A Universal Session Type for Untyped Asynchronous Communication

Stephanie Balzer
Carnegie Mellon University, USA

Frank Pfenning
Carnegie Mellon University, USA

Bernardo Toninho
NOVA LINCS, Universidade Nova de Lisboa, Portugal

Abstract

In the simply-typed λ-calculus we can recover the full range of expressiveness of the untyped λ-calculus solely by adding a single recursive type $\mathcal{U} = \mathcal{U} \to \mathcal{U}$. In contrast, in the session-typed π-calculus, recursion alone is insufficient to recover the untyped π-calculus, primarily due to linearity: each channel just has two unique endpoints. In this paper, we show that shared channels with a corresponding sharing semantics (based on the language $\mathsf{SILL_S}$ developed in prior work) are enough to embed the untyped asynchronous π-calculus via a universal shared session type $\mathcal{U}_\mathsf{S}$. We show that our encoding of the asynchronous π-calculus satisfies operational correspondence and preserves observable actions (i.e., processes are weakly bisimilar to their encoding). Moreover, we clarify the expressiveness of $\mathsf{SILL_S}$ by developing an operationally correct encoding of $\mathsf{SILL_S}$ in the asynchronous π-calculus.

2012 ACM Subject Classification Theory of computation → Models of computation → Concurrency → Process calculi; Theory of computation → Logic → Linear logic

Keywords and phrases Session types, sharing, π-calculus, bisimulation

1 Introduction

Session types [20, 22, 23] prescribe the protocols of message exchange between processes that interact along channels. The recent discovery of a Curry-Howard isomorphism between *linear logic* and the *session-typed π-calculus* [8, 9, 42, 38] has given message-passing concurrency a firm logical foundation. Programming languages [40, 19] building on this isomorphism not only guarantee *session fidelity* (i.e., protocol compliance) but also a form of *global progress*, since the process graph forms a tree and is acyclic by construction.

While the linear logic session framework allows for persistent servers through the exponential modality (i.e., replicated sessions that may be used an arbitrary number of times), it enforces a strict separation between server instances by means of a copying semantics [8, 42]. For instance, interactions between a client and a server cannot affect future client-server interactions. Thus, this session discipline fundamentally excludes programming scenarios that require *sharing* of server resources such as shared databases or shared output devices. This observation triggered the realization that linear session-typed calculi lag behind the untyped asynchronous π-calculus in expressiveness and the question of whether the full expressiveness of the untyped asynchronous π-calculus could be recovered in such a logical setting [42].

In this paper, we answer this question *positively*. In prior work we have introduced *manifest sharing* [3], a modal-typing discipline that orchestrates the coexistence of linear *and* shared channels while maintaining session fidelity, at the expense of generalized deadlock-freedom. In this work we show that manifest sharing recovers the expressiveness of the untyped asynchronous π-calculus. Given our language $\mathsf{SILL_S}$ [3, 4] that supports manifest sharing, we provide an encoding of the untyped asynchronous π-calculus into $\mathsf{SILL_S}$, showing that our encoding satisfies *operational correspondence* and that π-calculus processes are *weakly bisimilar* to their $\mathsf{SILL_S}$ encodings. To clarify the expressiveness of $\mathsf{SILL_S}$, we moreover develop an operationally correct encoding in the other direction, which embeds $\mathsf{SILL_S}$ into the asynchronous (polyadic) π-calculus.

Key to our encoding of the untyped asynchronous π-calculus into $\mathsf{SILL_S}$ is the representation of a π-calculus channel as a recursive *shared* session type $\mathcal{U}_{\mathsf{S}}$, reminiscent of the encoding of the untyped λ-calculus into the simply-typed λ-calculus via the type $\mathcal{U} = \mathcal{U} \to \mathcal{U}$. While the addition of a single recursive type is sufficient to recover the expressiveness of the untyped λ-calculus in the simply-typed λ-calculus, our result reveals that *both* shared and recursive session types are necessary to achieve the analogous result in the session-typed π-calculus.

The contributions of this paper are:

- A proof that our encoding of the untyped asynchronous π-calculus into $\mathsf{SILL_S}$ is operationally sound and complete and preserves observable actions (i.e., processes are weakly bisimilar to their encoding);[1]
- A formulation of a weak bisimulation between a labelled transition system for the asynchronous π-calculus and a multiset rewriting system for *closed terms* of $\mathsf{SILL_S}$;
- Evidence of the instrumental role shared channels take in the expressiveness of session-typed process calculi;
- An operationally correct encoding of $\mathsf{SILL_S}$ into the untyped asynchronous polyadic π-calculus.

