Termination in Concurrency, Revisited

Joseph W. N. Paulus
University of Groningen
Groningen, The Netherlands

Jorge A. Pérez
University of Groningen
Groningen, The Netherlands

Daniele Nantes-Sobrinho*
Imperial College London
London, UK

ABSTRACT

Termination is a central property in sequential programming models: a term is terminating if all its reduction sequences are finite. Termination is also important in concurrency in general, and for message-passing programs in particular. A variety of type systems that enforce termination by typing have been developed. In this paper, we rigorously compare several type systems for π-calculus processes from the unifying perspective of termination. Adopting *session types* as reference framework, we consider two different type systems: one follows Deng and Sangiorgi's weight-based approach; the other is Caires and Pfenning's Curry-Howard correspondence between linear logic and session types. Our technical results precisely connect these very different type systems, and shed light on the classes of client/server interactions they admit as correct.

CCS CONCEPTS

• **Theory of computation** → **Process calculi**; **Type structures**.

KEYWORDS

Concurrency, Process Calculi, Session Types, Expressiveness

ACM Reference Format:
Joseph W. N. Paulus, Jorge A. Pérez, and Daniele Nantes-Sobrinho. 2023. Termination in Concurrency, Revisited. In *International Symposium on Principles and Practice of Declarative Programming (PPDP 2023), October 22–23, 2023, Lisboa, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3610612.3610615

1 INTRODUCTION

The purpose of this paper is to present the first comparative study of type systems that enforce *termination* for message-passing processes in the π-calculus, the paradigmatic model of concurrency.

Termination is a cornerstone of sequential programming models: a term is terminating if all its reduction sequences are finite. Termination is also an important property in concurrency in general, and in message-passing programs in particular. In such a setting, infinite sequences of internal steps are rather undesirable, as they could jeopardize the reliable interaction between a process and its environment. That is, we would like processes that exhibit *infinite* sequences of observable actions, possibly intertwined with *finite* sequences of internal/unobservable steps (i.e., reductions).

In the (un)typed π-calculus, infinite behavior can be expressed via operators for recursion (or recursive definitions) or replication. We are interested in replication, and in particular in *input-guarded* replication, denoted $!x(y).P$. Input-guarded replication neatly captures the essence of *servers* that are persistently available to spawn interactive behavior upon invocations by concurrent *clients*. This way, it precisely expresses the controlled invocation of (shared) resources. To understand its operation, let us write $x\langle z\rangle$ to denote an output prefix, intended as an invocation to a server such as $!x(y).P$. The corresponding reduction rule is then roughly as follows:

$$!x(y).P \mid x\langle z\rangle.Q \longrightarrow\ !x(y).P \mid P[{}^{z}/y] \mid Q$$

Thus, after a synchronization on x, the server $!x(y).P$ continues to be available, and a copy of P is spawned (where $[{}^{z}/y]$ denotes the substitution of y with z, as usual), enabling interaction with Q.

In this setting, an obvious source of non-terminating behaviors is when clients and servers invoke each other indefinitely. This situation arises, in particular, when client invocations occur in the body of a server, which can easily trigger infinite "ping-pong" reductions, as in the following process (where $\mathbf{0}$ denotes inaction):

$$!x(y).x\langle y\rangle.\mathbf{0} \mid x\langle w\rangle.\mathbf{0} \longrightarrow\ !x(y).x\langle y\rangle.\mathbf{0} \mid x\langle w\rangle.\mathbf{0} \mid \mathbf{0} \longrightarrow \cdots \quad (1)$$

The challenge of statically ruling out processes such as (1) while enabling expressive client/server interactions has been addressed by multiple authors via various type systems, see, e.g., [5, 6, 9, 10, 13–15, 17]. Their underlying approaches are vastly diverse. For instance, Yoshida et al. [17] adopt a type-theoretical approach based on logical relations and linear action types. Deng and Sangiorgi [6] transport ideas from rewriting systems (well-founded measures) into a π-calculus with simple types. Caires and Pfenning's Curry-Howard correspondence between linear logic and session types [1] represents yet another approach: their type system enforces termination based purely on proof-theoretical principles, by interpreting the exponential '$!A$' as the type of a server and by connecting cut elimination with process synchronization. Several natural questions arise. How do these type disciplines compare? What are their relative strengths? More concretely, are there terminating processes detected as such by one type system but not by some other? If so, where is the difference?

As inviting and intriguing these questions are, a technical approach to a formal comparison is far from obvious. An immediate obstacle concerns the underlying formal models: all the type systems mentioned above operate on *different dialects* of the π-calculus, involving, e.g., synchronous/asynchronous communication, and monadic/polyadic message passing. These differences quickly escalate at the level of the respective type systems, with the presence/absence of *linearity* unsurprisingly playing a key distinguishing role. How do we even start formulating the intended comparison?

*Also with University of Brasília, Brazil.

PPDP 2023, October 22–23, 2023, Lisboa, Portugal

ACM ISBN 979-8-4007-0812-1/23/10…$15.00
https://doi.org/10.1145/3610612.3610615

We frame our formal comparison as follows. As baseline for comparison we take the π-calculus processes typable with Vasconcelos's session type system [16]. This is a quite liberal type system, which induces a broad class of session processes (including non-terminating ones), which is convenient for our purposes. In the following, this baseline class of processes is denoted $\mathcal{S}$.

We then consider two representative classes of processes, both terminating by typing. One is based on Deng and Sangiorgi's *weight-based* type system; the other is Caires and Pfenning's linear-logic type system. Because these type systems are so different from Vasconcelos's, to connect them with $\mathcal{S}$ we require typed translations. This leads to two classes of terminating processes:

- $\mathcal{W}$ contains all processes in $\mathcal{S}$ (i.e., typable under Vasconcelos's type system) which are also typable (up to a translation) under the weight-based type system.
- $\mathcal{L}$ contains all processes in $\mathcal{S}$ which are also typable (up to another translation) by the Curry-Howard correspondence.

This way, because Vasconcelos's system can type non-terminating processes, both $\mathcal{W} \subset \mathcal{S}$ and $\mathcal{L} \subset \mathcal{S}$ hold by definition. Our technical contributions are two-fold.

(1) Because the type systems by Vasconcelos and by Deng and Sangiorgi are so different, to define $\mathcal{W}$ we develop a *new weight-based type system* that combines elements from both: it ensures termination by enforcing well-founded measures (as Deng and Sangiorgi's) while accounting for linearity and sessions (as Vasconcelos's). The translation involved in bridging $\mathcal{S}$ and this new type system determines a technique for ensuring termination of session-typed processes, which is new and of independent interest.

(2) We prove that $\mathcal{L} \subset \mathcal{W}$ but $\mathcal{W} \not\subset \mathcal{L}$, thus determining the exact relationship between these classes of typed processes. Our discovery is that there are terminating session-typed processes that are typable with the weight-based approach but not under the Curry-Howard correspondence. In other words, techniques based on well-founded measures turn out to be more powerful for enforcing termination than proof-theoretical foundations.

Next, we introduce the class $\mathcal{S}$. §3 develops the new weight-based type system and §4 studies its corresponding class $\mathcal{W}$. The Curry-Howard correspondence for concurrency is recalled in §5, and its corresponding class $\mathcal{L}$ is presented in §6. Finally, §7 collects concluding remarks. The full version of the paper [11] contains omitted technical material.

2 THE CLASS $\mathcal{S}$ OF SESSION PROCESSES

We present the process language that we shall consider as reference in our comparisons, and its corresponding session type system. We distinguish between (i) the processes induced by this process model and (ii) the class of well-typed processes (Definition 2.8); in the following, these classes are denoted by π_S and $\mathcal{S}$, respectively. We consider the type system by Vasconcelos [16], which ensures communication safety and session fidelity, but not progress/deadlock-freedom nor termination. Our presentation closely follows [16], pointing out differences where appropriate.

$P, Q ::=$		(Processes)
	$\overline{x}\langle v\rangle.P$	(output)
	$q\ x(y).P$	(input)
	$P \mid Q$	(composition)
	$(\nu xy)P$	(restriction)
	$\mathbf{0}$	(inaction)
$q ::=$		(Qualifiers)
	lin	(linear)
	un	(unrestricted)
$v ::=$		(Values)
	x	(variables)

Figure 1: Syntax of the session π-calculus π_S

2.1 The Process Model π_S

Definition 2.1 (Processes). *Let $x, y, \ldots$ range over* variables*, denoting* channel names *(or* session endpoints*), and $v, v', \ldots$ over* values*; for simplicity, the sets of values and variables coincide. Also, let $P, Q, \ldots$ range over* processes*, defined by the grammar of Figure 1, which induces the class π_S.*

The output process $\overline{x}\langle v\rangle.P$ sends value v across channel x and then continues as P. In the input process $q\ x(y).P$, the qualifier q can be either lin (denoting a linear input) or un (denoting an unrestricted input, i.e., a replicated server). In either case, x expects to receive a value that will replace free occurrences of y in P. Parallel composition $P \mid Q$ denotes the concurrent execution of processes P and Q. The process $(\nu xy)P$ denotes the restriction of the *co-variables* x and y with scope P. This declares them as dual endpoints, which are expected to behave complementarily to each other. We write $(\nu zv : S)P$ when either z or v have session type S in P. As we will see, a synchronization always occurs across a pairs of co-variables. Finally, the inactive process is denoted as $\mathbf{0}$.

As usual, the set of free variables in a process P is denoted $\mathsf{fv}(P)$, and similarly $\mathsf{bv}(P)$ for bound variables. The capture-free substitution of the variable z by the value v is denoted as $[v/z]$. We adopt Barendregt's variable convention.

With respect to [16], the above the process syntax leaves out boolean values, conditional expressions, and labeled choices, which are all inessential for our comparative study of termination.

Definition 2.2 (Reduction Semantics). *The reduction relation $\longrightarrow$ of π_S is defined in Figure 2.*

The reduction semantics for π_S follows standard lines for (session) π-calculi; it is closed under a structural congruence, denoted $\equiv$, which captures expected principles for parallel composition and restriction. The reduction rule (R-LinCom) captures the linear communication across co-variables x and y, appropriately declared by restriction, in which value v is exchanged. Similarly, rule (R-UnCom) denotes unrestricted communication across co-variables; in this case, the input prefix is persistent, and remains ready for further synchronizations after reduction. The contextual rules (R-Par) and (R-Res) express that concurrent processes can reduce within the scope of parallel composition and restriction. Finally, rule (R-Str) denotes that reductions are closed under structural congruence.

$$
\begin{array}{llcll}
P \mid Q & \equiv Q \mid P & & P \mid \mathbf{0} & \equiv P \\
(P \mid Q) \mid R & \equiv P \mid (Q \mid R) & & (\nu xy)\mathbf{0} & \equiv \mathbf{0} \\
(\nu xy)(\nu zw)P & \equiv (\nu zw)(\nu xy)P & & (\nu xy)P & \equiv (\nu yx)P \\
((\nu xy)P) \mid Q & \multicolumn{4}{l}{\equiv (\nu xy)(P \mid Q)\ [x, y \notin \mathsf{fv}(Q)]}
\end{array}
$$

$$
\begin{array}{rl}
(\text{R-LinCom}) & (\nu xy)(\overline{x}\langle v\rangle.P \mid \mathsf{lin}\ y(z).Q \mid R) \\
& \quad \longrightarrow (\nu xy)(P \mid Q[v/z] \mid R) \\
(\text{R-UnCom}) & (\nu xy)(\overline{x}\langle v\rangle.P \mid \mathsf{un}\ y(z).Q \mid R) \\
& \quad \longrightarrow (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}\ y(z).Q \mid R) \\
(\text{R-Par}) & P \longrightarrow Q \Longrightarrow P \mid R \longrightarrow Q \mid R \\
(\text{R-Res}) & P \longrightarrow Q \Longrightarrow (\nu xy)P \longrightarrow (\nu xy)Q \\
(\text{R-Str}) & P \equiv P',\ P \longrightarrow Q,\ Q' \equiv Q \Longrightarrow P' \longrightarrow Q'
\end{array}
$$

Figure 2: Reduction semantics for π_{S}

$$
\begin{array}{rll}
q ::= & & \text{(Qualifiers)} \\
& \mathsf{lin} & \text{(linear)} \\
& \mathsf{un} & \text{(unrestricted)} \\
T, S ::= & & \text{(Types)} \\
& \mathbf{end} & \text{(termination)} \\
& q\ p & \text{(pretypes)} \\
& a & \text{(type variable)} \\
& \mu a.T & \text{(recursive types)} \\
p ::= & & \text{(Pretypes)} \\
& ?T.S & \text{(receive)} \\
& !T.S & \text{(send)} \\
\Gamma ::= & & \text{(Contexts)} \\
& \emptyset & \text{(empty)} \\
& \Gamma, x : T & \text{(assumption)}
\end{array}
$$

Figure 3: Session Types of π_{S}

2.2 Session Types

We endow π_{S} with the session type system by Vasconcelos [16], which ensures that well-typed processes respect their protocols but does not ensure deadlock-freedom nor termination guarantees. With respect to the syntax of types in [16], we only consider channel endpoint types (no ground types such as bool).

Definition 2.3 (Session Types). *The syntax of session types ($T, S, \ldots$) is given in Figure 3.*

Session types T, S describe protocols as *sequences* of actions for an endpoint; they do not admit the parallel usage of an endpoint. They have the following forms:

(1) Type **end** is given to an endpoint with a completed protocol.
(2) Type $q\ p$ denotes pre-type p with qualifier q, which indicates either a linear or an unrestricted behavior (lin and un, respectively). The pre-type $?T.S$ is given to an endpoint that first receives a value of type T and then continues according to type S. Dually, the pre-type $!T.S$ is intended for an endpoint that first outputs a value of type T and then continues according to S.
(3) Type $\mu a.T$ is a recursive type, with type variable a. A recursive type is required to be *contractive*, i.e., it contains no subexpression of type $\mu a_1. \ldots .\mu a_n.a_1$; and a is bound with scope T. Notions of bound and free type variables, alpha-conversion and capture-avoiding substitutions (denoted $[S/a]$) is defined as usual. Type equality is based on regular infinite trees [16].

Recursive types that are *tail-recursive* are expressive enough to type servers and clients; we have a dedicated notation for them.

Notation 2.1 (Server and Client Types). *We shall write $*?T$ to denote the server type $\mu a.\mathsf{un}\ ?T.a$, where variable a does not occur in T. Similarly, we write $*!T$ to denote the client type $\mu a.\mathsf{un}\ !T.a$*

In the following, we shall work with tail-recursive types only. A central notion in session-based concurrency is *duality*, which relates session types offering opposite (i.e., complementary) behaviors; it stands at the basis of communication safety and session fidelity.

Definition 2.4 (Duality). *Given a (tail-recursive) session type T, its dual type $\overline{T}$ is defined as follows:*

$$
\begin{array}{lllllllll}
\overline{\mathbf{end}} & = & \mathbf{end} & \overline{!T.S} & = & ?T.\overline{S} & \overline{*?T} & = & *!T \\
\overline{q\ p} & = & q\ \overline{p} & \overline{?T.S} & = & !T.\overline{S} & \overline{*!T} & = & *?T
\end{array}
$$

We now collect definitions and results from [16] that will lead to state the main properties of typable processes.

Definition 2.5 (Predicates on Types/Contexts). *We consider two predicates on types, denoted $\mathsf{lin}(T)$ and $\mathsf{un}(T)$, defined as follows:*

- $\mathsf{un}(T)$ *if and only if* $T = \mathbf{end}$ *or* $T = \mathsf{un}\ p$.
- $\mathsf{lin}(T)$ *if and only if* `true`.

