_1 = (\boldsymbol{\nu}\ldots)(b_1(\_, e_2); \ldots \mid \ldots).$$

  Thus, the synchronization is between the receive on b_1 and a send on y in $[\![C_1]\!]z$. A send on a variable y can only occur in the translation of that variable directly, under some reduction context. Since y is of type $\overline{S} = \&\{i : \overline{S_2^i}\}_{i\in I}$ and its translation appears under a reduction context, the only well-typed way for y to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_{1.i}\}_{i\in I}]\big]$. We then have

$$C \equiv_{\mathtt{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}', j\rangle y)(\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_{1.i}\}_{i\in I}] \mid C_2)\big].$$

  Hence, the observed reduction is the first step of executing Rule [RED-CASE].
- Case $S = \&\{i : S_2^i\}_{i\in I}$. By well typedness, then $\vec{m} = \epsilon$; this case is analogous to Case $S = \mathsf{end}$ above. □

5.3. **Deadlock Free $\mathtt{LAST}^n$.** By virtue of Section 3.3, well-typed APCP processes that are typable under empty contexts ($\vdash P :: \emptyset$) are deadlock free. We may transfer this result to $\mathtt{LAST}^n$ configurations by appealing to the operational correctness of our translation (Section 5.2 above). Each deadlock-free configuration in $\mathtt{LAST}^n$ obtained via this transference satisfies two requirements:

- The configuration is typable $\emptyset \vdash^{\blacklozenge}_{\mathtt{C}} C : \mathbf{1}$, i.e., it needs no external resources and has no external behavior.
- The typed translation of the configuration satisfies priority requirements in APCP: it is well typed under '$\vdash$', not only under '$\vdash^*$' (cf. Section 5.2).

We rely on soundness (Section 5.2) to transfer deadlock freedom to configurations:

Theorem 5.8 (Deadlock Freedom for $\texttt{LAST}^n$)**.** *Given* $\emptyset \vdash^{\blacklozenge}_{\mathsf{C}} C : \mathbf{1}$*, if* $\vdash [\![C]\!]z :: \Gamma$ *for some* Γ*, then* $C \equiv \blacklozenge\,()$ *or* $C \to_{\mathsf{C}} D$ *for some* D*.*

Proof. By Section 5.2 (type preservation for the translation), $\Gamma = z : [\![\mathbf{1}]\!] = z : \bullet$. Then $\vdash (\boldsymbol{\nu} z\_)[\![C]\!]z :: \emptyset$. Hence, by Section 3.3 (deadlock freedom), either $(\boldsymbol{\nu} z\_)[\![C]\!]z \equiv \mathbf{0}$ or $(\boldsymbol{\nu} z\_)[\![C]\!]z \to Q$ for some Q. The rest of the analysis depends on which possibility holds:

- We have $(\boldsymbol{\nu} z\_)[\![C]\!]z \equiv \mathbf{0}$. We straightforwardly deduce from the well typedness and translation of C that $C \equiv_{\mathsf{C}} \blacklozenge\,()$, proving the thesis.
- We have $(\boldsymbol{\nu} z\_)[\![C]\!]z \to Q$ for some Q. We argue that this implies that $[\![C]\!]z \to Q_0$, for some Q_0.
  - The reduction involves z. Since z is of type $\bullet$ in $[\![C]\!]z$, then z must occur in a forwarder, and the reduction involves a forwarder $[z \leftrightarrow x]$ for some x. Since x is not free in $[\![C]\!]z$, there must be a restriction on x. Hence, the forwarder $[z \leftrightarrow x]$ can also reduce with this restriction, instead of with the restriction $(\boldsymbol{\nu} z\_)$. This means that $[\![C]\!]z \to Q_0$ for some Q_0.
  - The reduction does not involve z, in which case the reduction must be internal to $[\![C]\!]z$. That is, $[\![C]\!]z \to Q_0$ for some Q_0 and $Q \equiv (\boldsymbol{\nu} z\_)Q_0$.

  First, by contradiction, we show that z is not involved in this reduction: Since z is of type $\bullet$ in $[\![C]\!]z$, then z must occur in a forwarder, and the reduction involves a forwarder $[z \leftrightarrow x]$ for some x. However, our translation (Figures 17 to 19) does not generate forwarders, so this is clearly a contradiction. Hence, the reduction does not involve z.

  Thus, the reduction must be internal to $[\![C]\!]z$. That is, $[\![C]\!]z \to Q_0$ for some Q_0 and $Q \equiv (\boldsymbol{\nu} z\_)Q_0$. Then, by Section 5.2 (soundness), there exists D such that $C \to^*_{\mathsf{C}} D$. Looking at the proof of Section 5.2 (in Appendix A.4.2), it is easy to see that in fact $C \to^+_{\mathsf{C}} D$. That is, there exists D_0 such that $C \to_{\mathsf{C}} D_0$, proving the thesis. □

Example 5.9. To illustrate the insights gained from deadlock analysis in APCP through translation from $\texttt{LAST}^n$, we consider a seemingly deadlock-free configuration that is not. To this end, we first discuss how we can assign priorities to the translated types of the configuration, and extract from the translation requirements on those priorities. Next, we show that our example leads to unsatisfiable priority requirements, indicating the elusive deadlock in the original configuration. We close the example by considering an alternative configuration that is indeed deadlock free.

Our example is a classic showcase of deadlock due to cyclic connections under synchronous communication. Though one would expect this example to be deadlock free under asynchronous communication, it is not: there is a deadlock induced by the call-by-name semantics of $\texttt{LAST}^n$. Let

$$M(x,y) \triangleq \begin{array}{l}\texttt{let}\, x_1 = \texttt{send}\,()\,x\,\texttt{in}\\ \texttt{let}\,(v, y_1) = \texttt{recv}\,y\,\texttt{in}\\ \texttt{close}\,x_1; \texttt{close}\,y_1; v,\end{array} \qquad C \triangleq \blacklozenge\,\begin{array}{l}\texttt{let}\,(x,x') = \texttt{new}\,\texttt{in}\\ \texttt{let}\,(y,y') = \texttt{new}\,\texttt{in}\\ \texttt{fork}\,M(x,y); M(y',x').\end{array}$$

Before showing the deadlock and its source, we first analyze the deadlock through translation into APCP.

By Section 5.2 and assigning priority variables to each connective, we have

$$\vdash^* [\![M(x,y)]\!]z_1 :: y : \bullet \otimes^{\circ_1} (\bullet \otimes^{\circ_2} \bullet) \parr^{\circ_3} \bullet \otimes^{\circ_4} \bullet \parr^{\circ_5} \bullet,\ x : \bullet \otimes^{\circ_6} \bullet \parr^{\circ_7} (\bullet \parr^{\circ_8} \bullet) \otimes^{\circ_9} \bullet \otimes^{\circ_{10}} \bullet \parr^{\circ_{11}} \bullet, z_1 : \bullet$$

$$\vdash^* [\![M(y',x')]\!]z_2 :: x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet,\ y' : \bullet \otimes^{\pi_6} \bullet \parr^{\pi_7} (\bullet \parr^{\pi_8} \bullet) \otimes^{\pi_9} \bullet \otimes^{\pi_{10}} \bullet \parr^{\pi_{11}} \bullet, z_2 : \bullet$$

$$\vdash^* [\![C]\!]z :: z : \bullet$$

To determine requirements on these priorities, we analyze the translations of typing rules into APCP derivations in Figures 17 to 19 and the priority requirements induced by them. Indeed, a deadlock is detected, because APCP requires $\circ_6 < \pi_1 < \pi_6 < \circ_1 < \circ_6$, which is unsatisfiable.

Before detailing the origin of these requirements, we first give an intuition behind this unsatisfiable chain of priority requirements, and how they reflect the deadlock that is indeed present in C. From left to right, the requirements denote that (1) the send on x occurs before the receive on x', (2) the receive on x' occurs before the send on y', (3) the send on y' occurs before the receive on y, (4) the receive on y occurs before the send on x. Steps (2) and (3) stand out: $M(x,y)$ and $M(y',x')$ seem to define an opposite order. This is because in, e.g., $M(x,y)$, the call-by-name semantics of $\texttt{LAST}^n$ transforms the $\texttt{let}\, x_1$ into an explicit substitution that can only be resolved—thus enabling the $\texttt{send}\,()\,x$—once the $\texttt{close}\, x_1$ is unblocked by the $\texttt{let}\,(v, y_1)$, which in turn is waiting for the $\texttt{recv}\, y$ to be resolved. To be precise:

$$M(x,y) \to_{\mathtt{M}} \big(\,\texttt{let}\,(v,y_1) = \texttt{recv}\, y\, \texttt{in}\ldots\big)\{\!\{\texttt{send}\,()\,x/x_1\}\!\} \triangleq M'(x,y) \not\to_{\mathtt{M}}$$

$$M(y',x') \to_{\mathtt{M}} \big(\,\texttt{let}\,(v',x_1') = \texttt{recv}\, x'\, \texttt{in}\ldots\big)\{\!\{\texttt{send}\,()\,y'/y_1'\}\!\} \triangleq M'(y',x') \not\to_{\mathtt{M}}$$

$$C \to_{\mathtt{C}}^{9} (\boldsymbol{\nu} s[\epsilon\rangle s')(\boldsymbol{\nu} t[\epsilon\rangle t')\big(\Diamond\, M'(x,t)\{\!\{s/x\}\!\} \parallel \blacklozenge\, M'(y',s')\{\!\{t'/y'\}\!\}\big) \not\to_{\mathtt{C}}$$

We now detail the origin of the conflicting priority requirements; that is, we spell out the rules of the translation and the induced requirements. Note that many of these rules include names and priority variables that are not visible outside the derivation, being created from within certain parts of the translation.

(1) Requirement $\circ_6 < \pi_1$ originates from the translation of $\texttt{new}$ in the $\texttt{let}\,(x,x')$. More precisely, the translation of $\texttt{new}$ prepares the translation of the upcoming buffer, which follows the involved types. The requirement is induced by a step in the translation of this buffer (Rule [TYP-BUF] in Figure 18):

$$\frac{\vdash^* [\![[\epsilon\rangle : \mathsf{end}\rangle \mathsf{end}]\!]g_3\rangle k_3 :: g_3 : \bullet \parr^{\rho_5} \bullet \otimes^{\circ_{11}} \bullet, k_3 : \bullet \parr^{\rho_6} \bullet \otimes^{\pi_5} \bullet \quad \begin{array}{l}\circ_8 = \pi_2, \underline{\circ_6 < \pi_1}, \circ_9 < \pi_1,\\ \pi_1 < \circ_8, \pi_1 < \pi_5, \circ_7 < \circ_9,\\ \pi_3 < \pi_2, \pi_3 < \pi_4\end{array}}{\vdash^* [\![[\epsilon\rangle : \mathsf{!1.end}\rangle \mathsf{!1.end}]\!]d_2\rangle e_2 :: d_2 : \bullet \parr^{\circ_6} \bullet \otimes^{\circ_7} (\bullet \otimes^{\circ_8} \bullet) \parr^{\circ_9} \bullet \parr^{\circ_{10}} \bullet \otimes^{\circ_{11}} \bullet,\ e_2 : \bullet \parr^{\pi_1} (\bullet \parr^{\pi_2} \bullet) \otimes^{\pi_3} \bullet \parr^{\pi_4} \bullet \otimes^{\pi_5} \bullet}$$

(2) Requirement $\pi_1 < \pi_6$ originates from four substeps, including intermediate priorities: $\pi_1 < \pi_3 < \pi_{14} < \pi_{12} < \pi_6$.

(a) $\pi_1 < \pi_3$ originates from the translation of the variable x' (Rule [TYP-VAR] in Figure 17):

$$\frac{\underline{\pi_1 < \pi_3}}{\vdash^* [\![x']\!]a_4' :: x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet, a_4' : (\bullet \parr^{\pi_2} \bullet) \otimes^{\pi_3} \bullet \parr^{\pi_4} \bullet \otimes^{\pi_5} \bullet}$$

(b) $\pi_3 < \pi_{14}$ originates from the translation of the $\texttt{recv}\, x'$ (Rule [TYP-RECV] in Figure 17):

$$\frac{\begin{array}{l}x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet, \\ \vdash^* [\![x']\!] a'_4 :: a'_4 : (\bullet \parr^{\pi_2} \bullet) \otimes^{\pi_3} \bullet \parr^{\pi_4} \bullet \otimes^{\pi_5} \bullet\end{array} \qquad \pi_{14} < \pi_2, \pi_{14} < \pi_{15}, \pi_4 < \pi_5, \underline{\pi_3 < \pi_{14}}}{\vdash^* [\![\texttt{recv}\, x']\!] b'_3 :: x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet, b'_3 : (\bullet \parr^{\pi_2} \bullet) \otimes^{\pi_{14}} \bullet \parr^{\pi_{15}} \bullet \otimes^{\pi_5} \bullet}$$

(c) $\pi_{14} < \pi_{12}$ originates from the translation of the $\texttt{let}\,(v', x'_1)$ (Rule [TYP-SPLIT] in Figure 17):

$$\frac{\begin{array}{l}\vdash^* [\![\texttt{recv}\, x']\!] b'_3 \\ :: x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet, \\ \quad b'_3 : (\bullet \parr^{\pi_2} \bullet) \otimes^{\pi_{14}} \bullet \parr^{\pi_{15}} \bullet \otimes^{\pi_5} \bullet\end{array} \qquad \begin{array}{l}\vdash^* [\![\ldots]\!] a'_2 \\ :: y'_1 : \bullet \otimes^{\pi_{12}} \bullet \parr^{\pi_{11}} \bullet, \\ \quad v' : \bullet \otimes^{\pi_2} \bullet, \\ \quad x'_1 : \bullet \otimes^{\pi_{15}} \bullet \parr^{\pi_5} \bullet\end{array} \qquad \underline{\pi_{14} < \pi_{12}}}{\begin{array}{r}\vdash^* [\![\texttt{let}\,(v', x'_1) = \texttt{recv}\, x' \,\texttt{in}\ldots]\!] a'_2 :: x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet, \\ y'_1 : \bullet \otimes^{\pi_{12}} \bullet \parr^{\pi_{11}} \bullet, a'_2 : \bullet\end{array}}$$

(d) Finally, $\pi_{12} < \pi_6$ originates from the translation of the $\texttt{let}\, y'_1$. To be precise, this syntactic sugar breaks down into an abstraction and an application; the requirement originates from the application (Rule [TYP-APP] in Figure 17):

$$\frac{\begin{array}{l}\vdash^* [\![\lambda y'_1 \ldots]\!] a'_1 \\ :: x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet, \\ \quad a'_1 : (\bullet \otimes^{\pi_{12}} \bullet \parr^{\pi_{11}} \bullet) \parr^{\pi_{13}} \bullet\end{array} \qquad \begin{array}{l}\vdash^* [\![\texttt{send}\,()\, y']\!] e'_1 \\ :: y' : \bullet \otimes^{\pi_6} \bullet \parr^{\pi_7} (\bullet \parr^{\pi_8} \bullet) \\ \qquad \otimes^{\pi_9} \bullet \otimes^{\pi_{10}} \bullet \parr^{\pi_{11}} \bullet, \\ \quad e'_1 : \bullet \otimes^{\pi_{11}} \bullet\end{array} \qquad \begin{array}{l}\underline{\pi_{12} < \pi_6}, \\ \underline{\pi_{13} < \pi_{12}}\end{array}}{\begin{array}{r}\vdash^* [\![\texttt{let}\, y'_1 = \texttt{send}\, y' \,\texttt{in}\ldots]\!] z_2 :: x' : \bullet \otimes^{\pi_1} (\bullet \otimes^{\pi_2} \bullet) \parr^{\pi_3} \bullet \otimes^{\pi_4} \bullet \parr^{\pi_5} \bullet, \\ y' : \bullet \otimes^{\pi_6} \bullet \parr^{\pi_7} (\bullet \parr^{\pi_8} \bullet) \otimes^{\pi_9} \bullet \otimes^{\pi_{10}} \bullet \parr^{\pi_{11}} \bullet, \\ z_2 : \bullet\end{array}}$$

(3) Requirement $\pi_6 < \circ_1$ originates from the buffer prepared by the translation of `new` in the $\texttt{let}\,(y, y')$. It is derived similar to Requirement (1).
(4) Requirement $\circ_1 < \circ_6$ originates from four similar substeps as in Requirement (2).

We conclude that the translation of C cannot be typed under $\vdash$, because the priority requirements cannot be satisfied. Hence, Theorem 5.8 does not apply, and so deadlock freedom cannot be guaranteed for C.

Note that it is straightforward to define variants of C whose translations are well typed under $\vdash$, thus guaranteeing deadlock freedom through Theorem 5.8. An example is the variant of $M(x, y)$ above in which the $\texttt{close}\, x_1$ occurs before the $\texttt{let}\,(v, y_1)$.

Remark 5.10. Determining the deadlock freedom of well-typed $\texttt{LAST}^n$ programs by means of translation into APCP is of similar complexity as a direct, priority-based approach. It is easy to see that the translation is $O(n)$ in the size of the $\texttt{LAST}^n$ typing derivation. We could add priorities to the types of $\texttt{LAST}^n$ and derive priority requirements from the translation; this would boil down to roughly the same amount of checking. Hence, the indirect approach via APCP is similarly complex. This justifies keeping the type system of $\texttt{LAST}^n$ simple, compared to type systems that use priorities for deadlock-free functional programs such as those by, e.g., Padovani and Novara [PN15] and Kokke and Dardha [KD21a].

6. Related Work

Closely related work on the two themes of the paper (deadlock freedom by typing and functional calculi with concurrency) has been already discussed throughout the paper. Here we comment on other related literature.

Asynchronous Communication. Asynchronous communication has a longstanding history in process algebras (see, e.g., [BKT85, HJH90, BKP92]). The first accounts of asynchronous communication for the π-calculus were developed independently by Honda and Tokoro [HT91, HT92] and by Boudol [Bou92]. Palamidessi [Pal03] shows that the synchronous π-calculus is strictly more expressive than its asynchronous variant, due to *mixed choices*: non-deterministic choices involving both inputs and outputs. Beauxis *et al.* [BPV08] study the exact form of asynchronous communication modeled by the asynchronous π-calculus; they examine communication mediated by different mechanisms (bags, stacks, queues) in the synchronous π-calculus, and prove that bags lead to the strongest operational correspondence with the asynchronous π-calculus.

As discussed already, asynchrony is a relevant phenomenon in session π-calculi; the communication structures delineated by sessions strongly influence the expected ordering of messages. The expressiveness gap between asynchronous and synchronous communication shown by Palamidessi in the untyped setting does not hold in this context, since session-typed π-calculi consider only deterministic choices and do not account for mixed choices. A notable exception is the work on *mixed sessions* by Casal *et al.* [VCAM20, CMV22], which does not address deadlock freedom. Nevertheless, fundamental differences between mixed choices in untyped and session-typed settings remain, as established by Peters and Yoshida [PY22].

In the context of session types, the first formal theory of asynchronous communication for the π-calculus is by Kouzapas *et al.* [KYH11], using a buffered semantics. Their focus is on the behavioral theory induced by asynchrony and the program transformations it enables. Follow-up work [KYHH16] goes beyond to consider asynchronous communication in combination with constructs for event-driven programming, and develops a corresponding type system and behavioral theory. Unlike our work, these two works do not consider deadlock freedom for asynchronous session processes.

DeYoung *et al.* [DCPT12] give the first connection between linear logic and a session-typed π-calculus with asynchronous communication using a continuation-passing semantics, and show that this semantics is equivalent to a buffered semantics. As mentioned before, `APCP`'s semantics is based on this work, except that our typing rules for sending and selection are axiomatic and that we consider recursion. On a similar line, Jia *et al.* [JGP16] consider a session-typed language with asynchronous communication; their focus is on the dynamic monitoring of session behaviors, not on deadlock freedom. Both these works are based on the correspondence with intuitionistic linear logic, which restricts the kind of process networks allowed by typing. Pruiksma and Pfenning [PP19, PP21] derive a "propositions-as-sessions" type system from adjoint logic, which combines several logics with different structural rules through modalities [Ree09, PCPR18]. Their process language features asynchronous communication with continuation passing and their type system treats asynchronous, non-blocking outputs via axiomatic typing rules, similar to Rules [TYP-SEND] and [TYP-SEL] in Figure 4.

Padovani's linear type system for the asynchronous π-calculus [Pad14] has been already mentioned in the context of deadlock freedom for cyclic process networks. His language is different from `APCP`, as it lacks session constructs and does not have continuation passing

	LAST [GV10]	GV [Wad12]	EGV [FLMD19]	PGV [KD21a, KD21b]	LAST^n **(this paper)**
Communication	Async.	Sync.	Async.	Sync.	Async.
Cyclic Topologies	✓	×	×	✓	✓
Deadlock Freedom	×	✓ (by typing)	✓ (by typing)	✓ (by typing)	✓ (via APCP)

Table 1: The features of CGV compared to related works.

baked into the type system. While it should be possible to encode sessions in his typed framework (using communication of pairs to model continuation passing), it seems unclear how to transfer the analysis of deadlock freedom from this setting to a language such as APCP via such a translation.

