Comparing Deadlock-Free Session Typed Processes

Ornela Dardha
University of Glasgow, United Kingdom

Jorge A. Pérez
University of Groningen, The Netherlands

Besides respecting prescribed protocols, communication-centric systems should never "get stuck". This requirement has been expressed by liveness properties such as progress or (dead)lock freedom. Several typing disciplines that ensure these properties for mobile processes have been proposed. Unfortunately, very little is known about the precise relationship between these disciplines–and the classes of typed processes they induce.

In this paper, we compare $\mathscr{L}$ and $\mathscr{K}$, two classes of deadlock-free, session typed concurrent processes. The class $\mathscr{L}$ stands out for its canonicity: it results naturally from interpretations of linear logic propositions as session types. The class $\mathscr{K}$, obtained by encoding session types into Kobayashi's usage types, includes processes not typable in other type systems.

We show that $\mathscr{L}$ is strictly included in $\mathscr{K}$. We also identify the precise condition under which $\mathscr{L}$ and $\mathscr{K}$ coincide. One key observation is that the *degree of sharing* between parallel processes determines a new expressiveness hierarchy for typed processes. We also provide a type-preserving rewriting procedure of processes in $\mathscr{K}$ into processes in $\mathscr{L}$. This procedure suggests that, while effective, the degree of sharing is a rather subtle criteria for distinguishing typed processes.

1 Introduction

The goal of this work is to formally relate different type systems for the π-calculus. Our interest is in *session-based concurrency*, a type-based approach to communication correctness: dialogues between participants are structured into *sessions*, basic communication units; descriptions of interaction sequences are then abstracted as *session types* [12] which are checked against process specifications. We offer the first formal comparison between different type systems that enforce *(dead)lock freedom*, the liveness property that ensures session communications never "get stuck". Our approach relates the classes of typed processes that such systems induce. To this end, we identify a property on the structure of typed parallel processes, the *degree of sharing*, which is key in distinguishing two salient classes of deadlock-free session processes, and in shedding light on their formal underpinnings.

In session-based concurrency, types enforce correct communications through different safety and liveness properties. Basic correctness properties are *communication safety* and *session fidelity*: while the former ensures absence of errors (e.g., communication mismatches), the latter ensures that well-typed processes respect the protocols prescribed by session types. Moreover, a central (liveness) property for safe processes is that they should never "get stuck". This is the well-known *progress* property, which asserts that a well-typed term either is a final value or can further reduce [17]. In calculi for concurrency, this property has been formalized as *deadlock freedom* ("a process is deadlock-free if it can always reduce until it eventually terminates, unless the whole process diverges" [15]) or as *lock freedom* ("a process is lock free if it can always reduce until it eventually terminates, even if the whole process diverges" [13]). Notice that in the absence of divergent behaviors, deadlock and lock freedom coincide.

(Dead)lock freedom guarantees that all communications will eventually succeed, an appealing requirement for communicating processes. Several advanced type disciplines that ensure deadlock-free processes have been proposed (see, e.g., [2, 3, 5, 10, 13, 15, 16, 20]). Unfortunately, these disciplines consider different process languages and/or are based on rather different principles. As a result, very little

To appear in EPTCS.

is known about how they relate to each other. This begs several research questions: What is the formal relationship between these type disciplines? What classes of deadlock-free processes do they induce?

In this paper, we tackle these open questions by comparing $\mathscr{L}$ and $\mathscr{K}$, two salient classes of deadlock-free, session typed processes (Definition 4.2):

- $\mathscr{L}$ contains all session processes that are well-typed according to the Curry-Howard correspondence of linear logic propositions as session types [2, 3, 21]. This suffices, because the type system derived from such a correspondence ensures communication safety, session fidelity, and deadlock freedom.
- $\mathscr{K}$ contains all session processes that enjoy communication safety and session fidelity (as ensured by the type system of Vasconcelos [19]) and are (dead)lock-free by combining Kobayashi's type system based on *usages* [13, 15] with Dardha et al.'s encodability result [8].

There are good reasons for considering $\mathscr{L}$ and $\mathscr{K}$. On the one hand, due to its deep logical foundations, $\mathscr{L}$ appears to us as the *canonic* class of deadlock-free session processes, upon which all other classes should be compared. Indeed, this class arguably offers the most principled yardstick for comparisons. On the other hand, $\mathscr{K}$ integrates session type checking with the sophisticated usage discipline developed by Kobayashi for π-calculus processes. This indirect approach to deadlock freedom (first suggested in [14], later developed in [4, 7, 8]) is fairly general, as it may capture sessions with subtyping, polymorphism, and higher-order communication. Also, as informally shown in [4], $\mathscr{K}$ strictly includes classes of typed processes induced by other type systems for deadlock freedom in sessions [5, 10, 16].

One key observation in our development is that $\mathscr{K}$ corresponds to a *family* of classes of deadlock-free processes, denoted $\mathscr{K}_0, \mathscr{K}_1, \cdots, \mathscr{K}_n$, which is defined by the *degree of sharing* between their parallel components. Intuitively, $\mathscr{K}_0$ is the subclass of $\mathscr{K}$ with *independent parallel composition*: for all processes $P \mid Q \in \mathscr{K}_0$, subprocesses P and Q do not share any sessions. Then, $\mathscr{K}_1$ is the subclass of $\mathscr{K}$ which contains $\mathscr{K}_0$ but admits also processes with parallel components that share at most one session. Then, $\mathscr{K}_n$ contains deadlock-free session processes whose parallel components share at most n sessions.

Contributions. In this paper, we present three main contributions:

1. We show that the inclusion between the constituent classes of $\mathscr{K}$ is *strict* (Theorem 4.4). We have:

$$\mathscr{K}_0 \subset \mathscr{K}_1 \subset \mathscr{K}_2 \subset \cdots \subset \mathscr{K}_n \subset \mathscr{K}_{n+1} \tag{1}$$

   Although not extremely surprising, the significance of this result lies in the fact that it talks about concurrency (via the degree of sharing) but implicitly also about the potential sequentiality of parallel processes. As such, processes in $\mathscr{K}_k$ are necessarily "more parallel" than those in $\mathscr{K}_{k+1}$. Interestingly, the degree of sharing in $\mathscr{K}_0, \ldots, \mathscr{K}_n$ can be defined in a very simple way, via a natural condition in the rule for parallel composition in Kobayashi's type system for deadlock freedom.

2. We show that $\mathscr{L}$ and $\mathscr{K}_1$ coincide (Theorem 4.6). That is, there are deadlock-free session processes that cannot be typed by systems derived from the Curry-Howard interpretation of session types [2, 3, 21], but that can be admitted by the (indirect) approach of [8]. This result is significant: it establishes the precise status of systems based on [3,21] with respect to previous (non Curry-Howard) disciplines. Indeed, it formally confirms that linear logic interpretations of session types naturally induce the most basic form of concurrent cooperation (sharing of exactly one session), embodied as the principle of "composition plus hiding", a distinguishing feature of such interpretations.

3. We define a rewriting procedure of processes in $\mathscr{K}$ into $\mathscr{L}$ (Defintion 5.7). Intuitively, due to our previous observation and characterization of the degree of sharing in session typed processes, it is

quite natural to convert a process in $\mathscr{K}$ into another, more parallel process in $\mathscr{L}$. In essence, the procedure replaces sequential prefixes with representative parallel components. The rewriting procedure satisfies type-preservation, and enjoys the compositionality and operational correspondence criteria as stated in [11] (cf. Theorems 5.8 and 5.10). These properties not only witness the significance of the rewriting procedure; they also confirm that the degree of sharing is a rather subtle criteria for formally distinguishing deadlock-free, session typed processes.

To the best of our knowledge, these contributions define the first formal comparison between fundamentally distinct type systems for deadlock freedom in session communications. Previous comparisons, such as the ones in [4] and [3, §6], are informal: they are based on representative "corner cases", i.e., examples of deadlock-free session processes typable in one system but not in some other.

The paper is structured as follows. § 2 summarizes the session π-calculus and associated type system of [19]. In § 3 we present the two typed approaches to deadlock freedom for sessions. § 4 defines the classes $\mathscr{L}$ and $\mathscr{K}$, formalizes the hierarchy (1), and shows that $\mathscr{L}$ and $\mathscr{K}_1$ coincide. In § 5 we give the rewriting procedure of $\mathscr{K}_n$ into $\mathscr{L}$ and establish its properties. § 6 collects some concluding remarks. Due to space restrictions, details of definitions and proofs are omitted; they can be found online [9].

2 Session π-calculus

Following Vasconcelos [19], we introduce the session π-calculus and its associated type system which ensures communication safety and session fidelity. The syntax is given in Figure 1 (upper part). Let P, Q range over processes x, y over channels and v over values; for simplicity, the set of values coincides with that of channels. In examples, we often use $\mathbf{n}$ to denote a terminated channel that cannot be further used.