Paper Structure. Section 2 provides the necessary background on $\mathsf{SILL_S}$. Section 3 introduces the encoding of the untyped asynchronous π-calculus into $\mathsf{SILL_S}$ and states and proves operational and observational correspondence. Section 4 develops an operationally correct encoding of $\mathsf{SILL_S}$ into the untyped asynchronous polyadic π-calculus. Section 5 summarizes related work, and Section 6 concludes the paper. The Appendix lists proofs.

2 Manifest Sharing with Session Types

In this section, we provide an introduction to *manifest sharing* [3] and the programming language $\mathsf{SILL_S}$ [3, 4], to the extent necessary for the development in this paper. *Session types* [20, 22, 23, 8, 40, 9, 42, 38] prescribe the protocols of message exchange between processes that interact along channels. For example, the recursive linear session type

$$\mathsf{queue}\ A = \&\{\mathsf{enq} : A \multimap \mathsf{queue}\ A,\ \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : A \otimes \mathsf{queue}\ A\}\}$$

defines the protocol of how to interact with a provider of a queue data structure that contains elements of some variable type A. In a session-typed interpretation of intuitionistic linear logic, session types are expressed from the point of view of the providing process, with the channel along which the process provides the session behavior being defined by the session type. This choice avoids the explicit dualization of a session type present in the original presentations of session types [20, 22] and those based on classical linear logic [42]. We adopt an *equi-recursive* [11] interpretation for recursive session types, silently equating a recursive session type with its unfolding and requiring types to be *contractive* [16].

Table 1 provides an overview of $\mathsf{SILL_S}$'s session types and their operational reading. For each type constructor, Table 1 lists the points of view of the *provider* and *client* of the given

[1] A preliminary version of our encoding of the untyped asynchronous π-calculus into $\mathsf{SILL_S}$ has been published in [3] for illustration purposes, but without proof.