Deadlock Freedom for Cyclic Process Networks. We have already discussed the related works by Kobayashi [Kob06], Padovani [Pad14], and Dardha and Gay [DG18]. The work of Kobayashi and Laneve [KL17] is related to APCP in that it addresses deadlock freedom for *unbounded* process networks. Toninho and Yoshida's work [TY18] addresses deadlock freedom for cyclic process networks by generating global types from binary types. The work by Balzer, Toninho and Pfenning [BP17, BTP19] is also worth mentioning: it guarantees deadlock freedom for processes with shared, mutable resources by means of manifest sharing, i.e., explicitly acquiring and releasing access to resources.

Functional Languages with Sessions. We have already discussed the related works by Gay and Vasconcelos [GV10], Wadler [Wad12], Kokke and Dardha [KD21a, KD21b], and Padovani and Novara [PN15]. Lindley and Morris [LM15] formally define a semantics for Wadler's GV, prove deadlock freedom and operational correspondence of the translation into Wadler's CP, and give a translation from CP into GV. Other related work in this line is Fowler *et al.*'s *Asynchronous* GV with buffered asynchronous communication and a call-by-value semantics, and *Exceptional* GV (EGV) which extends the former with exception handling [Fow19, FLMD19].

Table 1 summarizes the comparison of LAST^n to some of these related works. Note that none of the mentioned works supports sending anything but values, unlike LAST^n. Our work ensures deadlock freedom for configurations with cyclic topologies by means of a translation into a system with an established deadlock-freedom result (cf. Remark 5.10); the alternative approach is to enhance GV's type system with priorities, as done by Padovani and Novara [PN15] and Kokke and Dardha [KD21a, KD21b].

Typed Encodings between (Concurrent) λ-calculi and π-calculi. Several prior works develop typed encodings between λ- and π-calculi; to our knowledge, all of them consider call-by-value semantics, most do not consider λ with message passing, and none translate variables as sends, as we do. Some of these works have been already discussed in Section 4.3, where we justified the notions of completeness and soundness relevant for our translation of LAST^n into APCP.

Vasconcelos [Vas00] translates an untyped λ-calculus into an input/output-typed π-calculus; the translation is sound, up to barbed congruence. The aforementioned work by

Wadler [Wad12, Wad14] translates the session-typed GV with synchronous message passing into the session-typed CP, without giving a semantics for GV nor associated completeness/soundness results. Lindley and Morris [LM15] formalize a semantics for GV and its translation into CP, and prove completeness. Similarly to our approach, they use closures/explicit substitutions in GV to obtain a direct, more controlled correspondence with cut reduction in linear logic; however, no soundness result is given. Kokke and Dardha [KD21a] define PGV, an extension of GV with priority annotations to support deadlock-free cyclic thread configurations; they give a translation from PCP into PGV that is both sound and complete, but no translation in the other direction is studied. Toninho *et al.* [TCP12] translate a linearly typed λ-calculus into a session-typed π-calculus derived from intuitionistic linear logic; they prove completeness, but for soundness they extend reduction in λ to call by name. Toninho and Yoshida [TY21] translate a linearly typed λ-calculus without message passing into a session-typed polymorphic π-calculus. Their translation is fully abstract: it is complete and sound, with the latter result requiring an extension of reduction in π with commuting conversions. Fowler *et al.* [FKD$^+$23] introduce a variant of (call-by-value) GV based on *hypersequents*, and consider its relationship with hypersequent CP: they exhibit translations that preserve and reflect reduction, up to weak bisimilarity.

Other Applications of APCP. As mentioned in the introduction, two salient aspects of Curry-Howard correspondences for session types, namely analysis of deadlock freedom and connections with functional calculi, define the main themes of this paper. Yet another highlight of the logical correspondences is their suitability for the analysis of *multiparty protocols*. In separate work [HP22b], we have devised a methodology for the analysis of multiparty session types (MPST) based on APCP [HYC16]. In the multiparty context, two or more participants interact following a common protocol. APCP is well suited for the analysis of process implementations of MPST, which rely on asynchrony and recursion. In our analysis, multiple separate processes implement the roles of one or more participants of a multiparty protocol. The support for cyclic process networks in APCP allows us to connect these implementations with each other directly, without requiring an additional process (as used in similar prior works [CP16, CMSY15, CLM$^+$16]). The asynchrony in APCP captures the asynchronous nature of MPST, and APCP's recursion enables an expressive class of supported protocols. This way, our methodology unlocks the transfer of (static) analysis of deadlock freedom and protocol conformance from APCP to distributed implementations of MPST. The key ideas underlying the methodology in [HP22b] can be applied to the *runtime* verification of such distributed implementations, as recently shown in [HPD23].

7. Closing Remarks

In this paper, we have presented two contributions to the challenging issue of ensuring deadlock-free, message-passing interactions in a session-typed setting. Our *first contribution* is APCP: a typed framework for deadlock-free cyclic process networks with asynchronous communication and recursion. The design of APCP and its type system has a solid basis, with tight ties to logic ("propositions as sessions"): our syntax, semantics and type system harmoniously integrate insights separately presented in prior works for different typed variants of the π-calculus.

The step from synchronous communication (as in PCP [DG18]) to asynchronous communication (in APCP) is significant. In combination with cyclic process networks, asynchronous

communication enables checking a larger class of deadlock-free processes: processes that under synchronous communication would be deadlocked due to blocking outputs may be deadlock free in APCP (see, e.g., Example 3.4). Perhaps more significantly, asynchronous communication simplifies the priority management involved in the detection of cyclic dependencies (cf. Remark 3.15).

In our *second contribution*, we move to consider cyclic process networks with asynchronous communication in the setting of a prototypical functional language. We introduced $\texttt{LAST}^n$, a new functional language with asynchronous, session-typed communication. Inspired by Gay and Vasconcelos's LAST, the design of $\texttt{LAST}^n$ combines buffered communication with a call-by-name reduction strategy, explicit substitutions, and explicit session closing. In our opinion, this design makes $\texttt{LAST}^n$ an interesting object of study on its own. In the spirit of Gay and Vasconcelos's work, we defined a deliberately simple type system for $\texttt{LAST}^n$, whose type preservation property ensures protocol fidelity and communication safety, but not deadlock freedom. This means that well-typed threads in $\texttt{LAST}^n$ can perform session protocols that may lead to stuck terms/configurations. To connect our contributions, and to transfer the deadlock-freedom guarantee to the functional realm, we presented a translation from $\texttt{LAST}^n$ to APCP that is *operationally correct*. In particular, operational correctness includes the *soundness* property, which ensures that the translation reflects reduction steps and is critical to the transfer of deadlock freedom formalized by Theorem 5.8. This way, we were able to analyze deadlock freedom in $\texttt{LAST}^n$ by leaning on the established results for APCP (Section 3.3) without sacrificing complexity (cf. Remark 5.10).

In future work, it would be interesting to study the extension of $\texttt{LAST}^n$ with recursion/recursive types, and to extend our translation into APCP (and its operational correspondence results) accordingly. We do not expect technical difficulties, in particular if $\texttt{LAST}^n$ is enhanced with the kind of tail-recursive behavior present in APCP. In the same spirit, it would be insightful to explore how to accommodate APCP's support for priorities into a process language with inductive and coinductive types (least and greatest fixed points, respectively), such as the one studied by Rocha and Caires [RC23]. Finally, following [Kob05], it would be interesting to study type inference algorithms for APCP which could (automatically) determine priorities for typed processes.

Acknowledgements. We are grateful to the anonymous reviewers for their careful reading of our paper and their useful feedback. We also thank Ornela Dardha for clarifying the typing rules of PCP to us, and Simon Fowler and Simon Gay 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

[ALM16] Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation Confers Concurrency. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, pages 32–55. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-30936-1_2.

[BKP92] F.S. de Boer, J.W. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 137–147, June 1992. doi:10.1109/LICS.1992.185528.

[BKT85] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Process algebra with asynchronous communication mechanisms. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors,

Seminar on Concurrency, Lecture Notes in Computer Science, pages 76–95, Berlin, Heidelberg, 1985. Springer. doi:10.1007/3-540-15670-4_4.

[Bor98] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, March 1998. doi:10.1016/S0304-3975(97)00220-X.

[Bou92] Gérard Boudol. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992.

[BP17] Stephanie Balzer and Frank Pfenning. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.*, 1(ICFP):37:1–37:29, August 2017. doi:10.1145/3110281.

[BPV08] Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. On the Asynchronous Nature of the Asynchronous π-Calculus. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science, pages 473–492. Springer, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-68679-8_29.

[BTP19] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest Deadlock-Freedom for Shared Session Types. In Luís Caires, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 611–639, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-17184-1_22.

[Cai14] Luís Caires. Types and Logic, Concurrency and Non-Determinism. Technical Report MSR-TR-2014-104, In Essays for the Luca Cardelli Fest, Microsoft Research, September 2014.

[CLM+16] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CONCUR.2016.33.

[CMSY15] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty Session Types as Coherence Proofs. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 412–426, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CONCUR.2015.412.

[CMV22] Filipe Casal, Andreia Mordido, and Vasco T. Vasconcelos. Mixed sessions. *Theoretical Computer Science*, 897:23–48, 2022. doi:10.1016/J.TCS.2021.08.005.

[CP10] Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, pages 222–236, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-15375-4_16.

[CP16] Luís Caires and Jorge A. Pérez. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, pages 74–95. Springer International Publishing, 2016. doi:10.1007/978-3-319-39570-8_6.

[CP17] Luís Caires and Jorge A. Pérez. Linearity, Control Effects, and Behavioral Types. In Hongseok Yang, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 229–259, Berlin, Heidelberg, 2017. Springer. doi:10.1007/978-3-662-54434-1_9.

[CPT16] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, March 2016. doi:10.1017/S0960129514000218.

[DCPT12] 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*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 228–242, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2012.228.

[DG18] Ornela Dardha and Simon J. Gay. A New Linear Logic for Deadlock-Free Session-Typed Processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 91–109. Springer International Publishing, 2018. doi:10.1007/978-3-319-89366-2_5.

[DP15] Ornela Dardha and Jorge A. Pérez. Comparing Deadlock-Free Session Typed Processes. *Electronic Proceedings in Theoretical Computer Science*, 190:1–15, August 2015. arXiv:1508.06707, doi:10.4204/EPTCS.190.1.

[DP22] Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *Journal of Logical and Algebraic Methods in Programming*, 124:100717, January 2022. doi:10.1016/j.jlamp.2021.100717.

[FKD+23] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. *Log. Methods Comput. Sci.*, 19(3), 2023. doi:10.46298/LMCS-19(3:3)2023.

[FLMD19] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *Proceedings of the ACM on Programming Languages*, January 2019. doi:10.1145/3290341.

[Fow19] Simon Fowler. *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. PhD thesis, University of Edinburgh, 2019.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987. doi:10.1016/0304-3975(87)90045-4.

[Gor10] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, September 2010. doi:10.1016/j.ic.2010.05.002.

[GTV20] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of Session Types: The Final Cut. *Electronic Proceedings in Theoretical Computer Science*, 314:23–33, April 2020. arXiv:2004.01322, doi:10.4204/EPTCS.314.3.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010. doi:10.1017/S0956796809990268.

[HJH90] Jifeng He, Mark B. Josephs, and Charles Antony Richard Hoare. A theory of synchrony and asynchrony. In Manfred Broy and Cliff B. Jones, editors, *Programming Concepts and Methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, pages 459–478. North-Holland, 1990.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science, pages 509–523, Berlin, Heidelberg, 1993. Springer. doi:10.1007/3-540-57208-2_35.

[HP21] Bas van den Heuvel and Jorge A. Pérez. Deadlock freedom for asynchronous and cyclic process networks. In Julien Lange, Anastasia Mavridou, Larisa Safina, and Alceste Scalas, editors, *Proceedings 14th Interaction and Concurrency Experience, Online, 18th June 2021*, volume 347 of *Electronic Proceedings in Theoretical Computer Science*, pages 38–56. Open Publishing Association, 2021. doi:10.4204/EPTCS.347.3.

[HP22a] Bas van den Heuvel and Jorge A. Pérez. Asynchronous functional sessions: Cyclic and concurrent. In Valentina Castiglioni and Claudio A. Mezzina, editors, *Proceedings Combined 29th International Workshop on Expressiveness in Concurrency and 19th Workshop on Structural Operational Semantics, Warsaw, Poland, 12th September 2022*, volume 368 of *Electronic Proceedings in Theoretical Computer Science*, pages 75–94. Open Publishing Association, 2022. doi:10.4204/EPTCS.368.5.

[HP22b] Bas van den Heuvel and Jorge A. Pérez. A decentralized analysis of multiparty protocols. *Science of Computer Programming*, page 102840, June 2022. doi:10.1016/j.scico.2022.102840.

[HPD23] Bas van den Heuvel, Jorge A. Pérez, and Rares A. Dobre. Monitoring Blackbox Implementations of Multiparty Session Protocols. In Panagiotis Katsaros and Laura Nenzi, editors, *Runtime Verification*, Lecture Notes in Computer Science, pages 66–85, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-44267-4_4.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 133–147, Berlin, Heidelberg, 1991. Springer. doi:10.1007/BFb0057019.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, Lecture Notes in Computer Science, pages 21–51, Berlin, Heidelberg, 1992. Springer. doi:10.1007/3-540-55613-3_2.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 122–138, Berlin, Heidelberg, 1998. Springer. doi:10.1007/BFb0053567.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1), March 2016. doi:10.1145/2827695.

[JGP16] Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and Blame Assignment for Higher-order Session Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 582–594, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837662.

[KD21a] Wen Kokke and Ornela Dardha. Prioritise the Best Variation. In Kirstin Peters and Tim A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, pages 100–119, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-78089-0_6.

[KD21b] Wen Kokke and Ornela Dardha. Prioritise the Best Variation, December 2021. arXiv:2103.14466, doi:10.48550/arXiv.2103.14466.

[KL17] Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Information and Computation*, 252:48–70, February 2017. doi:10.1016/j.ic.2016.03.004.

[Kob05] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005. doi:10.1007/S00236-005-0179-X.

[Kob06] Naoki Kobayashi. A New Type System for Deadlock-Free Processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, Lecture Notes in Computer Science, pages 233–247. Springer Berlin Heidelberg, 2006. doi:10.1007/11817949_16.

[KYH11] Dimitrios Kouzapas, Nobuko Yoshida, and Kohei Honda. On Asynchronous Session Semantics. In Roberto Bruni and Juergen Dingel, editors, *Formal Techniques for Distributed Systems*, Lecture Notes in Computer Science, pages 228–243, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-21461-5_15.

[KYHH16] Dimitrios Kouzapas, Nobuko Yoshida, Raymond Hu, and Kohei Honda. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science*, 26(2):303–364, February 2016. doi:10.1017/S096012951400019X.

[LM99] Jean-Jacques Lévy and Luc Maranget. Explicit Substitutions and Programming Languages. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 181–200, Berlin, Heidelberg, 1999. Springer. doi:10.1007/3-540-46691-6_14.

[LM15] Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In Jan Vitek, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 560–584, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-46669-8_23.

[Mil89] Robin Milner. *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall, 1989.

[Mil90] Robin Milner. Functions as processes. In Michael S. Paterson, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 167–180, Berlin, Heidelberg, 1990. Springer. doi:10.1007/BFb0032030.

[Mil92] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, June 1992. doi:10.1017/S0960129500001407.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992. doi:10.1016/0890-5401(92)90008-4.

[Pad14] Luca Padovani. 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, pages 72:1–72:10, New York, NY, USA, 2014. ACM. doi:10.1145/2603088.2603116.

[Pal03] Catuscia Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, October 2003. arXiv:1307.2062, doi:10.1017/S0960129503004043.

[PCPR18] Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. *Unpublished manuscript*, April 2018.

[Pet19] Kirstin Peters. Comparing process calculi using encodings. In Jorge A. Pérez and Jurriaan Rot, editors, *Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019*, volume 300 of *EPTCS*, pages 19–38, 2019. doi:10.4204/EPTCS.300.2.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[PN15] Luca Padovani and Luca Novara. Types for Deadlock-Free Higher-Order Programs. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, pages 3–18, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-19195-9_1.

[PP19] Klaas Pruiksma and Frank Pfenning. A Message-Passing Interpretation of Adjoint Logic. In *Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES)*, volume 291 of *Electronic Proceedings in Theoretical Computer Science*, pages 60–79. Open Publishing Association, April 2019. arXiv:1904.01290, doi:10.4204/EPTCS.291.6.

[PP21] Klaas Pruiksma and Frank Pfenning. A Message-Passing Interpretation of Adjoint Logic. *Journal of Logical and Algebraic Methods in Programming*, 120(100637), April 2021. doi:10.1016/j.jlamp.2020.100637.

[PY22] Kirstin Peters and Nobuko Yoshida. On the Expressiveness of Mixed Choice Sessions. *Electronic Proceedings in Theoretical Computer Science*, 368:113–130, September 2022. arXiv:2209.06819, doi:10.4204/EPTCS.368.7.

[RC23] Pedro Rocha and Luís Caires. Safe session-based concurrency with shared linear state. In Thomas Wies, editor, *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, volume 13990 of *Lecture Notes in Computer Science*, pages 421–450. Springer, 2023. doi:10.1007/978-3-031-30044-8_16.

[Ree09] Jason Reed. A judgmental deconstruction of modal logic. *Unpublished manuscript, January*, 2009.

[TCP12] Bernardo Toninho, Luis Caires, and Frank Pfenning. Functions as Session-Typed Processes. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures*, Lecture Notes in Computer Science, pages 346–360, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-28729-9_23.

[TCP14] Bernardo Toninho, Luis Caires, and Frank Pfenning. Corecursion and Non-divergence in Session-Typed Processes. In Matteo Maffei and Emilio Tuosto, editors, *Trustworthy Global Computing*, Lecture Notes in Computer Science, pages 159–175, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-662-45917-1_11.

[TY18] Bernardo Toninho and Nobuko Yoshida. Interconnectability of Session-Based Logical Processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(4):17, December 2018. doi:10.1145/3242173.

[TY21] Bernardo Toninho and Nobuko Yoshida. On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings. *ACM Transactions on Programming Languages and Systems*, 43(2):7:1–7:55, June 2021. doi:10.1145/3457884.

[Vas00] Vasco T. Vasconcelos. The call-by-value λ-calculus, the SECD machine, and the π-calculus. report di-fcul-tr-00-3, Department of Informatics, University of Lisbon, May 2000.

[Vas12] Vasco T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, August 2012. doi:10.1016/j.ic.2012.05.002.

[VCAM20] Vasco T. Vasconcelos, Filipe Casal, Bernardo Almeida, and Andreia Mordido. Mixed Sessions. In Peter Müller, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 715–742, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-44914-8_26.

[Wad12] Philip Wadler. Propositions As Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012. ACM. doi:10.1145/2364527.2364568.

[Wad14] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, May 2014. doi:10.1017/S095679681400001X.

[YV07] Nobuko Yoshida and Vasco T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, July 2007. doi:10.1016/j.entcs.2007.02.056.

Contents

1. Introduction 1
2. Motivating Examples 4
2.1. Milner's Cyclic Scheduler in APCP 4
2.2. A Bookshop Scenario in LAST^n 6
3. APCP: Asynchronous Priority-based Classical Processes 8
3.1. The Process Language 8
3.2. The Type System 12
3.3. Type Preservation and Deadlock Freedom 21
3.4. Reactivity 28
3.5. Typing Milner's Cyclic Scheduler 30
3.6. Extensions: Explicit Session Closing and Replicated Servers 32
4. An Intermezzo: From APCP to LAST 33
4.1. The Syntax and Semantics of LAST* 34
4.2. The Type System of LAST* 38
4.3. Towards a Faithful Translation of LAST* into APCP 42
5. LAST^n and a Faithful Translation into APCP 43
5.1. The Language of LAST^n 44
5.2. Faithfully Translating LAST^n into APCP 52
5.3. Deadlock Free LAST^n 64
6. Related Work 68
7. Closing Remarks 70
References 71
Appendix A. LAST^n: Detailed Definitions and Proofs 78
A.1. Self-contained Definition of LAST^n and its Type System 78
A.2. Type Preservation 81
A.3. Translation: Type Preservation 87
A.4. Operational Correspondence 89

Terms ($M, N, \ldots$) and reduction contexts ($\mathcal{R}$):

$$
\begin{array}{rcl|rcl}
M, N & ::= & x & \text{variable} & \mid \ \texttt{new} & \text{create new channel} \\
& \mid & () & \text{unit value} & \mid \ \texttt{fork}\, M; N & \text{fork } M \text{ in parallel to } N \\
& \mid & \lambda x.M & \text{abstraction} & \mid \ (M, N) & \text{pair construction} \\
& \mid & M\ N & \text{application} & \mid \ \texttt{let}\,(x, y) = M\,\texttt{in}\,N & \text{pair deconstruction} \\
& \mid & \texttt{send}\, M\, N & \text{send } M \text{ along } N & \mid \ \texttt{select}\,\ell\, M & \text{select label } \ell \text{ along } M \\
& \mid & \texttt{recv}\, M & \text{receive along } M & \mid \ \texttt{case}\, M\, \texttt{of}\, \{i : M\}_{i \in I} & \text{offer labels in } I \text{ along } M \\
& \mid & \texttt{close}\, M; N & \text{close } M & \mid \ M\{\!\{N/x\}\!\} & \text{explicit substitution} \\
\mathcal{R} & ::= & [\cdot] \mid \mathcal{R}\ M \mid \texttt{send}\, M\, \mathcal{R} \mid \texttt{recv}\, \mathcal{R} \mid \texttt{let}\,(x, y) = \mathcal{R}\,\texttt{in}\,M & & & \\
& \mid & \texttt{select}\,\ell\,\mathcal{R} \mid \texttt{case}\,\mathcal{R}\,\texttt{of}\,\{i : M\}_{i \in I} \mid \texttt{close}\,\mathcal{R}; M \mid \mathcal{R}\{\!\{M/x\}\!\} & & &
\end{array}
$$

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

$$
\frac{x \notin \mathrm{fv}(\mathcal{R})}{(\mathcal{R}[M])\{\!\{N/x\}\!\} \equiv_{\mathtt{M}} \mathcal{R}\,[M\{\!\{N/x\}\!\}]}\ [\textsc{sc-sub-ext}] \qquad \frac{}{(\lambda x.M)\ N \rightarrow_{\mathtt{M}} M\{\!\{N/x\}\!\}}\ [\textsc{red-lam}]
$$

$$
\frac{}{\texttt{let}\,(x, y) = (M_1, M_2)\,\texttt{in}\,N \rightarrow_{\mathtt{M}} N\{\!\{M_1/x, M_2/y\}\!\}}\ [\textsc{red-pair}] \qquad \frac{}{x\{\!\{M/x\}\!\} \rightarrow_{\mathtt{M}} M}\ [\textsc{red-name-sub}]
$$

$$
\frac{M \rightarrow_{\mathtt{M}} N}{\mathcal{R}[M] \rightarrow_{\mathtt{M}} \mathcal{R}[N]}\ [\textsc{red-lift}] \qquad \frac{M \equiv_{\mathtt{M}} M' \qquad M' \rightarrow_{\mathtt{M}} N' \qquad N' \equiv_{\mathtt{M}} N}{M \rightarrow_{\mathtt{M}} N}\ [\textsc{red-lift-sc}]
$$

Figure 20: The $\texttt{LAST}^n$ term language.