Process $\overline{x}\langle v\rangle.P$ denotes the output of v along x, with continuation P. Dually, process $x(y).P$ denotes an input along x with continuation P, with y denoting a placeholder. Process $x \triangleleft l_j.P$ uses x to select l_j from a labelled choice process, being $x \triangleright \{l_i : P_i\}_{i\in I}$, so as to trigger P_j; labels indexed by the finite set I are pairwise distinct. We also have the inactive process (denoted $\mathbf{0}$), the parallel composition of P and Q (denoted $P \mid Q$), and the (double) restriction operator, noted $(\nu xy)P$: the intention is that x and y denote *dual session endpoints* in P. We omit $\mathbf{0}$ whenever possible and write, e.g., $\overline{x}\langle\mathbf{n}\rangle$ instead of $\overline{x}\langle\mathbf{n}\rangle.\mathbf{0}$. Notions of bound/free variables in processes are standard; we write $\mathsf{fn}(P)$ to denote the set of free names of P. Also, we write $P[v/z]$ to denote the (capture-avoiding) substitution of free occurrences of z in P with v.

The operational semantics is given in terms of a reduction relation, noted $P \to Q$, and defined by the rules in Figure 1 (lower part). It relies on a standard notion of structural congruence, noted $\equiv$ (see [19]). We write $\to^*$ to denote the reflexive, transitive closure of $\to$. Observe that interaction involves prefixes with different channels (endpoints), and always occurs in the context of an outermost (double) restriction. Key rules are (R-Com) and (R-Case), denoting the interaction of output/input prefixes and selection/branching constructs, respectively. Rules (R-Par), (R-Res), and (R-Str) are standard.

The syntax of session types, ranged over $T, S, \ldots$, is given by the following grammar.

$$T, S ::= \mathbf{end} \;\mid\; ?T.S \;\mid\; !T.S \;\mid\; \&\{l_i : S_i\}_{i\in I} \;\mid\; \oplus\{l_i : S_i\}_{i\in I}$$

Above, **end** is the type of an endpoint with a terminated protocol. The type $?T.S$ is assigned to an endpoint that first receives a value of type T and then continues according to the protocol described by S. Dually, type $!T.S$ is assigned to an endpoint that first outputs a value of type T and then continues according to the protocol described by S. Type $\oplus\{l_i : S_i\}_{i\in I}$, an *internal choice*, generalizes output types; type $\&\{l_i : S_i\}_{i\in I}$, an *external choice*, generalizes input types. Notice that session types describe *sequences* of structured behaviors; they do not admit parallel composition operators.

$$
\begin{array}{rlll}
P,Q ::= & \overline{x}\langle v\rangle.P & \text{(output)} & \mathbf{0} \quad \text{(inaction)} \\
& x(y).P & \text{(input)} & P \mid Q \quad \text{(composition)} \\
& x \triangleleft l_j.P & \text{(selection)} & (\nu xy)P \quad \text{(session restriction)} \\
& x \triangleright \{l_i : P_i\}_{i\in I} & \text{(branching)} & \\
v ::= & x & \text{(channel)} &
\end{array}
$$

$$
\begin{array}{ll}
\text{(R-Com)} & (\nu xy)(\overline{x}\langle v\rangle.P \mid y(z).Q) \to (\nu xy)(P \mid Q[v/z]) \\
\text{(R-Case)} & (\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}) \to (\nu xy)(P \mid P_j) \quad j \in I \\
\text{(R-Str)} & P \equiv P',\ P \to Q,\ Q' \equiv Q \Longrightarrow P' \to Q'
\end{array}
\qquad
\begin{array}{ll}
\text{(R-Par)} & P \to Q \Longrightarrow P \mid R \to Q \mid R \\
\text{(R-Res)} & P \to Q \Longrightarrow (\nu xy)P \to (\nu xy)Q
\end{array}
$$

Figure 1: Session π-calculus: syntax and semantics.

$$
\text{(T-Nil)}\ \frac{}{x : \mathbf{end} \vdash_{\mathrm{ST}} \mathbf{0}}
\qquad
\text{(T-Par)}\ \frac{\Gamma_1 \vdash_{\mathrm{ST}} P \quad \Gamma_2 \vdash_{\mathrm{ST}} Q}{\Gamma_1 \circ \Gamma_2 \vdash_{\mathrm{ST}} P \mid Q}
\qquad
\text{(T-Res)}\ \frac{\Gamma, x : T, y : \overline{T} \vdash_{\mathrm{ST}} P}{\Gamma \vdash_{\mathrm{ST}} (\nu xy)P}
\qquad
\text{(T-In)}\ \frac{\Gamma, x : S, y : T \vdash_{\mathrm{ST}} P}{\Gamma, x : ?T.S \vdash_{\mathrm{ST}} x(y).P}
$$

$$
\text{(T-Out)}\ \frac{\Gamma, x : S \vdash_{\mathrm{ST}} P}{\Gamma, x : !T.S, y : T \vdash_{\mathrm{ST}} \overline{x}\langle y\rangle.P}
\qquad
\text{(T-Brch)}\ \frac{\Gamma, x : S_i \vdash_{\mathrm{ST}} P_i \quad \forall i \in I}{\Gamma, x : \&\{l_i : S_i\}_{i\in I} \vdash_{\mathrm{ST}} x \triangleright \{l_i : P_i\}_{i\in I}}
\qquad
\text{(T-Sel)}\ \frac{\Gamma, x : S_j \vdash_{\mathrm{ST}} P \quad \exists j \in I}{\Gamma, x : \oplus\{l_i : S_i\}_{i\in I} \vdash_{\mathrm{ST}} x \triangleleft l_j.P}
$$

Figure 2: Typing rules for the π-calculus with sessions.

A central notion in session-based concurrency is *duality*, which relates session types offering opposite (i.e., complementary) behaviors. Duality stands at the basis of communication safety and session fidelity. Given a session type T, its dual type $\overline{T}$ is defined as follows:

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\overline{!T.S} & \triangleq & ?T.\overline{S} & \overline{?T.S} & \triangleq & !T.\overline{S} & \overline{\mathbf{end}} & \triangleq & \mathbf{end} \\
\overline{\oplus\{l_i : S_i\}_{i\in I}} & \triangleq & \&\{l_i : \overline{S}_i\}_{i\in I} & \overline{\&\{l_i : S_i\}_{i\in I}} & \triangleq & \oplus\{l_i : \overline{S}_i\}_{i\in I} & & &
\end{array}
$$

Typing contexts, ranged over by Γ, Γ', are sets of typing assignments $x : T$. Given a context Γ and a process P, a session typing judgement is of the form $\Gamma \vdash_{\mathrm{ST}} P$. Typing rules are given in Figure 2. Rule (T-Nil) states that $\mathbf{0}$ is well-typed under a terminated channel. Rule (T-Par) types the parallel composition of two processes by composing their corresponding typing contexts using a splitting operator, noted $\circ$ [19]. Rule (T-Res) types a restricted process by requiring that the two endpoints have dual types. Rules (T-In) and (T-Out) type the receiving and sending of a value over a channel x, respectively. Finally, rules (T-Brch) and (T-Sel) are generalizations of input and output over a labelled set of processes.

The main guarantees of the type system are *communication safety* and *session fidelity*, i.e., typed processes respect their ascribed protocols, as represented by session types.

Theorem 2.1 (Type Preservation for Session Types)**.** *If $\Gamma \vdash_{\mathrm{ST}} P$ and $P \to Q$, then $\Gamma \vdash_{\mathrm{ST}} Q$.*

The following notion of well-formed processes is key to single out meaningful typed processes.

Definition 2.2 (Well-Formedness for Sessions)**.** *A process is* well-formed *if for any of its structural congruent processes of the form $(\nu\widetilde{xy})(P \mid Q)$ the following hold.*

- *If P and Q are prefixed at the same variable, then the variable performs the same action (input or output, branching or selection).*

- *If P is prefixed in x_i and Q is prefixed in y_i where $x_i y_i \in \widetilde{xy}$, then $P \mid Q \rightarrow$.*

It is important to notice that well-typedness of a process does not imply the process is well-formed. We have the following theorem:

Theorem 2.3 (Type Safety for Sessions [19])**.** *If $\vdash_{\mathtt{ST}} P$ then P is well-formed.*

We present the main result of the session type system. The following theorem states that a well-typed closed process does not reduce to an ill-formed one. It follows immediately from Theorems 2.1 and 2.3.

Theorem 2.4 ([19])**.** *If $\vdash_{\mathtt{ST}} P$ and $P \rightarrow^* Q$, then Q is well-formed.*

An important observation is that the session type system given above does not exclude *deadlocked processes*, i.e., processes which reach a "stuck state." This is because the interleaving of communication prefixes in typed processes may create extra causal dependencies not described by session types. (This intuitive definition of deadlocked processes will be made precise below.) A particularly insidious class of deadlocks is due to cyclic interleaving of channels in processes. For example, consider a process such as $P \triangleq (\nu xy)(\nu wz)(\overline{x}\langle \mathbf{n} \rangle.\overline{w}\langle \mathbf{n} \rangle \mid z(t).y(s))$: it represents the implementation of two (simple) independent sessions, which get intertwined (blocked) due to the nesting induced by input and output prefixes. We have that $\mathbf{n} : \mathbf{end} \vdash_{\mathtt{ST}} P$ even if P is unable to reduce. A deadlock-free variant of P would be, e.g., process $P' \triangleq (\nu xy)(\nu wz)(\overline{x}\langle \mathbf{n} \rangle.\overline{w}\langle \mathbf{n} \rangle \mid y(s).z(t))$, which also is typable in $\vdash_{\mathtt{ST}}$.