Session type current	**cont**	**Process term current**	**cont**	**Description**
$c : \oplus\{\overline{l : A_{\mathsf{L}}}\}$	$c : A_{\mathsf{L}_h}$	$c.l_h\ ; P$	P	provider sends label l_h along c
		$\mathsf{case}\ c\ \mathsf{of}\ \overline{l \Rightarrow Q}$	Q_h	client receives label l_h along c
$c : \&\{\overline{l : A_{\mathsf{L}}}\}$	$c : A_{\mathsf{L}_h}$	$\mathsf{case}\ c\ \mathsf{of}\ \overline{l \Rightarrow P}$	P_h	provider receives label l_h along c
		$c.l_h\ ; Q$	Q	client sends label l_h along c
$c : A_{\mathsf{L}} \otimes B_{\mathsf{L}}$	$c : B_{\mathsf{L}}$	$\mathsf{send}\ c\ d\ ; P$	P	provider sends channel $d : A_{\mathsf{L}}$ along c
		$y_{\mathsf{L}} \leftarrow \mathsf{recv}\ c\ ; Q_{y_{\mathsf{L}}}$	$[d/y_{\mathsf{L}}]\, Q_{y_{\mathsf{L}}}$	client receives channel $d : A_{\mathsf{L}}$ along c
$c : A_{\mathsf{L}} \multimap B_{\mathsf{L}}$	$c : B_{\mathsf{L}}$	$y_{\mathsf{L}} \leftarrow \mathsf{recv}\ c\ ; P_{y_{\mathsf{L}}}$	$[d/y_{\mathsf{L}}]\, P_{y_{\mathsf{L}}}$	provider receives channel $d : A_{\mathsf{L}}$ along c
		$\mathsf{send}\ c\ d\ ; Q$	Q	client sends channel $d : A_{\mathsf{L}}$ along c
$c_{\mathsf{L}} : \Pi x{:}A_{\mathsf{S}}.B_{\mathsf{L}}$	$c_{\mathsf{L}} : B_{\mathsf{L}}$	$\mathsf{send}\ c_{\mathsf{L}}\ d_{\mathsf{S}}\ ; P$	P	provider sends channel $d_{\mathsf{S}} : A_{\mathsf{S}}$ along c_{L}
		$y_{\mathsf{S}} \leftarrow \mathsf{recv}\ c_{\mathsf{L}}\ ; Q_{y_{\mathsf{S}}}$	$[d_{\mathsf{S}}/y_{\mathsf{S}}]\, Q_{y_{\mathsf{S}}}$	client receives channel $d_{\mathsf{S}} : A_{\mathsf{S}}$ along c_{L}
$c_{\mathsf{L}} : \exists x{:}A_{\mathsf{S}}.B_{\mathsf{L}}$	$c_{\mathsf{L}} : B_{\mathsf{L}}$	$y_{\mathsf{S}} \leftarrow \mathsf{recv}\ c_{\mathsf{L}}\ ; P_{y_{\mathsf{S}}}$	$[d_{\mathsf{S}}/y_{\mathsf{S}}]\, P_{y_{\mathsf{S}}}$	provider receives channel $d_{\mathsf{S}} : A_{\mathsf{S}}$ along c_{L}
		$\mathsf{send}\ c_{\mathsf{L}}\ d_{\mathsf{S}}\ ; Q$	Q	client sends channel $d_{\mathsf{S}} : A_{\mathsf{S}}$ along c_{L}
$c : \mathbf{1}$	-	$\mathsf{close}\ c$	-	provider sends "end" along c
		$\mathsf{wait}\ c\ ; Q$	Q	provider receives "end" along c
$c_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}$	$c_{\mathsf{S}} : A_{\mathsf{S}}$	$c_{\mathsf{S}} \leftarrow \mathsf{detach}\ c_{\mathsf{L}}\ ; P_{x_{\mathsf{S}}}$	$[c_{\mathsf{S}}/x_{\mathsf{S}}]\, P_{x_{\mathsf{S}}}$	provider sends "detach c_{S}" along c_{L}
		$x_{\mathsf{S}} \leftarrow \mathsf{release}\ c_{\mathsf{L}}\ ; Q_{x_{\mathsf{S}}}$	$[c_{\mathsf{S}}/x_{\mathsf{S}}]\, Q_{x_{\mathsf{S}}}$	client receives "detach c_{S}" along c_{L}
$c_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$	$c_{\mathsf{L}} : A_{\mathsf{L}}$	$c_{\mathsf{L}} \leftarrow \mathsf{acquire}\ c_{\mathsf{S}}\ ; Q_{x_{\mathsf{L}}}$	$[c_{\mathsf{L}}/x_{\mathsf{L}}]\, Q_{x_{\mathsf{L}}}$	client sends "acquire c_{L}" along c_{S}
		$x_{\mathsf{L}} \leftarrow \mathsf{accept}\ c_{\mathsf{S}}\ ; P_{x_{\mathsf{L}}}$	$[c_{\mathsf{L}}/x_{\mathsf{L}}]\, P_{x_{\mathsf{L}}}$	provider receives "acquire c_{L}" along c_{S}

Table 1 Overview of session types in $\mathsf{SILL}_{\mathsf{S}}$ together with their operational meaning.

type, in the first and second lines, respectively. For each connective, its session type before the exchange (**Session type current**) and after the exchange (**Session type cont(inuation)**) is given. Likewise, the implementing process term is indicated before the exchange (**Process term current**) and after the exchange (**Process term continuation**). Table 1 shows that the process terms of a provider and a client for a connective come in matching pairs. Both participants' view of the session changes consistently.

For the linear session type $\mathsf{queue}\ A$ specified above, we have the following protocol: a process providing a service of type $\mathsf{queue}\ A$ gives a client the choice to either enqueue (enq) or dequeue (deq) an element of type A. Upon receipt of the label enq, the providing process expects to receive a channel of type A to be enqueued and recurs. Upon receipt of the label deq, the providing process either indicates that the queue is empty (none), in which case it terminates, or that there is a channel stored in the queue (some), in which case it dequeues this element, sends it to the client, and recurs.

Linearity restricts session type $\mathsf{queue}\ A$ to a single client. If we want the queue to be used in a classical consumer-producer scenario, where we have multiple producers and consumers accessing the queue, we can use the following *shared session type* instead:

$$\begin{aligned}\mathbf{queue}\ \mathbf{A}_{\mathsf{S}} = \uparrow^{\mathsf{S}}_{\mathsf{L}}\&\{&\mathsf{enq} : \Pi x : \mathbf{A}_{\mathsf{S}}.\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathbf{queue}\ \mathbf{A}_{\mathsf{S}},\\ &\mathsf{deq} : \oplus\{\mathsf{none} : \downarrow^{\mathsf{S}}_{\mathsf{L}}\mathbf{queue}\ \mathbf{A}_{\mathsf{S}},\ \mathsf{some} : \exists x : \mathbf{A}_{\mathsf{S}}.\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathbf{queue}\ \mathbf{A}_{\mathsf{S}}\}\}\end{aligned}$$