The definition extends to contexts as follows: we write $q(\Gamma)$ if and only if $x : T \in \Gamma$ implies $q(T)$.

This way, to express that T defines strictly linear behavior we write $\neg\mathsf{un}(T)$ (and similarly for a context Γ). The following notation is useful to separate the linear and unrestricted portions of a context:

Notation 2.2. *We write $\Gamma \circledast \Gamma'$ if $\mathsf{un}(\Gamma) \wedge \neg\mathsf{un}(\Gamma')$.*

Definition 2.6 (Context Split and Update). *The split and update operations on contexts, denoted $\circ$ and $+$, are defined as follows.*

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

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

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

The typing system considers two kinds of judgments, for processes and for variables, denoted $\Gamma \vdash_{\mathsf{S}} P$ and $\Gamma \vdash_{\mathsf{S}} x : T$, respectively. We write $\vdash_{\mathsf{S}} P$ when Γ is empty. The typing rules are given in Figure 4. We will explain Rule [S:In]: it is parametric on the qualifiers q_1 and q_2 and covers three different behaviours depending

$$[\texttt{S:Var}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash_{\mathsf{S}} x : T} \qquad [\texttt{S:Nil}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash_{\mathsf{S}} \mathbf{0}}$$

$$[\texttt{S:Par}]\ \frac{\Gamma_1 \vdash_{\mathsf{S}} P \qquad \Gamma_2 \vdash_{\mathsf{S}} Q}{\Gamma_1 \circ \Gamma_2 \vdash_{\mathsf{S}} P \mid Q} \qquad [\texttt{S:Res}]\ \frac{\Gamma, x : T, y : \overline{T} \vdash_{\mathsf{S}} P}{\Gamma \vdash_{\mathsf{S}} (\nu xy)P}$$

$$[\texttt{S:In}]\ \frac{q_1(\Gamma_1 \circ \Gamma_2) \qquad \Gamma_1 \vdash_{\mathsf{S}} x : q_2?T.S \qquad (\Gamma_2 + x : S), y : T \vdash_{\mathsf{S}} P}{\Gamma_1 \circ \Gamma_2 \vdash_{\mathsf{S}} q_1\ x(y).P}$$

$$[\texttt{S:Out}]\ \frac{\Gamma_1 \vdash_{\mathsf{S}} x : q!T.S \qquad \Gamma_2 \vdash_{\mathsf{S}} v : T \qquad \Gamma_3 + x : S \vdash_{\mathsf{S}} P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash_{\mathsf{S}} \overline{x}\langle v \rangle.P}$$

Figure 4: Typing rules for π_{S} (cf. [16]).

on whether q_i is lin or un, for $i = 1, 2$. In the case $q_1 = \mathsf{lin}$, to prove $\Gamma_1 \circ \Gamma_2 \vdash_{\mathsf{S}} \mathsf{lin}\ x(y).P$, we need to prove $\Gamma_1 \vdash_{\mathsf{S}} x : q_2?T.S$ and $(\Gamma_2 + x : S), y : T \vdash_{\mathsf{S}} P$; note that $\mathsf{lin}(\Gamma_1 \circ \Gamma_2)$ is true, by Definition 2.5. In the case $q_2 = \mathsf{lin}$, both judgments hold if $\Gamma_1 = \Gamma_1', x : \mathsf{lin}?T.S$, the assignment $x : \mathsf{lin}?T.S$ does not occur in Γ_2, by Definition 2.6, and $x : S$ is added to Γ_2 for the continuation. Differently, when $q_2 = \mathsf{un}$, both judgments hold if $\Gamma_1 = \Gamma_1', x : *?T$, the assignment $x : *?T$ also occurs in Γ_2 which with the addition of $x : S$ in Γ_2 implies $S = *?T$. Notice that the case $q_1 = \mathsf{un}$ and $q_2 = \mathsf{lin}$ is not possible since $\mathsf{un}(\Gamma_1 \circ \Gamma_2)$ implies that all assignments in $\Gamma_1 \circ \Gamma_2$ have types **end** or with 'un'; thus, in that case we cannot prove $\Gamma_1 \vdash_{\mathsf{S}} x : \mathsf{lin}?T.S$.

Similarly, Rule [S:Out] is parametric on the qualifier q.

The main property of the type system concerns *well-formed* processes, which are defined next.

Definition 2.7 (Redexes and Well-formedness). *A* redex *is a process of the form $q\ x(v).P \mid \overline{y}\langle z \rangle.Q$. Processes of the form $q\ x(v).P$ and $\overline{y}\langle z \rangle.Q$ have prefix x and y, respectively.*

A process is well-formed *if, for each of its structurally congruent processes of the form $(\nu x_1 y_1) \cdots (\nu x_n y_n)(P \mid Q \mid R)$, the following conditions hold. (1) If P and Q are processes prefixed at the same variable, then they are of the same nature (input, output). (2) If P is prefixed at x_1 and Q is prefixed at y_1 then $P \mid Q$ is a redex.*

Theorem 2.1 (Properties of the Type System). *The type system satisfies the following properties (see [16] for details):*

- *If $\Gamma \vdash_{\mathsf{S}} P$ and $P \equiv Q$, then $\Gamma \vdash_{\mathsf{S}} Q$.*
- *If $\Gamma \vdash_{\mathsf{S}} P$ and $P \longrightarrow Q$, then $\Gamma \vdash_{\mathsf{S}} Q$.*
- *If $\vdash_{\mathsf{S}} P$ then P is well-formed.*

For technical convenience, we rely on the *refined* typing rules for input and output in Figure 5, which are equivalent (but more fine-grained) than those in Figure 4.

We close this section by defining the class of processes $\mathcal{S}$.

Definition 2.8 ($\mathcal{S}$). *We define $\mathcal{S} = \{P \in \pi_{\mathsf{S}} \mid \exists\Gamma\ s.t.\ \Gamma \vdash_{\mathsf{S}} P\}$.*

Example 2.1 (A Non-Terminating Process in $\mathcal{S}$). *Consider the process $P_{2.1} = (\nu xy)(\overline{y}\langle w \rangle.\mathbf{0} \mid \mathsf{un}\ x(z).\overline{y}\langle w \rangle.\mathbf{0})$, which invokes itself ad infinitum. Process $P_{2.1}$ is in $\mathcal{S}$ because $w : \mathbf{end} \vdash_{\mathsf{S}} P_{2.1}$ holds with the following derivation:*

$$[\texttt{S:Res}]\ \dfrac{[\texttt{S:Par}]\ \dfrac{\Pi \qquad [\texttt{S:Un-In}]\ \dfrac{\dfrac{\mathsf{un}(\Gamma)}{\Gamma \vdash_{\mathsf{S}} x : *?\mathbf{end}} \qquad \Pi}{\Gamma \vdash_{\mathsf{S}} \mathsf{un}\ x(z).\overline{y}\langle w \rangle.\mathbf{0}}}{\Gamma \vdash_{\mathsf{S}} \overline{y}\langle w \rangle.\mathbf{0} \mid \mathsf{un}\ x(z).\overline{y}\langle w \rangle.\mathbf{0}}}{w : \mathbf{end} \vdash_{\mathsf{S}} (\nu xy)(\overline{y}\langle w \rangle.\mathbf{0} \mid \mathsf{un}\ x(z).\overline{y}\langle w \rangle.\mathbf{0})}$$

with $\Gamma = x : *?\mathbf{end}, y : *!\mathbf{end}, w : \mathbf{end}$ and Π is the derivation

$$[\texttt{S:Un-Out}]\ \dfrac{\dfrac{\mathsf{un}(\Gamma')}{\Gamma' \vdash_{\mathsf{S}} y : *!\mathbf{end}} \qquad \dfrac{\mathsf{un}(\Gamma')}{\Gamma' \vdash_{\mathsf{S}} w : \mathbf{end}} \qquad \dfrac{\mathsf{un}(\Gamma')}{\Gamma' \vdash_{\mathsf{S}} \mathbf{0}}}{\Gamma' \vdash_{\mathsf{S}} \overline{y}\langle w \rangle.\mathbf{0}}$$

with $\Gamma' = x : *?\mathbf{end}, y : *!\mathbf{end}, w : \mathbf{end}, z : \mathbf{end}$.

$$[\texttt{S:Lin-In}_1]\ \frac{\Gamma_1, x : \mathsf{lin}?T.S \vdash_{\mathsf{S}} x : \mathsf{lin}?T.S \qquad \Gamma_2, x : S, y : T \vdash_{\mathsf{S}} P}{\Gamma_1, x : \mathsf{lin}?T.S \circ \Gamma_2 \vdash_{\mathsf{S}} \mathsf{lin}\ x(y).P}$$

$$[\texttt{S:Lin-In}_2]\ \frac{\Gamma_1, x : *?T \vdash_{\mathsf{S}} x : *?T \qquad \Gamma_2, x : *?T, y : T \vdash_{\mathsf{S}} P}{(\Gamma_1, x : *?T) \circ (\Gamma_2, x : *?T) \vdash_{\mathsf{S}} \mathsf{lin}\ x(y).P}$$

$$[\texttt{S:Un-In}]\ \frac{\Gamma \vdash_{\mathsf{S}} x : *?T \qquad \Gamma, y : T \vdash_{\mathsf{S}} P}{\Gamma \vdash_{\mathsf{S}} \mathsf{un}\ x(y).P}$$

$$[\texttt{S:Un-Out}]\ \frac{\Gamma_1 \vdash_{\mathsf{S}} x : *!T \qquad \Gamma_2 \vdash_{\mathsf{S}} v : T \qquad \Gamma_3 \vdash_{\mathsf{S}} P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash_{\mathsf{S}} \overline{x}\langle v \rangle.P}$$

$$[\texttt{S:Lin-Out}]\ \frac{\Gamma_1 \vdash_{\mathsf{S}} x : \mathsf{lin}\,!T.S \qquad \Gamma_2 \vdash_{\mathsf{S}} v : T \qquad \Gamma_3, x : S \vdash_{\mathsf{S}} P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash_{\mathsf{S}} \overline{x}\langle v \rangle.P}$$

Figure 5: Refined typing rules for input and output.

3 A WEIGHT-BASED APPROACH TO TERMINATING PROCESSES

We move on to consider a type system that ensures termination for a class of π-calculus processes. Following Deng and Sangiorgi [7], the type system uses *weights* (or *levels*) to avoid infinite reduction sequences. This type system will induce a class of terminating π_{S} processes, denoted $\mathcal{W}$ (Definition 4.1), obtained via appropriate translations on processes and types. To ease the definition of such translations, here we define a type system that mildly modifies the system of [7] to account for linearity and synchronous/polyadic (tuple-based) communication. Our main result is that the weight-based system ensures termination (Theorem 3.2).

3.1 Processes

We introduce a process model for the weight-based type system, denoted π_{W}, formally defined next. In the following, we write $\tilde{y}$ to stand for the finite tuple $y_1, \cdots, y_n$.

Definition 3.1 (Processes). *The syntax of π_{W} processes is given by the grammar in Figure 7 (top).*

π_W is designed to stand in between π_S and the process model in [7]. Communication in π_W is polyadic, i.e., exchanges involve a tuple of names, rather than a single name as in Definition 2.1 and [7]. We shall often consider tuples of length two (i.e., dyadic communication), as this suffices for a continuation-passing encoding of sessions [2]. Another difference with respect to [7] is that inputs can be linear or unrestricted; this will facilitate the formal connection with π_S and its type system. The role of linearity is more prominent at the level of types, defined later on.

We give the operational semantics of π_W in terms of the (early) labeled transition system (LTS), with the following labels for input, output, bound output, and silent transitions (synchronizations):

$$\alpha ::= x(\tilde{v}) \mid \overline{x}\langle\tilde{y}\rangle \mid (\nu y, \widetilde{b})\overline{x}\langle\tilde{v}\rangle \mid \tau$$

The rules, given in Figure 6, are standard. Rules [W:Par] and [W:Tau] can be applied symmetrically across parallel composition.

3.2 Types

Definition 3.2 (Types for π_W). *The syntax of weight-based types for π_W is given by the grammar in Figure 7 (bottom).*

As in [7], our link types for π_W are *simple*, i.e., they do not admit the sequencing of actions enabled by session types. Our syntax of types extends that in [7] to account for (i) dyadic communication and (ii) explicit types for clients and servers. Concerning (ii), we purposefully adopt the tail-recursive types for clients and servers defined for π_S, rather than more general recursive types.

We introduce some notions borrowed from the type system from §2.2: duality, contexts, predicates on types, operations on contexts.

Definition 3.3 (Duality). *Duality on linked types is defined as:*

$$\overline{\#^n(V_1,V_2)} = \#^n\langle\overline{V_1},\overline{V_2}\rangle \qquad \overline{\#^n\langle V_1,V_2\rangle} = \#^n(\overline{V_1},\overline{V_2}) \qquad \overline{\mathbf{unit}} = \mathbf{unit}$$
$$\overline{*\#^n(V)} = *\#^n\langle\overline{V}\rangle \qquad \overline{*\#^n\langle V\rangle} = *\#^n(\overline{V})$$

Definition 3.4 (Contexts). *Contexts are given by the grammar:*

$$\Gamma,\Delta ::= \cdot \mid \Gamma, x : V \mid \Gamma, x : \langle V, \overline{V}\rangle$$

where $\Gamma, x : L$ *and* $\Gamma, x : \langle L, \overline{L}\rangle$ *imply* $x \notin \mathrm{dom}(\Gamma)$.

Following the sorts of [8], the assignment $x : \langle L, \overline{L}\rangle$ *denotes the pairing of* x *with two complementary protocols, where* $\langle L, \overline{L}\rangle = \langle \overline{L}, L\rangle$. *We use* $x :: L$ *to stand for* $x : \langle L, \overline{L}\rangle$ *when* L *is the main object of interest. We write* $x \diamond T$ *if either* $x : T$ *or* $x :: T$ *holds (i.e.,* $\diamond \in \{:, ::\}$*).*

Definition 3.5 (Unrestricted Types). *Predicate* $\mathrm{un}(T)$ *holds if* $T = *\#^n(V)$, $T = *\#^n\langle V\rangle$, $T = \mathbf{unit}$, *or* $x{:}\langle L, \overline{L}\rangle$ *with* $\mathrm{un}(L)$. *We write* $\mathrm{un}(\Gamma)$ *if* $\mathrm{un}(T)$ *holds for every* $x \diamond T \in \Gamma$.

Following Definition 2.6, the following definitions gives a relation to split contexts into two parts.

Definition 3.6 (Split Relation on Contexts). *The relation* $\circ$ *on contexts is defined in Figure 8.*

We now introduce notions on processes that are essential to Deng and Sangiorgi's approach to termination by typing.