Appendix A. $\texttt{LAST}^n$: Detailed Definitions and Proofs

A.1. **Self-contained Definition of $\texttt{LAST}^n$ and its Type System.** Figure 20 gives the term language of $\texttt{LAST}^n$, Figure 21 the configuration language, and Figure 22 the type system.

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

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

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

$$\frac{M \equiv_{\mathtt{M}} M'}{\phi\, M \equiv_{\mathtt{C}} \phi\, M'}\ [\textsc{sc-term-sc}] \qquad \frac{}{(\boldsymbol{\nu} x[\epsilon\rangle y)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C}\ [\textsc{sc-res-swap}]$$

$$\frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\boldsymbol{\nu} z[\vec{n}\rangle w)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} z[\vec{n}\rangle w)(\boldsymbol{\nu} x[\vec{m}\rangle y)C}\ [\textsc{sc-res-comm}] \qquad \frac{x, y \notin \mathrm{fv}(C)}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D) \equiv_{\mathtt{C}} C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y)D}\ [\textsc{sc-res-ext}]$$

$$\frac{}{C \parallel D \equiv_{\mathtt{C}} D \parallel C}\ [\textsc{sc-par-comm}] \qquad \frac{}{C \parallel (D \parallel E) \equiv_{\mathtt{C}} (C \parallel D) \parallel E}\ [\textsc{sc-par-assoc}] \qquad \frac{}{\phi\,(M\{\!\{N/x\}\!\}) \equiv_{\mathtt{C}} (\phi\, M)\{\!\{N/x\}\!\}}\ [\textsc{sc-conf-sub}]$$

$$\frac{x \notin \mathrm{fv}(\mathcal{G})}{(\mathcal{G}[C])\{\!\{M/x\}\!\} \equiv_{\mathtt{C}} \mathcal{G}[C\{\!\{M/x\}\!\}]}\ [\textsc{sc-conf-sub-ext}]$$

Configuration reduction ($\rightarrow_{\mathtt{C}}$):

$$\frac{}{\mathcal{F}[\mathtt{new}] \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)(\mathcal{F}[(x, y)])}\ [\textsc{red-new}] \qquad \frac{}{\hat{\mathcal{F}}[\mathtt{fork}\, M; N] \rightarrow_{\mathtt{C}} \hat{\mathcal{F}}[N] \parallel \lozenge\, M}\ [\textsc{red-fork}]$$

$$\frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[\mathtt{send}\, M\, x] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\hat{\mathcal{F}}[x] \parallel C)}\ [\textsc{red-send}]$$

$$\frac{}{(\boldsymbol{\nu} x[\vec{m}, M\rangle y)(\hat{\mathcal{F}}[\mathtt{recv}\, y] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[(M, y)] \parallel C)}\ [\textsc{red-recv}]$$

$$\frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{select}\, \ell\, x] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\ell, \vec{m}\rangle y)(\mathcal{F}[x] \parallel C)}\ [\textsc{red-select}]$$

$$\frac{j \in I}{(\boldsymbol{\nu} x[\vec{m}, j\rangle y)(\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_i\}_{i \in I}] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[M_j\ y] \parallel C)}\ [\textsc{red-case}]$$

$$\frac{}{(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{close}\, x; M] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} \Box[\vec{m}\rangle y)(\mathcal{F}[M] \parallel C)}\ [\textsc{red-close}] \qquad \frac{}{(\boldsymbol{\nu} \Box[\epsilon\rangle \Box)C \rightarrow_{\mathtt{C}} C}\ [\textsc{red-res-nil}]$$

$$\frac{}{C \parallel \lozenge\, () \rightarrow_{\mathtt{C}} C}\ [\textsc{red-par-nil}] \qquad \frac{C \rightarrow_{\mathtt{C}} C'}{\mathcal{G}[C] \rightarrow_{\mathtt{C}} \mathcal{G}[C']}\ [\textsc{red-lift-C}] \qquad \frac{M \rightarrow_{\mathtt{M}} M'}{\mathcal{F}[M] \rightarrow_{\mathtt{C}} \mathcal{F}[M']}\ [\textsc{red-lift-M}]$$

$$\frac{C \equiv_{\mathtt{C}} C' \qquad C' \rightarrow_{\mathtt{C}} D' \qquad D' \equiv_{\mathtt{C}} D}{C \rightarrow_{\mathtt{C}} D}\ [\textsc{red-conf-lift-sc}]$$

Figure 21: The $\mathtt{LAST}^n$ configuration language.

$$\frac{}{x : T \vdash_{\mathtt{M}} x : T}\ [\textsc{typ-var}] \qquad \frac{\Gamma, x : T \vdash_{\mathtt{M}} M : U}{\Gamma \vdash_{\mathtt{M}} \lambda x.M : T \multimap U}\ [\textsc{typ-abs}] \qquad \frac{\Gamma \vdash_{\mathtt{M}} M : T \multimap U \quad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} M\ N : U}\ [\textsc{typ-app}]$$

$$\frac{}{\emptyset \vdash_{\mathtt{M}} () : \mathbf{1}}\ [\textsc{typ-unit}] \qquad \frac{\Gamma \vdash_{\mathtt{M}} M : T \quad \Delta \vdash_{\mathtt{M}} N : U}{\Gamma, \Delta \vdash_{\mathtt{M}} (M, N) : T \times U}\ [\textsc{typ-pair}]$$

$$\frac{\Gamma \vdash_{\mathtt{M}} M : T \times T' \quad \Delta, x : T, y : T' \vdash_{\mathtt{M}} N : U}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{let}\ (x, y) = M\ \mathtt{in}\ N : U}\ [\textsc{typ-split}] \qquad \frac{}{\emptyset \vdash_{\mathtt{M}} \mathtt{new} : S \times \overline{S}}\ [\textsc{typ-new}]$$

$$\frac{\Gamma \vdash_{\mathtt{M}} M : \mathbf{1} \quad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{fork}\ M; N : T}\ [\textsc{typ-fork}] \qquad \frac{\Gamma \vdash_{\mathtt{M}} M : \mathsf{end} \quad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{close}\ M; N : T}\ [\textsc{typ-close}]$$

$$\frac{\Gamma \vdash_{\mathtt{M}} M : T \quad \Delta \vdash_{\mathtt{M}} N : !T.S}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{send}\ M\ N : S}\ [\textsc{typ-send}] \qquad \frac{\Gamma \vdash_{\mathtt{M}} M : ?T.S}{\Gamma \vdash_{\mathtt{M}} \mathtt{recv}\ M : T \times S}\ [\textsc{typ-recv}]$$

$$\frac{\Gamma \vdash_{\mathtt{M}} M : \oplus\{i : S_i\}_{i \in I} \quad j \in I}{\Gamma \vdash_{\mathtt{M}} \mathtt{select}\ j\ M : S_j}\ [\textsc{typ-sel}] \qquad \frac{\Gamma \vdash_{\mathtt{M}} M : \&\{i : S_i\}_{i \in I} \quad \forall i \in I.\ \Delta \vdash_{\mathtt{M}} N_i : S_i \multimap U}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{case}\ M\ \mathtt{of}\ \{i : N_i\}_{i \in I} : U}\ [\textsc{typ-case}]$$

$$\frac{\Gamma, x : T \vdash_{\mathtt{M}} M : U \quad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} M\{\!\{N/x\}\!\} : U}\ [\textsc{typ-sub}]$$

$$\frac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S'}\ [\textsc{typ-buf}] \qquad \frac{\Gamma \vdash_{\mathtt{M}} M : T \quad \Delta \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S}{\Gamma, \Delta \vdash_{\mathtt{B}} [\vec{m}, M\rangle : S'\rangle !T.S}\ [\textsc{typ-buf-send}] \qquad \frac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S_j \quad j \in I}{\Gamma \vdash_{\mathtt{B}} [\vec{m}, j\rangle : S'\rangle \oplus\{i : S_i\}_{i \in I}}\ [\textsc{typ-buf-sel}]$$

$$\frac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \Box}\ [\textsc{typ-buf-end-L}] \qquad \frac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \Box\rangle \mathsf{end}}\ [\textsc{typ-buf-end-R}]$$

$$\frac{\Gamma \vdash_{\mathtt{M}} M : \hat{T}}{\Gamma \vdash_{\mathtt{C}}^{\blacklozenge} \blacklozenge M : \hat{T}}\ [\textsc{typ-main}] \qquad \frac{\Gamma \vdash_{\mathtt{M}} M : \mathbf{1}}{\Gamma \vdash_{\mathtt{C}}^{\Diamond} \Diamond M : \mathbf{1}}\ [\textsc{typ-child}] \qquad \frac{\Gamma \vdash_{\mathtt{C}}^{\phi_1} C : T_1 \quad \Delta \vdash_{\mathtt{C}}^{\phi_2} D : T_2}{\Gamma, \Delta \vdash_{\mathtt{C}}^{\phi_1 + \phi_2} C \parallel D : T_1 + T_2}\ [\textsc{typ-par}]$$

$$\frac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S \quad \Delta, x : S' \vdash_{\mathtt{C}}^{\phi} C : T \quad \Gamma', y : \overline{S} = \Gamma, \Delta}{\Gamma' \vdash_{\mathtt{C}}^{\phi} (\boldsymbol{\nu} x[\vec{m}\rangle y) C : T}\ [\textsc{typ-res}]$$

$$\frac{\Gamma, x : T \vdash_{\mathtt{C}}^{\phi} C : U \quad \Delta \vdash_{\mathtt{M}} M : T}{\Gamma, \Delta \vdash_{\mathtt{C}}^{\phi} C\{\!\{M/x\}\!\} : U}\ [\textsc{typ-conf-sub}]$$

Figure 22: $\mathtt{LAST}^n$ typing rules for terms (top), buffers (center), and configurations (bottom).

A.2. **Type Preservation.** Here, we prove type preservation for $\texttt{LAST}^n$:

[Type Preservation for $\texttt{LAST}^n$] Given $\Gamma \vdash^{\phi}_{\texttt{C}} C : T$, if $C \equiv_{\texttt{C}} D$ or $C \to_{\texttt{C}} D$, then $\Gamma \vdash^{\phi}_{\texttt{C}} D : T$.

The proof is split into two parts: subject congruence and subject reduction. These and intermediate results are organized as follows:

- Theorems A.1 and A.2 prove subject congruence and subject reduction for terms, respectively.
- Theorems A.3 and A.4 then prove subject congruence and subject reduction for configurations, respectively, from which Section 5.1 follows.

Theorem A.1 (Subject Congruence for Terms)**.** *If* $\Gamma \vdash_{\texttt{M}} M : T$ *and* $M \equiv_{\texttt{M}} N$*, then* $\Gamma \vdash_{\texttt{M}} N : T$.

Proof. By induction on the derivation of $M \equiv_{\texttt{M}} N$. The inductive cases follow from the IH directly. We consider the only Rule [SC-SUB-EXT]:

$$x \notin \mathrm{fv}(\mathcal{R}) \implies (\mathcal{R}[M])\{\!\{N/x\}\!\} \equiv_{\texttt{M}} \mathcal{R}[M\{\!\{N/x\}\!\}]$$

We apply induction on the structure of the reduction context $\mathcal{R}$. As an interesting, representative case, consider $\mathcal{R} = L\{\!\{\mathcal{R}'/y\}\!\}$. Assuming $x \notin \mathrm{fv}(\mathcal{R})$, we have $x \notin \mathrm{fv}(L) \cup \mathrm{fv}(\mathcal{R}')$. We apply inversion of typing:

$$\dfrac{\dfrac{\Gamma, y : U \vdash_{\texttt{M}} L : T \qquad \Delta, x : U' \vdash_{\texttt{M}} \mathcal{R}'[M] : U}{\Gamma, \Delta, x : U' \vdash_{\texttt{M}} L\{\!\{(\mathcal{R}'[M])/y\}\!\} : T}\,[\text{TYP-SUB}] \qquad \Delta' \vdash_{\texttt{M}} N : U'}{\Gamma, \Delta, \Delta' \vdash_{\texttt{M}} (L\{\!\{(\mathcal{R}'[M])/y\}\!\})\{\!\{N/x\}\!\} : T}\,[\text{TYP-SUB}]$$

We can derive:

$$\dfrac{\Delta, x : U' \vdash_{\texttt{M}} \mathcal{R}'[M] : U \qquad \Delta' \vdash_{\texttt{M}} N : U'}{\Delta, \Delta' \vdash_{\texttt{M}} (\mathcal{R}'[M])\{\!\{N/x\}\!\} : U}\,[\text{TYP-SUB}]$$

Since $x \notin \mathrm{fv}(\mathcal{R}')$, by Rule [SC-SUB-EXT], $(\mathcal{R}'[M])\{\!\{N/x\}\!\} \equiv_{\texttt{M}} \mathcal{R}'[M\{\!\{N/x\}\!\}]$. Then, by the IH, $\Delta, \Delta' \vdash_{\texttt{M}} \mathcal{R}'[M\{\!\{N/x\}\!\}] : U$. Hence, we can conclude the following:

$$\dfrac{\Gamma, y : U \vdash_{\texttt{M}} L : T \qquad \Delta, \Delta' \vdash_{\texttt{M}} \mathcal{R}'[M\{\!\{N/x\}\!\}] : U}{\Gamma, \Delta, \Delta' \vdash_{\texttt{M}} L\{\!\{(\mathcal{R}'[M\{\!\{N/x\}\!\}])/y\}\!\} : T}\,[\text{TYP-SUB}]$$

It is straightforward to see that this reasoning works in opposite direction as well. □

Theorem A.2 (Subject Reduction for Terms)**.** *If* $\Gamma \vdash_{\texttt{M}} M : T$ *and* $M \to_{\texttt{M}} N$*, then* $\Gamma \vdash_{\texttt{M}} N : T$.

Proof. By induction on the derivation of $M \to_{\texttt{M}} N$ (IH_1). The case of Rule [RED-LIFT] follows by induction on the structure of the reduction context $\mathcal{R}$, where the base case ($\mathcal{R} = [\cdot]$) follows from IH_1. The case of Rule [RED-LIFT-SC] follows from IH_1 and Theorem A.1 (subject congruence for terms). We consider the other cases, applying inversion of typing and deriving the typing of the term after reduction:

- Rule [RED-LAM]: $(\lambda x.M)\ N \to_{\texttt{M}} M\{\!\{N/x\}\!\}$.

$$\dfrac{\dfrac{\Gamma, x : T \vdash_{\texttt{M}} M : U}{\Gamma \vdash_{\texttt{M}} \lambda x.M : T \multimap U}\,[\text{TYP-ABS}] \qquad \Delta \vdash_{\texttt{M}} N : T}{\Gamma, \Delta \vdash_{\texttt{M}} (\lambda x.M)\ N : U}\,[\text{TYP-APP}]$$

$$\to_{\texttt{M}}$$

$$\dfrac{\Gamma, x : T \vdash_{\texttt{M}} M : U \qquad \Delta \vdash_{\texttt{M}} N : T}{\Gamma, \Delta \vdash_{\texttt{M}} M\{\!\{N/x\}\!\} : U}\,[\text{TYP-SUB}]$$

- Rule [RED-PAIR]: $\mathtt{let}\,(x,y) = (M_1, M_2)\,\mathtt{in}\,N \to_{\mathtt{M}} N\{\!\{M_1/x, M_2/y\}\!\}$.

$$\dfrac{\dfrac{\Gamma \vdash_{\mathtt{M}} M_1 : T \qquad \Gamma' \vdash_{\mathtt{M}} M_2 : T'}{\Gamma,\Gamma' \vdash_{\mathtt{M}} (M_1, M_2) : T \times T'}\,[\text{TYP-PAIR}] \qquad \Delta, x : T, y : T' \vdash_{\mathtt{M}} N : U}{\Gamma,\Gamma',\Delta \vdash_{\mathtt{M}} \mathtt{let}\,(x,y) = (M_1,M_2)\,\mathtt{in}\,N : U}\,[\text{TYP-SPLIT}]$$

$$\to_{\mathtt{M}}$$

$$\dfrac{\dfrac{\Delta, x : T, y : T' \vdash_{\mathtt{M}} N : U \qquad \Gamma \vdash_{\mathtt{M}} M_1 : T}{\Gamma,\Delta, y : T' \vdash_{\mathtt{M}} N\{\!\{M_1/x\}\!\} : U}\,[\text{TYP-SUB}] \qquad \Gamma' \vdash_{\mathtt{M}} M_2 : T'}{\Gamma,\Gamma',\Delta \vdash_{\mathtt{M}} N\{\!\{M_1/x, M_2/y\}\!\} : U}\,[\text{TYP-SUB}]$$

- Rule [RED-NAME-SUB]: $x\{\!\{M/x\}\!\} \to_{\mathtt{M}} M$.

$$\dfrac{\dfrac{}{x : U \vdash_{\mathtt{M}} x : U}\,[\text{TYP-VAR}] \qquad \Gamma \vdash_{\mathtt{M}} M : U}{\Gamma \vdash_{\mathtt{M}} x\{\!\{M/x\}\!\} : U}\,[\text{TYP-SUB}]$$

$$\to_{\mathtt{M}}$$

$$\Gamma \vdash_{\mathtt{M}} M : U$$

□

Theorem A.3 (Subject Congruence for Configurations)**.** *If* $\Gamma \vdash^{\phi}_{\mathtt{C}} C : T$ *and* $C \equiv_{\mathtt{C}} D$*, then* $\Gamma \vdash^{\phi}_{\mathtt{C}} D : T$.

Proof. By induction on the derivation of $C \equiv_{\mathtt{C}} D$. The inductive cases follow from the IH directly. The case for Rule [SC-TERM-SC] follows from Theorem A.1 (subject congruence for terms). The cases for Rules [SC-RES-COMM], [SC-PAR-NIL], [SC-PAR-COMM], and [PAR-ASSOC] are straightforward. We consider the other cases:

- Rule [SC-RES-SWAP]: $(\boldsymbol{\nu} x[\epsilon\rangle y)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C$.
  The analysis depends on whether exactly one of x, y is $\square$ or not. We discuss both cases.
  – Exactly one of x, y is $\square$; w.l.o.g., assume $x = \square$. We have the following:

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \square\rangle \mathsf{end}}\,[\text{TYP-BUF-END-R}] \qquad \Gamma, x : \mathsf{end} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu}\square[\epsilon\rangle y)C : T}\,[\text{TYP-RES}]$$

$$\equiv_{\mathtt{C}}$$

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \square}\,[\text{TYP-BUF-END-L}] \qquad \Gamma, x : \mathsf{end} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle \square)C : T}\,[\text{TYP-RES}]$$

  – Neither or both of x, y are $\square$. We have the following:

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S'}\,[\text{TYP-BUF}] \qquad \Gamma, x : S', y : \overline{S'} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)C : T}\,[\text{TYP-RES}]$$

$$\equiv_{\mathtt{C}}$$

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S'}\rangle \overline{S'}}\,[\text{TYP-BUF}] \qquad \Gamma, x : S', y : \overline{S'} \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C : T}\,[\text{TYP-RES}]$$

- Rule [SC-RES-EXT]: $x, y \notin \mathrm{fv}(C) \implies (\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D) \equiv_{\mathtt{C}} C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y)D$.