For ease of reading, we typeset shared session types and channels in programs in red and **bold** font as opposed to linear session types and channels, which we typeset in black and regular font. Session type $\mathsf{queue}\ A_{\mathsf{S}}$ now describes the session offered by a shared process. Since a shared process can have multiple clients that refer to the process by a *shared channel*, state-altering communication with a shared process must only happen once exclusive access to the process has been obtained. Otherwise, session fidelity would be endangered. To this end, $\mathsf{SILL}_{\mathsf{S}}$ imposes an *acquire-release* discipline on shared processes, where an acquire yields exclusive access to a shared process, if the process is available, and a release relinquishes exclusive access. As a result, processes can alternate between linear and shared, where a successful acquire of a shared process turns the process into a linear one, and conversely, a release of a linear process turns the process into a shared one.

A potential producer process can now interact with a process that implements session type $\mathsf{queue}\ A_{\mathsf{S}}$ according to Figure 1, assuming that q is of type $\mathsf{queue}\ A_{\mathsf{S}}$ and x is of type A_{S}. The statement $q' \leftarrow \mathsf{acquire}\ q$, yields, if successful, the queue's linear channel q' along which the producer process can enqueue the element. The statement $q \leftarrow \mathsf{release}\ q'$ releases the now linear queue process providing along q', giving turn to another producer or consumer process, and yields the queue's shared channel q. As indicated by Table 1, there exist the dual notions of an *accept* and *detach* for an acquire and release, respectively, denoting the matching statements by a provider.

$q' \leftarrow \mathsf{acquire}\ \mathbf{q}$;
$q'.\mathsf{enq}$;
$\mathsf{send}\ q'\ \mathbf{x}$;
$\mathbf{q} \leftarrow \mathsf{release}\ q'$

Figure 1

A key contribution of manifest sharing is not only to support acquire-release as a programming primitive but also to make it manifest in the type system. Generalizing the idea of type *stratification* [35, 6, 36], session types are stratified into a linear and shared layer with two *adjoint modalities* going back and forth between them:

$$\begin{array}{lll} A_{\mathsf{S}} & \triangleq & \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}} \\ A_{\mathsf{L}}, B_{\mathsf{L}} & \triangleq & A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \mathbf{1} \mid \oplus\{\overline{l : A_{\mathsf{L}}}\} \mid \exists x : A_{\mathsf{S}}.B_{\mathsf{L}} \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid \Pi x : A_{\mathsf{S}}.B_{\mathsf{L}} \mid \&\{\overline{l : A_{\mathsf{L}}}\} \mid \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}} \end{array}$$

The modal operator $\downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}}$ shifting *down* from the shared to the linear layer is then interpreted as a *release* (and, dually, detach) and the operator $\uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}}$ shifting *up* from the linear to the shared layer as an *acquire* (and, dually, accept). As a result, we obtain a type system where a session type dictates any form of synchronization, including the acquisition and release of a shared process.

Returning to the shared session type $\mathsf{queue}\ A_{\mathsf{S}}$ defined above, we can see that any exchange of labels or channels with the queue is now guarded by a preceding acquire, and that the queue must be released before it recurs. The shared session type further deviates from its linear version in that it contains shared elements, as the entire queue is shared, and by recurring in the empty case of a dequeueing request, as there are now multiple clients.

We briefly discuss the typing and the dynamics of acquire-release. The typing and the dynamics of the residual linear connectives are standard. As is usual for an intuitionistic interpretation, each connective gives rise to a left and a right rule, denoting the use and provision, respectively, of a session of the given type:

$$\text{(T-}\uparrow_{\mathsf{L}}^{\mathsf{S}}\text{L)}\ \frac{\Gamma, x_{\mathsf{S}} : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}};\ \Delta, x_{\mathsf{L}} : A_{\mathsf{L}} \vdash_{\Sigma} Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma, x_{\mathsf{S}} : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}};\ \Delta \vdash_{\Sigma} x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ x_{\mathsf{S}}\ ; Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})} \qquad \text{(T-}\uparrow_{\mathsf{L}}^{\mathsf{S}}\text{R)}\ \frac{\Gamma;\ \cdot \vdash_{\Sigma} P_{x_{\mathsf{L}}} :: (x_{\mathsf{L}} : A_{\mathsf{L}})}{\Gamma \vdash_{\Sigma} x_{\mathsf{L}} \leftarrow \mathsf{accept}\ x_{\mathsf{S}}\ ; P_{x_{\mathsf{L}}} :: (x_{\mathsf{S}} : \uparrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{L}})}$$