We will say that a process is *deadlock-free* if any communication action that becomes active during execution is eventually consumed; that is, there is a corresponding co-action that eventually becomes available. Below we define deadlock freedom in the session π-calculus; we follow [13, 15] and consider *fair* reduction sequences [6]. For simplicity, we omit the symmetric cases for input and branching.

Definition 2.5 (Deadlock Freedom for Session π-Calculus)**.** *A process P_0 is* deadlock-free *if for any fair reduction sequence $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots$, we have that*

1. *$P_i \equiv (\nu \widetilde{xy})(\overline{x}\langle v \rangle.Q \mid R)$, for $i \geq 0$, implies that there exists $n \geq i$ such that $P_n \equiv (\nu \widetilde{x'y'})(\overline{x}\langle v \rangle.Q \mid y(z).R_1 \mid R_2)$ and $P_{n+1} \equiv (\nu \widetilde{x'y'})(Q \mid R_1[v/z] \mid R_2)$;*
2. *$P_i \equiv (\nu \widetilde{xy})(x \triangleleft l_j.Q \mid R)$, for $i \geq 0$, implies that there exists $n \geq i$ such that $P_n \equiv (\nu \widetilde{x'y'})(x \triangleleft l_j.Q \mid y \triangleright \{l_k : R_k\}_{k \in I \cup \{j\}} \mid S)$ and $P_{n+1} \equiv (\nu \widetilde{x'y'})(Q \mid R_j \mid S)$.*

3 Two Approaches to Deadlock Freedom

We introduce two approaches to deadlock-free, session typed processes. The first one, given in § 3.1, comes from interpretations of linear logic propositions as session types [1–3, 21]; the second approach, summarized in § 3.2, combines usage types for the standard π-calculus with encodings of session processes and types [8]. Based on these two approaches, in § 4 we will define the classes $\mathscr{L}$ and $\mathscr{K}$.

3.1 Linear Logic Foundations of Session Types

The linear logic interpretation of session types was introduced by Caires and Pfenning [3], and developed by Wadler [21] and others. Initially proposed for intutitionistic linear logic, here we consider an interpretation based on classical linear logic with mix principles, following a recent presentation by Caires [1].

The syntax and semantics of processes are as in § 2 except for the following differences. First, we have the standard restriction construct $(\nu x)P$, which replaces the double restriction. Second, we have a so-called *forwarding process*, denoted $[x \leftrightarrow y]$, which intuitively "fuses" names x and y. Besides these

$$\text{(T-}\mathbf{1}\text{)}\ \frac{}{\mathbf{0} \vdash_{\mathtt{CH}} x{:}\bullet} \qquad \text{(T-}\bot\text{)}\ \frac{P \vdash_{\mathtt{CH}} \Delta}{P \vdash_{\mathtt{CH}} x{:}\bullet, \Delta} \qquad \text{(T-id)}\ \frac{}{[x \leftrightarrow y] \vdash_{\mathtt{CH}} x{:}A, y{:}\overline{A}}$$

$$\text{(T-}\parr\text{)}\ \frac{P \vdash_{\mathtt{CH}} \Delta, y{:}A, x{:}B}{x(y).P \vdash_{\mathtt{CH}} \Delta, x{:}A \parr B} \qquad \text{(T-}\otimes\text{)}\ \frac{P \vdash_{\mathtt{CH}} \Delta, y{:}A \qquad Q \vdash_{\mathtt{CH}} \Delta', x{:}B}{\overline{x}(y).(P \mid Q) \vdash_{\mathtt{CH}} \Delta, \Delta', x{:}A \otimes B} \qquad \text{(T-cut)}\ \frac{P \vdash_{\mathtt{CH}} \Delta, x{:}\overline{A} \qquad Q \vdash_{\mathtt{CH}} \Delta', x{:}A}{(\nu x)(P \mid Q) \vdash_{\mathtt{CH}} \Delta, \Delta'}$$

$$\text{(T-}\oplus\text{)}\ \frac{P \vdash_{\mathtt{CH}} \Delta, x{:}A_j \qquad j \in I}{x \triangleleft l_j.P \vdash_{\mathtt{CH}} \Delta, x{:} \oplus \{l_i : A_i\}_{i \in I}} \qquad \text{(T-\&)}\ \frac{P_i \vdash_{\mathtt{CH}} \Delta, x{:}A_i \qquad \forall i \in I}{x \triangleright \{l_i : P_i\}_{i \in I} \vdash_{\mathtt{CH}} \Delta, x{:}\&\{l_i : A_i\}_{i \in I}} \qquad \text{(T-mix)}\ \frac{P \vdash_{\mathtt{CH}} \Delta \qquad Q \vdash_{\mathtt{CH}} \Delta'}{P \mid Q \vdash_{\mathtt{CH}} \Delta, \Delta'}$$

Figure 3: Typing rules for the π-calculus with C-types.

differences in syntax, we have also some minor modifications in reduction rules. Differences with respect to the language considered in § 2 are summarized in the following:

$$P, Q ::= \quad (\nu x)P \quad \text{(channel restriction)} \quad \mid \quad [x \leftrightarrow y] \quad \text{(forwarding)}$$

$$\begin{array}{llll} \text{(R-ChCom)} & \overline{x}\langle v\rangle.P \mid x(z).Q \to P \mid Q[v/z] & \text{(R-Fwd)} & (\nu x)([x \leftrightarrow y] \mid P) \to P[y/x] \\ \text{(R-ChCase)} & x \triangleleft l_j.P \mid x \triangleright \{l_i : P_i\}_{i \in I} \to P \mid P_j \quad j \in I & \text{(R-ChRes)} & P \to Q \Longrightarrow (\nu x)P \to (\nu x)Q \end{array}$$

Observe how interaction of input/output prefixes and selection/branching is no longer covered by an outermost restriction. As for the type system, we consider the so-called C-types which correspond to linear logic propositions. They are given by the following grammar:

$$A, B ::= \bot \ \mid\ \mathbf{1} \ \mid\ A \otimes B \ \mid\ A \parr B \ \mid\ \oplus\{l_i : A_i\}_{i \in I} \ \mid\ \&\{l_i : A_i\}_{i \in I}$$

Intuitively, $\bot$ and $\mathbf{1}$ are used to type a terminated endpoint. Type $A \otimes B$ is associated to an endpoint that first outputs an object of type A and then behaves according to B. Dually, type $A \parr B$ is the type of an endpoint that first inputs an object of type A and then continues as B. The interpretation of $\oplus\{l_i : A_i\}_{i \in I}$ and $\&\{l_i : A_i\}_{i \in I}$ as select and branch behaviors follows as expected.

We define a full duality on C-types, which exactly corresponds to the negation operator of CLL $(\cdot)^{\bot}$. The *dual* of type A, denoted $\overline{A}$, is inductively defined as follows:

$$\begin{array}{rclcrclcrcl} \overline{\mathbf{1}} & = & \bot & \qquad & \overline{\bot} & = & \mathbf{1} & \qquad & \overline{\oplus\{l_i : A_i\}_{i \in I}} & = & \&\{l_i : \overline{A}_i\}_{i \in I} \\ \overline{A \otimes B} & = & \overline{A} \parr \overline{B} & & \overline{A \parr B} & = & \overline{A} \otimes \overline{B} & & \overline{\&\{l_i : A_i\}_{i \in I}} & = & \oplus\{l_i : \overline{A}_i\}_{i \in I} \end{array}$$

Recall that $A \multimap B \triangleq \overline{A} \parr B$. As explained in [1], considering mix principles means admitting $\bot \multimap \mathbf{1}$ and $\mathbf{1} \multimap \bot$, and therefore $\bot = \mathbf{1}$. We write $\bullet$ to denote either $\bot$ or $\mathbf{1}$, and decree that $\overline{\bullet} = \bullet$.

Typing contexts, sets of typing assignments $x : A$, are ranged over $\Delta, \Delta', \ldots$. The empty context is denoted '$\cdot$'. Typing judgments are then of the form $P \vdash_{\mathtt{CH}} \Delta$. Figure 3 gives the typing rules associated to the linear logic interpretation. Salient points include the use of bound output $(\nu y)\overline{x}\langle y\rangle.P$, which is abbreviated as $\overline{x}(y)P$. Another highlight is the "composition plus hiding" principle implemented by rule (T-cut), which integrates parallel composition and restriction in a single rule. Indeed, there is no dedicated rule for restriction. Also, rule (T-mix) enables the typing of *independent parallel compositions*, i.e., the composition of two processes that do not share sessions.

We now collect main results for this type system; see [1, 3] for details. For any P, define $\mathit{live}(P)$ if and only if $P \equiv (\nu \widetilde{n})(\pi.Q \mid R)$, where π is an input, output, selection, or branching prefix.

$$
\begin{array}{rlll}
U ::= & ?^{\circ}_{\kappa}.U & \text{(used in input)} & \emptyset \quad \text{(not usable)} \\
 & !^{\circ}_{\kappa}.U & \text{(used in output)} & (U_1 \mid U_2) \quad \text{(used in parallel)} \\
T ::= & U[\widetilde{T}] & \text{(channel types)} & \langle l : T \rangle_{i \in I} \quad \text{(variant type)}
\end{array}
$$

Figure 4: Syntax of usage types for the π-calculus.