Definition 3.7 (Level Function, $l(x)$). *Let* $\mathcal{N}$ *denote the set of all names. We define the function* $l(\cdot) : \mathcal{N} \to \mathbb{N}$ *to map names of a process (free and bound) to naturals. We assume* α*-conversion is silently used to avoid name capture and ensure uniqueness of bound names. Given a (typed) process, we define this function as follows:*

$$l(x) = \begin{cases} n & \text{if } x : T \text{ or } x :: T \\ & \text{with } T \in \{\#^n(V_1,V_2), \#^n\langle V_1,V_2\rangle, *\#^n(V), *\#^n\langle V\rangle\} \\ m & \text{if } x : \mathbf{unit}, \text{ for any } m \in \mathbb{N} \end{cases}$$

Definition 3.8 (Active Outputs, $\mathrm{os}(\cdot)$). *Given a process* P*, the set of names with active outputs* $\mathrm{os}(P)$ *is defined inductively:*

$$\mathrm{os}(\overline{x}\langle\tilde{y}\rangle.P) = \{x\} \cup \mathrm{os}(P) \qquad \mathrm{os}(x(\tilde{y}).P) = \mathrm{os}(P)$$
$$\mathrm{os}(P \mid Q) = \mathrm{os}(P) \cup \mathrm{os}(Q) \qquad \mathrm{os}((\nu x)P) = \mathrm{os}(P)$$
$$\mathrm{os}(\mathbf{0}) = \emptyset \qquad \mathrm{os}(!x(\tilde{y}).P) = \emptyset$$

Typing judgments are of the form $\Gamma \vdash_W P$, with corresponding typing given in Figure 9. Typability is contingent on a level function: we say a process P is well-typed if there exists a level function $l(\cdot)$ such that a typing derivation $\Gamma \vdash_W P$ holds, for some Γ.

We comment on some of the rules in Figure 9 for π_W, contrasting them with those in Figure 4 for π_S. Rule [W:Var$_1$] is similar to rule [S:Var]. Rule [W:Var$_2$] is the corresponding rule for complementary interaction: if $x : \langle V, \overline{V}\rangle$, then we can assign the type $x : V$. Intuitively, name x encapsulates the types of its two endpoints, denoted as V and $\overline{V}$. As long as x respects one of these types, the channel is considered correctly typed.

Rule [W:Lin − In$_1$] acts as the linear counterpart to [S:In]. Importantly, there is no direct counterpart for x as a linear complementary interaction. Instead, the context split $\Gamma, x : \langle V, \overline{V}\rangle = (\Gamma_1, x : V) \circ (\Gamma_2, x : \overline{V})$ allows for the application of rule [W:Lin − In$_1$]. This structural mechanism operates silently within the rules where V is linear, achieved through context split. As a result, this disallows linear channels from consuming linear complementary interactions.

Rules [W:Lin − In$_2$] and [W:Lin − In$_3$], the first with ':' and the second with '::', are counterparts to rule [S:In] for unrestricted types with linear qualifier. Similarly, [W:Lin − Out], [W:Un − Out$_1$], and [W:Un − Out$_2$] represent the rule [S:Out]. Furthermore, [W:Un − In$_1$] and [W:Un − In$_2$] are the unrestricted counterparts to rule [S:In] with unrestricted qualifier. These rules adopt the main condition from [7], i.e., the weight of types of the active outputs must be strictly less than the weight of the type of the channel of the server providing them. Finally, rule [W:Res] types a restricted channel through a complementary interaction.

We state the type preservation property:

Theorem 3.1 (Type Preservation). *Suppose* $\Gamma \vdash_W P$ *for a level function* l*. If* $P \xrightarrow{\tau} P'$ *then* $\Gamma \vdash_W P'$ *for the same level function* l*.*

3.3 Termination by Typing

A process terminates if all its reduction sequences are finite. We show that our formulation of the type system in [7] also enforces termination by typing. The proof follows the same lines as in [7]: a weight is associated with a well-typed process; this weight is then shown to strictly reduce when the the process synchronizes. The weight is actually a *vector* constructed from the observable active outputs of a channel within a typed process.

Definition 3.9 (Vectors). *We define vectors and their operations:*

$$\texttt{[W:In]}\ \frac{}{x(\tilde{y}).P \xrightarrow{x(\tilde{v})} P[v_1/y_1][v_2/y_2]} \qquad \texttt{[W:Par]}\ \frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha)\cap\mathsf{fn}(Q)=\emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \texttt{[W:Res]}\ \frac{P \xrightarrow{\alpha} P' \qquad x \notin \mathsf{n}(\alpha)}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \qquad \texttt{[W:Rep]}\ \frac{}{!x(\tilde{y}).P \xrightarrow{x(\tilde{v})} !x(\tilde{y}).P \mid P[v_1/y_1]}$$

$$\texttt{[W:Out]}\ \frac{}{\overline{x}\langle\tilde{y}\rangle.P \xrightarrow{\overline{x}\langle\tilde{y}\rangle} P} \qquad \texttt{[W:Tau]}\ \frac{P \xrightarrow{(\nu\tilde{b})\overline{x}\langle\tilde{v}\rangle} P' \qquad Q \xrightarrow{x(\tilde{v})} Q' \qquad \tilde{b}\cap\mathsf{fn}(Q)=\emptyset}{P \mid Q \xrightarrow{\tau} (\nu\tilde{b})(P' \mid Q')} \qquad \texttt{[W:Open]}\ \frac{P \xrightarrow{(\nu\tilde{b})\overline{x}\langle\tilde{v}\rangle} P' \qquad y \in (\mathsf{fn}(v_1)\cup\mathsf{fn}(v_2)) - \{\tilde{b}, x\}}{(\nu y)P \xrightarrow{(\nu y,\tilde{b})\overline{x}\langle\tilde{v}\rangle} P'}$$

Figure 6: An LTS for π_{W}

$$\begin{array}{lll}
P, Q ::= & & \text{(Processes)} \\
& \overline{x}\langle y_1, y_2\rangle.P & \text{(output)} \\
& x(y_1, y_2).P & \text{(linear input)} \\
& P \mid Q & \text{(parallel)} \\
& !x(y_1, y_2).P & \text{(server)} \\
& (\nu x)P & \text{(restriction)} \\
& \mathbf{0} & \text{(inaction)} \\
\hline
S, T, V, L ::= & & \text{(Link Types)} \\
& \#^n(V_1, V_2) & \text{(linear input type)} \\
& \#^n\langle V_1, V_2\rangle & \text{(linear output type)} \\
& *\#^n(V) & \text{(unrestricted server type)} \\
& *\#^n\langle V\rangle & \text{(unrestricted client type)} \\
& \mathbf{unit} & \text{(termination)} \\
n ::= & 1, 2, \cdots & \text{(weights)}
\end{array}$$

Figure 7: Syntax of processes and types for π_{W}.

- *Given $k \geq 1$, we write $\mathtt{0}_i$ to denote the vector $\langle n_k, n_{k-1}, \cdots, n_1\rangle$ where $n_i = 1$ and $n_j = 0$ for every other j. Also, $\mathtt{0}$ denotes the zero vector where $n_i = 0$ for every i.*
- *Given vectors $v_1 = \langle n_k, n_{k-1}, \cdots, n_1\rangle$ and $v_2 = \langle m_l, m_{l-1}, \cdots, m_1\rangle$, with $k \geq l$, the sum $v_1 + v_2$ is defined in two steps. Firstly, if $k > l$ then the shorter vector v_2 is extended into v_2' by adding zeroes to match the size of v_1, i.e., $v_2' = \langle m_k, m_{k-1}, \cdots, m_l, \cdots, m_1\rangle$, with $\langle m_k, m_{k-1}, \cdots, m_{l+1}\rangle = \mathtt{0}$. Then, addition of v_1 and v_2' is applied pointwise.*
- *Given vectors $v_1 = \langle n_k, n_{k-1}, \cdots, n_1\rangle$ and $v_2 = \langle m_k, m_{k-1}, \cdots, m_1\rangle$ of equal size k, the ordering $v_1 \prec v_2$ is defined iff $\exists i \leq k,\ n_i < m_i$ and $\forall j > i,\ n_j = m_j$.*

Using vectors, we define the weight of a well-typed process:

Definition 3.10 (Weights). *Given a well-typed process P with level function l, the weight of P is the vector defined inductively as:*

$$\begin{array}{ll}
\mathsf{wt}(\mathbf{0}) = 0 & \mathsf{wt}(!x(\tilde{y}).P) = 0 \\
\mathsf{wt}(x(\tilde{y}).P) = \mathsf{wt}(P) & \mathsf{wt}(\overline{x}\langle\tilde{y}\rangle.P) = \mathsf{wt}(P) + \mathtt{0}_{l(x)} \\
\mathsf{wt}(P \mid Q) = \mathsf{wt}(P) + \mathsf{wt}(Q) & \mathsf{wt}((\nu x)P) = \mathsf{wt}(P)
\end{array}$$

We have the following results, whose proof is as in [7]:

Proposition 3.1. *If $\Gamma \vdash_{\mathsf{W}} P$ and $P \xrightarrow{\tau} P'$ then $\mathsf{wt}(P') \prec \mathsf{wt}(P)$.*

Theorem 3.2 (Termination). *If $\Gamma \vdash_{\mathsf{W}} P$ then P terminates.*

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

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathsf{un}(V)}{\Gamma, x : \langle V, \overline{V}\rangle = (\Gamma_1, x : \langle V, \overline{V}\rangle) \circ (\Gamma_2, x : \langle V, \overline{V}\rangle)}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \neg\mathsf{un}(V)}{\Gamma, x : \langle V, \overline{V}\rangle = (\Gamma_1, x : V) \circ (\Gamma_2, x : \overline{V})}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \neg\mathsf{un}(V)}{\Gamma, x : \langle V, \overline{V}\rangle = (\Gamma_1, x : \langle V, \overline{V}\rangle) \circ \Gamma_2}$$

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \neg\mathsf{un}(V)}{\Gamma, x : \langle V, \overline{V}\rangle = \Gamma_1 \circ (\Gamma_2, x : \langle V, \overline{V}\rangle)} \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \neg\mathsf{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ \Gamma_2}$$

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

Figure 8: Splitting of Contexts for π_{W}

4 $\mathcal{W}$: A CLASS OF TERMINATING PROCESSES

Here we define and study $\mathcal{W}$, a class of terminating π_{S} processes induced by the weight-based type system given in §3, which leverages translations on processes and types/contexts, denoted $\langle\!| \cdot |\!\rangle$ and $(\!| \cdot |\!)_l$, respectively. Concretely, $\mathcal{W}$ is defined as follows:

Definition 4.1 ($\mathcal{W}$). *We define:*

$$\mathcal{W} = \{P \in \pi_{\mathsf{S}} \mid \exists \Gamma, l \text{ s.t. } (\Gamma \vdash_{\mathsf{S}} P) \wedge (\!|\Gamma|\!)_l \vdash_{\mathsf{W}} \langle\!|P|\!\rangle\}$$

Hence, $\mathcal{W}$ contains those processes from $\mathcal{S}$ (Definition 2.8) whose translation gives typable π_{W} processes. By Theorem 3.2, $\mathcal{W}$ thus provides a characterization of terminating processes in $\mathcal{S}$. In the following we formally define the translations $\langle\!| \cdot |\!\rangle$ and $(\!| \cdot |\!)_l$, and establish their main properties. Our main result is that $\mathcal{W} \subset \mathcal{S}$ (Theorem 4.3): there are typable processes in $\mathcal{S}$ which are not terminating under the weight-based approach.

4.1 The Typed Translation

Our translation is *typed*, i.e., the translation of a π_{S} process depends on its associated (session) types. We first present the translation on

$$[\mathtt{W{:}Var_1}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma, x : V \vdash_{\mathrm{W}} x : V} \qquad [\mathtt{W{:}Var_2}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma, x : \langle V, \overline{V}\rangle \vdash_{\mathrm{W}} x : V} \qquad [\mathtt{W{:}Nil}]\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash_{\mathrm{W}} \mathbf{0}} \qquad [\mathtt{W{:}Par}]\ \frac{\Gamma_1 \vdash_{\mathrm{W}} P \quad \Gamma_2 \vdash_{\mathrm{W}} Q}{\Gamma_1 \circ \Gamma_2 \vdash_{\mathrm{W}} P \mid Q} \qquad [\mathtt{W{:}Res}]\ \frac{\Gamma, x : \langle V, \overline{V}\rangle \vdash_{\mathrm{W}} P}{\Gamma \vdash_{\mathrm{W}} (\nu x)P}$$

$$[\mathtt{W{:}Lin-In_1}]\ \frac{\Gamma_1, x : \#^n(V_1, V_2) \vdash_{\mathrm{W}} x : \#^n(V_1, V_2) \qquad \Gamma_2, y_1 : V_1, y_2 : V_2 \vdash_{\mathrm{W}} P \qquad l(x) = l(y_2)}{\Gamma_1, x : \#^n(V_1, V_2) \circ \Gamma_2 \vdash_{\mathrm{W}} x(y_1, y_2).P}$$

$$[\mathtt{W{:}Lin-In_2}]\ \frac{\Gamma_1, x : *\#^n(V) \vdash_{\mathrm{W}} x : *\#^n(V) \qquad \Gamma_2, x : *\#^n(V), y_1 : V, y_2 : \mathbf{unit} \vdash_{\mathrm{W}} P}{(\Gamma_1, x : *\#^n(V)) \circ (\Gamma_2, x : *\#^n(V)) \vdash_{\mathrm{W}} x(y_1, y_2).P}$$

$$[\mathtt{W{:}Lin-In_3}]\ \frac{\Gamma, x :: *\#^n(V) \vdash_{\mathrm{W}} x : *\#^n(V) \qquad \Gamma, x :: *\#^n(V), y_1 : V, y_2 : \mathbf{unit} \vdash_{\mathrm{W}} P}{\Gamma, x :: *\#^n(V) \vdash_{\mathrm{W}} x(y_1, y_2).P}$$

$$[\mathtt{W{:}Lin-Out}]\ \frac{\Gamma_1, x : \#^n\langle V_1, V_2\rangle \vdash_{\mathrm{W}} x : \#^n\langle V_1, V_2\rangle \qquad \Gamma_2, y_1 : V_1 \vdash_{\mathrm{W}} y_1 : V_1 \qquad \Gamma_3, y_2 : V_2 \vdash_{\mathrm{W}} P \qquad l(x) = l(y_2)}{(\Gamma_1, x : \#^n\langle V_1, V_2\rangle) \circ (\Gamma_2, y_1 : V_1) \circ (\Gamma_3, y_2 :: V_2) \vdash_{\mathrm{W}} \overline{x}\langle y_1, y_2\rangle.P}$$

$$[\mathtt{W{:}Un-Out_1}]\ \frac{\Gamma_1, x : *\#^n\langle V\rangle \vdash_{\mathrm{W}} x : *\#^n\langle V\rangle \qquad \Gamma_2, x : *\#^n\langle V\rangle, y : V \vdash_{\mathrm{W}} y : V \qquad \Gamma_3, x : *\#^n\langle V\rangle, y_2 : \mathbf{unit} \vdash_{\mathrm{W}} P}{(\Gamma_1, x : *\#^n\langle V\rangle) \circ (\Gamma_2, x : *\#^n\langle V\rangle, y : V) \circ (\Gamma_3, x : *\#^n\langle V\rangle) \vdash_{\mathrm{W}} \overline{x}\langle y_1, y_2\rangle.P}$$

$$[\mathtt{W{:}Un-Out_2}]\ \frac{\Gamma_1, x :: *\#^n\langle V\rangle \vdash_{\mathrm{W}} x : *\#^n\langle V\rangle \qquad \Gamma_2, x :: *\#^n\langle V\rangle, y : V \vdash_{\mathrm{W}} y : V \qquad \Gamma_3, x :: *\#^n\langle V\rangle, y_2 : \mathbf{unit} \vdash_{\mathrm{W}} P}{(\Gamma_1, x :: *\#^n\langle V\rangle) \circ (\Gamma_2, x :: *\#^n\langle V\rangle, y : V) \circ (\Gamma_3, x :: *\#^n\langle V\rangle) \vdash_{\mathrm{W}} \overline{x}\langle y_1, y_2\rangle.P}$$

$$[\mathtt{W{:}Un-In_1}]\ \frac{\Gamma, x : *\#^n(V) \vdash_{\mathrm{W}} x : *\#^n(V) \qquad \Gamma, x : *\#^n(V), y_1 : V, y_2 : \mathbf{unit} \vdash_{\mathrm{W}} P \qquad \forall b \in \mathrm{os}(P),\ l(b) < n}{\Gamma, x : *\#^n(V) \vdash_{\mathrm{W}} !x(y_1, y_2).P}$$

$$[\mathtt{W{:}Un-In_2}]\ \frac{\Gamma, x :: *\#^n(V) \vdash_{\mathrm{W}} x : *\#^n(V) \qquad \Gamma, x :: *\#^n(V), y_1 : V, y_2 : \mathbf{unit} \vdash_{\mathrm{W}} P \qquad \forall b \in \mathrm{os}(P),\ l(b) < n}{\Gamma, x :: *\#^n(V) \vdash_{\mathrm{W}} !x(y_1, y_2).P}$$

Figure 9: Typing rules for π_{W}

processes and types separately; then, we combine them to define the translation of a typing judgment.