$$\text{(T-}\downarrow_{\mathsf{L}}^{\mathsf{S}}\text{L)}\ \frac{\Gamma, x_{\mathsf{S}} : A_{\mathsf{S}};\ \Delta \vdash_{\Sigma} Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma;\ \Delta, x_{\mathsf{L}} : \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}} \vdash_{\Sigma} x_{\mathsf{S}} \leftarrow \mathsf{release}\ x_{\mathsf{L}}\ ; Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})} \qquad \text{(T-}\downarrow_{\mathsf{L}}^{\mathsf{S}}\text{R)}\ \frac{\Gamma \vdash_{\Sigma} P_{x_{\mathsf{S}}} :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Gamma;\ \cdot \vdash_{\Sigma} x_{\mathsf{S}} \leftarrow \mathsf{detach}\ x_{\mathsf{L}}\ ; P_{x_{\mathsf{S}}} :: (x_{\mathsf{L}} : \downarrow_{\mathsf{L}}^{\mathsf{S}} A_{\mathsf{S}})}$$

The typing judgments $\Gamma \vdash_{\Sigma} P :: (x_{\mathsf{S}} : A_{\mathsf{S}})$ and $\Gamma;\ \Delta \vdash_{\Sigma} P :: (x_{\mathsf{L}} : A_{\mathsf{L}})$ indicate that process P provides a service of session type A along channel x, given the typing of services provided by processes along the channels in typing contexts Γ (and Δ). Γ and Δ consist of hypotheses on the typing of shared and linear channels, respectively, where Γ is a structural and Δ a linear context. To allow for recursive process definitions, the typing judgment depends on a signature Σ that is populated with all process definitions prior to type-checking. The adjoint formulation forbids a shared process from depending on linear channels [3, 35]. Thus, when a shared session accepts an acquire and shifts to linear, it starts with an empty linear context.

Operationally, the dynamics of $\mathsf{SILL_S}$ is captured by *multiset rewriting rules* [10], which denote computation in terms of state transitions between configurations of processes. Multiset rewriting rules are local in that they only mention the parts of a configuration they rewrite. For acquire-release we have the rules of Figure 2.

$\mathsf{proc}(c_{\mathsf{L}},\, x_{\mathsf{L}} \leftarrow \mathsf{acquire}\; a_{\mathsf{S}}\,;\, Q_{x_{\mathsf{L}}}),\; \mathsf{proc}(a_{\mathsf{S}},\, x_{\mathsf{L}} \leftarrow \mathsf{accept}\; a_{\mathsf{S}}\,;\, P_{x_{\mathsf{L}}})$
$\longrightarrow\; \mathsf{proc}(c_{\mathsf{L}},\, [a_{\mathsf{L}}/x_{\mathsf{L}}]\, Q_{x_{\mathsf{L}}}),\; \mathsf{proc}(a_{\mathsf{L}},\, [a_{\mathsf{L}}/x_{\mathsf{L}}]\, P_{x_{\mathsf{L}}}),\; \mathsf{unavail}(a_{\mathsf{S}})$

$\mathsf{proc}(c_{\mathsf{L}},\, x_{\mathsf{S}} \leftarrow \mathsf{release}\; a_{\mathsf{L}}\,;\, Q_{x_{\mathsf{S}}}),\; \mathsf{proc}(a_{\mathsf{L}},\, x_{\mathsf{S}} \leftarrow \mathsf{detach}\; a_{\mathsf{L}}\,;\, P_{x_{\mathsf{S}}}),\; \mathsf{unavail}(a_{\mathsf{S}})$
$\longrightarrow\; \mathsf{proc}(c_{\mathsf{L}},\, [a_{\mathsf{S}}/x_{\mathsf{S}}]\, Q_{x_{\mathsf{S}}}),\; \mathsf{proc}(a_{\mathsf{S}},\, [a_{\mathsf{S}}/x_{\mathsf{S}}]\, P_{x_{\mathsf{S}}})$

Figure 2

Configuration states are defined by the predicates $\mathsf{proc}(c_m,\, P)$ and $\mathsf{unavail}(a_{\mathsf{S}})$. The former denotes a process with process term P providing along channel c_m, the latter a placeholder for a shared process providing along channel a_{S} that is currently not available. The above rule exploits the invariant that a process' providing channel a can come at one of two modes, a linear one, a_{L}, and a shared one, a_{S}. While the process is linear, it provides along a_{L}, while it is shared, along a_{S}. When a process shifts between modes, it switches between the two modes of its offering channel. This channel at the appropriate mode is substituted for the variables occurring in process terms.