Theorem 3.1 (Type Preservation for C-Types)**.** *If $P \vdash_{\mathtt{CH}} \Delta$ and $P \longrightarrow Q$ then $Q \vdash_{\mathtt{CH}} \Delta$.*

Theorem 3.2 (Progress)**.** *If $P \vdash_{\mathtt{CH}} \cdot$ and $live(P)$ then $P \longrightarrow Q$, for some Q.*

3.2 Deadlock Freedom by Encodability

As mentioned above, the second approach to deadlock-free session processes is *indirect*, in the sense that establishing deadlock freedom for session processes appeals to usage types for the π-calculus [13,15], for which type systems enforcing deadlock freedom are well-established. Formally, this reduction exploits encodings of processes and types: a session process $\Gamma \vdash_{\mathtt{ST}} P$ is encoded into a (standard) π-calculus process $[\![\Gamma]\!]_f \vdash^n_{\mathtt{KB}} [\![P]\!]_f$. Next we introduce the syntax of standard π-calculus processes with variant values (§ 3.2.1), the discipline of usage types (§ 3.2.2), and the encodings of session processes and types into standard π-calculus processes and usage types, respectively (§ 3.2.3).

3.2.1 Processes

The syntax and semantics of the π-calculus with usage types build upon those in § 2. We require some modifications. First, the encoding of terms presented in § 3.2.3, requires polyadic communication. Rather than branching and selection constructs, the π-calculus that we consider here includes a *case* construct $\mathbf{case}\, v\, \mathbf{of}\, \{l_i\_x_i \triangleright P_i\}_{i \in I}$ that uses *variant value* l_j_v. Moreover, we consider the standard channel restriction, rather than double restriction. These modifications are summarized below:

$$
\begin{array}{rlll}
P,Q ::= & (\nu x)P & \text{(channel restriction)} & \mid \quad \mathbf{case}\, v\, \mathbf{of}\, \{l_i\_x_i \triangleright P_i\}_{i \in I} \quad \text{(case)} \\
v ::= & l_j\_v & \text{(variant value)} &
\end{array}
$$

$$
\begin{array}{ll}
(\mathrm{R}\pi\text{-}\textsc{Com}) & \overline{x}\langle \widetilde{v} \rangle.P \mid x(\widetilde{z}).Q \to P \mid Q[\widetilde{v}/\widetilde{z}] \\
(\mathrm{R}\pi\text{-}\textsc{Res}) & P \to Q \Longrightarrow (\nu x)P \to (\nu x)Q \\
(\mathrm{R}\pi\text{-}\textsc{Case}) & \mathbf{case}\, l_j\_v\, \mathbf{of}\, \{l_i\_x_i \triangleright P_i\}_{i \in I} \to P_j[v/x_i] \quad j \in I
\end{array}
$$

The definition of deadlock-freedom for the π-calculus follows [13, 15]:

Definition 3.3 (Deadlock Freedom for Standard π-Calculus)**.** *A process P_0 is* deadlock-free *under fair scheduling, if for any fair reduction sequence $P_0 \to P_1 \to P_2 \to \cdots$ the following hold*

1. *if $P_i \equiv (\nu \tilde{x})(\overline{x}\langle \widetilde{v} \rangle.Q \mid R)$ for $i \geq 0$, implies that there exists $n \geq i$ such that $P_n \equiv (\nu \tilde{x})(\overline{x}\langle \widetilde{v} \rangle.Q \mid x(\widetilde{z}).R_1 \mid R_2)$ and $P_{n+1} \equiv (\nu \tilde{x})(Q \mid R_1[\widetilde{v}/\widetilde{z}] \mid R_2)$;*
2. *if $P_i \equiv (\nu \tilde{x})(x(\widetilde{z}).Q \mid R)$ for $i \geq 0$, implies that there exists $n \geq i$ such that $P_n \equiv (\nu \tilde{x})(x(\widetilde{z}).Q \mid \overline{x}\langle \widetilde{v} \rangle.R_1 \mid R_2)$ and $P_{n+1} \equiv (\nu \tilde{x})(Q[\widetilde{v}/\widetilde{z}] \mid R_1 \mid R_2)$.*

3.2.2 Usage Types

The syntax of usage types is defined in Figure 4. For simplicity, we let α range over input ? or output ! actions. The usage $\emptyset$ describes a channel that cannot be used at all. We will often omit $\emptyset$, and so we

will write U instead of $U.\emptyset$. Usages $?^{\mathrm{o}}_{\kappa}.U$ and $!^{\mathrm{o}}_{\kappa}.U$ describe channels that can be used once for input and output, respectively and then used according to the continuation usage U. The *obligation* o and *capability* κ range over the set of natural numbers. The usage $U_1 \mid U_2$ describes a channel that is used according to U_1 by one process and U_2 by another processes in parallel.

Intuitively, obligations and capabilities describe inter-channel dependencies:

- An obligation of level n must be fulfilled by using only capabilities of level *less than* n. Said differently, an action of obligation n must be prefixed by actions of capabilities less than n.
- For an action with capability of level n, there must exist a co-action with obligation of level *less than or equal to* n.

Typing contexts are sets of typing assignments and are ranged over Γ, Γ'. A typing judgement is of the form $\Gamma \vdash^{n}_{\mathrm{KB}} P$: the annotation n explicitly denotes the greatest *degree of sharing* admitted in parallel processes. Before commenting on the typing rules (given in Figure 5), we discuss some important auxiliary notions, extracted from [13, 15]. First, the composition operation on types (denoted $\mid$, and used in rules Tπ-(Par)$_n$ and Tπ-(Out)) is based on the composition of usages and is defined as follows:

$$\langle l_i : T_i \rangle_{i\in I} \mid \langle l_i : T_i \rangle_{i\in I} = \langle l_i : T_i \rangle_{i\in I} \qquad U_1[\widetilde{T}] \mid U_2[\widetilde{T}] = (U_1 \mid U_2)[\widetilde{T}]$$

The generalization of $\mid$ to typing contexts, denoted $(\Gamma_1 \mid \Gamma_2)(x)$, is defined as expected. The unary operation $\uparrow^t$ applied to a usage U lifts its obligation level *up to* t; it is defined inductively as:

$$\uparrow^t \emptyset = \emptyset \qquad \uparrow^t \alpha^{\mathrm{o}}_{\kappa}.U = \alpha^{max(\mathrm{o},t)}_{\kappa}.U \qquad \uparrow^t (U_1 \mid U_2) = (\uparrow^t U_1 \mid \uparrow^t U_2)$$

The $\uparrow^t$ is extends to types/typing contexts as expected. *Duality* on usage types simply exchanges ? and !:

$$\overline{\emptyset[]} = \emptyset[] \qquad \overline{?^{\mathrm{o}}_{\kappa}.U[\widetilde{T}]} = !^{\mathrm{o}}_{\kappa}.\overline{U}[\widetilde{T}] \qquad \overline{!^{\mathrm{o}}_{\kappa}.U[\widetilde{T}]} = ?^{\mathrm{o}}_{\kappa}.\overline{U}[\widetilde{T}]$$

Operator " ; " in $\Delta = x : [T]\alpha^{\mathrm{o}}_{\kappa}$; Γ, used in rules (Tπ-In) and (Tπ-Out), is such that the following hold:

$$\mathsf{dom}(\Delta) = \{x\} \cup \mathsf{dom}(\Gamma) \qquad \Delta(x) = \begin{cases} \alpha^{\mathrm{o}}_{\kappa}.U[\widetilde{T}] & \text{if } \Gamma(x) = U[\widetilde{T}] \\ \alpha^{\mathrm{o}}_{\kappa}[\widetilde{T}] & \text{if } x \notin dom(\Gamma) \end{cases} \qquad \Delta(y) = \uparrow^{\kappa+1} \Gamma(y) \text{ if } y \neq x$$

The final required notion is that of a *reliable usage*. It builds upon the following definition:

Definition 3.4. *Let U be a usage. The input and output* obligation levels *(resp.* capability levels*) of U, written $\mathsf{ob}_?(U)$ and $\mathsf{ob}_!(U)$ (resp. $\mathsf{cap}_?(U)$ and $\mathsf{cap}_!(U)$), are defined as:*

$$\begin{array}{rclcrcl} \mathsf{ob}_\alpha(\alpha^{\mathrm{o}}_{\kappa}.U) & = & \mathrm{o} & & \mathsf{cap}_\alpha(\alpha^{\mathrm{o}}_{\kappa}.U) & = & \kappa \\ \mathsf{ob}_\alpha(U_1 \mid U_2) & = & min(\mathsf{ob}_\alpha(U_1), \mathsf{ob}_\alpha(U_2)) & & \mathsf{cap}_\alpha(U_1 \mid U_2) & = & min(\mathsf{cap}_\alpha(U_1), \mathsf{cap}_\alpha(U_2)) \end{array}$$

The definition of reliable usages depends on a reduction relation on usages, noted $U \to U'$. Intuitively, $U \to U'$ means that if a channel of usage U is used for communication, then after the communication occurs, the channel should be used according to usage U'. Thus, e.g., $?^{\mathrm{o}}_{\kappa}.U_1 \mid ?^{\mathrm{o}'}_{\kappa'}.U_2$ reduces to $U_1 \mid U_2$.