The analysis depends on whether C or D are child threads. W.l.o.g., we assume C is a child thread. Assuming $x, y \notin \mathrm{fn}(C)$, we apply inversion of typing:

$$\dfrac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S \qquad \dfrac{\Delta \vdash_{\mathtt{C}}^{\Diamond} C : \mathbf{1} \qquad \Lambda, x : S', y : \overline{S} \vdash_{\mathtt{C}}^{\phi} D : T}{\Delta, \Lambda, x : S', y : \overline{S} \vdash_{\mathtt{C}}^{\Diamond+\phi} C \parallel D : T}\ [\textsc{typ-par}]}{\Gamma, \Delta, \Lambda \vdash_{\mathtt{C}}^{\Diamond+\phi} (\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D) : T}\ [\textsc{typ-res}]$$

Then, we derive the typing of the structurally congruent configuration:

$$\dfrac{\Delta \vdash_{\mathtt{C}}^{\Diamond} C : \mathbf{1} \qquad \dfrac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S \qquad \Lambda, x : S', y : \overline{S} \vdash_{\mathtt{C}}^{\phi} D : T}{\Gamma, \Lambda \vdash_{\mathtt{C}}^{\phi} (\boldsymbol{\nu} x[\vec{m}\rangle y) D : T}\ [\textsc{typ-res}]}{\Gamma, \Delta, \Lambda \vdash_{\mathtt{C}}^{\Diamond+\phi} C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y) D : T}\ [\textsc{typ-par}]$$

The other direction is analogous.

- Rule [SC-CONF-SUB]: $\phi\,(M\{\!\{N/x\}\!\}) \equiv_{\mathtt{C}} (\phi\, M)\{\!\{N/x\}\!\}$.
  This case follows by a straightforward inversion of typing on both terms:

$$\dfrac{\dfrac{\Gamma, x : T \vdash_{\mathtt{M}} M : U \qquad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} M\{\!\{N/x\}\!\} : U}\ [\textsc{typ-sub}]}{\Gamma, \Delta \vdash_{\mathtt{C}}^{\phi} \phi\,(M\{\!\{N/x\}\!\}) : U}\ [\textsc{typ-main/child}]$$

$$\equiv_{\mathtt{C}}$$

$$\dfrac{\dfrac{\Gamma, x : T \vdash_{\mathtt{M}} M : U}{\Gamma, x : T \vdash_{\mathtt{C}}^{\phi} \phi\, M : U}\ [\textsc{typ-main/child}] \qquad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{C}}^{\phi} (\phi\, M)\{\!\{N/x\}\!\} : U}\ [\textsc{typ-conf-sub}]$$

- Rule [SC-CONF-SUB-EXT]: $x \notin \mathrm{fv}(\mathcal{G}) \implies (\mathcal{G}[C])\{\!\{M/x\}\!\} \equiv_{\mathtt{C}} \mathcal{G}[C\{\!\{M/x\}\!\}]$.
  This case follows by induction on the structure of $\mathcal{G}$. The inductive cases follow from the IH straightforwardly. For the base case ($\mathcal{G} = [\cdot]$), the structural congruence is simply an equality. □

Theorem A.4 (Subject Reduction for Configurations)**.** *If* $\Gamma \vdash_{\mathtt{C}}^{\phi} C : T$ *and* $C \to_{\mathtt{C}} D$*, then* $\Gamma \vdash_{\mathtt{C}}^{\phi} D : T$.

Proof. By induction on the derivation of $C \to_{\mathtt{C}} D$ (IH_1). The case of Rule [RED-LIFT-C] ($C \to_{\mathtt{C}} C' \implies \mathcal{G}[C] \to_{\mathtt{C}} \mathcal{G}[C']$) follows by induction on the structure of $\mathcal{G}$, directly from IH_1. The case of Rule [RED-LIFT-M] ($M \to_{\mathtt{M}} M' \implies \mathcal{F}[M] \to_{\mathtt{C}} \mathcal{F}[M']$) follows by induction on the structure of $\mathcal{F}$, where the base case ($\mathcal{F} = \phi\,[\cdot]$) follows from Theorem A.2 (subject reduction for terms). The case for Rule [RED-CONF-LIFT-SC] ($C \equiv_{\mathtt{C}} C' \wedge C' \to_{\mathtt{C}} D' \wedge D' \equiv_{\mathtt{C}} D \implies C \to_{\mathtt{C}} D$) follows from IH_1 and Theorem A.3 (subject congruence for configurations). We consider the other cases:

- Rule [RED-NEW]: $\mathcal{F}[\mathtt{new}] \to_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)(\mathcal{F}[(x, y)])$.
  This case follows by induction on the structure of $\mathcal{F}$. The inductive cases follow from the IH directly. For the base case ($\mathcal{F} = \phi\,[\cdot]$), we apply inversion of typing and derive the

typing of the reduced configuration:

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{M}} \mathtt{new} : S \times \overline{S}}\,[\textsc{typ-new}]}{\emptyset \vdash^{\phi}_{\mathtt{C}} \phi\ \mathtt{new} : S \times \overline{S}}\,[\textsc{typ-main/child}]$$

$$\rightarrow_{\mathtt{C}}$$

$$\dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S\rangle S}\,[\textsc{typ-buf}] \qquad \dfrac{\dfrac{\dfrac{}{x : S \vdash_{\mathtt{M}} x : S}\,[\textsc{typ-var}] \quad \dfrac{}{y : \overline{S} \vdash_{\mathtt{M}} y : \overline{S}}\,[\textsc{typ-var}]}{x : S, y : \overline{S} \vdash_{\mathtt{M}} (x, y) : S \times \overline{S}}\,[\textsc{typ-pair}]}{x : S, y : \overline{S} \vdash^{\phi}_{\mathtt{C}} \phi\,(x, y) : S \times \overline{S}}\,[\textsc{typ-main/child}]}{\emptyset \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)(\phi\,(x, y)) : S \times \overline{S}}\,[\textsc{typ-res}]$$

- Rule [RED-FORK]: $\hat{\mathcal{F}}[\mathtt{fork}\,M; N] \rightarrow_{\mathtt{C}} \hat{\mathcal{F}}[N] \parallel \Diamond M$. By induction on the structure of $\hat{\mathcal{F}}$, which excludes holes under explicit substitution and so no names of M are captured by $\hat{\mathcal{F}}$. The inductive cases follows from the IH directly; we detail the base case ($\hat{\mathcal{F}} = \phi\,[\cdot]$):

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\mathtt{M}} M : \mathbf{1} \qquad \Delta \vdash_{\mathtt{M}} N : T}{\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{fork}\,M; N : T}\,[\textsc{typ-fork}]}{\Gamma, \Delta \vdash^{\phi}_{\mathtt{C}} \phi\ \mathtt{fork}\,M; N : T}\,[\textsc{typ-main/-child}]$$

$$\rightarrow_{\mathtt{C}}$$

$$\dfrac{\dfrac{\Delta \vdash_{\mathtt{M}} N : T}{\Delta \vdash^{\phi}_{\mathtt{C}} \phi\,N : T}\,[\textsc{typ-main/-child}] \qquad \dfrac{\Gamma \vdash_{\mathtt{M}} M : \mathbf{1}}{\Gamma \vdash^{\Diamond}_{\mathtt{C}} \Diamond M : \mathbf{1}}\,[\textsc{typ-child}]}{\Gamma, \Delta \vdash^{\phi+\Diamond}_{\mathtt{C}} \phi\,N \parallel \Diamond M : T}\,[\textsc{typ-par}]$$

  Clearly, $\phi + \Diamond = \phi$, proving the thesis.
- Rule [RED-SEND]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[\mathtt{send}\,M\,x] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\hat{\mathcal{F}}[x] \parallel C)$.

  This case follows by induction on the structure of $\hat{\mathcal{F}}$. The inductive cases follow from the IH straightforwardly. The fact that the hole in $\hat{\mathcal{F}}$ does not occur under an explicit substitution guarantees that we can move M out of the context of $\hat{\mathcal{F}}$ and into the buffer. We consider the base case ($\hat{\mathcal{F}} = \phi\,\mathcal{R}$). By well typedness, it must be that $\mathcal{R} = \mathcal{R}_1[\mathtt{close}\,\mathcal{R}_2; M]$. We apply induction on the structures of $\mathcal{R}_1, \mathcal{R}_2$ and consider the base cases: $\mathcal{R}_1 = \mathcal{R}_2 = [\cdot]$. We apply inversion of typing, w.l.o.g. assuming that $\phi = \blacklozenge$ and $y \in \mathrm{fv}(C)$ (omitting "TYP-" from rule labels):

$$\dfrac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : !T.\mathsf{end}\rangle S \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta_1 \vdash_{\mathtt{M}} M : T \quad \dfrac{}{x : !T.\mathsf{end} \vdash_{\mathtt{M}} x : !T.\mathsf{end}}\,[\textsc{var}]}{\Delta_1, x : !T.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{send}\,M\,x : \mathsf{end}}\,[\textsc{send}] \quad \Delta_2 \vdash_{\mathtt{M}} N : U}{\Delta_1, \Delta_2, x : !T.\mathsf{end} \vdash_{\mathtt{M}} \mathtt{close}\,(\mathtt{send}\,M\,x); N : U}\,[\textsc{close}]}{\Delta_1, \Delta_2, x : !T.\mathsf{end} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{close}\,(\mathtt{send}\,M\,x); N) : U}\,[\textsc{main}] \quad \Lambda, y : \overline{S} \vdash^{\Diamond}_{\mathtt{C}} C : \mathbf{1}}{\Delta_1, \Delta_2, x : !T.\mathsf{end}, \Lambda, y : \overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{close}\,(\mathtt{send}\,M\,x); N) \parallel C : U}\,[\textsc{par}]}{\Gamma, \Delta_1, \Delta_2, \Lambda \vdash^{\blacklozenge}_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\blacklozenge\,(\mathtt{close}\,(\mathtt{send}\,M\,x); N) \parallel C) : U}\,[\textsc{res}]$$

  Note that the derivation of $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : !T.\mathsf{end}\rangle S$ depends on the size of $\vec{m}$. By induction on the size of $\vec{m}$ (IH_2), we derive $\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}\rangle : \mathsf{end}\rangle S$:
  – If $\vec{m}$ is empty, it follows by inversion of typing that $\Gamma = \emptyset$ and $S = !T.\mathsf{end}$:

$$\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : !T.\mathsf{end}\rangle !T.\mathsf{end}}\,[\textsc{typ-buf}]$$

Then, we derive the following:

$$\frac{\Delta_1 \vdash_{\mathtt{M}} M : T \qquad \dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle\mathsf{end}}\,[\textsc{typ-buf}]}{\Delta_1 \vdash_{\mathtt{B}} [M\rangle : \mathsf{end}\rangle !T.\mathsf{end}}\,[\textsc{typ-buf-send}]$$

– If $\vec{m} = \vec{m}', L$, it follows by inversion of typing that $\Gamma = \Gamma', \Gamma''$ and $S = !T'.\mathsf{end}'$:

$$\frac{\Gamma' \vdash_{\mathtt{M}} L : T' \qquad \Gamma'' \vdash_{\mathtt{B}} [\vec{m}'\rangle : !T.\mathsf{end}\rangle\mathsf{end}'}{\Gamma', \Gamma'' \vdash_{\mathtt{B}} [\vec{m}', L\rangle : !T.\mathsf{end}\rangle !T'.\mathsf{end}'}\,[\textsc{typ-buf-send}]$$

By IH_2, $\Gamma'', \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle\mathsf{end}'$, allowing us to derive the following:

$$\frac{\Gamma' \vdash_{\mathtt{M}} L : T' \qquad \Gamma'', \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle\mathsf{end}'}{\Gamma', \Gamma'', \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}', L\rangle : \mathsf{end}\rangle !T'.\mathsf{end}'}\,[\textsc{typ-buf-send}]$$

– If $\vec{m} = \vec{m}', j$, it follows by inversion of typing that there exist types S_i for each i in a set of labels I, where $j \in I$, such that $S = \oplus\{i : S_i\}_{i \in I}$:

$$\frac{\Gamma \vdash_{\mathtt{B}} [\vec{m}'\rangle : !T.\mathsf{end}\rangle S_j}{\Gamma \vdash_{\mathtt{B}} [\vec{m}, j\rangle : !T.\mathsf{end}\rangle \oplus\{i : S_i\}_{i \in I}}\,[\textsc{typ-buf-sel}]$$

By IH_2, $\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle S_j$, so we derive the following:

$$\frac{\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}'\rangle : \mathsf{end}\rangle S_j}{\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}', j\rangle : \mathsf{end}\rangle \oplus\{i : S_i\}_{i \in I}}\,[\textsc{typ-buf-sel}]$$

Now, we can derive the typing of the structurally congruent configuration (omitting "TYP-" from rule labels):

$$\frac{\Gamma, \Delta_1 \vdash_{\mathtt{B}} [M, \vec{m}\rangle : \mathsf{end}\rangle S \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{end} \vdash_{\mathtt{M}} x : \mathsf{end}}\,[\textsc{var}] \quad \Delta_2 \vdash_{\mathtt{M}} N : U}{\Delta_2, x : \mathsf{end} \vdash_{\mathtt{M}} \mathtt{close}\, x; N : U}\,[\textsc{close}]}{\Delta_2, x : \mathsf{end} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge(\mathtt{close}\, x; N) : U}\,[\textsc{main}] \qquad \Lambda, y : \overline{S} \vdash^{\Diamond}_{\mathtt{C}} C : \mathbf{1}}{\Delta_2, \Lambda, x : \mathsf{end}, y : \overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge(\mathtt{close}\, x; N) \parallel C : U}\,[\textsc{par-r}]}{\Gamma, \Delta_1, \Delta_2, \Lambda \vdash^{\blacklozenge}_{\mathtt{C}} (\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\blacklozenge(\mathtt{close}\, x; N) \parallel C) : U}\,[\textsc{res}]$$

- Rule [RED-RECV]: $(\boldsymbol{\nu} x[\vec{m}, M\rangle y)(\hat{\mathcal{F}}[\mathtt{recv}\, y] \parallel C) \to_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[(M, y)] \parallel C)$.

  For this case, we apply induction on the structure of $\hat{\mathcal{F}}$. The inductive cases follow from the IH directly. We consider the base case ($\hat{\mathcal{F}} = \phi[\cdot]$). We apply inversion of typing,

w.l.o.g. assuming that $\phi = \blacklozenge$, and then derive the typing of the reduced configuration:

$$\pi \triangleq \dfrac{\dfrac{\dfrac{\dfrac{}{y : ?T.\overline{S} \vdash_{\mathtt{M}} y : ?T.\overline{S}}\,[\text{TYP-VAR}]}{y : ?T.\overline{S} \vdash_{\mathtt{M}} \mathtt{recv}\, y : T \times \overline{S}}\,[\text{TYP-RECV}]}{y : ?T.\overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{recv}\, y) : T \times \overline{S}}\,[\text{TYP-MAIN}] \qquad \Lambda, x : S' \vdash^{\lozenge}_{\mathtt{C}} C : \mathbf{1}}{\Lambda, x : S', y : ?T.\overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(\mathtt{recv}\, y) \parallel C : T \times \overline{S}}\,[\text{TYP-PAR}]$$

$$\dfrac{\dfrac{\Gamma \vdash_{\mathtt{M}} M : T \qquad \Delta \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S}{\Gamma, \Delta \vdash_{\mathtt{B}} [\vec{m}, M\rangle : S'\rangle !T.S}\,[\text{TYP-BUF-SEND}] \qquad \pi}{\Gamma, \Delta, \Lambda \vdash^{\blacklozenge}_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}, M\rangle y)(\blacklozenge\,(\mathtt{recv}\, y) \parallel C) : T \times \overline{S}}\,[\text{TYP-RES}]$$

$$\rightarrow_{\mathtt{C}}$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\mathtt{M}} M : T \qquad \dfrac{}{y : \overline{S} \vdash_{\mathtt{M}} y : \overline{S}}\,[\text{TYP-VAR}]}{\Gamma, y : \overline{S} \vdash_{\mathtt{M}} (M, y) : T \times \overline{S}}\,[\text{TYP-PAIR}]}{\Gamma, y : \overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(M, y) : T \times \overline{S}}\,[\text{TYP-MAIN}] \qquad \Lambda, x : S' \vdash^{\lozenge}_{\mathtt{C}} C : \mathbf{1}}{\Gamma, \Lambda, x : S', y : \overline{S} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\,(M, y) \parallel C : T \times \overline{S}}\,[\text{TYP-PAR}] \qquad \Delta \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S}{\Gamma, \Delta, \Lambda \vdash^{\blacklozenge}_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\blacklozenge\,(M, y) \parallel C) : T \times \overline{S}}\,[\text{TYP-RES}]$$

- Rule [RED-SELECT] is similar to the case of Rule [RED-SEND]:

$$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{select}\, \ell\, x] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\ell, \vec{m}\rangle y)(\mathcal{F}[x] \parallel C).$$

- Rule [RED-CASE] is similar to the case of Rule [RED-RECV]:

$$j \in I \implies (\boldsymbol{\nu} x[\vec{m}, j\rangle y)(\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_i\}_{i \in I}] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[M_j\ y] \parallel C)$$

- Rule [RED-CLOSE]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{close}\, x; M] \parallel C) \rightarrow_{\mathtt{C}} (\boldsymbol{\nu} \Box[\vec{m}\rangle y)(\mathcal{F}[M] \parallel C)$. By induction on the structure of $\mathcal{F}$. The inductive cases follows from the IH straightforwardly; we detail the base case ($\mathcal{F} = \phi\,[\cdot]$). Assume, w.l.o.g., that $\phi = \blacklozenge$. We first derive the typing of the configuration before reduction:

$$\dfrac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : \mathsf{end}\rangle S \qquad \dfrac{\dfrac{\dfrac{x : \mathsf{end} \vdash_{\mathtt{M}} x : \mathsf{end} \qquad \Delta \vdash_{\mathtt{M}} M : T}{\Delta, x : \mathsf{end} \vdash_{\mathtt{M}} \mathtt{close}\, x; M : T}\,[\text{TYP-CLOSE}]}{\Delta, x : \mathsf{end} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\ \mathtt{close}\, x; M : T}\,[\text{TYP-MAIN}] \qquad \Theta \vdash^{\lozenge}_{\mathtt{C}} C : \mathbf{1}}{\Delta, \Theta, x : \mathsf{end} \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge\ \mathtt{close}\, x; M \parallel C : T}\,[\text{TYP-PAR}]}{\Gamma' \vdash^{\blacklozenge}_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\blacklozenge\ \mathtt{close}\, x; M \parallel C) : T}\,[\text{TYP-RES/-BUF}]$$

  We prove that $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : \Box\rangle S$. The analysis depends on whether $y = \Box$.
  - If $y = \Box$, then $\vec{m} = \epsilon$, $S = \Box$, $\Gamma' = \Gamma, \Delta, \Theta$, and $\Gamma = \emptyset$, because $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : \mathsf{end}\rangle S$ must be derived as follows:

$$\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \Box}\,[\text{TYP-BUF-END-L}]$$

    The thesis holds as follows:

$$\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \Box\rangle \Box}\,[\text{TYP-BUF}]$$

  - If $y \neq \Box$, then $\Gamma', y : \overline{S} = \Gamma, \Delta, \Theta$. We apply induction on the size of $\vec{m}$ (IH_2):
    * If $\vec{m} = \epsilon$, then $\Gamma = \emptyset$, $S = \mathsf{end}$, because $\Gamma \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle S$ must be derived as follows:

$$\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \mathsf{end}}\,[\text{TYP-BUF}]$$

The thesis then holds as follows:

$$\frac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \Box\rangle \mathsf{end}} \text{[TYP-BUF-END-R]}$$

* If $\vec{m} = \vec{m}', N$, then $\Gamma = \Gamma_1, \Gamma_2$, and $S = !U.S'$, because $\Gamma \vdash_{\mathtt{B}} [\vec{m}', N\rangle : \mathsf{end}\rangle S$ must be derived as follows:

$$\frac{\Gamma_1 \vdash_{\mathtt{M}} N : U \qquad \Gamma_2 \vdash_{\mathtt{B}} [\vec{m}'\rangle : \mathsf{end}\rangle S'}{\Gamma_1, \Gamma_2 \vdash_{\mathtt{B}} [\vec{m}', N\rangle : \mathsf{end}\rangle !U.S'} \text{[TYP-BUF-SEND]}$$

By IH_2, $\Gamma_2 \vdash_{\mathtt{B}} [\vec{m}'\rangle : \Box\rangle S'$. The thesis then holds as follows:

$$\frac{\Gamma_1 \vdash_{\mathtt{M}} N : U \qquad \Gamma_2 \vdash_{\mathtt{B}} [\vec{m}'\rangle : \Box\rangle S'}{\Gamma_1, \Gamma_2 \vdash_{\mathtt{B}} [\vec{m}', N\rangle : \Box\rangle !U.S'} \text{[TYP-BUF-SEND]}$$

Now, we derive the typing of the reduced configuration:

$$\frac{\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : \Box\rangle S \qquad \dfrac{\dfrac{\Delta \vdash_{\mathtt{M}} M : T}{\Delta \vdash_{\mathtt{C}}^{\blacklozenge} \blacklozenge M : T} \text{[TYP-MAIN]} \qquad \Theta \vdash_{\mathtt{C}}^{\Diamond} C : \mathbf{1}}{\Delta, \Theta \vdash_{\mathtt{C}}^{\blacklozenge} \blacklozenge M \parallel C : T} \text{[TYP-PAR]}}{\Gamma' \vdash_{\mathtt{C}}^{\blacklozenge} (\boldsymbol{\nu} \Box[\vec{m}\rangle y)(\blacklozenge M \parallel C) : T} \text{[TYP-RES/-BUF]}$$

- Rule [RED-RES-NIL]: $(\boldsymbol{\nu} \Box[\epsilon\rangle \Box) C \to_{\mathtt{C}} C$. We have

$$\frac{\dfrac{}{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \Box\rangle \Box} \text{[TYP-BUF]} \qquad \Gamma \vdash_{\mathtt{C}}^{\phi} C : T}{\Gamma \vdash_{\mathtt{C}}^{\phi} (\boldsymbol{\nu} \Box[\epsilon\rangle \Box) C : T} \text{[TYP-RES/-RES-BUF]}$$

$$\to_{\mathtt{C}}$$

$$\Gamma \vdash_{\mathtt{C}}^{\phi} C : T$$

- Rule [RED-PAR-NIL]: $C \parallel \Diamond\,() \to_{\mathtt{C}} C$. We have the following:

$$\frac{\Gamma \vdash_{\mathtt{C}}^{\phi} C : T \qquad \dfrac{\dfrac{}{\emptyset \vdash_{\mathtt{M}} () : \mathbf{1}} \text{[TYP-UNIT]}}{\emptyset \vdash_{\mathtt{C}}^{\Diamond} \Diamond\,() : \mathbf{1}} \text{[TYP-CHILD]}}{\Gamma \vdash_{\mathtt{C}}^{\phi} C \parallel \Diamond\,() : T} \text{[TYP-PAR]}$$

$$\to_{\mathtt{C}}$$

$$\Gamma \vdash_{\mathtt{C}}^{\phi} C : T$$

□

A.3. **Translation: Type Preservation.** [Type Preservation for the Translation]

- If $\Gamma \vdash_{\mathtt{M}} M : T$, then $\vdash^* [\![M]\!] z :: (\!|\Gamma|\!), z : [\![T]\!]$;
- If $\Gamma \vdash_{\mathtt{C}}^{\phi} C : T$, then $\vdash^* [\![C]\!] z :: (\!|\Gamma|\!), z : [\![T]\!]$;
- If $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S$, then $\vdash^* [\![[\vec{m}\rangle]\!] a\rangle b :: (\!|\Gamma|\!), a : \overline{(\!|S'|\!)}, b : \overline{(\!|S|\!)}$.