3 Recovering the Untyped Asynchronous π-calculus in $\mathsf{SILL_S}$

We now detail our encoding of the asynchronous π-calculus into $\mathsf{SILL_S}$, show that it satisfies *operational correspondence* and that processes are *weakly bisimilar* to their $\mathsf{SILL_S}$ encodings.

3.1 Encoding the Untyped Asynchronous π-calculus in $\mathsf{SILL_S}$

The essence of linear session-typed process calculi — treating channels as stateful resources — is fundamental in facilitating reasoning about session-typed programs and to guarantee strong properties, such as session fidelity and possibly deadlock-freedom. However, where channels in linear session-typed process calculi connect exactly one *sending* process with one *receiving* process, in the untyped π-calculus they may connect *multiple* sending and receiving processes, giving rise to *non-determinism*. For example, the π-calculus process $c(x).P \mid \overline{c}\langle a\rangle \mid c(y).Q$, made up of three parallel components, where the first and third seek to input along channel c and the second outputs the name a along c, may reduce to either $[a/x]\,P \mid c(y).Q$ or $c(x).P \mid [a/y]\,Q$.

In purely linear session-typed process calculi, on the other hand, message exchange is completely *deterministic*, even in the presence of replicated or persistent sessions (this argument is made precise through a typed contextual equivalence for intuitionistic linear logic sessions in [34]). The addition of sharing to session-typed calculi — and with it non-determinism — suggests that it should now be possible to faithfully encode the untyped π-calculus. In previous work we have postulated this conjecture by providing an encoding of the untyped asynchronous π-calculus into $\mathsf{SILL_S}$ [3], without any further proof. We now refine the encoding and prove it operationally and behaviorally correct.

The basic idea of our encoding is to represent a π-calculus *process* by a *linear* $\mathsf{SILL_S}$ process and a π-calculus *channel* by a *shared* $\mathsf{SILL_S}$ process. Reminiscent of the encoding of the untyped λ-calculus into the typed λ-calculus, we type π-calculus channels with a *universal recursive shared* session type $\mathcal{U}_{\mathsf{S}}$:

$$\mathcal{U}_{\mathsf{S}} = {\uparrow}^{\mathsf{S}}_{\mathsf{L}}\,\&\{\mathsf{ins} : \Pi x{:}\mathcal{U}_{\mathsf{S}}.\,{\downarrow}^{\mathsf{S}}_{\mathsf{L}}\,\mathcal{U}_{\mathsf{S}},\; \mathsf{del} : \oplus\{\mathsf{none} : {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\,\mathcal{U}_{\mathsf{S}},\; \mathsf{some} : \exists x{:}\mathcal{U}_{\mathsf{S}}.\,{\downarrow}^{\mathsf{S}}_{\mathsf{L}}\,\mathcal{U}_{\mathsf{S}}\}\}$$