Definition 3.5 (Reliability)**.** *We write $\mathsf{con}_\alpha(U)$ when $\mathsf{ob}_{\overline{\alpha}}(U) \leq \mathsf{cap}_\alpha(U)$. We write $\mathsf{con}(U)$ when $\mathsf{con}_?(U)$ and $\mathsf{con}_!(U)$ hold. Usage U is* reliable*, noted $\mathsf{rel}(U)$, if $\mathsf{con}(U')$ holds $\forall U'$ such that $U \to^* U'$.*

$$\frac{}{x : \emptyset[] \vdash^n_{\mathtt{KB}} \mathbf{0}}\ (\text{T}\pi\text{-Nil}) \qquad \frac{\Gamma, x : U[\widetilde{T}] \vdash^n_{\mathtt{KB}} P \quad \mathsf{rel}(U)}{\Gamma \vdash^n_{\mathtt{KB}} (\nu x)P}\ (\text{T}\pi\text{-Res}) \qquad \frac{\Gamma_1 \vdash^n_{\mathtt{KB}} P \qquad \Gamma_2 \vdash^n_{\mathtt{KB}} Q \qquad |\Gamma_1 \cap \Gamma_1| \leq n}{\Gamma_1 \mid \Gamma_2 \vdash^n_{\mathtt{KB}} P \mid Q}\ (\text{T}\pi\text{-Par}_n) \qquad \frac{\Gamma, \widetilde{y} : \widetilde{T} \vdash^n_{\mathtt{KB}} P}{x : ?^0_\kappa[\widetilde{T}]\ ; \Gamma \vdash^n_{\mathtt{KB}} x(\widetilde{y}).P}\ (\text{T}\pi\text{-In})$$

$$\frac{\Gamma_1 \vdash^n_{\mathtt{KB}} \widetilde{v} : \widetilde{T} \qquad \Gamma_2 \vdash^n_{\mathtt{KB}} P}{x : !^0_\kappa[\widetilde{T}]\ ; (\Gamma_1 \mid \Gamma_2) \vdash^n_{\mathtt{KB}} \overline{x}\langle\widetilde{v}\rangle.P}\ (\text{T}\pi\text{-Out}) \qquad \frac{\Gamma \vdash^n_{\mathtt{KB}} v : T_j \quad \exists j \in I}{\Gamma \vdash^n_{\mathtt{KB}} l_j\_v : \langle l_i : T_i\rangle_{i\in I}}\ (\text{T}\pi\text{-LVal}) \qquad \frac{\Gamma_1 \vdash^n_{\mathtt{KB}} v : \langle l_i : T_i\rangle_{i\in I} \qquad \Gamma_2, x_i : T_i \vdash^n_{\mathtt{KB}} P_i \quad \forall i \in I}{\Gamma_1, \Gamma_2 \vdash^n_{\mathtt{KB}} \mathbf{case}\, v\, \mathbf{of}\, \{l_i\_x_i \triangleright P_i\}_{i\in I}}\ (\text{T}\pi\text{-Case})$$

Figure 5: Typing rules for the π-calculus with usage types with degree of sharing n.

Typing Rules. The typing rules for the standard π-calculus with usage types are given in Figure 5. The only difference with respect to the rules in Kobayashi's systems [13, 15] is that we annotate typing judgements with the degree of sharing, explicitly stated in rule (Tπ-Par$_n$)–see below. Rule (Tπ-Nil) states that the terminated process is typed under a terminated channel. Rule (Tπ-Res) states that process $(\nu x)P$ is well-typed if the usage for x is reliable (cf. Definition 3.5). Rules (Tπ-In) and (Tπ-Out) type input and output processes in a typing context where the " ; " operator is used in order to increase the obligation level of the channels in continuation P. Rules (Tπ-LVal) and (Tπ-Case) type a choice: the first types a variant value with a variant type; the second types a case process using a variant value as its guard.

Given a degree of sharing n, rule (Tπ-Par$_n$) states that the parallel composition of processes P and Q (typable under contexts Γ_1 and Γ_2, respectively) is well-typed under the typing context $\Gamma_1 \mid \Gamma_2$ only if $|\Gamma_1 \cap \Gamma_2| \leq n$. This allows to simply characterize the "concurrent cooperation" between P and Q. As a consequence, if $P \vdash^n_{\mathtt{KB}}$ then $P \vdash^k_{\mathtt{KB}}$, for any $k \leq n$. Observe that the typing rule for parallel composition in [13, 15] is the same as (Tπ-Par$_n$), except for condition $|\Gamma_1 \cap \Gamma_2| \leq n$, which is not specified.

The next theorems imply that well-typed processes by the type system in Figure 5 are deadlock-free.

Theorem 3.6 (Type Preservation for Usage Types)**.** *If $\Gamma \vdash^n_{\mathtt{KB}} P$ and $P \to Q$, then $\Gamma' \vdash^n_{\mathtt{KB}} Q$ for some Γ' such that $\Gamma \to \Gamma'$.*

Theorem 3.7 (Deadlock Freedom)**.** *If $\emptyset \vdash^n_{\mathtt{KB}} P$ and either $P \equiv (\nu\tilde{x})(x(\widetilde{z}).Q \mid R)$ or $P \equiv (\nu\tilde{x})(\overline{x}\langle\widetilde{v}\rangle.Q \mid R)$, then $P \to Q$, for some Q.*

Corollary 3.8. *If $\emptyset \vdash^n_{\mathtt{KB}} P$, then P is deadlock-free, in the sense of Definition 3.3.*

Theorem 3.2 (progress for the linear logic system) and Theorem 3.7 (deadlock freedom for the standard π-calculus) have a rather similar formulation: both properties state that processes can always reduce if they are well-typed (under the empty typing context) and have an appropriate structure (i.e., condition $live(P)$ in Theorem 3.2 and condition $P \equiv (\nu\tilde{x})(x(\widetilde{z}).Q \mid R)$ or $P \equiv (\nu\tilde{x})(\overline{x}\langle\widetilde{v}\rangle.Q \mid R)$ in Theorem 3.7).

3.2.3 Encodings of Processes and Types

Encoding Processes. To relate classes of processes obtained by the different type systems given so far, we rewrite a session typed or C-typed process into a usage typed process by following a continuation-passing style: this allows us to mimic the structure of a session or C-type by sending its continuation as a payload over a channel. This idea, suggested in [14] and developed in [8], is recalled in Figure 6.

$$
\begin{array}{lcllcl}
[\![\overline{x}\langle v\rangle.P]\!]_f & \triangleq & (\nu c)\overline{f_x}\langle v,c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & [\![x\triangleright\{l_i:P_i\}_{i\in I}]\!]_f & \triangleq & f_x(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I} \\
[\![x(y).P]\!]_f & \triangleq & f_x(y,c).[\![P]\!]_{f,\{x\mapsto c\}} & [\![(\nu xy)P]\!]_f & \triangleq & (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}} \\
[\![x\triangleleft l_j.P]\!]_f & \triangleq & (\nu c)\overline{f_x}\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & [\![P\mid Q]\!]_f & \triangleq & [\![P]\!]_f \mid [\![Q]\!]_f
\end{array}
$$

Figure 6: Encoding of session processes into π-calculus processes.

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\mathbf{end}]\!]_{\mathsf{su}} & = & \emptyset[] & [\![\mathbf{end}]\!]_{\mathsf{c}} & = & \bullet \\
[\![?T.S]\!]_{\mathsf{su}} & = & ?^{\mathrm{o}}_{\kappa}[[\![T]\!]_{\mathsf{su}}, [\![S]\!]_{\mathsf{su}}] & [\![?T.S]\!]_{\mathsf{c}} & = & [\![T]\!]_{\mathsf{c}} \parr [\![S]\!]_{\mathsf{c}} \\
[\![!T.S]\!]_{\mathsf{su}} & = & !^{\mathrm{o}}_{\kappa}[[\![T]\!]_{\mathsf{su}}, [\![\overline{S}]\!]_{\mathsf{su}}] & [\![!T.S]\!]_{\mathsf{c}} & = & [\![\overline{T}]\!]_{\mathsf{c}} \otimes [\![S]\!]_{\mathsf{c}} \\
[\![\&\{l_i:S_i\}_{i\in I}]\!]_{\mathsf{su}} & = & ?^{\mathrm{o}}_{\kappa}[\langle l_i : [\![S_i]\!]_{\mathsf{su}}\rangle_{i\in I}] & [\![\&\{l_i:S_i\}_{i\in I}]\!]_{\mathsf{c}} & = & \&\{l_i : [\![S_i]\!]_{\mathsf{c}}\}_{i\in I} \\
[\![\oplus\{l_i:S_i\}_{i\in I}]\!]_{\mathsf{su}} & = & !^{\mathrm{o}}_{\kappa}[\langle l_i : [\![\overline{S_i}]\!]_{\mathsf{su}}\rangle_{i\in I}] & [\![\oplus\{l_i:S_i\}_{i\in I}]\!]_{\mathsf{c}} & = & \oplus\{l_i : [\![S_i]\!]_{\mathsf{c}}\}_{i\in I}
\end{array}
$$

Figure 7: Encodings of session types into usage types (Left) and C-types (Right).