Proof. By induction on the $\mathtt{LAST}^n$ typing derivation. It is sufficient to give the typing of the translations of typing rules in Figures 17 to 19 as follows; checking the derivations is straightforward.

[TYP-VAR]	$\vdash^* [\![x : T \vdash_{\mathtt{M}} x : T]\!] z$	$:: x : (\!|T|\!), z : [\![T]\!]$
[TYP-ABS]	$\vdash^* [\![\Gamma \vdash_{\mathtt{M}} \lambda x.M : T \multimap U]\!] z$	$:: (\!|\Gamma|\!), z : (\!|T|\!) \parr [\![U]\!] = [\![T \multimap U]\!]$
[TYP-APP]	$\vdash^* [\![\Gamma, \Delta \vdash_{\mathtt{M}} M\ N : U]\!] z$	$:: (\!|\Gamma|\!), (\!|\Delta|\!), z : [\![U]\!]$

[TYP-UNIT]	$\vdash^* [\![\emptyset \vdash_{\mathtt{M}} () : \mathbf{1}]\!] z$	$:: z : \bullet = [\![\mathbf{1}]\!]$
[TYP-PAIR]	$\vdash^* [\![\Gamma, \Delta \vdash_{\mathtt{M}} (M, N) : T \times U]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : (\bullet \parr [\![T]\!]) \otimes (\bullet \parr [\![U]\!]) = \overline{(\!\vert T\vert\!)} \otimes \overline{(\!\vert U\vert\!)} = [\![T \times U]\!]$
[TYP-SPLIT]	$\vdash^* [\![\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{let}\,(x, y) = M\,\mathtt{in}\,N : U]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![U]\!]$
[TYP-NEW]	$\vdash^* [\![\emptyset \vdash_{\mathtt{M}} \mathtt{new} : S \times \overline{S}]\!] z$	$:: z : \overline{(\!\vert S\vert\!)} \otimes \overline{(\!\vert \overline{S}\vert\!)} = [\![S \times \overline{S}]\!]$
[TYP-FORK]	$\vdash^* [\![\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{fork}\,M; N : T]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![T]\!]$
[TYP-CLOSE]	$\vdash^* [\![\Gamma, \Delta \vdash_{\mathtt{M}} \mathtt{close}\,M; N : T]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![T]\!]$
[TYP-SEND]	$\vdash^* [\![\Gamma \vdash_{\mathtt{M}} \mathtt{send}\,M\,N : S]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![S]\!]$
[TYP-RECV]	$\vdash^* [\![\Gamma \vdash_{\mathtt{M}} \mathtt{recv}\,M : T \times S]\!] z$	$:: (\!\vert\Gamma\vert\!), z : (\bullet \parr [\![T]\!]) \otimes (\bullet \parr [\![S]\!]) = \overline{(\!\vert T\vert\!)} \otimes \overline{(\!\vert S\vert\!)} = [\![T \times S]\!]$
[TYP-SEL]	$\vdash^* [\![\Gamma \vdash_{\mathtt{M}} \mathtt{select}\,j\,M : S_j]\!] z$	$:: (\!\vert\Gamma\vert\!), z : [\![S_j]\!]$
[TYP-CASE]	$\vdash^* [\![\Gamma \vdash_{\mathtt{M}} \mathtt{case}\,M\,\mathtt{of}\,\{i : N_i\}_{i\in I} : U]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![U]\!]$
[TYP-SUB]	$\vdash^* [\![\Gamma, \Delta \vdash_{\mathtt{M}} M\{\!\{N/x\}\!\} : U]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![U]\!]$
[TYP-MAIN]	$\vdash^* [\![\Gamma \vdash^{\blacklozenge}_{\mathtt{C}} \blacklozenge M : \hat{T}]\!] z$	$:: (\!\vert\Gamma\vert\!), z : [\![\hat{T}]\!]$
[TYP-CHILD]	$\vdash^* [\![\Gamma \vdash^{\lozenge}_{\mathtt{C}} \lozenge M : \mathbf{1}]\!] z$	$:: (\!\vert\Gamma\vert\!), z : \bullet = [\![\mathbf{1}]\!]$
[TYP-PAR] $(T_1 = \mathbf{1})$	$\vdash^* [\![\Gamma, \Delta \vdash^{\phi_1+\phi_2}_{\mathtt{C}} C \parallel D : T_2]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![T_2]\!]$
[TYP-PAR] $(T_2 = \mathbf{1})$	$\vdash^* [\![\Gamma, \Delta \vdash^{\phi_1+\phi_2}_{\mathtt{C}} C \parallel D : T_1]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![T_1]\!]$
[TYP-RES]	$\vdash^* [\![\Gamma' \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y) C : T]\!] z$	$:: (\!\vert\Gamma'\vert\!), z : [\![T]\!]$
[TYP-CONF-SUB]	$\vdash^* [\![\Gamma, \Delta \vdash^{\phi}_{\mathtt{C}} C\{\!\{M/x\}\!\} : U]\!] z$	$:: (\!\vert\Gamma\vert\!), (\!\vert\Delta\vert\!), z : [\![U]\!]$
[TYP-BUF] $(S' = !T.S)$	$\vdash^* [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S']\!] a\rangle b$	$:: a : \bullet \parr \bullet \otimes \overline{(\!\vert T\vert\!)} \parr \overline{(\!\vert S\vert\!)} = \bullet \parr [\![S']\!] = \overline{(\!\vert S'\vert\!)}$, $b : \bullet \parr \overline{(\!\vert T\vert\!)} \otimes \overline{(\!\vert \overline{S}\vert\!)} = \bullet \parr [\![?T.\overline{S}]\!] = \bullet \parr [\![\overline{S'}]\!] = \overline{(\!\vert \overline{S'}\vert\!)}$
[TYP-BUF] $(S' = \oplus\{i : S_i\}_{i\in I})$	$\vdash^* [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S']\!] a\rangle b$	$:: a : \bullet \parr \bullet \otimes \&\{i : \overline{(\!\vert S_i\vert\!)}\}_{i\in I} = \bullet \parr [\![S']\!] = \overline{(\!\vert S'\vert\!)}$, $b : \bullet \parr \oplus\{i : \overline{(\!\vert S_i\vert\!)}\}_{i\in I} = \bullet \parr [\![\&\{i : \overline{S_i}\}_{i\in I}]\!] = \bullet \parr [\![\overline{S'}]\!] = \overline{(\!\vert \overline{S'}\vert\!)}$
[TYP-BUF] $(S' \in \{?T.S, \&\{i : S_i\}_{i\in I}\})$	$\vdash^* [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S']\!] a\rangle b$	$:: a : \overline{\overline{(\!\vert S'\vert\!)}} = \overline{(\!\vert S'\vert\!)}, b : \overline{(\!\vert S'\vert\!)}$
[TYP-BUF] $(S' = \mathsf{end})$	$\vdash^* [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S']\!] a\rangle b$	$:: a : \bullet \parr \bullet \otimes \bullet = \overline{(\!\vert S'\vert\!)}$, $b : \bullet \parr \bullet \otimes \bullet = \overline{(\!\vert S'\vert\!)} = \overline{(\!\vert \overline{S'}\vert\!)}$
[TYP-BUF] $(S' = \square)$	$\vdash^* [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S']\!] a\rangle b$	$:: a : \bullet = \overline{(\!\vert S'\vert\!)}, b : \bullet = \overline{(\!\vert S'\vert\!)} = \overline{(\!\vert \overline{S'}\vert\!)}$

[TYP-BUF-SEND] $\vdash^* [\![\Gamma, \Delta \vdash_{\mathtt{B}} [\vec{m}, M\rangle : S'\rangle !T.S]\!]a\rangle b$ $:: (\!|\Gamma|\!), (\!|\Delta|\!), a : \overline{(\!|S'|\!)}, b : \bullet \parr \overline{(\!|T|\!)} \otimes \overline{(\!|S|\!)} = \bullet \parr [\![?T.\overline{S}]\!] = \bullet \parr [\![!T.S]\!] = \overline{(\!|!T.S|\!)}$

[TYP-BUF-SEL] $\vdash^* [\![\Gamma \vdash_{\mathtt{B}} [\vec{m}, j\rangle : S'\rangle \oplus\{i : S_i\}_{i\in I}]\!]a\rangle b$ $:: (\!|\Gamma|\!), a : \overline{(\!|S'|\!)}, b : \bullet \parr \oplus\{i : \overline{(\!|S_i|\!)}\}_{i\in I} = \bullet \parr [\![\&\{i : \overline{S_i}\}_{i\in I}]\!] = \bullet \parr \overline{[\![\oplus\{i : S_i\}_{i\in I}]\!]} = \overline{(\!|\oplus\{i : S_i\}_{i\in I}|\!)}$

[TYP-BUF-END-L] $\vdash^* [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \square]\!]a\rangle b$ $:: a : \bullet \parr \bullet \otimes \bullet = \overline{(\!|\mathsf{end}|\!)}, b : \bullet = \overline{(\!|\square|\!)} = \overline{(\!|\square|\!)}$

[TYP-BUF-END-R] $\vdash^* [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \square\rangle \mathsf{end}]\!]a\rangle b$ $:: a : \bullet = \overline{(\!|\square|\!)}, b : \bullet \parr \bullet \otimes \bullet = \overline{(\!|\mathsf{end}|\!)} = \overline{(\!|\mathsf{end}|\!)}$ □

A.4. **Operational Correspondence.** Appendices A.4.1 and A.4.2 prove completeness and soundness, respectively. Both results rely on the following lemma, which ensures that $\mathtt{LAST}^n$ contexts translate to evaluation contexts in APCP.

- $[\![\mathcal{R}[M]]\!]z = \mathcal{E}[[\![M]\!]z']$ for some $\mathcal{E}, z'$;
- $[\![\mathcal{F}[M]]\!]z = \mathcal{E}[[\![M]\!]z']$ for some $\mathcal{E}, z'$;
- $[\![\mathcal{G}[C]]\!]z = \mathcal{E}[[\![C]\!]z']$ for some $\mathcal{E}, z'$;
- $x \notin \mathrm{fv}(C)$ implies $x \notin \mathrm{fn}([\![C]\!]z)$;
- $x \in \mathrm{fv}(C)$ implies $[\![C\{y/x\}]\!]z = [\![C]\!]z\{y/x\}$.

Proof (Sketch). By induction on the structure of the contexts. □

A.4.1. *Completeness.* Here we prove the completeness of the translation. The proof relies on the following intermediate results:

- Appendix A.4.1 prove that the translation preserves structural congruence for terms and configurations, respectively.
- Appendix A.4.1 then shows that the translation is complete with respect to term reduction.
- Finally, we prove completeness (Section 5.2).

[Preservation of Structural Congruence for Terms] Given $\Gamma \vdash_{\mathtt{M}} M : T$, if $M \equiv_{\mathtt{M}} N$, then $[\![M]\!]z \equiv [\![N]\!]z$.

Proof. By induction on the derivation of $M \equiv_{\mathtt{M}} N$ (IH_1). The inductive cases follow from IH_1 and Section 5.2 directly. We detail the (only) base case of Rule [SC-SUB-EXT]: $x \notin \mathrm{fn}(\mathcal{R})$ implies $(\mathcal{R}[M])\{\!|N/x|\!\} \equiv_{\mathtt{M}} \mathcal{R}[M\{\!|N/x|\!\}]$.

The analysis is by induction on the structure of $\mathcal{R}$ (IH_2), assuming $x \notin \mathrm{fn}(\mathcal{R})$. The base case where $\mathcal{R} = [\cdot]$ is immediate. We detail one representative inductive case: $\mathcal{R} = \mathcal{R}'\ M'$.

The thesis holds as follows:

$$
\begin{aligned}
&[\![(\mathcal{R}'[M]\ M')\{\!\{N/x\}\!\}]\!]z \\
&= (\boldsymbol{\nu} x a_1)\big((\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)([\![\mathcal{R}'[M]]\!]a_2 \mid b_2[c_2, z] \mid d_2(\_, e_2); [\![M']\!]e_2) \mid a_1(\_, b_1); [\![N]\!]b_1\big) \\
&\equiv (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)\big((\boldsymbol{\nu} x a_1)([\![\mathcal{R}'[M]]\!]a_2 \mid a_1(\_, b_1); [\![N]\!]b_1) \mid b_2[c_2, z] \mid d_2(\_, e_2); [\![M']\!]e_2\big) \\
&= (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)\big([\![(\mathcal{R}'[M])\{\!\{N/x\}\!\}]\!]a_2 \mid b_2[c_2, z] \mid d_2(\_, e_2); [\![M']\!]e_2\big) \\
&\equiv (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)\big([\![\mathcal{R}'[M\{\!\{N/x\}\!\}]]\!]a_2 \mid b_2[c_2, z] \mid d_2(\_, e_2); [\![M']\!]e_2\big) && (\mathrm{IH}_2) \\
&= [\![(\mathcal{R}'[M\{\!\{N/x\}\!\}])\ M']\!]z = [\![\mathcal{R}[M\{\!\{N/x\}\!\}]]\!]z && 1
\end{aligned}
$$

□

[Preservation of Structural Congruence for Configurations] Given $\Gamma \vdash^{\phi}_{\mathtt{C}} C : T$, if $C \equiv_{\mathtt{C}} D$, then $[\![C]\!]z \equiv [\![D]\!]z$.

Proof. By induction on the derivation of $C \equiv_{\mathtt{C}} D$ (IH_1). The inductive cases follow from IH_1 and Section 5.2 straightforwardly. We detail the base cases, induced by the ten rules in Figure 21:

- Rule [SC-TERM-SC]: $M \equiv_{\mathtt{M}} M'$ implies $\phi\, M \equiv_{\mathtt{C}} \phi\, M'$. We have $[\![\phi\, M]\!]z = [\![M]\!]z$ and $[\![\phi\, M']\!]z = [\![M']\!]z$. By the assumption that $M \equiv_{\mathtt{M}} M'$ and Appendix A.4.1, $[\![M]\!]z \equiv [\![M']\!]z$. The thesis follows immediately.
- Rule [SC-RES-SWAP]: $(\boldsymbol{\nu} x[\epsilon\rangle y)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} y[\epsilon\rangle x)C$. Both directions are analogous; we detail the left to right direction. We first infer the typing of the left configuration:

$$\frac{\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S \qquad \Gamma \vdash^{\phi}_{\mathtt{C}} C : T}{\Gamma' \vdash^{\phi}_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)C : T}\ [\text{TYP-RES/-BUF}]$$

  Here, $\Gamma' = \Gamma \setminus x : S', y : \overline{S}$. The analysis depends on whether $x = \square$ and/or $y = \square$. In each case, we show that

$$[\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S]\!]a\rangle b = [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S}\rangle \overline{S'}]\!]b\rangle a : \tag{A.1}$$

  - Case $x = y = \square$, or $x \neq \square$ and $x \neq \square$. Either way, $S' = S$. If $S' = \square$, both translations are $\mathbf{0}$, from which the thesis follows immediately. Otherwise, the thesis follows by induction on the structure of S'; clearly, the resulting translations are exactly the same.
  - Case $x = \square$ and $y \neq \square$, or $x \neq \square$ and $y = \square$. W.l.o.g., assume the former. Then $S' = \square$ and $S = \mathsf{end}$. The thesis then holds as follows:

$$
\begin{aligned}
[\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \square\rangle \mathsf{end}]\!]a\rangle b &= b(\_, c); c[\_, \_] \\
&= [\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle \square]\!]b\rangle c
\end{aligned}
$$

  The thesis then holds as follows:

$$
\begin{aligned}
[\![(\boldsymbol{\nu} x[\epsilon\rangle y)C]\!]z &= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S'\rangle S]\!]a\rangle b \mid [\![C]\!]z) \\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S}\rangle \overline{S'}]\!]b\rangle a \mid [\![C]\!]z) && (\mathrm{A.1}) \\
&\equiv (\boldsymbol{\nu} b y)(\boldsymbol{\nu} a x)([\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \overline{S}\rangle \overline{S'}]\!]b\rangle a \mid [\![C]\!]z) \\
&= [\![(\boldsymbol{\nu} y[\epsilon\rangle x)C]\!]z
\end{aligned}
$$

- Rule [SC-RES-COMM]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\boldsymbol{\nu} z[\vec{n}\rangle w)C \equiv_{\mathtt{C}} (\boldsymbol{\nu} z[\vec{n}\rangle w)(\boldsymbol{\nu} x[\vec{m}\rangle y)C$. The thesis holds as follows:

$$\begin{aligned}
[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(\boldsymbol{\nu} z[\vec{n}\rangle w)C]\!]z &= (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)\big([\![[\vec{m}\rangle]\!]a_1\rangle b_1 \mid (\boldsymbol{\nu} a_2 z)(\boldsymbol{\nu} b_2 w)([\![[\vec{n}\rangle]\!]a_2\rangle b_2 \mid [\![C]\!]z)\big)\\
&\equiv (\boldsymbol{\nu} a_2 z)(\boldsymbol{\nu} b_2 w)\big([\![[\vec{n}\rangle]\!]a_2\rangle b_2 \mid (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)([\![[\vec{m}\rangle]\!]a_1\rangle b_1 \mid [\![C]\!]z)\big)\\
&= [\![(\boldsymbol{\nu} z[\vec{n}\rangle w)(\boldsymbol{\nu} x[\vec{m}\rangle y)C]\!]z
\end{aligned}$$

- Rule [SC-RES-EXT]: $x, y \notin \mathrm{fv}(C)$ implies $(\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D) \equiv_{\mathtt{C}} C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y)D$. The analysis depends on which of C, D is a child thread; w.l.o.g., assume that C is. Assume the condition; by Section 5.2, then $x, y \notin \mathrm{fn}([\![C]\!]\_)$ $(*)$. The thesis holds as follows:

$$\begin{aligned}
[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(C \parallel D)]\!]z &= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![[\vec{m}\rangle]\!]a\rangle b \mid (\boldsymbol{\nu}__)[\![C]\!]_ \mid [\![D]\!]z)\\
&\equiv (\boldsymbol{\nu}__)[\![C]\!]_ \mid (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![[\vec{m}\rangle]\!]a\rangle b \mid [\![D]\!]z) \qquad (*)\\
&= [\![C \parallel (\boldsymbol{\nu} x[\vec{m}\rangle y)D]\!]z
\end{aligned}$$