Similar to the type queue A_{S} of Section 2, the type $\mathcal{U}_{\mathsf{S}}$ represents a buffer that stores elements, but with the elements being of type $\mathcal{U}_{\mathsf{S}}$ themselves and *without* maintaining any order. Figure 3 shows the processes *empty* and *elem* that implement session type $\mathcal{U}_{\mathsf{S}}$. In SILL$_{\mathsf{S}}$, we declare the type of a defined process X with $X : \{A \leftarrow A_1, \ldots, A_n\}$, indicating that the process provides a service of type A, using channels of type $A_1, \ldots, A_n$. The definition of the process is then given by $x \leftarrow X \leftarrow y_1, \ldots, y_n = P$, where P is the body of the process with occurrences of channels $y_1 : A_1, \ldots, y_n : A_n$. A new process X providing along channel x is spawned with an expression of the form $x \leftarrow X \leftarrow y_1, \ldots, y_n \ ; Q_x$, where Q_x is the continuation binding x. We refer to Table 1 for the meaning of the process terms.

```
empty : {U_S}
c ← empty =
  c′ ← accept c ;
  case c′ of
  | ins → x ← recv c′ ;
          c ← detach c′ ;
          e ← empty ;
          c ← elem ← x, e
  | del → c′.none ;
          c ← detach c′ ;
          c ← empty

elem : {U_S ← U_S, U_S}
c ← elem ← x, d =
  c′ ← accept c ;
  case c′ of
  | ins → y ← recv c′ ;
          c ← detach c′ ;
          e ← elem ← x, d ;
          c ← elem ← y, e
  | del → c′.some ;
          c′ ← nd_pick ← x, d

nd_pick : {∃x:U_S. ↓^S_L U_S ← U_S, U_S}
c′ ← nd_pick ← x, d =
  ndc ← nd_choice ;
  case ndc of
  | yes → send c′ x ;
          c ← detach c′ ;
          wait ndc ;
          fwd c d
  | no → d′ ← acquire d ;
         d′.del ;
         case d′ of
         | none → d ← release d′ ;
                  send c′ x ;
                  c ← detach c′ ;
                  wait ndc ;
                  fwd c d
         | some → y ← recv d′ ;
                  d ← release d′ ;
                  send c′ y ;
                  c ← detach c′ ;
                  wait ndc ;
                  c ← elem ← x, d

nd_choice : {⊕{yes : 1, no : 1}}

coin = ↑^S_L ⊕ {head : ↓^S_L coin,
                tail : ↓^S_L coin}
coin_head : {coin}
coin_tail : {coin}
coin_flipper : {1 ← coin}
```

Figure 3 Processes *empty* and *elem* implementing a π-calculus channel with auxiliary processes (see Appendix A.1 for omitted process bodies).

The buffer is implemented as a sequence of *elem* processes, ending in an *empty* process. The recursive process *elem* provides a buffer sequence along channel c and uses a channel $x : \mathcal{U}_{\mathsf{S}}$ (the buffer element at the current position in the sequence) as well as a channel $d : \mathcal{U}_{\mathsf{S}}$ (the next *elem* of the sequence). Process *empty*, on the other hand, provides an empty buffer sequence along channel c, without using any other channels. Both processes insert the received element at the head of the buffer sequence in the ins case, but handle the del case differently. Whereas process *empty* responds with label none, process *elem* responds with label some, followed by sending and deleting an *arbitrary* element from the buffer. Process *elem* achieves arbitrary deletion by recurring as process *nd_pick*. Process *nd_pick*, in turn, uses process *nd_choice* to nondeterministically choose between sending and deleting the element at the current position in the sequence (case yes) or, possibly recursively, propagating the deletion request to the next element in the sequence (case no). While linear session-typed calculi are deterministic, *non-determinism* arises in SILL$_{\mathsf{S}}$ from the acquisition of shared channels, since it is unknown which client among all those competing to acquire a shared process will succeed. Process *nd_choice* uses this fact and achieves non-determinism by reading a coin that it shares with process *coin_flipper*. Both processes then try to acquire the coin concurrently, which switches sides when read, with the result that the value read by *nd_choice* depends on the order in which the coin is acquired. The interested reader

$$
\begin{array}{lcl}
[\![0]\!] & = & \cdot \\
[\![\overline{c}\langle a\rangle]\!] & = & x \leftarrow \mathit{snd} \leftarrow \mathbf{c} \; ; \\
 & & \mathsf{send}\ x\ \mathbf{a} \; ; \\
 & & \mathsf{wait}\ x \; ; \cdot \\
[\![c(x).P]\!] & = & y \leftarrow \mathit{poll\_rcv} \leftarrow \mathbf{c} \; ; \\
 & & \mathbf{z} \leftarrow \mathsf{recv}\ y \; ; \\
 & & \mathsf{wait}\ y \; ; \\
 & & [\mathbf{z}/x]\,[\![P]\!] \\
[\![\nu x\, P]\!] & = & \mathbf{y} \leftarrow \mathit{empty} \; ; \\
 & & [\mathbf{y}/x]\,[\![P]\!] \\
[\![P_1 \mid P_2]\!] & = & \_ \leftarrow [\![P_1]\!] \; ; \\
 & & [\![P_2]\!] \\
[\![!P]\!] & = & \mathit{Rec}_{!P} \quad \text{where} \\
\mathit{Rec}_{!P} & = & \_ \leftarrow [\![P]\!] \; ; \\
 & & \mathit{Rec}_{!P}
\end{array}
$$