Definition 4.2 (Translating Processes). *The translation $\langle\!\vert \cdot \vert\!\rangle : \pi_{\mathrm{S}} \to \pi_{\mathrm{W}}$ is given in Figure 10 (top), where we assume z is fresh.*

We discuss some interesting cases in the translation of processes:

- The shape of process $\langle\!\vert \mathsf{lin}\ x(y).P \vert\!\rangle$ depends on whether x has a linear or an unrestricted type: this is due to rule [S:In] (Figure 4) which depends on a qualifier q_2 that can be linear or unrestricted. If $x : \mathsf{lin}?T.S$ then the translation is $x(y,z).\langle\!\vert P[{}^{z}\!/x] \vert\!\rangle$, with the continuation along z; otherwise, in case $x : *?T$, the translation is $x(y,z).\langle\!\vert P \vert\!\rangle$, since there is no continuation in x, as explained in the description of rule [S:In] in Figure 4.
- The process $\langle\!\vert \mathsf{un}\ x(y).P \vert\!\rangle$ is simply an unrestricted input process $!x(y,z).\langle\!\vert P \vert\!\rangle$.
- The process $\langle\!\vert \overline{x}\langle y\rangle.P \vert\!\rangle$, the translation of a bound send, also depends on the type of x and the justification for it is similar to the translation of linear inputs described above.
- The process $\langle\!\vert (\nu xy)P \vert\!\rangle$ is simply $(\nu z)\langle\!\vert P[{}^{z}\!/x][{}^{z}\!/y] \vert\!\rangle$: the co-variables x, y are replaced by the restricted (fresh) name z. The duality between the types of x and y, say $x : L$ and $y : \overline{L}$, must be preserved by the type of z in π_{W}. This correspondence will become evident when discussing the translation of judgements (Definition 4.4).

Definition 4.3 (Translating Types/Contexts). *The translation $(\!\vert - \vert\!)_l$ of session types and contexts is given in Figure 10 (bottom). The translation of contexts is parametric on a level function l. In particular, the translation of a type assignment $(\!\vert x : T \vert\!)_l$, relies on an auxiliary translation $x : (\!\vert T \vert\!)_l^x$, which is deemed to be assigned a level $l(x)$ in the translated type $(\!\vert T \vert\!)_l^x$, depending on the shape of T. Other names, denoted $\alpha, \beta, \gamma \ldots$, are necessary when translating within types.*

The translation $(\!\vert - \vert\!)_l$ follows the continuation-passing approach of [2] to encode session types into link types. The translation of tail-recursive types is rather direct, and self-explanatory.

By combining the translations of types and processes in Figure 10 we obtain a translation of type judgements / derivations in π_{S} into type judgements / derivations in π_{W}. We use an auxiliary notation:

Definition 4.4 (Translating Judgements/Derivations). *The translation of a type judgment for π_{S} into a type judgment for π_{W} is*

$$
\begin{array}{rcl}
\llparenthesis \mathbf{0} \rrparenthesis & = & \mathbf{0} \\
\llparenthesis P \mid Q \rrparenthesis & = & \llparenthesis P \rrparenthesis \mid \llparenthesis Q \rrparenthesis \\
\llparenthesis (\nu xy)P \rrparenthesis & = & (\nu z)\llparenthesis P[^z/x][^z/y] \rrparenthesis \\
\llparenthesis \mathsf{lin}\ x(y).P \rrparenthesis & = & \begin{cases} x(y,z).\llparenthesis P[^z/x] \rrparenthesis & \text{If } x : \mathsf{lin}?T.S \\ x(y,z).\llparenthesis P \rrparenthesis & \text{If } x : *?T \end{cases} \\
\llparenthesis \mathsf{un}\ x(y).P \rrparenthesis & = & !x(y,z).\llparenthesis P \rrparenthesis \\
\llparenthesis \overline{x}\langle y \rangle.P \rrparenthesis & = & \begin{cases} (\nu z)\overline{x}\langle y,z \rangle.\llparenthesis P[^z/x] \rrparenthesis & \text{If } x : \mathsf{lin}!T.S \\ \overline{x}\langle y,z \rangle.\llparenthesis P \rrparenthesis & \text{If } x : *!T \end{cases} \\
\hline
\llparenthesis \Gamma \rrparenthesis_l & = & \llparenthesis x_1 : T_1 \rrparenthesis_l, \cdots, \llparenthesis x_n : T_n \rrparenthesis_l \\
\llparenthesis x : T \rrparenthesis_l & = & x : \llparenthesis T \rrparenthesis_l^x \\
\llparenthesis \mathbf{end} \rrparenthesis_l^x & = & \mathbf{unit} \\
\llparenthesis \mathsf{lin}\ ?T.S \rrparenthesis_l^x & = & \#^{l(x)}(\llparenthesis T \rrparenthesis_l^\alpha, \llparenthesis S \rrparenthesis_l^x) \\
\llparenthesis \mathsf{lin}\ !T.S \rrparenthesis_l^x & = & \#^{l(x)}\langle \llparenthesis T \rrparenthesis_l^\beta, \llparenthesis S \rrparenthesis_l^x \rangle \\
\llparenthesis *\ ?T \rrparenthesis_x^l & = & *\#^{l(x)}(\llparenthesis T \rrparenthesis_l^\gamma) \\
\llparenthesis *\ !T \rrparenthesis_x^l & = & *\#^{l(x)}\langle \llparenthesis T \rrparenthesis_l^\gamma \rangle
\end{array}
$$

Figure 10: From π_S to π_W (Definition 4.2 and 4.3)

parametric on the level function $l : \mathcal{N} \to \mathbb{N}$, and is defined as:

$$\llbracket \Gamma \vdash_\mathsf{S} P \rrbracket_l = \llparenthesis \Gamma \rrparenthesis_l \vdash_\mathsf{W} \llparenthesis P \rrparenthesis$$
$$\llbracket \Gamma, x : T \vdash_\mathsf{S} x : T \rrbracket_l = \llparenthesis \Gamma \rrparenthesis_l, x : \llparenthesis T \rrparenthesis_l^x \vdash_\mathsf{W} x : \llparenthesis T \rrparenthesis_l^x$$

This translation induces an inductive construction of the translation of type derivations in π_W from type derivations in π_S, denoted as:

$$\left\llbracket [\mathsf{S{:}Rule}]\ \frac{\Upsilon_i \qquad \forall i \in I}{\Gamma \vdash_\mathsf{S} P} \right\rrbracket_l = [\mathsf{W{:}Rule}]\ \frac{\llbracket \Upsilon_i \rrbracket_l \qquad \forall i \in I}{\llparenthesis \Gamma \rrparenthesis_l \vdash_\mathsf{W} \llparenthesis P \rrparenthesis}$$

where Υ_i denotes a set of derivations used to prove $\Gamma \vdash_\mathsf{S} P$.

The translation, of which Figure 11 gives an excerpt, relies on analyzing the last rule [S:Rule] *applied in the derivation $\Gamma \vdash_\mathsf{S} P$ and the unfolding of the translation of judgements, mapping to a derivation $\llparenthesis \Gamma \rrparenthesis_l \vdash_\mathsf{W} \llparenthesis P \rrparenthesis$ in π_W, in which the last rule applied is* [`W:rule`].

4.2 Results

In general, the translation of a $P \in \mathcal{S}$ is not necessarily typable in π_W; this occurs when, e.g., P is non-terminating. We focus on processes in $\mathcal{S}$ that are typable in π_W, and therefore, are terminating.

Notation 4.1. *We write $\llparenthesis \Gamma \rrparenthesis_l \vdash_\mathsf{W} \llparenthesis P \rrparenthesis$ if $\llbracket \Gamma \vdash_\mathsf{S} P \rrbracket_l$ holds, for some l.*

Our translations are correct, in the following sense:

Theorem 4.1 (Operational Completeness). *Let $P \in \mathcal{W}$ such that $\llparenthesis \Gamma \rrparenthesis_l \vdash_\mathsf{W} \llparenthesis P \rrparenthesis$, for some level function l. Then there exists $R \in \mathcal{W}$ such that $P \longrightarrow Q \implies \llparenthesis P \rrparenthesis \xrightarrow{\tau} \llparenthesis R \rrparenthesis$ and $R \equiv Q$.*

Theorem 4.2 (Operational Soundness). *Let $P \in \mathcal{W}$ with $\llparenthesis \Gamma \rrparenthesis_l \vdash_\mathsf{W} \llparenthesis P \rrparenthesis$, for some level function l. If $\llparenthesis P \rrparenthesis \xrightarrow{\tau} U$ Then there exists $R, Q \in \mathcal{W}$ such that $P \longrightarrow Q \wedge R \equiv Q \wedge U = \llparenthesis R \rrparenthesis$.*

An immediate corollary of Theorem 4.1 is that our translation preserves (non-)terminating behaviour, i.e., does not map non-terminating processes in $\mathcal{S}$ into terminating processes in π_W.

Corollary 4.1. *$\llparenthesis \cdot \rrparenthesis$ preserves (non-)terminating behaviour.*

The following result corroborates our informal intuitions about $\mathcal{S}$ and $\mathcal{W}$. It also precisely characterizes a class of terminating processes based on our correct translations $\llparenthesis \cdot \rrparenthesis$ and $\llparenthesis \cdot \rrparenthesis_l$.

Theorem 4.3. *$\mathcal{W} \subset \mathcal{S}$.*

Proof (Sketch). The inclusion $\mathcal{W} \subseteq \mathcal{S}$ is immediate by definition. To prove that the inclusion is strict, we consider a counterexample, i.e., a process P typable in π_S but not typable in π_W. Process $P_{2.1}$ from Example 2.1 suffices for this purpose. □

5 PROPOSITIONS AS SESSIONS

We now introduce π_DILL, the process model induced by the Curry-Howard correspondence between linear types and session types (*propositions-as-sessions*) [1]. π_DILL is a synchronous π-calculus extended with (binary) guarded choice and forwarding.

Definition 5.1 (Processes and Types). *Processes in π_DILL are given by the grammar in Figure 12 (top). Types coincide with linear logic propositions, as given in the grammar in Figure 12 (bottom).*

Definition 5.2 (Reduction in π_DILL). *The reduction semantics of π_DILL is defined in Figure 13 (bottom), relying on structural congruence, the least congruence relation defined in Figure 13 (top).*

Notation 5.1 (Process Abbreviations). *We adopt the following abbreviations for bound outputs and replicated forwarders:*

$$\overline{x}(z).P = (\nu z)x\langle z \rangle.P$$
$$![x \leftrightarrow y] = !y(z).\overline{x}(k).[k \leftrightarrow z]$$

As usual, a type environment is a collection of type assignments $x : A$ where x is a name and A a type, the names being pairwise disjoint. The empty environment is denoted '$\cdot$'. We consider *unrestricted* environments (denoted Γ, Γ') and *linear* environments (denoted as Δ, Δ'); while the former satisfy weakening and contraction, the latter do not.

We denote by $dom(\Gamma)$, the *domain of* Γ, the set of names whose type assignments are in Γ, i.e., $dom(\Gamma) = \{x \mid x : A \in \Gamma\}$. Also, $\Gamma(x)$ denotes the type of the name $x \in dom(\Gamma)$, i.e., $\Gamma(x) = A$, if $x : A \in \Gamma$. The domain of Δ and $\Delta(x)$ are similarly defined.

Typing judgments for π_DILL are of the form $\Gamma; \Delta \vdash_\ell P :: x : A$. Such a judgment is intuitively read as: "P provides protocol A along x by using the protocols described in the assignments in Γ and Δ". The domains of Γ, Δ and $x : A$ are pairwise disjoint. The corresponding type rules are given in Figure 14. Each logical operator is represented by right and left rules: the former explains how to *offer* a behavior (according to the operator's interpretation, cf. Figure 12 (bottom)); the latter explains how to *make use* of a behavior typed with the operator. In particular, the behavior of clients and servers is governed by four typing rules: [L:cut$^!$], [L:copy], [L:!L], and [L:!R].