Encoding of Types. We formally relate session types and logic propositions to usage types by means of the encodings given in Figure 7. The former one, denoted as denoted $[\![\cdot]\!]_{\mathsf{su}}$, is taken from [8].

Definition 3.9. *Let Γ be a session typing context. The encoding $[\![\cdot]\!]_f$ into usage typing context and $[\![\cdot]\!]_{\mathsf{c}}$ into C-typing context is inductively defined as follows:*

$$[\![\emptyset]\!]_f = [\![\emptyset]\!]_{\mathsf{c}} \triangleq \emptyset \qquad [\![\Gamma, x:T]\!]_f \triangleq [\![\Gamma]\!]_f, f_x : [\![T]\!]_{\mathsf{su}} \qquad [\![\Gamma, x:T]\!]_{\mathsf{c}} \triangleq [\![\Gamma]\!]_{\mathsf{c}}, x : [\![T]\!]_{\mathsf{c}}$$

Lemma 3.10 (Duality and encoding of session types)**.** *Let T, S be finite session types. Then: (i) $\overline{T} = S$ if and only if $\overline{[\![T]\!]_{\mathsf{c}}} = [\![S]\!]_{\mathsf{c}}$; (ii) $\overline{T} = S$ if and only if $\overline{[\![T]\!]_{\mathsf{su}}} = [\![S]\!]_{\mathsf{su}}$.*

On Deadlock Freedom by Encoding. The next results relate deadlock freedom, typing and encoding.

Proposition 3.11. *Let P be a deadlock-free session process, then $[\![P]\!]_f$ is a deadlock-free π-process.*

Proof. Follows by the encoding of terms given in Figure 6, Definition 2.5 and Definition 3.3. □

Next we recall an important result relating deadlock freedom and typing, by following [4].

Corollary 3.12. *Let $\vdash_{\mathtt{ST}} P$ be a session process. If $\vdash^{n}_{\mathtt{KB}} [\![P]\!]_f$ is deadlock-free then P is deadlock-free.*

4 A Hierarchy of Deadlock-Free Session Typed Processes

Preliminaries. To formally define the classes $\mathscr{L}$ and $\mathscr{K}$, we require some auxiliary definitions. The following translation addresses minor syntactic differences between session typed processes (cf. § 2) and the processes typable in the linear logic interpretation of session types (cf. § 3.1). Such differences concern output actions and the restriction operator:

Definition 4.1. *Let P be a session process. The translation $\{\!\{\cdot\}\!\}$ is defined as*

$$\{\!\{\overline{x}\langle y\rangle.P\}\!\} = \overline{x}(z).([z\leftrightarrow y] \mid \{\!\{P\}\!\}) \qquad \{\!\{(\nu xy)P\}\!\} = (\nu w)\{\!\{P\}\!\}[w/x][w/y] \quad w \notin \mathsf{fn}(P)$$

and as an homomorphism for the other process constructs.

Let $[\![\cdot]\!]_{\mathsf{c}}$ denote the encoding of session types into linear logic propositions in Figure 7 (right). Recall that $[\![\cdot]\!]_f$ stands for the encoding of processes and $[\![\cdot]\!]_{\mathsf{su}}$ for the encoding of types, both defined in [8], and given here in Figure 6 and Figure 7 (left), respectively. We may then formally define the languages under comparison as follows:

Definition 4.2 (Typed Languages)**.** *The languages $\mathscr{L}$ and $\mathscr{K}_n$ ($n \geq 0$) are defined as follows:*

$$\begin{array}{rcl} \mathscr{L} & = & \{P \mid \exists \Gamma.\ (\Gamma \vdash_{\mathtt{ST}} P \wedge \{\!\{P\}\!\} \vdash_{\mathtt{CH}} [\![\Gamma]\!]_{\mathsf{c}})\} \\ \mathscr{K}_n & = & \{P \mid \exists \Gamma, f.\ (\Gamma \vdash_{\mathtt{ST}} P \wedge [\![\Gamma]\!]_f \vdash^n_{\mathtt{KB}} [\![P]\!]_f)\} \end{array}$$

Main Results. Our first observation is that there are processes in $\mathscr{K}_2$ but not in $\mathscr{K}_1$:

Lemma 4.3. $\mathscr{K}_1 \subset \mathscr{K}_2$.

Proof. $\mathscr{K}_2$ contains (deadlock-free) session processes not captured in $\mathscr{K}_1$. A representative example is:

$$P_2 = (\nu a_1 b_1)(\nu a_2 b_2)(a_1(x).\, \overline{a_2}\langle x\rangle \mid \overline{b_1}\langle \mathbf{n}\rangle.\, b_2(z))$$

This process is not in $\mathscr{K}_1$ because it involves the composition of two parallel processes which share two sessions. As such, it is typable in $\vdash^n_{\mathtt{KB}}$ (with $n \geq 2$) but not in $\vdash^1_{\mathtt{KB}}$. □

The previous result generalizes easily, so as to define a hierarchy of deadlock-free, session processes:

Theorem 4.4. *For all $n \geq 1$, we have that $\mathscr{K}_n \subset \mathscr{K}_{n+1}$.*

Proof. Immediate by considering one of the following processes, which generalize process P_2 in Lemma 4.3:

$$\begin{array}{rcl} P_{n+1} & = & (\nu a_1 b_1)(\nu a_2 b_2)\cdots(\nu a_{n+1} b_{n+1})(a_1(x).\, \overline{a_2}\langle x\rangle.\cdots.\, \overline{a_{n+1}}\langle y\rangle \mid \overline{b_1}\langle \mathbf{n}\rangle.\, b_2(z).\, \cdots\, b_{n+1}(z)) \\ Q_{n+1} & = & (\nu a_1 b_1)(\nu a_2 b_2)\cdots(\nu a_{n+1} b_{n+1})(a_1(x).\, \overline{a_2}\langle x\rangle.\cdots.\, a_{n+1}(y) \mid \overline{b_1}\langle \mathbf{n}\rangle.\, b_2(z).\, \cdots\, \overline{b_{n+1}}\langle \mathbf{n}\rangle) \end{array}$$

To distinguish $\mathscr{K}_{n+1}$ from $\mathscr{K}_n$, we consider P_{n+1} if $n+1$ is even and Q_{n+1} otherwise. □

One main result of this paper is that $\mathscr{L}$ and $\mathscr{K}_1$ coincide. Before stating this result, we make the following observations. The typing rules for processes in $\mathscr{L}$ do not directly allow free output. However, free output is representable (and typable) by linear logic types by means of the transformation in Definition 4.1. Thus, considered processes are not syntactically equal. In $\mathscr{L}$ there is cooperating composition (enabled by rule (T-cut) in Figure 3); independent composition can only be enabled by rule (T-mix). Arbitrary restriction is not allowed; only restriction of parallel processes.

The following property is key in our developments: it connects our encodings of (dual) session types into usage types with reliability (Definition 3.5), a central notion to the type system for deadlock freedom in Figure 5. Recall that, unlike usage types, there is no parallel composition operator at the level of session types.

Proposition 4.5. *Let T be a session type. Then* $\mathsf{rel}([\![T]\!]_{\mathsf{su}} \mid [\![\overline{T}]\!]_{\mathsf{su}})$ *holds.*

Proof (Sketch). By induction on the structure of session type T and the definitions of $[\![\cdot]\!]_{\mathsf{su}}$ and predicate $\mathsf{rel}(\cdot)$, using Lemma 3.10 (encodings of types preserve session type duality). See [9] for details. □

We then have the following main result, whose proof is detailed in [9]:

Theorem 4.6. $\mathscr{L} = \mathscr{K}_1$.

Therefore, we have the following corollary, which attests that the class of deadlock-free session processes naturally induced by linear logic interpretations of session types is strictly included in the class induced by the indirect approach of Dardha et al. [8] (cf. § 3.2).

Corollary 4.7. $\mathscr{L} \subset \mathscr{K}_n$, $n > 1$.

The fact that (deadlock-free) processes such as P_2 (cf. Lemma 4.3) are not in $\mathscr{L}$ is informally discussed in [3, §6]. However, [3] gives no formal comparisons with other classes of deadlock-free processes.

5 Rewriting $\mathscr{K}_n$ into $\mathscr{L}$

The hierarchy of deadlock-free session processes established by Theorem 4.4 is *subtle* in the following sense: if $P \in \mathscr{K}_{k+1}$ but $P \notin \mathscr{K}_k$ (with $k \geq 1$) then we know that there is a subprocess of P that needs to be "adjusted" in order to "fit in" $\mathscr{K}_k$. More precisely, we know that such a subprocess of P must become more independent in order to be typable under the lesser degree of sharing k.