- Rule [SC-PAR-COMM]: $C \parallel D \equiv_{\mathtt{C}} D \parallel C$. The analysis depends on which of C, D is a child thread; w.l.o.g., assume that C is. The thesis holds as follows:

$$\begin{aligned}
[\![C \parallel D]\!]z &= (\boldsymbol{\nu}__)[\![C]\!]_ \mid [\![D]\!]z\\
&\equiv [\![D]\!]z \mid (\boldsymbol{\nu}__)[\![C]\!]_\\
&= [\![D \parallel C]\!]z
\end{aligned}$$

- Rule [SC-PAR-ASSOC]: $C \parallel (D \parallel E) \equiv_{\mathtt{C}} (C \parallel D) \parallel E$. The analysis depends on which of C, D, E are child threads; w.l.o.g., assume that C, D are. The thesis holds as follows:

$$\begin{aligned}
[\![C \parallel (D \parallel E)]\!]z &= (\boldsymbol{\nu}__)[\![C]\!]_ \mid ((\boldsymbol{\nu}__)[\![D]\!]__ \mid [\![E]\!]z)\\
&\equiv (\boldsymbol{\nu}__)((\boldsymbol{\nu}__)[\![C]\!]_ \mid [\![D]\!]__) \mid [\![E]\!]z\\
&= [\![(C \parallel D) \parallel E]\!]z
\end{aligned}$$

- Rule [SC-CONF-SUB]: $\phi\,(M\{\!\{N/x\}\!\}) \equiv_{\mathtt{C}} (\phi\, M)\{\!\{N/x\}\!\}$. The thesis holds as follows:

$$\begin{aligned}
[\![\phi\,(M\{\!\{N/x\}\!\})]\!]z &= (\boldsymbol{\nu} x a)([\![M]\!]z \mid a(_, b); [\![N]\!]b)\\
&= (\boldsymbol{\nu} x a)([\![\phi\, M]\!]z \mid a(_, b); [\![N]\!]b)\\
&= [\![(\phi\, M)\{\!\{N/x\}\!\}]\!]z
\end{aligned}$$

- Rule [SC-CONF-SUB-EXT]: $x \notin \mathrm{fv}(\mathcal{G})$ implies $(\mathcal{G}[C])\{\!\{M/x\}\!\} \equiv_{\mathtt{C}} \mathcal{G}[C\{\!\{M/x\}\!\}]$. By Section 5.2, for any D, $[\![\mathcal{G}[D]]\!]z = \mathcal{E}[[\![D]\!]z']$ for some $\mathcal{E}, z'$ $(*_1)$. Assume the condition; by Section 5.2, then $x \notin \mathrm{fn}(\mathcal{E})$ $(*_2)$. The thesis holds as follows:

$$\begin{aligned}
[\![(\mathcal{G}[C])\{\!\{M/x\}\!\}]\!]z &= (\boldsymbol{\nu} x a)([\![\mathcal{G}[C]]\!]z \mid a(_, b); [\![M]\!]b)\\
&= (\boldsymbol{\nu} x a)(\mathcal{E}[[\![C]\!]z'] \mid a(_, b); [\![M]\!]b) \qquad (*_1)\\
&\equiv \mathcal{E}[(\boldsymbol{\nu} x a)([\![C]\!]z' \mid a(_, b); [\![M]\!]b)] \qquad (*_2)\\
&= \mathcal{E}[[\![C\{\!\{M/x\}\!\}]\!]z']\\
&= [\![\mathcal{G}[C\{\!\{M/x\}\!\}]]\!]z \qquad (*_1)
\end{aligned}$$

□

[Completeness of Reduction for Terms] Given $\Gamma \vdash_{\mathtt{M}} M : T$, if $M \to_{\mathtt{M}} N$, then $[\![M]\!]z \to^* [\![N]\!]z$.

Proof. By induction on the derivation of $M \to_{\mathtt{M}} N$. We detail each rule:

- Rule [red-lam]: $(\lambda x.M)\ N \to_{\mathtt{M}} M\{\!\{N/x\}\!\}$. The thesis holds as follows:

$$\begin{aligned}
[\![(\lambda x.M)\ N]\!]z &= (\boldsymbol{\nu} a_1 b_1)(\boldsymbol{\nu} c_1 d_1)(a_1(x,a_2); [\![M]\!]a_2 \mid b_1[c_1,z] \mid d_1(_,e_1); [\![N]\!]e_1) \\
&\to (\boldsymbol{\nu} c_1 d_1)([\![M]\!]z\{c_1/x\} \mid d_1(_,e_1); [\![N]\!]e_1) \\
&\equiv (\boldsymbol{\nu} x d_1)([\![M]\!]z \mid d_1(_,e_1); [\![N]\!]e_1) \\
&= [\![M\{\!\{N/x\}\!\}]\!]z
\end{aligned}$$

- Rule [red-pair]: $\mathtt{let}\,(x,y) = (M_1,M_2)\,\mathtt{in}\,N \to_{\mathtt{M}} N\{\!\{M_1/x, M_2/y\}\!\}$. The thesis holds as follows:

$$\begin{aligned}
&[\![\mathtt{let}\,(x,y) = (M_1,M_2)\,\mathtt{in}\,N]\!]z \\
&= (\boldsymbol{\nu} a_1 b_1)\big(a_1(x,y); [\![N]\!]z \\
&\qquad \mid (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)(b_1[a_2,c_2] \\
&\qquad\qquad \mid b_2(_,e_2); [\![M_1]\!]e_2 \\
&\qquad\qquad \mid d_2(_,f_2); [\![M_2]\!]f_2)\big) \\
&\to (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)([\![N]\!]z\{a_2/x, c_2/y\} \\
&\qquad\qquad \mid b_2(_,e_2); [\![M_1]\!]e_2 \mid d_2(_,f_2); [\![M_2]\!]f_2) \\
&\equiv (\boldsymbol{\nu} y d_2)\big((\boldsymbol{\nu} x b_2)([\![N]\!]z \mid b_2(_,e_2); [\![M_1]\!]e_2) \mid d_2(_,f_2); [\![M_2]\!]f_2\big) \\
&= [\![N\{\!\{M_1/x, M_2/y\}\!\}]\!]z
\end{aligned}$$

- Rule [red-name-sub]: $x\{\!\{M/x\}\!\} \to_{\mathtt{M}} M$. The thesis holds as follows:

$$\begin{aligned}
[\![x\{\!\{M/x\}\!\}]\!]z &= (\boldsymbol{\nu} x a)(x[_,z] \mid a(_,b); [\![M]\!]b) \\
&\to [\![M]\!]z
\end{aligned}$$

- Rule [red-lift]: $M \to_{\mathtt{M}} N$ implies $\mathcal{R}[M] \to_{\mathtt{M}} \mathcal{R}[N]$. By Section 5.2, for any L, $[\![\mathcal{R}[L]]\!]z = \mathcal{E}[[\![L]\!]z']$ for some $\mathcal{E}, z'$ $(*_1)$. Assume the condition; by the IH, $[\![M]\!]z' \to^* [\![N]\!]z'$ $(*_2)$. The thesis holds as follows:

$$\begin{aligned}
[\![\mathcal{R}[M]]\!]z &= \mathcal{E}[[\![M]\!]z'] && (*_1) \\
&\to^* \mathcal{E}[[\![N]\!]z'] && (*_2) \\
&= [\![\mathcal{R}[N]]\!]z && (*_1)
\end{aligned}$$

- Rule [red-lift-sc]: $M \equiv_{\mathtt{M}} M'$, $M' \to_{\mathtt{M}} N'$, and $N' \equiv_{\mathtt{M}} N$ imply $M \to_{\mathtt{M}} N$. Assume the conditions. By Appendix A.4.1, $[\![M]\!]z \equiv [\![M']\!]z$ $(*_1)$ and $[\![N']\!]z \equiv [\![N]\!]z$ $(*_2)$. By the IH, $[\![M']\!]z \to^* [\![N']\!]z$ $(*_3)$. The thesis holds as follows:

$$\begin{aligned}
[\![M]\!]z &\equiv [\![M']\!]z && (*_1) \\
&\to^* [\![N']\!]z && (*_3) \\
&\equiv [\![N]\!]z && (*_2)
\end{aligned}$$

□

[Completeness] Given $\Gamma \vdash^{\phi}_{\mathtt{C}} C : T$, if $C \to_{\mathtt{C}} D$, then $[\![C]\!]z \to^* [\![D]\!]z$.

Proof. By induction on the derivation of $C \to_{\mathtt{C}} D$. We detail every rule:

- Rule [RED-NEW]: $\mathcal{F}[\mathtt{new}] \to_{\mathtt{C}} (\boldsymbol{\nu} x[\epsilon\rangle y)(\mathcal{F}[(x,y)])$. By Section 5.2, for any M, $[\![\mathcal{F}[M]]\!]z = \mathcal{E}[[\![M]\!]z']$ for some $\mathcal{E}, z'$ $(*)$. The thesis holds as follows:

$$
\begin{aligned}
[\![\mathcal{F}[\mathtt{new}]]\!]z &= \mathcal{E}[[\![\mathtt{new}]\!]z'] && (*)\\
&= \mathcal{E}[(\boldsymbol{\nu} ab)\big(a[\_,z'] \mid b(\_,c); (\boldsymbol{\nu} dx)(\boldsymbol{\nu} ey)([\![[\epsilon\rangle]\!]d\rangle e \mid [\![(x,y)]\!]c)\big)]\\
&\to \mathcal{E}[(\boldsymbol{\nu} dx)(\boldsymbol{\nu} ey)([\![[\epsilon\rangle]\!]d\rangle e \mid [\![(x,y)]\!]z')]\\
&\equiv (\boldsymbol{\nu} dx)(\boldsymbol{\nu} ey)([\![[\epsilon\rangle]\!]d\rangle e \mid \mathcal{E}[[\![(x,y)]\!]z'])\\
&= (\boldsymbol{\nu} dx)(\boldsymbol{\nu} ey)([\![[\epsilon\rangle]\!]d\rangle e \mid [\![\mathcal{F}[(x,y)]]\!]z) && (*)\\
&= [\![(\boldsymbol{\nu} x[\epsilon\rangle y)(\mathcal{F}[(x,y)])]\!]z
\end{aligned}
$$

- Rule [RED-FORK]: $\hat{\mathcal{F}}[\mathtt{fork}\, M; N] \to_{\mathtt{C}} \hat{\mathcal{F}}[N] \parallel \Diamond M$. By Section 5.2, for any L, $[\![\mathcal{F}[L]]\!]z = \mathcal{E}[[\![L]\!]z']$ for some $\mathcal{E}, z'$ $(*_1)$. Moreover, since $\hat{\mathcal{F}}$ does not have its hole under an explicit substitution, it does not capture any free variables of M; hence, by Section 5.2, $\mathcal{E}$ does not capture any free names of $[\![M]\!]u$ for any u $(*_2)$. The thesis holds as follows:

$$
\begin{aligned}
[\![\hat{\mathcal{F}}[\mathtt{fork}\, M; N]]\!]z &= \mathcal{E}[[\![\mathtt{fork}\, M; N]\!]z'] && (*_1)\\
&= \mathcal{E}[(\boldsymbol{\nu} ab)\big(a[\_,z'] \mid b(\_,c); ((\boldsymbol{\nu} \_\_)[\![M]\!]\_ \mid [\![N]\!]c)\big)]\\
&\to \mathcal{E}[(\boldsymbol{\nu} \_\_)[\![M]\!]\_ \mid [\![N]\!]z']\\
&\equiv \mathcal{E}[[\![N]\!]z'] \mid (\boldsymbol{\nu} \_\_)[\![M]\!]\_ && (*_2)\\
&= [\![\hat{\mathcal{F}}[N]]\!]z \mid (\boldsymbol{\nu} \_\_)[\![M]\!]\_ && (*_1)\\
&= [\![\hat{\mathcal{F}}[N] \parallel \Diamond M]\!]z
\end{aligned}
$$

- Rule [RED-SEND]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[\mathtt{send}\, M\, x] \parallel C) \to_{\mathtt{C}} (\boldsymbol{\nu} x[M,\vec{m}\rangle y)(\hat{\mathcal{F}}[x] \parallel C)$. W.l.o.g., assume C is a child thread. By Section 5.2, for any L, $[\![\mathcal{F}[L]]\!]z = \mathcal{E}_1\big[[\![L]\!]z'\big]$ for some $\mathcal{E}_1, z'$ $(*_1)$. Moreover, since $\hat{\mathcal{F}}$ does not have its hole under an explicit substitution, it does not capture any free variables of M; hence, by Section 5.2, $\mathcal{E}_1$ does not capture any free names of $[\![M]\!]u$ for any u $(*_2)$. By inversion of typing, $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S_1\rangle S$ where $S_1 = !T.S_2$ $(*_3)$. By Section 5.2, $[\![\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S_1\rangle S]\!]a\rangle b = \mathcal{E}_2\big[[\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S_1\rangle S_1]\!]a\rangle c\big]$ $(*_4)$ and $[\![\Gamma, \Delta \vdash_{\mathtt{B}} [M,\vec{m}\rangle : S_1\rangle S]\!]a\rangle b = \mathcal{E}_2\big[[\![\Delta \vdash_{\mathtt{B}} [M\rangle : S_1\rangle S_1]\!]a\rangle c\big]$ $(*_5)$ for some $\mathcal{E}_2, c$. Below, we

omit types from translations of buffers. The thesis holds as follows:

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[\mathtt{send}\, M\, x] \parallel C)]\!]z \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!]a\rangle b \mid [\![\hat{\mathcal{F}}[\mathtt{send}\, M\, x]]\!]z \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\mathcal{E}_2[[\![[\epsilon\rangle]\!]a\rangle c] \mid \mathcal{E}_1[[\![\mathtt{send}\, M\, x]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_1,*_4) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[a(\_,c_1); (\boldsymbol{\nu} d_1e_1)(c_1[\_,d_1] \mid e_1(f_1,g_1); (\boldsymbol{\nu} h_1k_1)(c(\_,l_1); l_1[f_1,h_1] \mid [\![[\epsilon\rangle]\!]g_1\rangle k_1))] \\
&\quad \mid \mathcal{E}_1[(\boldsymbol{\nu} a_2b_2)(\boldsymbol{\nu} c_2d_2)(a_2(\_,e_2); [\![M]\!]e_2 \mid x[\_,c_2] \mid d_2(\_,f_2); (\boldsymbol{\nu} g_2h_2)(f_2[b_2,g_2] \mid h_2[\_,z']))] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_3) \\
&\to (\boldsymbol{\nu} c_2d_2)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[(\boldsymbol{\nu} d_1e_1)(c_2[\_,d_1] \mid e_1(f_1,g_1); (\boldsymbol{\nu} h_1k_1)(c(\_,l_1); l_1[f_1,h_1] \mid [\![[\epsilon\rangle]\!]g_1\rangle k_1))] \\
&\quad \mid \mathcal{E}_1[(\boldsymbol{\nu} a_2b_2)(a_2(\_,e_2); [\![M]\!]e_2 \mid d_2(\_,f_2); (\boldsymbol{\nu} g_2h_2)(f_2[b_2,g_2] \mid h_2[\_,z']))] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\to (\boldsymbol{\nu} e_1d_1)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[e_1(f_1,g_1); (\boldsymbol{\nu} h_1k_1)(c(\_,l_1); l_1[f_1,h_1] \mid [\![[\epsilon\rangle]\!]g_1\rangle k_1)] \\
&\quad \mid \mathcal{E}_1[(\boldsymbol{\nu} a_2b_2)(a_2(\_,e_2); [\![M]\!]e_2 \mid (\boldsymbol{\nu} g_2h_2)(d_1[b_2,g_2] \mid h_2[\_,z']))] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\to (\boldsymbol{\nu} b_2a_2)(\boldsymbol{\nu} g_2h_2)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[(\boldsymbol{\nu} h_1k_1)(c(\_,l_1); l_1[b_2,h_1] \mid [\![[\epsilon\rangle]\!]g_2\rangle k_1)] \\
&\quad \mid \mathcal{E}_1[a_2(\_,e_2); [\![M]\!]e_2 \mid h_2[\_,z']] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\equiv (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(&& (*_2) \\
&\quad \mathcal{E}_2[(\boldsymbol{\nu} a_2b_2)(\boldsymbol{\nu} h_1k_1)(a_2(\_,e_2); [\![M]\!]e_2 \mid c(\_,l_1); l_1[b_2,h_1] \mid [\![[\epsilon\rangle]\!]a\rangle k_1)] \\
&\quad \mid \mathcal{E}_1[x[\_,z']] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\mathcal{E}_2[[\![[M\rangle]\!]a\rangle c] \mid \mathcal{E}_1[[\![x]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[M,\vec{m}\rangle]\!]a\rangle b \mid [\![\hat{\mathcal{F}}[x]]\!]z \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_1,*_5) \\
&= [\![(\boldsymbol{\nu} x[M,\vec{m}\rangle y)(\hat{\mathcal{F}}[x] \parallel C)]\!]z
\end{aligned}
$$

- Rule [RED-RECV]: $(\boldsymbol{\nu} x[\vec{m}, M\rangle y)(\hat{\mathcal{F}}[\mathtt{recv}\, y] \parallel C) \to_{\mathtt{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[(M,y)] \parallel C)$. W.l.o.g., assume C is a child thread. By Section 5.2, for any L, $[\![\mathcal{F}[L]]\!]z = \mathcal{E}[[\![L]\!]z']$ for some $\mathcal{E}, z'$ $(*_1)$. Moreover, since $\hat{\mathcal{F}}$ does not have its hole under an explicit substitution, it does not capture any free variables of M; hence, by Section 5.2, $\mathcal{E}$ does not capture any free names of $[\![M]\!]u$ for any u $(*_2)$. The thesis holds as follows:

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x[\vec{m}, M\rangle y)(\hat{\mathcal{F}}[\mathtt{recv}\, y] \parallel C)]\!]z \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}, M\rangle]\!]a\rangle b \mid [\![\hat{\mathcal{F}}[\mathtt{recv}\, y]]\!]z \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}, M\rangle]\!]a\rangle b \mid \mathcal{E}[[\![\mathtt{recv}\, y]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_1) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\\
&\quad (\boldsymbol{\nu} c_1d_1)(\boldsymbol{\nu} e_1f_1)(c_1(\_,g_1); [\![M]\!]g_1 \mid b(\_,h_1); h_1[d_1,e_1] \mid [\![[\vec{m}\rangle]\!]a\rangle f_1) \\
&\quad \mid \mathcal{E}[(\boldsymbol{\nu} a_2b_2)(y[\_,a_2] \mid b_2(c_2,d_2); (\boldsymbol{\nu} e_2,f_2)(z'[c_2,e_2] \mid f_2(\_,g_2); d_2[\_,g_2]))] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_)
\end{aligned}
$$

$$
\begin{aligned}
&\to (\boldsymbol{\nu} ax)(\boldsymbol{\nu} a_2 b_2)(\\
&\quad (\boldsymbol{\nu} c_1 d_1)(\boldsymbol{\nu} e_1 f_1)(c_1(\_, g_1); [\![M]\!] g_1 \mid a_2[d_1, e_1] \mid [\![[\vec{m}\rangle]\!] a\rangle f_1)\\
&\quad \mid \mathcal{E}\big[b_2(c_2, d_2); (\boldsymbol{\nu} e_2, f_2)(z'[c_2, e_2] \mid f_2(\_, g_2); d_2[\_, g_2])\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\to (\boldsymbol{\nu} ax)(\boldsymbol{\nu} d_1 c_1)(\boldsymbol{\nu} f_1 e_1)(\\
&\quad c_1(\_, g_1); [\![M]\!] g_1 \mid [\![[\vec{m}\rangle]\!] a\rangle f_1\\
&\quad \mid \mathcal{E}\big[(\boldsymbol{\nu} e_2, f_2)(z'[d_1, e_2] \mid f_2(\_, g_2); e_1[\_, g_2])\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\equiv (\boldsymbol{\nu} ax)(\boldsymbol{\nu} f_1 e_1)(\qquad (*_2)\\
&\quad [\![[\vec{m}\rangle]\!] a\rangle f_1\\
&\quad \mid \mathcal{E}\big[(\boldsymbol{\nu} d_1 c_1)(\boldsymbol{\nu} e_2, f_2)(z'[d_1, e_2] \mid c_1(\_, g_1); [\![M]\!] g_1 \mid f_2(\_, g_2); e_1[\_, g_2])\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\equiv (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\\
&\quad [\![[\vec{m}\rangle]\!] a\rangle b\\
&\quad \mid \mathcal{E}\big[(\boldsymbol{\nu} d_1 c_1)(\boldsymbol{\nu} e_2, f_2)(z'[d_1, e_2] \mid c_1(\_, g_1); [\![M]\!] g_1 \mid f_2(\_, g_2); y[\_, g_2])\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!] a\rangle b \mid \mathcal{E}\big[[\![(M, y)]\!] z'\big] \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!] a\rangle b \mid [\![\hat{\mathcal{F}}[(M, y)]]\!] z \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_) \qquad (*_1)\\
&= [\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}[(M, y)] \parallel C)]\!] z
\end{aligned}
$$