The Curry-Howard correspondence connects the logical principle of cut elimination with process synchronization. As a result, we have the fundamental property ensured by typing:

Theorem 5.1 (Type Preservation). *If $\Gamma; \Delta \vdash_\ell P :: x : A$ and $P \longrightarrow Q$ then $\Gamma; \Delta \vdash_\ell Q :: x : A$.*

Case: Input

$$\left\llbracket \frac{\begin{array}{l}[\mathsf{S{:}Lin-In_1}]\\ \Gamma_1' \vdash_{\mathsf{S}} x : \mathsf{lin}?T.S \qquad \Gamma_2, x : S, y : T \vdash_{\mathsf{S}} P\end{array}}{\Gamma_1' \circ \Gamma_2 \vdash_{\mathsf{S}} \mathsf{lin}\, x(y).P} \right\rrbracket_l = [\mathsf{W{:}Lin-In_1}]\ \frac{\begin{array}{c}\llparenthesis\Gamma_1'\rrparenthesis_l \vdash_{\mathsf{W}} x : \#^{l(x)}(\llparenthesis T\rrparenthesis_l^y, \llparenthesis S\rrparenthesis_l^z)\\ \llparenthesis\Gamma_2\rrparenthesis_l, y : \llparenthesis T\rrparenthesis_l^y, z : \llparenthesis S\rrparenthesis_l^z \vdash_{\mathsf{W}} \langle\!| P[z/x] |\!\rangle \qquad l(x) = l(z)\end{array}}{\llparenthesis\Gamma_1'\rrparenthesis_l \circ \llparenthesis\Gamma_2\rrparenthesis_l \vdash_{\mathsf{W}} x(y,z).\langle\!| P[z/x] |\!\rangle}$$

where $\Gamma_1' = \Gamma_1, x : \mathsf{lin}?T.S$ where $\llparenthesis\Gamma_1'\rrparenthesis_l = \llparenthesis\Gamma_1\rrparenthesis_l, x : \#^{l(x)}(\llparenthesis T\rrparenthesis_l^y, \llparenthesis S\rrparenthesis_l^z)$

$$\left\llbracket \frac{\begin{array}{l}[\mathsf{S{:}Lin-In_2}]\\ \Gamma_1' \vdash_{\mathsf{S}} x : *?T \qquad \Gamma_2', y : T \vdash_{\mathsf{S}} P\end{array}}{\Gamma_1' \circ \Gamma_2' \vdash_{\mathsf{S}} \mathsf{lin}\, x(y).P} \right\rrbracket_l = [\mathsf{W{:}Lin-In_2}]\ \frac{\llparenthesis\Gamma_1'\rrparenthesis_l \vdash_{\mathsf{W}} x : *\#^{l(x)}(\llparenthesis T\rrparenthesis_l^y) \qquad \llparenthesis\Gamma_2'\rrparenthesis_l, z : \mathbf{unit} \vdash_{\mathsf{W}} \langle\!| P |\!\rangle}{\llparenthesis\Gamma_1'\rrparenthesis_l \circ \llparenthesis\Gamma_2'\rrparenthesis_l \vdash_{\mathsf{W}} x(y,z).\langle\!| P |\!\rangle}$$

where $\Gamma_1' = \Gamma_1, x : *?T$ and $\Gamma_2' = \Gamma_2, x : *?T$ where $\llparenthesis\Gamma_1'\rrparenthesis_l = \llparenthesis\Gamma_1\rrparenthesis_l, x : *\#^{l(x)}(\llparenthesis T\rrparenthesis_l^y)$ and $\llparenthesis\Gamma_2'\rrparenthesis_l = \llparenthesis\Gamma_2\rrparenthesis_l, x : *\#^{l(x)}(\llparenthesis T\rrparenthesis_l^y)$

$$\left\llbracket \frac{\begin{array}{l}[\mathsf{S{:}Un-In}]\\ \Gamma' \vdash_{\mathsf{S}} x : *?T \qquad \Gamma', y : T \vdash_{\mathsf{S}} P\end{array}}{\Gamma, x : *?T \vdash_{\mathsf{S}} \mathsf{un}\, x(y).P} \right\rrbracket_l = [\mathsf{W{:}Un-In_1}]\ \frac{\begin{array}{c}\llparenthesis\Gamma'\rrparenthesis_l \vdash_{\mathsf{W}} x : *\#^{l(x)}(\llparenthesis T\rrparenthesis_l^y)\\ \llparenthesis\Gamma'\rrparenthesis_l, y : \llparenthesis T\rrparenthesis_l^y, z : \mathbf{unit} \vdash_{\mathsf{W}} \langle\!| P |\!\rangle \qquad \forall b \in \mathrm{os}(\langle\!| P |\!\rangle),\ l(b) < l(x)\end{array}}{\llparenthesis\Gamma\rrparenthesis_l, x : *\#^{l(x)}(\llparenthesis T\rrparenthesis_l^y) \vdash_{\mathsf{W}} !x(y,z).\langle\!| P |\!\rangle}$$

where $\Gamma' = \Gamma, x : *?T$ where $\llparenthesis\Gamma'\rrparenthesis_l = \llparenthesis\Gamma\rrparenthesis_l, x : *\#^{l(x)}\llparenthesis T\rrparenthesis_l^y$

Case: Output

$$\left\llbracket \frac{\begin{array}{l}[\mathsf{S{:}Lin-Out}]\\ \Gamma_1' \vdash_{\mathsf{S}} x : \mathsf{lin}!T.S \qquad \Gamma_2' \vdash_{\mathsf{S}} y : T \qquad \Gamma_3, x : S \vdash_{\mathsf{S}} P\end{array}}{\Gamma_1' \circ \Gamma_2' \circ \Gamma_3 \vdash_{\mathsf{S}} \overline{x}\langle y\rangle.P} \right\rrbracket_l = [\mathsf{W{:}Lin-Out}]\ \frac{\dfrac{\begin{array}{c}\llparenthesis\Gamma_1'\rrparenthesis_l \vdash_{\mathsf{W}} x : \#^{l(x)}\langle\llparenthesis T\rrparenthesis_l^y, \llparenthesis S\rrparenthesis_l^z\rangle\\ \llparenthesis\Gamma_2'\rrparenthesis_l \vdash_{\mathsf{W}} y : \llparenthesis T\rrparenthesis_l^y \qquad \llparenthesis\Gamma_3\rrparenthesis_l, z : \llparenthesis S\rrparenthesis_l^z \vdash_{\mathsf{W}} \langle\!| P[z/x] |\!\rangle \qquad l(x) = l(z)\end{array}}{(\llparenthesis\Gamma_1'\rrparenthesis_l \circ \llparenthesis\Gamma_2'\rrparenthesis_l \circ \llparenthesis\Gamma_3\rrparenthesis_l), z :: \llparenthesis S\rrparenthesis_l^z \vdash_{\mathsf{W}} \overline{x}\langle y,z\rangle.(\langle\!| P[z/x] |\!\rangle)}}{\llparenthesis\Gamma_1'\rrparenthesis_l \circ \llparenthesis\Gamma_2'\rrparenthesis_l \circ \llparenthesis\Gamma_3\rrparenthesis_l \vdash_{\mathsf{W}} (\nu z)\overline{x}\langle y,z\rangle.(\langle\!| P[z/x] |\!\rangle)}$$

where $\Gamma_1' = \Gamma_1, x : \mathsf{lin}!T.S$ and $\Gamma_2' = \Gamma_2, y : T$ where $\llparenthesis\Gamma_1'\rrparenthesis_l = \llparenthesis\Gamma_1\rrparenthesis_l, x : \#^{l(x)}\langle\llparenthesis T\rrparenthesis_l^y, \llparenthesis S\rrparenthesis_l^z\rangle$ and $\llparenthesis\Gamma_2'\rrparenthesis_l = \llparenthesis\Gamma_2\rrparenthesis_l, y : \llparenthesis T\rrparenthesis_l^y$

$$\left\llbracket \frac{\begin{array}{l}[\mathsf{S{:}Un-Out}]\\ \Gamma_1' \vdash_{\mathsf{S}} x : *!T \qquad \Gamma_2' \vdash_{\mathsf{S}} y : T \qquad \Gamma_3' \vdash_{\mathsf{S}} P\end{array}}{\Gamma_1' \circ \Gamma_2' \circ \Gamma_3' \vdash_{\mathsf{S}} \overline{x}\langle y\rangle.P} \right\rrbracket_l = [\mathsf{W{:}Un-Out_1}]\ \frac{\begin{array}{c}\llparenthesis\Gamma_1'\rrparenthesis_l \vdash_{\mathsf{W}} x : *\#^{l(x)}\llparenthesis T\rrparenthesis_l^y\\ \llparenthesis\Gamma_2'\rrparenthesis_l \vdash_{\mathsf{W}} y : \llparenthesis T\rrparenthesis_l^y \qquad \llparenthesis\Gamma_3'\rrparenthesis_l, z : \mathbf{unit} \vdash_{\mathsf{W}} \langle\!| P |\!\rangle\end{array}}{\llparenthesis\Gamma_1'\rrparenthesis_l \circ \llparenthesis\Gamma_2'\rrparenthesis_l \circ \llparenthesis\Gamma_3'\rrparenthesis_l \vdash_{\mathsf{W}} \overline{x}\langle y,z\rangle.\langle\!| P |\!\rangle}$$

where $\Gamma_1' = \Gamma_1, x : *!T$, $\Gamma_2' = \Gamma_2, x : *!T, y : T$, $\Gamma_3' = \Gamma_3, x : *!T$ where $\llparenthesis\Gamma_1'\rrparenthesis_l = \llparenthesis\Gamma_1\rrparenthesis_l, x : *\#^{l(x)}\llparenthesis T\rrparenthesis_l^y$, $\llparenthesis\Gamma_2'\rrparenthesis_l = \llparenthesis\Gamma_2\rrparenthesis_l, x : *\#^{l(x)}\llparenthesis T\rrparenthesis_l^y, y : \llparenthesis T\rrparenthesis_l^y$, $\llparenthesis\Gamma_3'\rrparenthesis_l = \llparenthesis\Gamma_3\rrparenthesis_l, x : *\#^{l(x)}\llparenthesis T\rrparenthesis_l^y$

Figure 11: From derivations in π_{S} to derivations in π_{W} (excerpt, cf. Definition 4.4)

The type system enforces also progress and termination. The latter property can be proven using logical relations [12].

6 $\mathcal{L}$: A CLASS OF TERMINATING PROCESSES

We now study $\mathcal{L}$, another class of terminating π_{S} processes. This class is induced by the Curry-Howard system given in §5, which leverages translations on processes and types/contexts, denoted $\langle\!| \cdot |\!\rangle$ and $\llparenthesis \cdot \rrparenthesis$, respectively. Roughly, $\mathcal{L}$ is defined as follows:

$$\mathcal{L} = \{P \in \pi_{\mathsf{S}} \mid \Gamma \circledast \Delta \vdash_{\mathsf{S}} P \ \wedge\ \llparenthesis\Gamma\rrparenthesis; \llparenthesis\Delta\rrparenthesis \vdash_{\ell} \langle\!| P |\!\rangle :: u : \llparenthesis\overline{S}\rrparenthesis\}$$

Definition 6.7 will give a formal definition. In the following we define the translations $\langle\!| \cdot |\!\rangle$ and $\llparenthesis \cdot \rrparenthesis$, and establish their properties. Our main result is that $\mathcal{L} \subset \mathcal{W}$ but $\mathcal{W} \not\subset \mathcal{L}$ (Theorems 6.3 and 6.2): there are terminating processes detected as such by the weight-based approach but not by the Curry-Howard correspondence.

We require some auxiliary definitions. The following predicates say whether a session type contains client or server behaviors.

Definition 6.1. *Given a session type T, we define predicates $\mathsf{svr}(T)$ and $\mathsf{cli}(T)$ as follows:*

$$\begin{array}{rlrl}
\mathsf{svr}(*?T) & = \mathtt{true} & \mathsf{cli}(*!T) & = \mathtt{true}\\
\mathsf{svr}(\mathbf{end}) & = \mathtt{false} & \mathsf{cli}(\mathbf{end}) & = \mathtt{false}\\
\mathsf{svr}(q\,!S.T) & = \mathsf{svr}(T) & \mathsf{cli}(q\,!S.T) & = \mathsf{cli}(T)\\
\mathsf{svr}(q\,?S.T) & = \mathsf{svr}(T) & \mathsf{cli}(q\,?S.T) & = \mathsf{cli}(T)\\
\mathsf{svr}(*!T) & = \mathsf{svr}(T) & \mathsf{cli}(*?T) & = \mathsf{cli}(T)
\end{array}$$

These predicates extend to contexts Γ as expected. This way, e.g., $\mathsf{svr}(\Gamma)$ stands for $\bigwedge_{x \in dom(\Gamma)} \Gamma(x)$. Also, we write $\mathsf{svr}(\Gamma; P)$ to stand

$P, Q ::=$	(Processes)	$x(y).P$	(linear input)
$x\langle y\rangle.P$	(output)	$P \mid Q$	(composition)
$!x(y).P$	(server)	$\mathbf{0}$	(inaction)
$(\nu x)P$	(restriction)	$x.\mathbf{inr}; P$	(select right)
$x.\mathbf{case}(P, Q)$	(branching)	$[x \leftrightarrow y]$	(forwarding)
$x.\mathbf{inl}; P$	(select left)		
$A, B ::=$	(Types)		
$\mathbf{1}$	(Termination)	$!A$	(Shared)
$A \multimap B$	(Receive)	$A \otimes B$	(Send)
$A \oplus B$	(Selection)	$A \& B$	(Branching)

Figure 12: Processes and types of the session π-calculus π_{DILL}

$$P \mid \mathbf{0} \equiv P \qquad P \equiv_\alpha Q \implies P \equiv Q \qquad P \mid Q \equiv Q \mid P$$
$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$$
$$x \notin \mathsf{fn}(P) \implies P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)$$

$(\mathrm{R}{\leftrightarrow})$	$(\nu x)([x \leftrightarrow y] \mid P) \longrightarrow P[y/x] \quad \text{if } x \neq y$
(RC)	$x\langle y\rangle.P \mid x(z).Q \longrightarrow P \mid Q[y/z]$
$(\mathrm{R!})$	$x\langle y\rangle.P \mid !x(z).Q \longrightarrow P \mid Q[y/z] \mid !x(z).Q$
(RL)	$x.\mathbf{inl}; P \mid x.\mathbf{case}(Q, R) \longrightarrow P \mid Q$
(RR)	$x.\mathbf{inr}; P \mid x.\mathbf{case}(Q, R) \longrightarrow P \mid R$
$(\mathrm{R}\mid)$	$Q \longrightarrow R \implies P \mid Q \longrightarrow P \mid R$
$(\mathrm{R}\nu)$	$P \longrightarrow Q \implies (\nu x)P \longrightarrow (\nu x)Q$
$(\mathrm{R}{\equiv})$	$P \equiv P' \wedge P' \longrightarrow Q' \wedge Q' \equiv Q \implies P \longrightarrow Q$

Figure 13: Structural congruence and reductions for π_{DILL}

for $\bigwedge_{x\in(\mathsf{fn}(P)\cap dom(\Gamma))} \mathsf{svr}(\Gamma(x))$, *returning* `true` *when* $(\mathsf{fn}(P) \cap dom(\Gamma)) = \emptyset$. *Analogous definitions for* $\mathsf{cli}(\cdot)$, $\neg\mathsf{svr}(\cdot)$, *and* $\neg\mathsf{cli}(\cdot)$ *arise similarly.*

This way, intuitively:

- $\neg\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T)$ means that T is an always-linear behavior, i.e., it does not contain server and client actions.
- $\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T)$ means that T contains some server behavior and that it does not contain client behaviors.
- $\neg\mathsf{svr}(T) \wedge \mathsf{cli}(T)$ means that T will at some point exhibit client behaviors and that it does not contain server behaviors.

Also, $\mathsf{svr}(T) \wedge \mathsf{cli}(T)$ means that T contains both server and client actions; this combination, however, is excluded by typing.

Example 6.1. *We further illustrate Definition 6.1 by example:*

	T	$\mathsf{svr}(T)$	$\mathsf{cli}(T)$
1	$*!(\mathsf{lin}\ !(*!S).(\mathsf{lin}\ ?(*?R).(*?T_0)))$	`true`	`true`
2	$\mathsf{lin}\ !(*!S).(\mathsf{lin}\ ?(*?R).(*?T_0))$	`true`	`false`
3	$*!(\mathsf{lin}\ !(*!S).(\mathsf{lin}\ ?(*?R).\mathbf{end}))$	`false`	`true`
4	$\mathsf{lin}\ !(*!S).(\mathsf{lin}\ ?(*?R).\mathbf{end})$	`false`	`false`

Both (1) *and* (2) *return* `true` *for* $\mathsf{svr}(T)$ *because of their final behavior (i.e., '$*?T_0$'), whereas* (3) *and* (4) *return* `false`*, because their final behavior is* **end**. *Both* (1) *and* (3) *return* `true` *for* $\mathsf{cli}(T)$ *as their initial type behavior (i.e., '$*!T'$') is that of a client, whereas* (2) *and* (4) *return* `false` *as they do not contain any client behavior.*

6.1 The Typed Translation

Definition 6.2 (Translating Processes). *The translation* $\llparenthesis \cdot \rrparenthesis : \pi_{\mathsf{S}} \to \pi_{\mathsf{DILL}}$ *is given in Figure 15.*

The translation of processes relies on type information; in particular, the translation of outputs and unrestricted inputs depends on whether the overall behavior of channels exhibits server or client behaviors (cf. Definition 6.1). In translating outputs, we check whether the output is free or bound. The translation of free outputs is further influenced by whether the sender is associated with a linear connection or acts as a client connected to a server. There are 5 cases to consider, and the translated processes are designed to preserve typability. Similar conditions apply to the translation of bound outputs.