Here we propose a *rewriting procedure* that converts processes in $\mathscr{K}_n$ into processes in $\mathscr{K}_1$ (that is, $\mathscr{L}$, by Theorem 4.6). The rewriting procedure follows a simple idea: given a parallel process as input, return as output a process in which one of the components is kept unchanged, but the other is replaced by parallel representatives of the sessions implemented in it. Such parallel representatives are formally defined as characteristic processes and catalyzers, introduced next. The rewriting procedure is type preserving and satisfies operational correspondence (cf. Theorems 5.8 and 5.10).

5.1 Preliminaries: Characteristic Processes and Catalyzers

Before presenting our rewriting procedure, let us first introduce some preliminary results.

Definition 5.1 (Characteristic Processes of a Session Type)**.** *Let T be a session type (cf. § 2). Given a name x, the set of* characteristic processes *of T, denoted $\{\!|T|\!\}^x$, is inductively defined as follows:*

$$\begin{array}{rcl}
\{\!|\mathbf{end}|\!\}^x & = & \{P \mid P \vdash_{\mathtt{CH}} x\colon \bullet\} \\
\{\!|?T.S|\!\}^x & = & \{x(y).P \mid P \vdash_{\mathtt{CH}} y\colon [\![T]\!]_{\mathsf{c}}, x\colon [\![S]\!]_{\mathsf{c}}\} \\
\{\!|!T.S|\!\}^x & = & \{\overline{x}(y).(P \mid Q) \mid P \in \{\!|\overline{T}|\!\}^y \wedge Q \in \{\!|S|\!\}^x\} \\
\{\!|\&\{l_i : S_i\}_{i\in I}|\!\}^x & = & \{x \triangleright \{l_i : P_i\}_{i\in I} \mid \forall i \in I.\ P_i \in \{\!|S_i|\!\}^x\} \\
\{\!|\oplus\{l_i : S_i\}_{i\in I}|\!\}^x & = & \bigcup_{i\in I} \{x \triangleleft l_i.P_i \mid P_i \in \{\!|S_i|\!\}^x\}
\end{array}$$

Definition 5.2 (Catalyzer)**.** *Given a session typing context Γ, we define its associated* catalyzer *as a process context $\mathscr{C}_\Gamma[\cdot]$, as follows:*

$$\mathscr{C}_\emptyset[\cdot] = [\cdot] \qquad \mathscr{C}_{\Gamma,x:T}[\cdot] = (\nu x)(\mathscr{C}_\Gamma[\cdot] \mid P) \quad \textit{with } P \in \{\!|\overline{T}|\!\}^x$$

We record the fact that characteristic processes are well-typed in the system of § 3.1:

Lemma 5.3. *Let T be a session type. For all $P \in \{\!|T|\!\}^x$, we have: $P \vdash_{\mathtt{CH}} x : [\![T]\!]_{\mathsf{c}}$*

We use $\{\!|T|\!\}^x \vdash_{\mathtt{CH}} x : [\![T]\!]_{\mathsf{c}}$ to denote the set of processes $P \in \{\!|T|\!\}^x$ such that $P \vdash_{\mathtt{CH}} x : [\![T]\!]_{\mathsf{c}}$.

Lemma 5.4 (Catalyzers Preserve Typability)**.** *Let $\Gamma \vdash_{\mathtt{ST}} P$ and $\Gamma' \subseteq \Gamma$. Then $\mathscr{C}_{\Gamma'}\big[P\big] \vdash_{\mathtt{CH}} [\![\Gamma]\!]_{\mathsf{c}} \setminus [\![\Gamma']\!]_{\mathsf{c}}$.*

Corollary 5.5. *Let $\Gamma \vdash_{\mathtt{ST}} P$. Then $\mathscr{C}_\Gamma\big[P\big] \vdash_{\mathtt{CH}} \emptyset$.*

5.2 Rewriting $\mathscr{K}_n$ in $\mathscr{L}$

We start this section with some notations. First, in order to represent pseudo-non deterministic binary choices between two equally typed processes, we introduce the following:

Notation 5.6. *Let P_1, P_2 be two processes such that $k \notin \mathsf{fn}(P_1,P_2)$. We write $P_1 \parallel_k P_2$ to stand for the process $(\nu k)(k \triangleleft \mathtt{inx}.\mathbf{0} \mid k \triangleright \{\mathtt{inl}:P_1, \mathtt{inr}:P_2\})$, where label $\mathtt{inx}$ stands for either $\mathtt{inl}$ or $\mathtt{inr}$.*

Clearly, since session execution is purely deterministic, notation $P_1 \parallel_k P_2$ denotes that either P_1 or P_2 will be executed (and that the actual deterministic choice is not relevant). It is worth adding that Caires has already developed the technical machinery required to include non deterministic behavior into the linear logic interpretation of session types; see [1]. Integrating such non-deterministic behavior into our rewriting procedure is interesting future work.

For syntactic convenience, we annotate bound names in processes with session types, and write $(\nu xy : T)P$ and $x(y:T).P$, for some session type T. When the reduction relation involves a left or right choice in a binary labelled choice, as in reductions due to pseudo-non deterministic choices (Notation 5.6), we sometimes annotate the reduction as $\rightarrow^{\mathtt{inl}}$ or $\rightarrow^{\mathtt{inr}}$. We let $\mathtt{C}$ denote a *process context*, i.e., a process with a hole. And finally, for a typing context Γ, we shall write $\{\!|\Gamma|\!\}$ to denote the process $\prod_{(w_i:T_i)\in\Gamma} \{\!|T_i|\!\}^{w_i}$. We are now ready to give the rewriting procedure from $\mathscr{K}_n$ to $\mathscr{L}$.

Definition 5.7 (Rewriting $\mathscr{K}_n$ into $\mathscr{L}$)**.** *Let $P \in \mathscr{K}_n$ such that $\Gamma \vdash_{\mathtt{ST}} P$, for some Γ. The encoding $(\!|\Gamma \vdash_{\mathtt{ST}} P|\!)$ is a process of $\mathscr{L}$ inductively defined as follows:*

$$
\begin{array}{rll}
(\!|x : \mathbf{end} \vdash_{\mathtt{ST}} \mathbf{0}|\!) \triangleq \mathbf{0} & & \\
(\!|\Gamma \vdash_{\mathtt{ST}} \overline{x}\langle v\rangle.P'|\!) \triangleq \overline{x}(z).\big([v \leftrightarrow z] \mid (\!|\Gamma', x : S \vdash_{\mathtt{ST}} P'|\!)\big) & & \Gamma = \Gamma', x : !T.S, v : T \\
(\!|\Gamma \vdash_{\mathtt{ST}} x(y:T).P'|\!) \triangleq x(y).(\!|\Gamma', x : S, y : T \vdash_{\mathtt{ST}} P'|\!) & & \Gamma = \Gamma', x : ?T.S \\
(\!|\Gamma \vdash_{\mathtt{ST}} x \triangleleft l_j.P'|\!) \triangleq x \triangleleft l_j.(\!|\Gamma', x : S_j \vdash_{\mathtt{ST}} P'|\!) & & \Gamma = \Gamma', x : \oplus\{l_i : S_i\}_{i\in I} \\
(\!|\Gamma \vdash_{\mathtt{ST}} x \triangleright \{l_i : P_i\}_{i\in I}|\!) \triangleq x \triangleright \{l_i : (\!|\Gamma', x : S_i \vdash_{\mathtt{ST}} P_i|\!)\}_{i\in I} & & \Gamma = \Gamma', x : \&\{l_i : S_i\}_{i\in I} \\
(\!|\Gamma \vdash_{\mathtt{ST}} (\nu \widetilde{xy} : \widetilde{S})(P \mid Q)|\!) \triangleq \{\!|\Gamma_2|\!\} \mid \mathscr{C}_{\widetilde{z}:\widetilde{S}}\big[(\!|\Gamma_1, \widetilde{x}{:}\widetilde{S} \vdash_{\mathtt{ST}} P|\!)[\widetilde{z}/\widetilde{x}]\big] & & \Gamma = \Gamma_1 \circ \Gamma_2 \ \wedge \Gamma_1, \widetilde{x} : \widetilde{S} \vdash_{\mathtt{ST}} P \\
\qquad \parallel_k \{\!|\Gamma_1|\!\} \mid \mathscr{C}_{\widetilde{z}:\widetilde{V}}\big[(\!|\Gamma_2, \widetilde{y}{:}\widetilde{V} \vdash_{\mathtt{ST}} Q|\!)[\widetilde{z}/\widetilde{y}]\big] & & \Gamma_2, \widetilde{y} : \widetilde{V} \vdash_{\mathtt{ST}} Q \wedge V_i = \overline{S_i}
\end{array}
$$

We illustrate the procedure in [9]. Notice that the rewriting procedure given in Definition 5.7 satisfies the compositionality criteria given in [11]. In particular, it is easy to see that the rewriting of a composition of terms is defined in terms of the rewriting of the constituent subterms. Indeed, e.g., $(\!|\Gamma_1 \circ \Gamma_2 \vdash_{\mathtt{ST}} (\nu xy : S)(P \mid Q)|\!)$ depends on a context including both $(\!|\Gamma_1, x : S \vdash_{\mathtt{ST}} P|\!)$ and $(\!|\Gamma_2, y : \overline{S} \vdash_{\mathtt{ST}} Q|\!)$.