- Rule [RED-SELECT]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{select}\,\ell\,x] \parallel C) \to_{\mathtt{C}} (\boldsymbol{\nu} x[\ell, \vec{m}\rangle y)(\mathcal{F}[x] \parallel C)$. W.l.o.g., assume C is a child thread. By Section 5.2, for any L, $[\![\mathcal{F}[L]]\!] z = \mathcal{E}_1\big[[\![L]\!] z'\big]$ for some $\mathcal{E}_1, z'$ $(*_1)$. By inversion of typing, $\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S_1\rangle S$ where $S_1 = \oplus\{i : S_i\} \cup \{\ell : S_2\}$ $(*_2)$. By Section 5.2,

$$[\![\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : S_1\rangle S]\!] a\rangle b = \mathcal{E}_2\big[[\![\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : S_1\rangle S_1]\!] a\rangle c\big] \qquad (*_3)$$

and

$$[\![\Gamma \vdash_{\mathtt{B}} [\ell, \vec{m}\rangle : S_0\rangle S]\!] a\rangle b = \mathcal{E}_2\big[[\![\emptyset \vdash_{\mathtt{B}} [\ell\rangle : S_1\rangle S_1]\!] a\rangle c\big] \qquad (*_4)$$

for some $\mathcal{E}_2, c$. Below, we omit types from translations of buffers. The thesis holds as follows:

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{select}\,\ell\,x] \parallel C)]\!] z\\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!] a\rangle b \mid [\![\mathcal{F}[\mathtt{select}\,\ell\,x]]\!] z \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\mathcal{E}_2\big[[\![[\epsilon\rangle]\!] a\rangle c\big] \mid \mathcal{E}_1\big[[\![\mathtt{select}\,\ell\,x]\!] z'\big] \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_) \qquad (*_1, *_3)\\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2\Big[a(\_, c_1); (\boldsymbol{\nu} d_1 e_1)\big(c_1[\_, d_1] \mid e_1(f_1) \triangleright \{i : \ldots\}_{i \in I} \cup \{\ell : (\boldsymbol{\nu} g_1 h_1) \begin{pmatrix} c(\_, k_1); k_1[g_1] \triangleleft \ell \\ \mid [\![[\epsilon\rangle]\!] f_1\rangle h_1 \end{pmatrix}\}\big)\Big]\\
&\quad \mid \mathcal{E}_1\big[(\boldsymbol{\nu} a_2 b_2)\big(x[\_, a_2] \mid b_2(\_, c_2); (\boldsymbol{\nu} d_2 e_2)(c_2[d_2] \triangleleft \ell \mid e_2[\_, z'])\big)\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\qquad (*_2)\\
&\to (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2\Big[(\boldsymbol{\nu} d_1 e_1)\big(a_2[\_, d_1] \mid e_1(f_1) \triangleright \{i : \ldots\}_{i \in I} \cup \{\ell : (\boldsymbol{\nu} g_1 h_1) \begin{pmatrix} c(\_, k_1); k_1[g_1] \triangleleft \ell \\ \mid [\![[\epsilon\rangle]\!] f_1\rangle h_1 \end{pmatrix}\}\big)\Big]\\
&\quad \mid \mathcal{E}_1\big[b_2(\_, c_2); (\boldsymbol{\nu} d_2 e_2)(c_2[d_2] \triangleleft \ell \mid e_2[\_, z'])\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)
\end{aligned}
$$

$$
\begin{aligned}
&\to (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} b y)(\\
&\quad \mathcal{E}_2\big[(\boldsymbol{\nu} d_1 e_1)\big(a_2[\_, d_1] \mid e_1(f_1) \triangleright \{i : \ldots\}_{i\in I} \cup \{\ell : (\boldsymbol{\nu} g_1 h_1) \begin{pmatrix} c(\_, k_1); k_1[g_1] \triangleleft \ell \\ \mid [\![[\epsilon\rangle]\!] f_1 \rangle h_1 \end{pmatrix}\}\big)\big]\\
&\quad \mid \mathcal{E}_1\big[b_2(\_, c_2); (\boldsymbol{\nu} d_2 e_2)(c_2[d_2] \triangleleft \ell \mid e_2[\_, z'])\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\to (\boldsymbol{\nu} d_2 e_2)(\boldsymbol{\nu} b y)(\\
&\quad \mathcal{E}_2\big[(\boldsymbol{\nu} g_1 h_1)(c(\_, k_1); k_1[g_1] \triangleleft \ell \mid [\![[\epsilon\rangle]\!] d_2 \rangle h_1)\big]\\
&\quad \mid \mathcal{E}_1\big[e_2[\_, z']\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\equiv (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)(\\
&\quad \mathcal{E}_2\big[(\boldsymbol{\nu} g_1 h_1)(c(\_, k_1); k_1[g_1] \triangleleft \ell \mid [\![[\epsilon\rangle]\!] a \rangle h_1)\big]\\
&\quad \mid \mathcal{E}_1\big[x[\_, z']\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)(\mathcal{E}_2\big[[\![[\ell\rangle]\!] a \rangle c)\big] \mid \mathcal{E}_1\big[[\![x]\!] z'\big) \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![[\ell, \vec{m}\rangle]\!] a \rangle b)] \mid [\![\mathcal{F}[x]]\!] z \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_) && (*_1, *_4)\\
&= [\![(\boldsymbol{\nu} x[\ell, \vec{m}\rangle y)(\mathcal{F}[x] \parallel C)]\!] z
\end{aligned}
$$

- Rule [RED-CASE]: $j \in I$ implies

$$(\boldsymbol{\nu} x[\vec{m}, j\rangle y)(\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_i\}_{i\in I}] \parallel C) \to_{\mathsf{C}} (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[M_j\ y] \parallel C).$$

W.l.o.g., assume C is a child thread. By Section 5.2, for any L, $[\![\mathcal{F}[L]]\!] z = \mathcal{E}\big[[\![L]\!] z'\big]$ for some $\mathcal{E}, z'$ $(*)$. Assume the condition. The thesis holds as follows:

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x[\vec{m}, j\rangle y)(\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_i\}_{i\in I}] \parallel C)]\!] z\\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![[\vec{m}, j\rangle]\!] a \rangle b \mid [\![\mathcal{F}[\mathtt{case}\, y\, \mathtt{of}\, \{i : M_i\}_{i\in I}]]\!] z \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![[\vec{m}, j\rangle]\!] a \rangle b \mid \mathcal{E}\big[[\![\mathtt{case}\, y\, \mathtt{of}\, \{i : M_i\}_{i\in I}]\!] z'\big] \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_) && (*)\\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)(\\
&\quad (\boldsymbol{\nu} c_1 d_1)(b(\_, e_1); e_1[c_1] \triangleleft j \mid [\![[\vec{m}\rangle]\!] a \rangle d_1)\\
&\quad \mid \mathcal{E}\big[(\boldsymbol{\nu} a_2 b_2)(y[\_, a_2] \mid b_2(c_2) \triangleright \{i : [\![M_i\ c_2]\!] z'\}_{i\in I})\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\to (\boldsymbol{\nu} a x)(\boldsymbol{\nu} a_2 b_2)(\\
&\quad (\boldsymbol{\nu} c_1 d_1)(a_2[c_1] \triangleleft j \mid [\![[\vec{m}\rangle]\!] a \rangle d_1)\\
&\quad \mid \mathcal{E}\big[b_2(c_2) \triangleright \{i : [\![M_i\ c_2]\!] z'\}_{i\in I}\big]\\
&\quad \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&\to (\boldsymbol{\nu} a x)(\boldsymbol{\nu} d_1 c_1)([\![[\vec{m}\rangle]\!] a \rangle d_1 \mid \mathcal{E}\big[[\![M_j\ c_2]\!] z'\{c_1/c_2\}\big] \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_)\\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} d_1 c_1)([\![[\vec{m}\rangle]\!] a \rangle d_1 \mid \mathcal{E}\big[[\![M_j\ c_1]\!] z'\big] \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_) && \text{(Section 5.2)}\\
&\equiv (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![[\vec{m}\rangle]\!] a \rangle b \mid \mathcal{E}\big[[\![M_j\ y]\!] z'\big] \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_) && \text{(Section 5.2)}\\
&= (\boldsymbol{\nu} a x)(\boldsymbol{\nu} b y)([\![[\vec{m}\rangle]\!] a \rangle b \mid [\![\mathcal{F}[M_j\ y]]\!] z \mid (\boldsymbol{\nu} \_\_)[\![C]\!]\_) && (*)\\
&= [\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[M_j\ y] \parallel C)]\!] z
\end{aligned}
$$

- Rule [RED-CLOSE]: $(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{close}\, x; M] \parallel C) \to_{\mathsf{C}} (\boldsymbol{\nu} \Box[\vec{m}\rangle y)(\mathcal{F}[M] \parallel C)$. W.l.o.g., assume C is a child thread. By Section 5.2, for any L, $[\![\mathcal{F}[L]]\!] z = \mathcal{E}_1\big[[\![L]\!] z'\big]$ for some $\mathcal{E}_1, z'$ $(*_1)$. The analysis depends on whether $y = \Box$; w.l.o.g., assume not. By inversion of typing, $\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : \mathsf{end}\rangle S$ where $S \neq \Box$ $(*_2)$. By Section 5.2, $[\![\Gamma \vdash_{\mathsf{B}} [\vec{m}\rangle : \mathsf{end}\rangle S]\!] a \rangle b =$

$\mathcal{E}_2[[\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \mathsf{end}\rangle\mathsf{end}]]a\rangle c]$ $(*_3)$ and $[\![\Gamma \vdash_{\mathtt{B}} [\vec{m}\rangle : \Box\rangle S]]a\rangle b = \mathcal{E}_2[[\emptyset \vdash_{\mathtt{B}} [\epsilon\rangle : \Box\rangle\mathsf{end}]]a\rangle c]$ $(*_4)$ for some $\mathcal{E}_2, c$. Below, we omit types from translations of buffers. The thesis holds as follows:

$$
\begin{aligned}
&[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{close}\, x; M] \parallel C)]\!]z \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!]a\rangle b \mid [\![\mathcal{F}[\mathtt{close}\, x; M]]\!]z \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(\mathcal{E}_2[[\![[\epsilon\rangle]\!]a\rangle c] \mid \mathcal{E}_1[[\![\mathtt{close}\, x; M]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_1,*_3) \\
&= (\boldsymbol{\nu} ax)(\boldsymbol{\nu} by)(&& (*_2) \\
&\quad \mathcal{E}_2[a(\_, c_1); c_1[\_, \_] \mid b(\_, d_1); d_1[\_, \_]] \\
&\quad \mid \mathcal{E}_1[(\boldsymbol{\nu} a_2 b_2)(x[\_, a_2] \mid b_2(\_, \_); [\![M]\!]z')] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\to (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} by)(\\
&\quad \mathcal{E}_2[a_2[\_, \_] \mid b(\_, d_1); d_1[\_, \_]] \\
&\quad \mid \mathcal{E}_1[b_2(\_, \_); [\![M]\!]z'] \\
&\quad \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\to (\boldsymbol{\nu} by)(\mathcal{E}_2[b(\_, d_1); d_1[\_, \_]] \mid \mathcal{E}_1[[\![M]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&\equiv (\boldsymbol{\nu} a\_)(\boldsymbol{\nu} by)(\mathcal{E}_2[b(\_, d_1); d_1[\_, \_]] \mid \mathcal{E}_1[[\![M]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} a\_)(\boldsymbol{\nu} by)(\mathcal{E}_2[[\![[\epsilon\rangle]\!]a\rangle c] \mid \mathcal{E}_1[[\![M]\!]z'] \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) \\
&= (\boldsymbol{\nu} a\_)(\boldsymbol{\nu} by)([\![[\vec{m}\rangle]\!]a\rangle b \mid [\![\mathcal{F}[M]]\!]z \mid (\boldsymbol{\nu}\_\_)[\![C]\!]\_) && (*_1,*_4) \\
&= [\![(\boldsymbol{\nu}\Box[\vec{m}\rangle y)(\mathcal{F}[M] \mid C)]\!]z
\end{aligned}
$$

- Rule [RED-PAR-NIL]: $C \parallel \Diamond() \to_{\mathtt{C}} C$. The thesis holds as follows:

$$
\begin{aligned}
[\![C \parallel \Diamond()]\!]z &= [\![C]\!]z \mid (\boldsymbol{\nu}\_\_)\mathbf{0} \\
&\equiv [\![C]\!]z
\end{aligned}
$$

- Rule [RED-RES-NIL]: $(\boldsymbol{\nu}\Box[\epsilon\rangle\Box)C \to_{\mathtt{C}} C$. The thesis holds as follows:

$$
\begin{aligned}
[\![(\boldsymbol{\nu}\Box[\epsilon\rangle\Box)C]\!]z &= (\boldsymbol{\nu} a\_)(\boldsymbol{\nu} b\_)(\mathbf{0} \mid [\![C]\!]z) \\
&\equiv [\![C]\!]z
\end{aligned}
$$

- Rule [RED-LIFT-C]: $C \to_{\mathtt{C}} C'$ implies $\mathcal{G}[C] \to_{\mathtt{C}} \mathcal{G}[C']$. By Section 5.2, for any D, $[\![\mathcal{G}[D]]\!]z = \mathcal{E}[[\![D]\!]z']$ for some $\mathcal{E}, z'$ $(*_1)$. Assume the condition. By the IH, $[\![C]\!]z' \to^* [\![C']\!]z'$ $(*_2)$. The thesis holds as follows:

$$
\begin{aligned}
[\![\mathcal{G}[C]]\!]z &= \mathcal{E}[[\![C]\!]z'] && (*_1) \\
&\to^* \mathcal{E}[[\![C']\!]z'] && (*_2) \\
&= [\![\mathcal{G}[C']]\!]z && (*_1)
\end{aligned}
$$

- Rule [RED-LIFT-M]: $M \to_{\mathtt{M}} M'$ implies $\mathcal{F}[M] \to_{\mathtt{C}} \mathcal{F}[M']$. By Section 5.2, for any N, $[\![\mathcal{F}[N]]\!]z = \mathcal{E}[[\![N]\!]z']$ for some $\mathcal{E}, z'$ $(*_1)$. Assume the condition. By Appendix A.4.1, $[\![M]\!]z' \to^* [\![M']\!]z'$ $(*_2)$. The thesis holds as follows:

$$
\begin{aligned}
[\![\mathcal{F}[M]]\!]z &= \mathcal{E}[[\![M]\!]z'] && (*_1) \\
&\to^* \mathcal{E}[[\![M']\!]z'] && (*_2) \\
&= [\![\mathcal{F}[M']]\!]z && (*_1)
\end{aligned}
$$

- Rule [RED-CONF-LIFT-SC]: $C \equiv_{\mathsf{C}} C'$, $C' \to_{\mathsf{C}} D'$, and $D' \equiv_{\mathsf{C}} D$ imply $C \to_{\mathsf{C}} D$. Assume the conditions. By Appendix A.4.1, $[\![C]\!]z \equiv [\![C']\!]z$ $(*_1)$ and $[\![D']\!]z \equiv [\![D]\!]z$ $(*_2)$. By the IH, $[\![C']\!]z \to^* [\![D']\!]z$ $(*_3)$. The thesis holds as follows:

$$\begin{aligned} [\![C]\!]z &\equiv [\![C']\!]z && (*_1)\\ &\to^* [\![D']\!]z && (*_3)\\ &\equiv [\![D]\!]z && (*_2) \end{aligned}$$

□

A.4.2. *Soundness.* [Soundness] Given $\Gamma \vdash^{\phi}_{\mathsf{C}} C : T$, if $[\![C]\!]z \to^* Q$, then there exists D such that $C \to^*_{\mathsf{C}} D$ and $Q \to^* [\![D]\!]z$.

Proof. By induction on the number k of steps $[\![C]\!]z \to^k Q$ (IH_1). We distinguish cases on all possible initial reductions $[\![C]\!]z \to Q_0$ and discuss all possible following reductions. Here, we rely on APCP's confluence of independent reductions, allowing us to focus on a specific sequence of reductions, postponing other possibilities that eventually lead to the same result.

We then use induction on the structure of C (IH_2). The goal is to identify some D_0 such that we can isolate $k_0 \geq 0$ reductions such that $C \to_{\mathsf{C}} D_0$ and $[\![C]\!]z \to Q_0 \to^{k_0} [\![D_0]\!]z$ (where k_0 may be different in each case). We then have $[\![D_0]\!]z \to^{k-k_0} Q$, so it follows from IH_1 that there exists D such that $D_0 \to^*_{\mathsf{C}} D$ and $[\![D_0]\!]z \to^* [\![D]\!]z$.

- Case $C = \phi\, M$. By construction, we can identify a maximal context $\mathcal{R}$ and a term M_0 such that $M = \mathcal{R}[M_0]$ and the observed reduction $[\![\phi\,(\mathcal{R}[M_0])]\!]z \to Q_0$ originates from the translation of M_0 directly (i.e., not from inside an evaluation context in the translation of M_0 or from interaction with the translation of $\mathcal{R}$).

  We detail every case for M_0, though not all cases may be applicable to show a reduction. In each case, we rely on Section 5.2 to work with $\mathcal{E}, z'$ such that $[\![\phi\,(\mathcal{R}[M_0])]\!]z = [\![\mathcal{R}[M_0]]\!]z = \mathcal{E}\big[[\![M_0]\!]z'\big]$. Also, in many cases, subterms that partake in the reduction may appear in sequences of explicit substitutions; since structural congruence can always extrude the scope of explicit substitutions, they can, w.l.o.g., be factored out of the proofs below.
  - Case $M_0 = x$. We have $[\![x]\!]z' = x[\_, z']$, so no reduction is possible.
  - Case $M_0 = ()$. We have $[\![()]\!]z' = \mathbf{0}$, so no reduction is possible.
  - Case $M_0 = \lambda x.M_1$. We have $[\![\lambda x.M_1]\!]z' = z'(x, a); [\![M_1]\!]a$, so no reduction is possible.
  - Case $M_0 = M_1\ M_2$. We have

$$[\![M_1\ M_2]\!]z' = (\boldsymbol{\nu} a_1 b_1)(\boldsymbol{\nu} c_1 d_1)([\![M_1]\!]a_1 \mid b_1[c_1, z'] \mid d_1(\_, e_1); [\![M_2]\!]e_1).$$

    The reduction can only originate from a synchronization between the send on b_1 and a receive on a_1 in $[\![M_1]\!]a_1$. By well typedness, M_1 must be of type $T_2 \multimap T_1$ and M_2 of type T_2. It must then be the case that $M_1 = \lambda x.M_{1.1}$: this is the only possibility for a receive on a_1 in $[\![M_1]\!]a_1$.
    Let $D_0 \triangleq \phi\,(\mathcal{R}[M_{1.1}\{\!\{M_2/x\}\!\}])$. We have $C \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned} [\![C]\!]z &= \mathcal{E}\big[(\boldsymbol{\nu} a_1 b_1)(\boldsymbol{\nu} c_1 d_1)(a_1(x, a_2); [\![M_{1.1}]\!]a_2 \mid b_1[c_1, z'] \mid d_1(\_, e_1); [\![M_2]\!]e_1)\big]\\ &\to \mathcal{E}\big[(\boldsymbol{\nu} x d_1)([\![M_{1.1}]\!]z' \mid d_1(\_, e_1); [\![M_2]\!]e_1)\big]\\ &= [\![D_0]\!]z \end{aligned}$$

– Case $M_0 = \texttt{new}$. We have

$$[\![\texttt{new}]\!]z' = (\boldsymbol{\nu} a_1 b_1)\big(a_1[\_, z'] \mid b_1(\_, c_1); (\boldsymbol{\nu} d_1 x)(\boldsymbol{\nu} e_1 y)([\![[\epsilon\rangle]\!]d_1\rangle e_1 \mid [\![(x,y)]\!]c_1)\big).$$

The reduction can only originate from a synchronization between the send on a_1 and the receive on b_1. Let $D_0 \triangleq (\boldsymbol{\nu} x[\epsilon\rangle y)(\phi\,(\mathcal{R}[(x,y)]))$. We have $C \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= \mathcal{E}\big[(\boldsymbol{\nu} a_1 b_1)\big(a_1[\_, z'] \mid b_1(\_, c_1); (\boldsymbol{\nu} d_1 x)(\boldsymbol{\nu} e_1 y)([\![[\epsilon\rangle]\!]d_1\rangle e_1 \mid [\![(x,y)]\!]c_1)\big)\big] \\ &\to (\boldsymbol{\nu} d_1 x)(\boldsymbol{\nu} e_1 y)([\![[\epsilon\rangle]\!]d_1\rangle e_1 \mid \mathcal{E}\big[[\![(x,y)]\!]z'\big]) \\ &= [\![D_0]\!]z\end{aligned}$$

– Case $M_0 = \texttt{fork}\, M_1; M_2$. We have

$$[\![\texttt{fork}\, M_1; M_2]\!]z' = (\boldsymbol{\nu} a_1)\big(a_1[\_, z'] \mid b_1(\_, c_1); ((\boldsymbol{\nu} \_\_)[\![M_1]\!]\_ \mid [\![M_2]\!]c_1)\big).$$

The reduction can only originate from a synchronization between the send on a_1 and the receive on b_1. Let $D_0 \triangleq \phi\,(\mathcal{R}[M_2]) \parallel \Diamond\, M_1$. We have $C \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= \mathcal{E}\big[(\boldsymbol{\nu} a_1)\big(a_1[\_, z'] \mid b_1(\_, c_1); ((\boldsymbol{\nu} \_\_)[\![M_1]\!]\_ \mid [\![M_2]\!]c_1)\big)\big] \\ &\to \mathcal{E}\big[[\![M_2]\!]z'\big] \mid (\boldsymbol{\nu} \_\_)[\![M_1]\!]\_ \\ &= [\![D_0]\!]z\end{aligned}$$

– Case $M_0 = (M_1, M_2)$. We have

$$[\![(M_1, M_2)]\!]z' = (\boldsymbol{\nu} a_1 b_1)(\boldsymbol{\nu} c_1 d_1)(z'[a_1, c_1] \mid b_1(\_, e_1); [\![M_1]\!]e_1 \mid d_1(\_, f_1); [\![M_2]\!]f_1),$$

so no reduction is possible.