Remark 1. *To ensure typability of the translated process, we explain some of the choices in Figure 15:*

1. *In a free output $\overline{x}\langle z\rangle.P$ the value z cannot have a server behavior. In Figure 15, this is ensured using the predicate $\neg\mathsf{svr}(T)$.*
2. *In an unrestricted bound output $(\nu xy)\overline{z}\langle y\rangle.P$, the value y cannot have a client behavior. In Figure 15, this is ensured using the predicate $\neg\mathsf{cli}(T)$.*

We illustrate what we mean by "client behavior" above. Consider the process $P = (\nu xy)((\nu wv)\overline{x}\langle v\rangle.\mathsf{un}\ w(a).\mathbf{0} \mid \mathsf{un}\ y(c).\overline{c}\langle b\rangle.\mathbf{0})$. In P, the output action on x is an unrestricted bound output, whose object v has a client behavior: after one reduction, an output on v will be ready to invoke the server on w. Notice that $P \in \mathcal{S}$, as P is typable with $b : \mathbf{end} \vdash_{\mathsf{S}} P$ and $x : *!(*!\mathbf{end}), y : *?(*!\mathbf{end}), w : *?\mathbf{end}$ and $v : *!\mathbf{end}$.

We want a typable translation of the judgement $[\![\Gamma \vdash_{\mathsf{S}} P]\!]_u$. Consider the partial translation of P, i.e., $\llparenthesis P \rrparenthesis = (\nu x)(\llparenthesis P_1 \rrparenthesis \mid \llparenthesis Q_1 \rrparenthesis[x/y])$, where we use the abbreviations

- *$P_1 = (\nu wv)\overline{x}\langle v\rangle.\mathsf{un}\ w(a).\mathbf{0}$, and*
- *$Q_1 = \mathsf{un}\ y(c).\overline{c}\langle b\rangle.\mathbf{0}$.*

Suppose we can apply $[\mathsf{L{:}cut}]$*, then there are derivations Π_1 and Π_2 such that*

$$\dfrac{\dfrac{\Pi_1}{b : \mathbf{1}; \cdot \vdash_\ell \llparenthesis Q_1 \rrparenthesis[x/y] :: y : !(\mathbf{1} \multimap \mathbf{1})} \qquad \dfrac{\Pi_2}{b : \mathbf{1}; x : !(\mathbf{1} \multimap \mathbf{1}) \vdash_\ell \llparenthesis P_1 \rrparenthesis :: u : T}}{b : \mathbf{1}; \cdot \vdash_\ell (\nu x)(\llparenthesis P_1 \rrparenthesis \mid \llparenthesis Q_1 \rrparenthesis[x/y]) :: u : T}$$

Consider the partial translation $\llparenthesis (\nu wv)\overline{x}\langle v\rangle.\mathsf{un}\ w(a).\mathbf{0} \rrparenthesis = \overline{x}(w).(!w(a').P_1')$ for some P_1' that we will leave opaque for now. Notice, however, that the following derivation is not possible: to type $!w(a').P_1'$ we would need $u = w$ to apply $[\mathsf{L{:}!R}]$ *(the application of* $[\mathsf{L{:}copy}]$*), but w already occurs in the context and this contradicts the domain restriction of $\vdash_\ell$ judgements.*

$$[\mathsf{L{:}!L}]\ \dfrac{[\mathsf{L{:}copy}]\ \dfrac{b : \mathbf{1}, x : (\mathbf{1} \multimap \mathbf{1}); w : (\mathbf{1} \multimap \mathbf{1}) \nvdash_\ell !w(a).0 :: u : T}{b : \mathbf{1}, x : (\mathbf{1} \multimap \mathbf{1}); \cdot \vdash_\ell !w(a').P_1' :: u : T}}{b : \mathbf{1}; x : !(\mathbf{1} \multimap \mathbf{1}) \vdash_\ell \overline{x}(w).(!w(a').P_1') :: u : T}$$

A similar argument and example can be used to justify the first item of this remark.

While the translation of linear inputs is straightforward, in translating unrestricted inputs we check whether the synchronization concerns a bound or free output. When the unrestricted input cannot discern the client or server behavior from the type, it offers both

$$[\mathsf{L{:}1L}]\ \frac{\Gamma;\Delta \vdash_\ell P :: T}{\Gamma;\Delta, x:\mathbf{1} \vdash_\ell P :: T} \qquad [\mathsf{L{:}1R}]\ \frac{}{\Gamma;\cdot \vdash_\ell \mathbf{0} :: x:\mathbf{1}} \qquad [\mathsf{L{:}fwd}]\ \frac{}{\Gamma; x:A \vdash_\ell [x \leftrightarrow y] :: y:A} \qquad [\mathsf{L{:}\otimes L}]\ \frac{\Gamma;\Delta, y:A, x:B \vdash_\ell P :: T}{\Gamma;\Delta, x:A\otimes B \vdash_\ell x(y).P :: T}$$

$$[\mathsf{L{:}\otimes R}]\ \frac{\Gamma;\Delta_1 \vdash_\ell P :: y:A \qquad \Gamma;\Delta_2 \vdash_\ell Q :: x:B}{\Gamma;\Delta_1,\Delta_2 \vdash_\ell \overline{x}(y).(P \mid Q) :: x:A\otimes B} \qquad [\mathsf{L{:}cut}]\ \frac{\Gamma;\Delta_1 \vdash_\ell P :: x:A \qquad \Gamma;\Delta_2, x:A \vdash_\ell Q :: T}{\Gamma;\Delta_1,\Delta_2 \vdash_\ell (\nu x)(P \mid Q) :: T}$$

$$[\mathsf{L{:}cut^!}]\ \frac{\Gamma;\cdot \vdash_\ell P :: y:A \qquad \Gamma, u:A;\Delta \vdash_\ell Q :: T}{\Gamma;\Delta \vdash_\ell (\nu u)(!u(y).P \mid Q) :: T} \qquad [\mathsf{L{:}copy}]\ \frac{\Gamma, u:A;\Delta, y:A \vdash_\ell P :: T}{\Gamma, u:A;\Delta \vdash_\ell \overline{u}(y).P :: T} \qquad [\mathsf{L{:}!L}]\ \frac{\Gamma, u:A;\Delta \vdash_\ell P[u/x] :: T}{\Gamma;\Delta, x:!A \vdash_\ell P :: T}$$

$$[\mathsf{L{:}!R}]\ \frac{\Gamma;\cdot \vdash_\ell Q :: y:A}{\Gamma;\cdot \vdash_\ell !x(y).Q :: x:!A} \qquad [\mathsf{L{:}\&L_1}]\ \frac{\Gamma;\Delta, x:A \vdash_\ell P :: T}{\Gamma;\Delta, x:A\&B \vdash_\ell x.\mathbf{inl};P :: T} \qquad [\mathsf{L{:}\oplus L}]\ \frac{\Gamma;\Delta, x:A \vdash_\ell P :: T \quad \Gamma;\Delta, x:B \vdash_\ell P :: T}{\Gamma;\Delta, x:A\oplus B \vdash_\ell x.\mathbf{case}(P,Q) :: T}$$

$$[\mathsf{L{:}\oplus R_2}]\ \frac{\Gamma;\Delta \vdash_\ell P :: x:B}{\Gamma;\Delta \vdash_\ell x.\mathbf{inr};P :: x:A\oplus B}$$

Figure 14: Type rules for π_{DILL} (selection)

$$\begin{aligned}
\llparenthesis \overline{x}\langle y\rangle.P \rrparenthesis &= \begin{cases} \overline{x}(z).([y\leftrightarrow z] \mid \llparenthesis P\rrparenthesis) & \text{If } x:\mathsf{lin}!(T).S \wedge \neg\mathsf{un}(T)\wedge\neg\mathsf{svr}(T).\\ \overline{x}(z).(![y\leftrightarrow z] \mid \llparenthesis P\rrparenthesis) & \text{If } x:\mathsf{lin}!(T).S \wedge \mathsf{un}(T)\wedge\neg\mathsf{svr}(T).\\ \overline{x}(z).\overline{z}(w).([y\leftrightarrow w] \mid \llparenthesis P\rrparenthesis) & \text{If } x:*!T \wedge \neg\mathsf{un}(T)\wedge\neg\mathsf{svr}(T)\wedge\mathsf{cli}(T)\\ \overline{x}(z).\overline{z}(w).(![y\leftrightarrow w] \mid \llparenthesis P\rrparenthesis) & \text{If } x:*!T \wedge \mathsf{un}(T)\wedge\neg\mathsf{svr}(T)\wedge\mathsf{cli}(T)\\ \overline{x}(z).z.\mathbf{inl};\overline{z}(w).(![y\leftrightarrow w] \mid \llparenthesis P\rrparenthesis) & \text{If } x:*!T \wedge \mathsf{un}(T)\wedge\neg\mathsf{svr}(T)\wedge\neg\mathsf{cli}(T)\\ \overline{x}(z).z.\mathbf{inl};\overline{z}(w).([y\leftrightarrow w] \mid \llparenthesis P\rrparenthesis) & \text{If } x:*!T \wedge \neg\mathsf{un}(T)\wedge\neg\mathsf{svr}(T)\wedge\neg\mathsf{cli}(T)\end{cases}\\
\llparenthesis (\nu xy)\overline{z}\langle y\rangle.P \rrparenthesis &= \begin{cases} \overline{z}(x).x.\mathbf{inr};\llparenthesis P\rrparenthesis & \text{If } z:*!T \wedge\neg\mathsf{svr}(T)\wedge\neg\mathsf{cli}(T).\\ \overline{z}(x).\llparenthesis P\rrparenthesis & \text{If } z:*!T \wedge\mathsf{svr}(T)\wedge\neg\mathsf{cli}(T).\end{cases}\\
\llparenthesis (\nu xy)\overline{z}\langle x\rangle.(P\mid Q) \rrparenthesis &= \overline{z}(y).(\llparenthesis P\rrparenthesis \mid \llparenthesis Q\rrparenthesis) \quad \text{If } z:\mathsf{lin}!T.S \wedge z\notin\mathsf{fn}(P)\wedge y\notin\mathsf{fn}(Q)\\
\llparenthesis \mathsf{lin}\ x(y).P \rrparenthesis &= x(y).\llparenthesis P\rrparenthesis \quad \text{If } x:\mathsf{lin}?T.S\\
\llparenthesis \mathsf{un}\ x(y).P \rrparenthesis &= \begin{cases} !x(z).\llparenthesis P[z/y]\rrparenthesis & \text{If } x:*?T\wedge\mathsf{svr}(T)\wedge\neg\mathsf{cli}(T).\\ !x(z).z(y).\llparenthesis P\rrparenthesis & \text{If } x:*?T\wedge\neg\mathsf{svr}(T)\wedge\mathsf{cli}(T).\\ !x(z).z.\mathbf{case}(z(y).\llparenthesis P\rrparenthesis, \llparenthesis P[z/y]\rrparenthesis) & \text{If } x:*?T\wedge\neg\mathsf{svr}(T)\wedge\neg\mathsf{cli}(T).\end{cases}\\
\llparenthesis (\nu xy)(P\mid Q) \rrparenthesis &= (\nu x)(\llparenthesis P\rrparenthesis \mid \llparenthesis Q\rrparenthesis[x/y]) \quad \text{If } y\notin\mathsf{fn}(P)\wedge x\notin\mathsf{fn}(Q)\\
\llparenthesis P\mid Q \rrparenthesis &= (\nu w)(\llparenthesis P\rrparenthesis \mid \llparenthesis Q\rrparenthesis) \quad \text{With } w \text{ fresh}\\
\llparenthesis \mathbf{0} \rrparenthesis &= \mathbf{0}
\end{aligned}$$

Figure 15: Translating processes in π_{S} into π_{DILL}

behaviors using a branching construct; the synchronizing party (i.e. the translation of output, free or bound) then determines the desired behavior using a corresponding selection construct.

Example 6.2. *Consider $P = (\nu xy)(\mathsf{un}\ x(z).\mathbf{0} \mid \overline{y}\langle w\rangle.\mathbf{0})$, a π_{S} process that implements a simple server-client communication. As in Example 2.1, one can verify that $x : *?\mathbf{end}, y : *!\mathbf{end}$, $w : \mathbf{end}$ and $z : \mathbf{end}$, which entail $w : \mathbf{end} \vdash_{\mathsf{S}} P$. Since $y \notin \mathsf{fn}(\mathsf{un}\ x(z).\mathbf{0})$ and $x \notin \mathsf{fn}(\overline{y}\langle w\rangle.\mathbf{0})$, the translation of P is as:*

$$\llparenthesis P\rrparenthesis = (\nu x)(\llparenthesis \mathsf{un}\ x(z).\mathbf{0}\rrparenthesis \mid \llparenthesis \overline{x}\langle w\rangle.\mathbf{0}\rrparenthesis)$$

Note that $\neg\mathsf{cli}(\mathbf{end}) \wedge \neg\mathsf{svr}(\mathbf{end}) \wedge \mathsf{un}(\mathbf{end})$ holds (cf. Definition 2.5 and Definition 6.1). Thus,

$$\llparenthesis \mathsf{un}\ x(z).\mathbf{0}\rrparenthesis = !x(v).v.\mathbf{case}(v(z).\mathbf{0}, \mathbf{0})$$
$$\llparenthesis \overline{x}\langle w\rangle.\mathbf{0}\rrparenthesis = \overline{x}(z).z.\mathbf{inl};\overline{z}(v).(![w\leftrightarrow v] \mid \mathbf{0})$$

Definition 6.3. *Given a session type/linear logic proposition A, we write $\dagger(A)$ to denote A without top-level occurrences of '$!$', i.e., $\dagger(!A) = A$ and is the identity function otherwise.*

Definition 6.4 (Translating Types/Contexts). *The translation $\llparenthesis\cdot\rrparenthesis$ from session types in π_{S} to logic propositions in π_{DILL} is given in Figure 16. The translation of types extends to contexts as expected; we shall write $\llparenthesis\Gamma\rrparenthesis^{\dagger}$ to stand for $\dagger(\llparenthesis\Gamma\rrparenthesis)$.*

The translation of **end** and linear input/output types is standard. As for client and servers, the translation of types follows the translation of processes. When the type of the client or server exhibits a server behavior, the type is encoded into an unrestricted type. Notice that a client type $*!T$ is translated into its dual behavior $!\llparenthesis\overline{T}\rrparenthesis$, but a server is not. This has to do with the left/right interpretation

$$
\begin{array}{lll}
(\!|\mathbf{end}|\!) & = !\mathbf{1} & \\
(\!|\mathsf{lin}!S.T|\!) & = (\!|S|\!) \multimap (\!|T|\!) & \\
(\!|\mathsf{lin}?S.T|\!) & = (\!|S|\!) \otimes (\!|T|\!) & \\
(\!| * ?T |\!) & = \begin{cases} !(\!|T|\!) & \text{If } \mathsf{svr}(T) \wedge \neg\mathsf{cli}(T). \\ !((\!|T|\!) \otimes \mathbf{1}) & \text{If } \neg\mathsf{svr}(T) \wedge \mathsf{cli}(T). \\ !(((\!|T|\!) \otimes \mathbf{1}) \oplus (\!|T|\!)) & \text{If } \neg\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T). \end{cases} & \\
(\!| * !T |\!) & = \begin{cases} !(\!|\overline{T}|\!) & \text{If } \mathsf{svr}(T) \wedge \neg\mathsf{cli}(T). \\ !((\!|T|\!) \multimap \mathbf{1}) & \text{If } \neg\mathsf{svr}(T) \wedge \mathsf{cli}(T). \\ !(((\!|T|\!) \multimap \mathbf{1}) \& (\!|\overline{T}|\!)) & \text{If } \neg\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T). \end{cases} &
\end{array}
$$

Figure 16: Translating session types into logical propositions

of judgments in π_{DILL}: servers always occur on the right-hand side; to provide a dual behavior, the client should itself be dual.