We present two important results about our rewriting procedure. First, we show it is type preserving:

Theorem 5.8 (Rewriting is Type Preserving)**.** *Let $(\Gamma \vdash_{\mathtt{ST}} P) \in \mathscr{K}_n$. Then, $(\!|\Gamma \vdash_{\mathtt{ST}} P|\!) \vdash_{\mathtt{CH}} [\![\Gamma]\!]_{\mathtt{c}}$.*

Notice that the inverse of the previous theorem is trivial by following the definition of typed encoding. Theorem 5.8 is meaningful, for it says that the type interface of a process (i.e., the set of sessions implemented in it) is not modified by the rewriting procedure. That is, the procedure modifies the process structure by closely following the causality relations described by (session) types. Notice that causality relations present in processes, but not described at the level of types, may be removed.

The rewriting procedure also satisfies an operational correspondence result. Let us write $\Gamma \vdash_{\mathtt{ST}} P_1, P_2$ whenever both $\Gamma \vdash_{\mathtt{ST}} P_1$ and $\Gamma \vdash_{\mathtt{ST}} P_2$ hold. We have the following auxiliary definition:

Definition 5.9. *Let P, P' be such that $\Gamma \vdash_{\mathtt{ST}} P, P'$. Then, we write $P \div P'$ if and only if $P = \mathtt{C}[Q]$ and $P' = \mathtt{C}[Q']$, for some context $\mathtt{C}$, and there is Γ' such that $\Gamma' \vdash_{\mathtt{ST}} Q, Q'$.*

Theorem 5.10 (Operational Correspondence)**.** *Let $P \in \mathscr{K}_n$ such that $\Gamma \vdash_{\mathtt{ST}} P$ for some Γ. Then we have:*

I) If $P \to P'$ then there exist Q, Q' s.t. (i) $(\!|\Gamma \vdash_{\mathtt{ST}} P|\!) \to^{\mathtt{inx}} \to^* \equiv Q$; (ii) $Q \div Q'$; (iii) $(\!|\Gamma \vdash_{\mathtt{ST}} P'|\!) \to^{\mathtt{inx}} Q'$.

II) If $(\!|\Gamma \vdash_{\mathtt{ST}} P|\!) \to^{\mathtt{inx}} \to^* \equiv Q$ then there exists P' s.t. $P \to P'$ and $Q \div (\!|\Gamma \vdash_{\mathtt{ST}} P'|\!)$.

6 Concluding Remarks

We have presented a formal comparison of fundamentally distinct type systems for deadlock-free, session typed processes. To the best of our knowledge, ours is the first work to establish precise relationships of this kind. Indeed, prior comparisons between type systems for deadlock freedom are informal, given in terms of representative examples typable in one type system but not in some other.

An immediate difficulty in giving a unified account of different typed frameworks for deadlock freedom is the variety of process languages, type structures, and typing rules that define each framework. Indeed, our comparisons involve: the framework of session processes put forward by Vasconcelos [19]; the interpretation of linear logic propositions as session types by Caires [1]; the π-calculus with usage types defined by Kobayashi in [13]. Finding some common ground for comparing these three frameworks is not trivial—several translations/transformations were required in our developments to account for numerous syntactic differences. We made an effort to follow the exact definitions in each framework. Overall, we believe that we managed to concentrate on essential semantic features of two salient classes of deadlock-free session processes, noted $\mathscr{L}$ and $\mathscr{K}$.

Our main contribution is identifying the *degree of sharing* as a subtle, important issue that underlies both session typing and deadlock freedom. We propose a simple definition of the degree of sharing: in essence, it arises via an explicit premise for the typing rule for parallel composition in the type system in [13]. The degree of sharing is shown to effectively induce a strict hierarchy of deadlock-free session processes in $\mathscr{K}$, as resulting from the approach of [8]. We showed that the most elementary (and non trivial) member of this hierarchy precisely corresponds to $\mathscr{L}$–arguably the most canonical class of session typed processes known to date. Furthermore, by exhibiting an intuitive rewriting procedure of processes in $\mathscr{K}$ into processes in $\mathscr{L}$, we demonstrated that the degree of sharing is a subtle criteria for distinguishing deadlock-free processes. As such, even if our technical developments are technically simple, in our view they substantially clarify our understanding of type systems for liveness properties (such as deadlock freedom) in the context of π-calculus processes.

As future work, we would like to obtain *semantic characterizations* of the degree of sharing, in the form of, e.g., preorders on typed processes that distinguish when one process "is more parallel" than another. We plan also to extend our formal relationships to cover typing disciplines with *infinite behavior*. We notice that the approach of [8] extends to recursive behavior [7] and that infinite (yet non divergent) behavior has been incorporated into logic-based session types [18]. Finally, we plan to explore whether the rewriting procedure given in § 5 could be adapted into a *deadlock resolution* procedure.

Acknowledgements. We are grateful to Luís Caires, Simon J. Gay, and the anonymous reviewers for their valuable comments and suggestions. This work was partially supported by the EU COST Action IC1201 (Behavioural Types for Reliable Large-Scale Software Systems). Dardha is supported by the UK EPSRC project EP/K034413/1 (From Data Types to Session Types: A Basis for Concurrency and Distribution). Pérez is also affiliated to NOVA Laboratory for Computer Science and Informatics, Universidade Nova de Lisboa, Portugal.

References

[1] Luís Caires (2014): *Types and Logic, Concurrency and Non-Determinism*. In *Essays for the Luca Cardelli Fest* - Microsoft Research Technical Report MSR-TR-2014-104. Available at `http://research.microsoft.com/apps/pubs/default.aspx?id=226237`.

[2] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *Proc. of CONCUR 2010*, *LNCS* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[3] Luís Caires, Frank Pfenning & Bernardo Toninho (2014): *Linear Logic Propositions as Session Types*. *MSCS*, doi:10.1017/S0960129514000218.

[4] Marco Carbone, Ornela Dardha & Fabrizio Montesi (2014): *Progress as Compositional Lock-Freedom*. In: *COORDINATION*, *LNCS* 8459, Springer, pp. 49–64, doi:10.1007/978-3-662-43376-8_4.

[5] Marco Carbone & Søren Debois (2010): *A Graphical Approach to Progress for Structured Communication in Web Services*. In: *Proc. of ICE 2010, Amsterdam, The Netherlands, 10th of June 2010.*, *EPTCS* 38, pp. 13–27, doi:10.4204/EPTCS.38.4.

[6] Gerardo Costa & Colin Stirling (1987): *Weak and Strong Fairness in CCS*. *Inf. Comput.* 73(3), pp. 207–244, doi:10.1016/0890-5401(87)90013-7.

[7] Ornela Dardha (2014): *Recursive Session Types Revisited*. In: *Proceedings Third Workshop on Behavioural Types, BEAT 2014, Rome, Italy, 1st September 2014.*, *EPTCS* 162, pp. 27–34, doi:10.4204/EPTCS.162.4.

[8] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2012): *Session types revisited*. In: *Proc. of PPDP'12*, ACM, pp. 139–150, doi:10.1145/2370776.2370794.

[9] Ornela Dardha & Jorge A. Pérez (2015): *Full version of this paper*. Technical Report. Available at `http://www.jorgeaperez.net`.

[10] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro & Nobuko Yoshida (2008): *On Progress for Structured Communications*. In: *Trustworthy Global Computing*, *LNCS* 4912, Springer, pp. 257–275, doi:10.1007/978-3-540-78663-4_18.

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

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

[13] Naoki Kobayashi (2002): *A Type System for Lock-Free Processes*. *Inf. Comput.* 177(2), pp. 122–159, doi:10.1006/inco.2002.3171.

[14] Naoki Kobayashi (2003): *Type Systems for Concurrent Programs*. In: *Formal Methods at the Crossroads*, *LNCS* 2757, Springer, pp. 439–453, doi:10.1007/978-3-540-40007-3_26.

[15] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In: *Proc. of CONCUR 2006*, *LNCS* 4137, Springer, pp. 233–247, doi:10.1007/11817949_16.

[16] Luca Padovani (2013): *From Lock Freedom to Progress Using Session Types*. In: *Proceedings of PLACES 2013, Rome, Italy, 23rd March 2013.*, *EPTCS* 137, pp. 3–19, doi:10.4204/EPTCS.137.2.

[17] Benjamin C. Pierce (2002): *Types and programming languages*. MIT Press, MA, USA.

[18] Bernardo Toninho, Luís Caires & Frank Pfenning (2014): *Corecursion and Non-divergence in Session-Typed Processes*. In: *Proc. of TGC 2014*, *LNCS* 8902, Springer, pp. 159–175, doi:10.1007/978-3-662-45917-1_11.

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

[20] Hugo Torres Vieira & Vasco Thudichum Vasconcelos (2013): *Typing Progress in Communication-Centred Systems*. In: *COORDINATION*, *LNCS* 7890, Springer, pp. 236–250, doi:10.1007/978-3-642-38493-6_17.

[21] Philip Wadler (2012): *Propositions as sessions*. In: *Proc. of ICFP'12*, pp. 273–286, doi:10.1145/2364527.2364568.