– Case $M_0 = \texttt{let}\,(x,y) = M_1\,\texttt{in}\,M_2$. We have

$$[\![\texttt{let}\,(x,y) = M_1\,\texttt{in}\,M_2]\!]z' = (\boldsymbol{\nu} a_1 b_1)(a_1(x,y); [\![M_2]\!]z' \mid [\![M_1]\!]b_1).$$

The reduction can only originate from a synchronization between the receive on a_1 and a send on b_1 in $[\![M_1]\!]b_1$. By well typedness, M_1 must be of type $T_{1.1} \times T_{1.2}$. It must then be the case that $M_1 = (M_{1.1}, M_{1.2})$: this is the only possibility for a send on b_1 in $[\![M_1]\!]b_1$.
Let $D_0 \triangleq \phi\,(\mathcal{R}[M_1\{\!\{M_{1.1}/x, M_{1.2}/y\}\!\}])$. We have $C \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= \mathcal{E}\Big[(\boldsymbol{\nu} a_1 b_1)(a_1(x,y); [\![M_2]\!]z' \mid (\boldsymbol{\nu} a_2 b_2)(\boldsymbol{\nu} c_2 d_2)\begin{pmatrix} b_1[a_2, c_2] \mid b_2(\_, e_2); [\![M_{1.1}]\!]e_2 \\ \mid d_2(\_, f_2); [\![M_{1.2}]\!]f_2\end{pmatrix})\Big] \\ &\to \mathcal{E}\big[(\boldsymbol{\nu} y d_2)\big((\boldsymbol{\nu} x b_2)([\![M_2]\!]z' \mid b_2(\_, e_2); [\![M_{1.1}]\!]e_2) \mid d_2(\_, f_2); [\![M_{1.2}]\!]f_2\big)\big] \\ &= [\![D_0]\!]z\end{aligned}$$

– Case $M_0 = \texttt{send}\, M_1\, M_2$. We have

$$[\![\texttt{send}\, M_1\, M_2]\!]z' = (\boldsymbol{\nu} a_1 b_1)(\boldsymbol{\nu} c_1 d_1)\begin{pmatrix} a_1(\_, e_1); [\![M_1]\!]e_1 \mid [\![M_2]\!]c_1 \\ \mid d_1(\_, f_1); (\boldsymbol{\nu} g_1 h_1)(f_1[b_1, g_1] \mid h_1[\_, z'])\end{pmatrix}.$$

The reduction can only originate from a synchronization between the receive on d_1 and a send on c_1 in $[\![M_2]\!]c_1$. By well typedness, M_2 must be of type $!T_1.S_2$. No reduction is possible: no M_2 can satisy these conditions.

– Case $M_0 = \texttt{recv}\, M_1$. We have

$$[\![\texttt{recv}\, M_1]\!]z' = (\boldsymbol{\nu} a_1 b_1)\big([\![M_1]\!]a_1 \mid b_1(c_1, d_1); (\boldsymbol{\nu} e_1 f_1)(z'[c_1, e_1] \mid f_1(\_, g_1); d_1[\_, g_1])\big).$$

The reduction can only originate from a synchronization between the receive on b_1 and a send on a_1 in $[\![M_1]\!]a_1$. By well typedness, M_1 must be of type $?T_1.S_1$. No reduction is possible: no M_1 can satisfy these conditions.

– Case $M_0 = \mathtt{select}\, j\, M_1$. We have

$$[\![\mathtt{select}\, j\, M_1]\!]z' = (\boldsymbol{\nu} a_1 b_1)\big([\![M_1]\!]a_1 \mid b_1(\_, c_1); (\boldsymbol{\nu} d_1 e_1)(c_1[d_1] \triangleleft j \mid e_1[\_, z'])\big).$$

The reduction can only originate from a synchronization between the receive on b_1 and a send on a_1 in $[\![M_1]\!]a_1$. By well typedness, M_1 must be of type $\oplus\{i : S_1^i\}_{i\in I}$ with $j \in I$. No reduction is possible: no M_1 can satisfy these conditions.

– Case $M_0 = \mathtt{case}\, M_1\, \mathtt{of}\, \{i : M_2^i\}_{i\in I}$. We have

$$[\![\mathtt{case}\, M_1\, \mathtt{of}\, \{i : M_2^i\}_{i\in I}]\!]z' = (\boldsymbol{\nu} a_1 b_1)([\![M_1]\!]a_1 \mid b_1(c_2) \triangleright \{i : [\![M_2^i\ c_2]\!]z'\}_{i\in I}).$$

The reduction can only originate from a synchronization between the branch on b_1 and a selection on a_1 in $[\![M_1]\!]a_1$. By well typedness, M_1 must be of type $\&\{i : S_2^i\}_{i\in I}$. No reduction is possible: no M_1 can satisfy these conditions.

– Case $M_0 = \mathtt{close}\, M_1; M_2$. We have

$$[\![\mathtt{close}\, M_1; M_2]\!]z' = (\boldsymbol{\nu} a_1 b_1)([\![M_1]\!]a_1 \mid b_1(\_, \_); [\![M_2]\!]z').$$

The reduction can only originate from a synchronization between the receive on b_1 and a send on a_1 in $[\![M_1]\!]a_1$. By well typedness, M_1 must be of type $\mathtt{end}$. No reduction is possible: no M_1 can satisfy these conditions.

– Case $M_0 = M_1\{\!\{M_2/x\}\!\}$. We have

$$[\![M_1\{\!\{M_2/x\}\!\}]\!]z' = (\boldsymbol{\nu} x a_1)([\![M_1]\!]z' \mid a_1(\_, b_1); [\![M_2]\!]b_1).$$

The reduction can only originate from a synchronization between the receive on a_1 and a send on x in $[\![M_1]\!]z'$. It must then be the case that $M_1 = \mathcal{R}_{\mathcal{U}}[x]$. By Section 5.2, $[\![\mathcal{R}_{\mathcal{U}}[x]]\!]z' = \mathcal{E}_{\mathcal{U}}[[\![x]\!]z_1] = \mathcal{E}_{\mathcal{U}}[x[\_, z_1]]$.
Let $D_0 \triangleq \phi\,(\mathcal{R}\big[\mathcal{R}_{\mathcal{U}}[M_2]\big])$. We have $C \equiv_{\mathsf{C}} \phi\,(\mathcal{R}\big[\mathcal{R}_{\mathcal{U}}[x\{\!\{M_1/x\}\!\}]\big]) \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= \mathcal{E}\big[(\boldsymbol{\nu} x a_1)(\mathcal{E}_{\mathcal{U}}[x[\_, z_1]] \mid a_1(\_, b_1); [\![M_2]\!]b_1)\big] \\ &\to \mathcal{E}\big[\mathcal{E}_{\mathcal{U}}[[\![M_2]\!]z_1]\big] \\ &= [\![D_0]\!]z\end{aligned}$$

- Case $C = C_1 \parallel C_2$. Assume, w.l.o.g., that C_2 is a child thread. We have

$$[\![C]\!]z = [\![C_1]\!]z \mid (\boldsymbol{\nu} \_\_)[\![C_2]\!]\_.$$

The reduction may originate from $[\![C_1]\!]z$ or from $[\![C_2]\!]\_$; w.l.o.g., assume the former.

We thus have $[\![C_1]\!]z \to Q_1$. By IH_2, there are $D_1, k_1 \geq 0$ such that $C_1 \to_{\mathsf{C}} D_1$ and $[\![C_1]\!]z \to Q_1 \to^{k_1} [\![D_1]\!]z$. Let $D_0 \triangleq D_1 \parallel C_2$. We have $C \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= [\![C_1]\!]z \mid (\boldsymbol{\nu} \_\_)[\![C_2]\!]\_ \\ &\to^{k_1+1} [\![D_1]\!]z \mid (\boldsymbol{\nu} \_\_)[\![C_2]\!]\_ \\ &= [\![D_0]\!]z\end{aligned}$$

- Case $C = (\boldsymbol{\nu} x[\vec{m}\rangle y)C_1$. We have

$$[\![(\boldsymbol{\nu} x[\vec{m}\rangle y)C_1]\!]z = (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)([\![\vec{m}]\!]a_1\rangle b_1 \mid [\![C_1]\!]z).$$

The reduction may originate from (i) $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$, (ii) $[\![C_1]\!]z$, (iii) a synchronization between a_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and x in $[\![C_1]\!]z$, or (iv) a synchronization between b_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and y in $[\![C_1]\!]z$. We detail each case:

(i) The reduction originates from $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$. No matter what $[\vec{m}\rangle$ is, no reduction is possible.

(ii) The reduction originates from $[\![C_1]\!]z$. We thus have $[\![C_1]\!]z \to Q_1$. By IH_2, there are $D_1, k_1 \geq 0$ such that $C_1 \to_{\mathtt{C}} D_1$ and $[\![C_1]\!]z \to Q_1 \to^{k_1} [\![D_1]\!]z$. Let $D_0 \triangleq (\boldsymbol{\nu} x[\vec{m}\rangle y)D_1$. We have $C \to_{\mathtt{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)([\![[\vec{m}\rangle]\!]a_1\rangle b_1 \mid [\![C_1]\!]z) \\ &\to^{k_1+1} (\boldsymbol{\nu} a_1 x)(\boldsymbol{\nu} b_1 y)([\![[\vec{m}\rangle]\!]a_1\rangle b_1 \mid [\![D_1]\!]z) \\ &= [\![D_0]\!]z\end{aligned}$$

(iii) The reduction originates from a synchronization between a_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and x in $[\![C_1]\!]z$. By well typedness, $\Delta \vdash_{\mathtt{B}} [\vec{m}\rangle : S'\rangle S$. Note first that, by Section 5.2, $S' \neq \Box$ implies that there are $\mathcal{E}_2, c_1$ such that $[\![[\vec{m}\rangle : S'\rangle S]\!]a_1\rangle b_1 = \mathcal{E}_2\big[[\![[\epsilon\rangle : S'\rangle S']\!]a_1\rangle c_1\big]$. The analysis depends on S', so we consider all possibilities. In each case, if the reduction is indeed possible, we show that the reduction is the first step in the execution of some rule such that $C \to_{\mathtt{C}} D_0$. The corresponding reduction $[\![C]\!]z \to Q_0 \to^{k_0} [\![D_0]\!]z$ follows the corresponding case in the proof of Section 5.2 (Faithfulness: Operational Correspondence).

– Case $S' = \Box$. Then $x = \Box$ is not free in C_1, and thus x is not free in $[\![C_1]\!]z$: the reduction is not possible.

– Case $S' = \mathsf{end}$. The analysis depends on whether $S = \Box$ or not; w.l.o.g., assume not. We have

$$[\![[\epsilon\rangle : \mathsf{end}\rangle\mathsf{end}]\!]a_1\rangle c_1 = a_1(\_, c_2); \ldots \mid \ldots$$

Thus, the synchronization is between the receive on a_1 and a send on x in $[\![C_1]\!]z$. A send on a variable x can only occur in the translation of that variable directly, under some reduction context. Since x is of type end and its translation appears under a reduction context, the only well-typed way for x to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}[\mathtt{close}\, x; M_1]\big]$. We then have

$$C \equiv_{\mathtt{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{close}\, x; M_1] \mid C_2)\big].$$

Hence, the observed reduction is the first step of executing Rule [RED-CLOSE].

– Case $S' = !T_2.S'_2$. We have

$$[\![[\epsilon\rangle : !T_2.S'_2\rangle !T_2.S'_2]\!]a_1\rangle c_1 = a_1(\_, c_2); \ldots$$

Thus, the synchronization is between the receive on a_1 and a send on x in $[\![C_1]\!]z$. A send on a variable x can only occur in the translation of that variable directly, under some reduction context. Since x is of type $!T_2.S'_2$ and its translation appears under a reduction context, the only well-typed way for x to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}[\mathtt{send}\, M_1\, x]\big]$. We then have

$$C \equiv_{\mathtt{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}\rangle y)(\hat{\mathcal{F}}'[\mathtt{send}\, M_1\, x] \mid C_2)\big].$$

Hence, the observed reduction is the first step of executing Rule [RED-SEND].

– Case $S' = ?T_2.S'_2$. We have

$$[\![[\epsilon\rangle : ?T_2.S'_2\rangle ?T_2.S'_2]\!]\, a_1\rangle c_1 = [\![[\epsilon\rangle : !T_2.\overline{S'_2}\rangle !T_2.\overline{S'_2}]\!]\, c_1\rangle a_1 = c_1(\_, c_2); \ldots$$

Thus, the reduction is not possible.

– Case $S' = \oplus\{i : S_2^i\}_{i\in I}$. We have

$$[\![[\epsilon\rangle : \oplus\{i : S_2^i\}_{i\in I}\rangle \oplus\{i : S_2^i\}_{i\in I}]\!]\, a_1\rangle c_1 = a_1(\_, c_2); \ldots$$

Thus, the synchronization is between the receive on a_1 and a send on x in $[\![C_1]\!]z$. A send on a variable x can only occur in the translation of that variable directly, under some reduction context. Since x is of type $\oplus\{i : S_2^i\}_{i\in I}$ and its translation appears under a reduction context, the only well-typed way for x to appear in C_1 is if $C_1 = \mathcal{G}[\mathcal{F}[\mathtt{select}\, j\, x]]$ where $j \in I$. We then have

$$C \equiv_{\mathsf{C}} \mathcal{G}'[(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathtt{select}\, j\, x] \mid C_2)].$$

Hence, the observed reduction is the first step of executing Rule [RED-SELECT].

– Case $S' = \&\{i : S_2^i\}_{i\in I}$. We have

$$\begin{aligned}[\![[\epsilon\rangle : \&\{i : S_2^i\}_{i\in I}\rangle \&\{i : S_2^i\}_{i\in I}]\!]\, a_1\rangle c_1 &= [\![[\epsilon\rangle : \oplus\{i : \overline{S_2^i}\}_{i\in I}\rangle \oplus\{i : \overline{S_2^i}\}_{i\in I}]\!]\, c_1\rangle a_1 \\ &= c_1(\_, c_2); \ldots\end{aligned}$$

Thus, the reduction is not possible.

(iv) The reduction originates from a synchronization between b_1 in $[\![[\vec{m}\rangle]\!]a_1\rangle b_1$ and y in $[\![C_1]\!]z$. By well typedness, $\Delta \vdash_{\mathsf{B}} [\vec{m}\rangle : S'\rangle S$. The analysis depends on S, so we consider all possibilities. In each case, if the reduction is indeed possible, we show that the reduction is the first step in the execution of some rule such that $C \to_{\mathsf{C}} D_0$. The corresponding reduction $[\![C]\!]z \to Q_0 \to^{k_0} [\![D_0]\!]z$ follows the corresponding case in the proof of Section 5.2 (Faithfulness: Operational Correspondence).

– Case $S = \Box$. Then $y = \Box$ is not free in C_1, and thus y is not free in $[\![C_1]\!]z$: the reduction is not possible.

– Case $S = \mathsf{end}$. By well typedness, then $\vec{m} = \epsilon$. Let $C' \triangleq (\boldsymbol{\nu} y[\epsilon\rangle x)C_1$; we have $C \equiv_{\mathsf{C}} C'$ and $[\![C]\!]z \equiv [\![C']\!]z$ (by Appendix A.4.1). The thesis then follows as in the analogous case under Subcase (iii) above.

– Case $S = !T_2.S_2$. By well typedness, then $\vec{m} = \vec{m}', M_1$. We have

$$[\![[\vec{m}', M_1\rangle : S'\rangle !T_2.S_2]\!]\, a_1\rangle b_1 = (\boldsymbol{\nu} \ldots)(\boldsymbol{\nu} \ldots)(\ldots \mid b_1(\_, h_2); \ldots \mid \ldots).$$

Thus, the synchronization is between the receive on b_1 and a send on y in $[\![C_1]\!]z$. A send on a variable y can only occur in the translation of that variable directly, under some reduction context. Since y is of type $\overline{S} = ?T_2.\overline{S_2}$ and its translation appears under a reduction context, the only well-typed way for y to appear in C_1 is if $C_1 = \mathcal{G}[\mathcal{F}[\mathtt{recv}\, y]]$. We then have

$$C \equiv_{\mathsf{C}} \mathcal{G}'[(\boldsymbol{\nu} x[\vec{m}', M_1\rangle y)(\mathcal{F}'[\mathtt{recv}\, y] \mid C_2)].$$

Hence, the observed reduction is the first step of executing Rule [RED-RECV].

– Case $S = ?T_2.S_2$. By well typedness, then $\vec{m} = \epsilon$; this case is analogous to Case $S = \mathsf{end}$ above.

– Case $S = \oplus\{i : S_2^i\}_{i\in I}$. By well typedness, then $\vec{m} = \vec{m}', j$ where $j \in I$. We have

$$[\![[\vec{m}', j\rangle : S'\rangle \oplus\{i : S_2^i\}_{i\in I}]\!]\, a_1\rangle b_1 = (\boldsymbol{\nu} \ldots)(b_1(\_, e_2); \ldots \mid \ldots).$$

Thus, the synchronization is between the receive on b_1 and a send on y in $[\![C_1]\!]z$. A send on a variable y can only occur in the translation of that variable directly, under some reduction context. Since y is of type $\overline{S} = \&\{i : \overline{S_2^i}\}_{i\in I}$ and its translation appears under a reduction context, the only well-typed way for y to appear in C_1 is if $C_1 = \mathcal{G}\big[\mathcal{F}\,[\mathtt{case}\,y\,\mathtt{of}\,\{i : M_{1.i}\}_{i\in I}]\big]$. We then have

$$C \equiv_{\mathsf{C}} \mathcal{G}'\big[(\boldsymbol{\nu} x[\vec{m}', j\rangle y)(\mathcal{F}\,[\mathtt{case}\,y\,\mathtt{of}\,\{i : M_{1.i}\}_{i\in I}] \mid C_2)\big].$$

Hence, the observed reduction is the first step of executing Rule [RED-CASE].

– Case $S = \&\{i : S_2^i\}_{i\in I}$. By well typedness, then $\vec{m} = \epsilon$; this case is analogous to Case $S = \mathsf{end}$ above.

- Case $C = C_1\{\!\{M/x\}\!\}$. We have

$$[\![C_1\{\!\{M/x\}\!\}]\!]z = (\boldsymbol{\nu} x a_1)([\![C_1]\!]z \mid a_1(\_, b_1); [\![M]\!]b_1).$$

The reduction may originate from (i) $[\![C_1]\!]z$ or (ii) a synchronization between the receive on a_1 and a send on x in $[\![C_1]\!]z$. We detail both cases:

(i) The reduction originates from $[\![C_1]\!]z$. We thus have $[\![C_1]\!]z \to Q_1$. By IH_2, there are $D_1, k_1 \geq 0$ such that $C_1 \to_{\mathsf{C}} D_1$ and $[\![C_1]\!]z \to Q_1 \to^{k_1} [\![D_1]\!]z$. Let $D_0 \triangleq D_1\{\!\{M/x\}\!\}$. We have $C \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= (\boldsymbol{\nu} x a_1)([\![C_1]\!]z \mid a_1(\_, b_1); [\![M]\!]b_1)\\ &\to^{k_1+1} (\boldsymbol{\nu} x a_1)([\![D_1]\!]z \mid a_1(\_, b_1); [\![M]\!]b_1)\\ &= [\![D_0]\!]z\end{aligned}$$

(ii) The reduction originates from a synchronization between the receive on a_1 and a send on x in $[\![C_1]\!]z$. It must then be the case that $C_1 = \mathcal{G}\big[\mathcal{F}\,[x]\big]$. By Section 5.2, $[\![\mathcal{G}\big[\mathcal{F}\,[x]\big]]\!]z = \mathcal{E}[[\![x]\!]z'] = \mathcal{E}[x[\_, z']]$.
Let $D_0 \triangleq \mathcal{G}\big[\mathcal{F}\,[M]\big]$. We have $C \equiv_{\mathsf{C}} \mathcal{G}\big[\mathcal{F}\,[x\{\!\{M/x\}\!\}]\big] \to_{\mathsf{C}} D_0$. Moreover:

$$\begin{aligned}[\![C]\!]z &= (\boldsymbol{\nu} x a_1)(\mathcal{E}[x[\_, z']] \mid a_1(\_, b_1); [\![M]\!]b_1)\\ &\to \mathcal{E}[[\![M]\!]z']\\ &= [\![D_0]\!]z\end{aligned}$$

□