Example 6.3 (Cont. Example 6.2). *Consider the type assignments* $x : *?\mathbf{end}, y : *!\mathbf{end}$, $w : \mathbf{end}$ *and* $z : \mathbf{end}$. *Since* $\neg\mathsf{cli}(\mathbf{end}) \wedge \neg\mathsf{svr}(\mathbf{end})$, *the translation in Figure 16 gives:*

- $x : (\!| * ?\mathbf{end}|\!) = !(((\!|\mathbf{end}|\!) \otimes \mathbf{1}) \oplus (\!|\mathbf{end}|\!)) = !((!\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1})$;
- $y : (\!| * !\mathbf{end}|\!) = !(((\!|\mathbf{end}|\!) \multimap \mathbf{1}) \& (\!|\overline{\mathbf{end}}|\!)) = !((!\mathbf{1} \multimap \mathbf{1}) \& \mathbf{1})$

The translations to $z : \mathbf{1}$ *and* $w : \mathbf{1}$ *are trivial.*

Armed with the translations of processes and types given in Figure 15 and Figure 16, we are now ready to translate a judgment $\Gamma, \Delta \vdash_{\mathsf{S}} P$ into $(\!|\Gamma|\!)^{\dagger}; (\!|\Delta|\!) \vdash_{\ell} \langle\!|P|\!\rangle : u :: A$, for some name u. This translation requires that $\mathsf{un}(\Gamma)$ and $\neg\mathsf{un}(\Delta)$, i.e., Γ is unrestricted and Δ is 'not' unrestricted; this is the abbreviation $\Gamma \circledast \Delta$ (Notation 2.2).

The following auxiliary notion relates contexts that may differ in exactly one assignment:

Definition 6.5. *Given contexts* Γ, Γ', *we write* $\Gamma \asymp^{z}_{T} \Gamma'$ *if* $(\Gamma = \Gamma' \wedge z \notin \mathsf{dom}(\Gamma)) \vee (\Gamma = \Gamma', z : T)$ *for some type* T.

Definition 6.6 (Translating Judgements). *Given a judgment* $\Gamma \circledast \Delta \vdash_{\mathsf{S}} P$ *and a name* u, *its translation* $[\![\Gamma \circledast \Delta \vdash_{\mathsf{S}} P]\!]_u$ *is defined as*

$$(\!|\Gamma'|\!)^{\dagger}; (\!|\Delta'|\!) \vdash_{\ell} \langle\!|P|\!\rangle :: u : A$$

where Γ', Δ', *and* A *are subject to one of the following conditions:*

- $A = (\!|\overline{T}|\!)$ *when* $\{u : T\} \subset \Gamma, \Delta$ *with* $(\Gamma \asymp^{u}_{T} \Gamma') \wedge (\Delta \asymp^{u}_{T} \Delta')$; *or*
- $A = \mathbf{1}$ *when* $u \notin \mathsf{dom}(\Gamma, \Delta)$ *with* $(\Gamma = \Gamma') \wedge (\Delta = \Delta')$.

Table 1 defines the translation by induction on P, *assuming that the contexts satisfy the appropriate requirements, i.e.,* $\mathsf{un}(\Gamma, \Gamma') \wedge \neg\mathsf{un}(\Delta, \Delta_1, \Delta')$ *and* A, Γ' *and* Δ' *are as one of the cases above.*

Using this translation of judgments, a translation of derivations can be defined exactly as in Definition 4.4.

We discuss some entries in Table 1 with the following example.

Example 6.4 (Cont. Example 6.2). *The translation of judgments defined in Table 1 relies on the typability in the source language* (S) *to determine the exact conditions to the typability of the translated process in* π_{DILL}. *First, the type derivation of* $w : \mathbf{end} \vdash_{\mathsf{S}} (\nu xy)(\mathsf{un}\, x(z).\mathbf{0} \mid \overline{y}\langle w\rangle.\mathbf{0})$ *is essential to build a type derivation for the translated judgment (if one exists):*

$$
\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{S}} x : *?\mathbf{end} \qquad \Gamma, z : \mathbf{end} \vdash_{\mathsf{S}} \mathbf{0}}{\Gamma \vdash_{\mathsf{S}} \mathsf{un}\, x(z).\mathbf{0}} \qquad \dfrac{\begin{array}{c}\Gamma \vdash_{\mathsf{S}} y : *!\mathbf{end} \\ \Gamma \vdash_{\mathsf{S}} w : \mathbf{end} \qquad \Gamma \vdash_{\mathsf{S}} \mathbf{0}\end{array}}{\Gamma \vdash_{\mathsf{S}} \overline{y}\langle w\rangle.\mathbf{0}}}{\Gamma \vdash_{\mathsf{S}} \mathsf{un}\, x(z).\mathbf{0} \mid \overline{y}\langle w\rangle.\mathbf{0}}}{w : \mathbf{end} \vdash_{\mathsf{S}} (\nu xy)(\mathsf{un}\, x(z).\mathbf{0} \mid \overline{y}\langle w\rangle.\mathbf{0})}
$$

where $\Gamma = x : *?\mathbf{end}, y : *!\mathbf{end}, w : \mathbf{end}$

Second, the translation $[\![w : \mathbf{end} \vdash_{\mathsf{S}} (\nu xy)(\mathsf{un}\, x(z).\mathbf{0} \mid \overline{y}\langle w\rangle.\mathbf{0})]\!]_u$ *corresponds to entry 3 of Table 1,*

$$w : \mathbf{1}; \cdot \vdash_{\ell} (\nu x)(\langle\!|\mathsf{un}\, x(z).\mathbf{0}|\!\rangle \mid \langle\!|\overline{x}\langle w\rangle.\mathbf{0}|\!\rangle) :: u : A$$

and we have previously observed that the side conditions hold. Since u *is not in the type context, we have* $A = \mathbf{1}$. *Now we proceed to build a type derivation for the translated judgment by applying rule* [L:cut]*:*

$$
\dfrac{\dfrac{\Pi_1}{w : \mathbf{1}; \cdot \vdash_{\ell} \langle\!|\mathsf{un}\, x(z).\mathbf{0}|\!\rangle :: x : !B} \qquad \dfrac{\begin{array}{c}\Pi_2 \\ w : \mathbf{1}, y : B; \cdot \vdash_{\ell} \langle\!|\overline{y}\langle w\rangle.\mathbf{0}|\!\rangle) :: u : \mathbf{1}\end{array}}{w : \mathbf{1}; x : !B \vdash_{\ell} \langle\!|\overline{x}\langle w\rangle.\mathbf{0}|\!\rangle) :: u : \mathbf{1}}}{w : \mathbf{1}; \cdot \vdash_{\ell} (\nu x)(\langle\!|\mathsf{un}\, x(z).\mathbf{0}|\!\rangle \mid \langle\!|\overline{x}\langle w\rangle.\mathbf{0}|\!\rangle) :: u : \mathbf{1}}
$$

and we will show that there exist derivations Π_1 *and* Π_2 *such that the derivation holds. We recall the translations* $\langle\!|\mathsf{un}\, x(z).0|\!\rangle$ *and* $\langle\!|\overline{x}\langle w\rangle.\mathbf{0}|\!\rangle$ *in Example 6.2.*

Third, the left premise is the translation $[\![\Gamma \vdash_{\mathsf{S}} \mathsf{un}\, x(z).\mathbf{0}]\!]_x$ *and corresponds to entry (4) of the Table 1. We recall Example 6.3 for* $(\!|\Gamma|\!)^{\dagger} = w : \mathbf{1}, x : ((!\mathbf{1} \otimes \mathbf{1}) \oplus \mathbf{1}), y : ((!\mathbf{1} \multimap \mathbf{1}) \& \mathbf{1})$ *and use the abbreviation* $B = ((!\mathbf{1} \multimap \mathbf{1}) \& \mathbf{1})$ *and strengthening of* $(\!|\Gamma|\!)^{\dagger}$. *The derivation* Π_1 *is as follows:*

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{w : \mathbf{1}; \cdot \vdash_{\ell} \mathbf{0} :: v : \mathbf{1}}{w : \mathbf{1}; z : \mathbf{1} \vdash_{\ell} \mathbf{0} :: v : !\mathbf{1}}}{w : \mathbf{1}; \cdot \vdash_{\ell} v(z).\mathbf{0} :: v : !\mathbf{1} \multimap \mathbf{1}} \qquad w : \mathbf{1}; \cdot \vdash_{\ell} \mathbf{0} :: v : \mathbf{1}}{w : \mathbf{1}; \cdot \vdash_{\ell} v.\mathbf{case}(v(z).\mathbf{0}, \mathbf{0}) :: v : (!\mathbf{1} \multimap \mathbf{1}) \& \mathbf{1}}\,[\mathsf{L{:}\&R}]}{w : \mathbf{1}; \cdot \vdash_{\ell} !x(v).v.\mathbf{case}(v(z).\mathbf{0}, \mathbf{0}) :: x : !((!\mathbf{1} \multimap \mathbf{1}) \& \mathbf{1})}\,[\mathsf{L{:}!R}]}{}
$$

Fourth, the right premise is the translation $[\![\Gamma \vdash_{\mathsf{S}} \overline{y}\langle w\rangle.0[{}^{x}\!/y]]\!]_u$ *and corresponds to the entry (11a) of the Table 1. The derivation* Π_2 *is as follows:*

$$
\dfrac{\dfrac{\dfrac{(\!|\Gamma'|\!)^{\dagger}; \cdot \vdash_{\ell} ![w \leftrightarrow v] :: v : !\mathbf{1} \qquad (\!|\Gamma'|\!)^{\dagger}; z : \mathbf{1} \vdash_{\ell} \mathbf{0} :: u : \mathbf{1}}{(\!|\Gamma'|\!)^{\dagger}; z : (!\mathbf{1} \multimap \mathbf{1}) \vdash_{\ell} \overline{z}(v).(![w \leftrightarrow v] \mid \mathbf{0}) :: u : \mathbf{1}}}{(\!|\Gamma'|\!)^{\dagger}; z : (!\mathbf{1} \multimap \mathbf{1}) \& \mathbf{1} \vdash_{\ell} z.\mathbf{inl}; \overline{z}(v).(![w \leftrightarrow v] \mid \mathbf{0}) :: u : \mathbf{1}}}{(\!|\Gamma'|\!)^{\dagger}; \cdot \vdash_{\ell} \overline{y}(z).z.\mathbf{inl}; \overline{z}(v).(![w \leftrightarrow v] \mid \mathbf{0}) :: u : \mathbf{1}}\,[\mathsf{L{:}copy}]
$$

where we strengthen $(\!|\Gamma|\!)^{\dagger}$ *to* $(\!|\Gamma'|\!)^{\dagger} = y : ((!\mathbf{1} \multimap \mathbf{1}) \& \mathbf{1}), w : \mathbf{1}$.

We have the following property, which holds by definition of the entries of Table 1:

Theorem 6.1 (Type preservation). *If* $\Gamma \circledast \Delta \vdash_{\mathsf{S}} P$ *then* $(\!|\Gamma'|\!)^{\dagger}; (\!|\Delta'|\!) \vdash_{\ell} \langle\!|P|\!\rangle :: u : A$ *is well-typed in* π_{DILL}, *with* A, Γ' *and* Δ' *as in Definition 6.6.*

Notice that the translations of typable π_{S} processes are not necessarily typable in π_{DILL}. We shall concentrate on processes in $\mathcal{S}$ that are typable in π_{DILL}:

	$\Gamma \circledast \Delta \vdash_{\mathsf{S}} P$	$\llparenthesis \Gamma \rrparenthesis^{\dagger}; \llparenthesis \Delta \rrparenthesis \vdash_{\ell} \langle\!| P |\!\rangle :: u : A$
1	$\Gamma \circledast \cdot \vdash_{\mathsf{S}} \mathbf{0}$	$\llparenthesis \Gamma \rrparenthesis^{\dagger}; \cdot \vdash_{\ell} \mathbf{0} :: u : \mathbf{1}$
2	$\Gamma \circledast \Delta_1, \Delta \vdash_{\mathsf{S}} P \mid Q$	$\llparenthesis \Gamma' \rrparenthesis^{\dagger}; \llparenthesis \Delta_1 \rrparenthesis, \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} (\nu w)(\langle\!| P |\!\rangle \mid \langle\!| Q |\!\rangle) :: u : A$ if $w \notin \mathsf{dom}(\Gamma, \Delta_1, \Delta) \wedge u \notin \mathsf{fn}(P)$
3	$\Gamma \circledast \Delta_1, \Delta \vdash_{\mathsf{S}} (\nu z v : V)(P \mid Q)$	If $(u \notin \mathsf{fn}(P)) \wedge ((\neg \mathsf{un}(V)) \vee (\mathsf{un}(V) \wedge v \notin \mathsf{fn}(P) \wedge z \notin \mathsf{fn}(Q)))$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}; \llparenthesis \Delta_1 \rrparenthesis, \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} (\nu z)(\langle\!| P |\!\rangle \mid \langle\!| Q |\!\rangle[z/v]) :: u : A$
4	$\Gamma, x : *?T \circledast \cdot \vdash_{\mathsf{S}} \mathsf{un}\ x(y).P$	If $u = x \wedge x \notin \mathsf{fn}(P)$ and one of the following holds: $\llparenthesis \Gamma \rrparenthesis^{\dagger}; \cdot \vdash_{\ell} !x(w).w.\mathbf{case}(w(y).\langle\!| P |\!\rangle, \langle\!| P[w/y] |\!\rangle) :: u : !((\llparenthesis T \rrparenthesis \multimap \mathbf{1}) \& \llparenthesis \overline{T} \rrparenthesis)$ if $\neg\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T)$ $\llparenthesis \Gamma \rrparenthesis^{\dagger}; \cdot \vdash_{\ell} !x(w).\langle\!| P[w/y] |\!\rangle :: u : !\llparenthesis \overline{T} \rrparenthesis$ if $\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T)$ $\llparenthesis \Gamma \rrparenthesis^{\dagger}; \cdot \vdash_{\ell} !x(w).w(y).\langle\!| P |\!\rangle :: u : !(\llparenthesis T \rrparenthesis \multimap \mathbf{1})$ if $\neg\mathsf{svr}(T) \wedge \mathsf{cli}(T)$
5	$\Gamma \circledast x : \mathsf{lin}?T.S, \Delta \vdash_{\mathsf{S}} \mathsf{lin}\ x(y).P$	$\llparenthesis \Gamma \rrparenthesis^{\dagger}; \llparenthesis \Delta \rrparenthesis \vdash_{\ell} x(y).\langle\!| P |\!\rangle :: u : \llparenthesis T \rrparenthesis \multimap \llparenthesis \overline{S} \rrparenthesis$ if $u = x$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}; \llparenthesis \Delta' \rrparenthesis, x : \llparenthesis T \rrparenthesis \otimes \llparenthesis S \rrparenthesis \vdash_{\ell} x(y).\langle\!| P |\!\rangle :: u : A$ otherwise
6	$\Gamma, z : *!T \circledast \Delta \vdash_{\mathsf{S}} (\nu x y)\overline{z}\langle y \rangle.P$	If $(\neg \mathsf{un}(T)) \vee (y \notin \mathsf{fn}(P) \wedge \mathsf{un}(T))$ and: $\llparenthesis \Gamma' \rrparenthesis^{\dagger}, z : \llparenthesis T \rrparenthesis \multimap \llparenthesis \mathbf{1} \rrparenthesis \& \llparenthesis \overline{T} \rrparenthesis; \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \overline{z}(x).x.\mathbf{inr}; \langle\!| P |\!\rangle :: u : A$ if $\neg\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}, z : \llparenthesis \overline{T} \rrparenthesis; \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \overline{z}(x).\langle\!| P |\!\rangle :: u : A$ if $\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T)$
7	$\Gamma \circledast z : \mathsf{lin}!T.S, \Delta_1, \Delta \vdash_{\mathsf{S}}$ $(\nu x y)\overline{z}\langle x \rangle.(P \mid Q)$	If $(\neg \mathsf{un}(T)) \vee (x \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q) \wedge \mathsf{un}(T))$ and: $\llparenthesis \Gamma \rrparenthesis^{\dagger}; \llparenthesis \Delta_1 \rrparenthesis, \llparenthesis \Delta \rrparenthesis \vdash_{\ell} \overline{z}(y).(\langle\!| P |\!\rangle \mid \langle\!| Q |\!\rangle) :: z : \llparenthesis T \rrparenthesis \otimes \llparenthesis \overline{S} \rrparenthesis$ if $z = u \wedge u, z \notin \mathsf{fn}(P) \wedge y \notin \mathsf{fn}(Q)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}; \llparenthesis \Delta_1 \rrparenthesis, \llparenthesis \Delta' \rrparenthesis, z : \llparenthesis T \rrparenthesis \multimap \llparenthesis S \rrparenthesis \vdash_{\ell} \overline{z}(y).(\langle\!| P |\!\rangle \mid \langle\!| Q |\!\rangle) :: u : A$ if $z \neq u \wedge u, z \notin \mathsf{fn}(P) \wedge y \notin \mathsf{fn}(Q)$
8	$\Gamma \circledast v : T, x : \mathsf{lin}!(T).S, \Delta \vdash_{\mathsf{S}} \overline{x}\langle v \rangle.P$	$\llparenthesis \Gamma \rrparenthesis^{\dagger}; v : \llparenthesis T \rrparenthesis, \llparenthesis \Delta \rrparenthesis \vdash_{\ell} \overline{x}(y).([v \leftrightarrow y] \mid \langle\!| P |\!\rangle) :: u : \llparenthesis T \rrparenthesis \otimes \llparenthesis \overline{S} \rrparenthesis$ if $u = x \wedge \neg\mathsf{un}(T) \wedge \neg\mathsf{svr}(T)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}; v : \llparenthesis T \rrparenthesis, \llparenthesis \Delta' \rrparenthesis, x : \llparenthesis T \rrparenthesis \multimap \llparenthesis S \rrparenthesis \vdash_{\ell} \overline{x}(y).([v \leftrightarrow y] \mid P) :: u : \llparenthesis \overline{R} \rrparenthesis$ if $\neg\mathsf{un}(T) \wedge \neg\mathsf{svr}(T)$
9	$\Gamma, v : T \circledast x : \mathsf{lin}!(T).S, \Delta \vdash_{\mathsf{S}} \overline{x}\langle v \rangle.P$	$\llparenthesis \Gamma \rrparenthesis^{\dagger}, v : \llparenthesis T \rrparenthesis; \llparenthesis \Delta \rrparenthesis \vdash_{\ell} \overline{x}(y).(![v \leftrightarrow y] \mid \langle\!| P |\!\rangle) :: x : \llparenthesis T \rrparenthesis \otimes \llparenthesis \overline{S} \rrparenthesis$ if $u = x \wedge \mathsf{un}(T) \wedge \neg\mathsf{svr}(T)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}, v : T; \llparenthesis \Delta' \rrparenthesis, x : \llparenthesis T \rrparenthesis \multimap \llparenthesis S \rrparenthesis \vdash_{\ell} \overline{x}(y).(![v \leftrightarrow y] \mid P) :: u : A$ if $\mathsf{un}(T) \wedge \neg\mathsf{svr}(T)$
10	$\Gamma, x : *!T \circledast v : T, \Delta \vdash_{\mathsf{S}} \overline{x}\langle v \rangle.P$	If $\neg\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T) \wedge \neg\mathsf{un}(T) \wedge z \notin \mathsf{fn}(P)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}, x : \llparenthesis T \rrparenthesis \multimap \mathbf{1} \& \llparenthesis \overline{T} \rrparenthesis; v : \llparenthesis T \rrparenthesis, \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \overline{x}(z).z.\mathbf{inl}; \overline{z}(w).([v \leftrightarrow w] \mid \langle\!| P |\!\rangle) :: u : A$ If $\neg\mathsf{svr}(T) \wedge \mathsf{cli}(T) \wedge \neg\mathsf{un}(T) \wedge z \notin \mathsf{fn}(P)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}, x : \llparenthesis T \rrparenthesis \multimap \mathbf{1}; v : \llparenthesis T \rrparenthesis, \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \overline{x}(z).\overline{z}(w).([v \leftrightarrow w] \mid \langle\!| P |\!\rangle) :: u : A$
11	$\Gamma, x : *!T, v : T \circledast \Delta \vdash_{\mathsf{S}} \overline{x}\langle v \rangle.P$	If $\neg\mathsf{svr}(T) \wedge \neg\mathsf{cli}(T) \wedge \mathsf{un}(T) \wedge z \notin \mathsf{fn}(P)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}, x : \llparenthesis T \rrparenthesis \multimap \mathbf{1} \& \llparenthesis \overline{T} \rrparenthesis; v : \llparenthesis T \rrparenthesis; \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \overline{x}(z).z.\mathbf{inl}; \overline{z}(w).(![v \leftrightarrow w] \mid \langle\!| P |\!\rangle) :: u : A$ If $\neg\mathsf{svr}(T) \wedge \mathsf{cli}(T) \wedge \mathsf{un}(T) \wedge z \notin \mathsf{fn}(P)$ $\llparenthesis \Gamma' \rrparenthesis^{\dagger}, x : \llparenthesis T \rrparenthesis \multimap \mathbf{1}, v : \llparenthesis T \rrparenthesis; \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \overline{x}(z).\overline{z}(w).(![v \leftrightarrow w] \mid \langle\!| P |\!\rangle) :: u : A$

Table 1: From judgments in π_{S} to judgments in π_{DILL} (Definition 6.6).

Notation 6.1. *We write $\llparenthesis \Gamma' \rrparenthesis^{\dagger}; \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \langle\!| P |\!\rangle :: u : \llparenthesis \overline{S} \rrparenthesis$ whenever $[\![\Gamma, \Delta \vdash_{\mathsf{S}} P]\!]_u$ holds, with $\Gamma \asymp_S^u \Gamma'$ and $\Delta \asymp_S^u \Delta'$.*

We can finally define $\mathcal{L}$:

Definition 6.7 ($\mathcal{L}$). *Let u be a name. We define:*

$$\mathcal{L} = \{P \in \pi_{\mathsf{S}} \mid \Gamma \circledast \Delta \vdash_{\mathsf{S}} P \wedge \Gamma \asymp_S^u \Gamma' \wedge \Delta \asymp_S^u \Delta' \wedge \llparenthesis \Gamma' \rrparenthesis^{\dagger}; \llparenthesis \Delta' \rrparenthesis \vdash_{\ell} \langle\!| P |\!\rangle :: u : \llparenthesis \overline{S} \rrparenthesis\}$$

where contexts and types mentioned are existentially quantified.

6.2 Results

Theorem 6.2 ($\mathcal{L} \subset \mathcal{W}$). *Let $P \in \mathcal{S}$ such that $\Gamma \vdash_{\mathsf{S}} P$, for some context Γ. If there exists u such that $[\![\Gamma \vdash_{\mathsf{S}} P]\!]_u$ holds, then there exists a level function l such that $[\![\Gamma \vdash_{\mathsf{S}} P]\!]_l$ holds.*

The proof of Theorem 6.2 is by induction on the structure of the typed process P. To show that the typing discipline for π_{DILL} induces an appropriate level function l in the setting of π_{W}, we first construct a strict partial order $>_1$, based on the structure of P. Then, we define a flattening procedure on $>_1$, which works on so-called connected channels, i.e., names that occur in a restriction or as the subject of an input or an output prefix. This flattening procedure returns a strict partial order, denoted $>_2$, that we use to represent a level function. Intuitively, the level function induced by $>_2$ measures the number of channels that a given channel can relate to. Finally, we show that this level function can be used to correctly encode P into $\mathcal{W}$.

Theorem 6.3 ($\mathcal{W} \not\subset \mathcal{L}$). *$\exists P \in \mathcal{W}$ with $\Gamma \vdash_{\mathsf{S}} P$ and $[\![\Gamma \vdash_{\mathsf{S}} P]\!]_l$ for some l such that $\nexists z$ s.t. $[\![\Gamma \vdash_{\mathsf{S}} P]\!]_z$.*

To prove Theorem 6.3, it suffices to consider the π_{S} process

$$P = (\nu x y)(\mathsf{lin}\ x(z).\mathsf{un}\ z(w).\mathbf{0} \mid (\nu s t)\overline{y}\langle s \rangle.((\nu u v)(\overline{t}\langle u \rangle.\mathbf{0}) \mid \mathbf{0}))$$

Clearly, P is terminating:

$$P \longrightarrow (\nu s t)(\mathsf{un}\ s(w).\mathbf{0} \mid (\nu u v)(\overline{t}\langle u \rangle.\mathbf{0})) \longrightarrow (\nu s t)\mathsf{un}\ s(w).\mathbf{0}$$

Process P can be typed so as to establish $P \in \mathcal{S}$. Also, there is a level function that makes its translation into π_{W} typable. Hence, $P \in \mathcal{W}$. However, its translation into π_{DILL} is not typable, so $P \notin \mathcal{L}$.

7 CLOSING REMARKS

We presented a comparative study of type systems for concurrent processes in the π-calculus, from the unifying perspective of termination and session types. To our knowledge, this is the first study of its kind.

Even by focusing on only three different type systems, we were confronted with technical challenges connected with the intrinsic differences between them. The typed process model π_{S} [16], focused on session-based concurrency, admits a rather broad class of processes, exploiting a clear distinction between linear and unrestricted resources, implemented via context splitting. The typed process model π_{W} combines features from type systems that target the termination property [7] and type systems for sessions. Finally, the typed process model π_{DILL} [1] rests upon a firm logical foundation, and its control of clients and servers is directly inherited from the logical principles of the exponential $!A$. Notice that π_{DILL} is unique among type systems for the π-calculus in that it ensures protocol fidelity, deadlock-freedom, confluence, and strong normalization/termination for typed processes.

The main take-away message is that the Curry-Howard correspondence is strictly weaker than weight-based approaches for enforcing the termination property. Hence, the control of server/client interactions that is elegantly enabled by the copying semantics of $!A$ turns out to be rather implicit when contrasted to weight-based techniques. Interestingly, Dardha and Pérez arrived to a similar conclusion in their comparative study of type systems focused on the deadlock-freedom property [3, 4]: type systems based on the Curry-Howard correspondence can detect strictly less deadlock-free processes than other, more sophisticated type systems. Notice that the study in [3, 4] considers only finite processes, without input-guarded replication (so all process are terminating).

Immediate items for current and future work include incorporating other type systems into our formal comparisons. The type systems by Sangiorgi [14] and by Yoshida et al. [17] are very appealing candidates. Also, Deng and Sangiorgi proposed several type systems for termination. Here we considered only the simplest variant, which induces the class $\mathcal{W}$ and is already different from $\mathcal{L}$; it would be interesting to consider the other variants.

ACKNOWLEDGMENTS

We acknowledge the support of the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software). Daniele Nantes-Sobrinho has been supported by the EPSRC Fellowship 'VeTSpec: Verified Trustworthy Software Specification' (EP/R034567/1).

We are grateful to Davide Sangiorgi, Nobuko Yoshida, and the anonymous reviewers for useful suggestions and remarks.

REFERENCES

[1] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 (LNCS, Vol. 6269)*. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

[2] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *PPDP'12*. ACM, 139–150. https://doi.org/10.1145/2370776.2370794

[3] Ornela Dardha and Jorge A. Pérez. 2015. Comparing Deadlock-Free Session Typed Processes. In *Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS (EPTCS, Vol. 190)*. 1–15. https://doi.org/10.4204/EPTCS.190.1

[4] Ornela Dardha and Jorge A. Pérez. 2022. Comparing type systems for deadlock freedom. *J. Log. Algebraic Methods Program.* 124 (2022), 100717. https://doi.org/10.1016/j.jlamp.2021.100717

[5] Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. 2010. Termination in Impure Concurrent Languages. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6269)*, Paul Gastin and François Laroussinie (Eds.). Springer, 328–342. https://doi.org/10.1007/978-3-642-15375-4_23

[6] Yuxin Deng and Davide Sangiorgi. 2004. Ensuring Termination by Typability. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France (IFIP, Vol. 155)*, Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell (Eds.). Kluwer/Springer, 619–632. https://doi.org/10.1007/1-4020-8141-3_47

[7] Yuxin Deng and Davide Sangiorgi. 2006. Ensuring termination by typability. *Information and Computation* 204, 7 (2006), 1045–1082.

[8] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP'98 (LNCS, Vol. 1381)*. Springer, 122–138. https://doi.org/10.1007/BFb0053567

[9] Naoki Kobayashi and Davide Sangiorgi. 2010. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.* 32, 5 (2010), 16:1–16:49. https://doi.org/10.1145/1745312.1745313

[10] Ugo Dal Lago, Marc de Visme, Damiano Mazza, and Akira Yoshimizu. 2019. Intersection types and runtime errors in the pi-calculus. *Proc. ACM Program. Lang.* 3, POPL (2019), 7:1–7:29. https://doi.org/10.1145/3290320

[11] Joseph W. N. Paulus, Jorge A. Pérez, and Daniele Nantes-Sobrinho. 2023. Termination in Concurrency, Revisited. (2023). arXiv:2308.01165 https://doi.org/10.48550/arXiv.2308.01165

[12] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logical relations and observational equivalences for session-based concurrency. *Inf. Comput.* 239 (2014), 254–302. https://doi.org/10.1016/j.ic.2014.08.001

[13] Mauro Piccolo. 2012. Strong Normalization in the π-calculus with Intersection and Union Types. *Fundam. Informaticae* 121, 1-4 (2012), 227–252. https://doi.org/10.3233/FI-2012-777

[14] Davide Sangiorgi. 2006. Termination of processes. *Math. Struct. Comput. Sci.* 16, 1 (2006), 1–39. https://doi.org/10.1017/S0960129505004810

[15] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2014. Corecursion and Non-divergence in Session-Typed Processes. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8902)*, Matteo Maffei and Emilio Tuosto (Eds.). Springer, 159–175. https://doi.org/10.1007/978-3-662-45917-1_11

[16] Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Inf. Comput.* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002

[17] Nobuko Yoshida, Martin Berger, and Kohei Honda. 2001. Strong Normalisation in the pi-Calculus. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 311–322. https://doi.org/10.1109/LICS.2001.932507