$$
\begin{array}{l}
\mathit{snd} : \{(\Pi x{:}\mathcal{U}_{\mathsf{S}}.\,\mathbf{1}) \leftarrow \mathcal{U}_{\mathsf{S}}\} \\
d \leftarrow \mathit{snd} \leftarrow \mathbf{c} = \\
\quad \mathbf{x} \leftarrow \mathsf{recv}\ d \; ; \\
\quad c' \leftarrow \mathsf{acquire}\ \mathbf{c} \; ; \\
\quad c'.\mathsf{ins} \; ; \\
\quad \mathsf{send}\ c'\ \mathbf{x} \; ; \\
\quad \mathbf{c} \leftarrow \mathsf{release}\ c' \; ; \\
\quad \mathsf{close}\ d
\end{array}
$$

$$
\begin{array}{l}
\mathit{poll\_rcv} : \{(\exists x{:}\mathcal{U}_{\mathsf{S}}.\,\mathbf{1}) \leftarrow \mathcal{U}_{\mathsf{S}}\} \\
d \leftarrow \mathit{poll\_rcv} \leftarrow \mathbf{c} = \\
\quad c' \leftarrow \mathsf{acquire}\ \mathbf{c} \; ; \\
\quad c'.\mathsf{del} \; ; \\
\quad \mathsf{case}\ c'\ \mathsf{of} \\
\quad \mid \mathsf{none} \rightarrow \mathbf{c} \leftarrow \mathsf{release}\ c' \; ; \\
\qquad\qquad\quad d \leftarrow \mathit{poll\_rcv} \leftarrow \mathbf{c} \\
\quad \mid \mathsf{some} \rightarrow \mathbf{x} \leftarrow \mathsf{recv}\ c' \; ; \\
\qquad\qquad\quad \mathbf{c} \leftarrow \mathsf{release}\ c' \; ; \\
\qquad\qquad\quad \mathsf{send}\ d\ \mathbf{x} \; ; \\
\qquad\qquad\quad \mathsf{close}\ d
\end{array}
$$

Figure 4 Translation of untyped asynchronous π-calculus processes into $\mathsf{SILL_S}$ and auxiliary processes *snd* and *poll_recv* ($\mathit{empty} : \{\mathcal{U}_{\mathsf{S}}\}$ is defined in Figure 3).

can find the omitted bodies of processes *nd_choice*, *coin_head*, *coin_tail*, and *coin_flipper* in Appendix A.1.

Given the buffer abstraction, encoded π-calculus processes in $\mathsf{SILL_S}$ simply amount to "producers" and "consumers" of shared channels of type $\mathcal{U}_{\mathsf{S}}$. Any such process can communicate along a π-calculus channel by acquiring the corresponding $\mathsf{SILL_S}$ channel of universal type. We are now ready to give the encoding of the untyped asynchronous monadic π-calculus [30, 37] into $\mathsf{SILL_S}$. The syntax of the asynchronous π-calculus is [5]:

$$P \quad \triangleq \quad \mathsf{0} \;\mid\; \overline{c}\langle a\rangle \;\mid\; c(x).P \;\mid\; \nu c\, P \;\mid\; P_1 \mid P_2 \;\mid\; !P$$

$\mathsf{0}$ denotes an inactive process. $\overline{c}\langle a\rangle$ represents an asynchronous send of channel a along channel c. $c(x).P$ amounts to a guarded input, where the channel received along c is bound to x in the continuation P. $\nu c\, P$ introduces a new channel c that is bound in P. $P_1 \mid P_2$ denotes parallel composition of P_1 and P_2, and $!P$ replication of P (i.e., an unbounded number of copies of P in parallel). We assume a standard reduction and labelled transition semantics, but where replication involves an explicit reduction (and τ transition) instead of expansion through structural congruence: $!P \longrightarrow P \mid !P$. Moreover, we enforce that structural congruence is only applied at the top-level of processes.

Our encoding, shown in Figure 4, yields for each π-calculus process P a corresponding linear process $[\![P]\!]$ in $\mathsf{SILL_S}$, satisfying the typing judgment: $\Gamma_{\mathcal{F}}; \Gamma_{\mathcal{B}}; \Gamma_{\mathcal{I}}; \cdot \vdash_\Sigma [\![P]\!] :: (\cdot)$. We use an empty succedent to denote that the process does not provide any session. Since all communication is going to happen along π-calculus channels, i.e., the shared $\mathsf{SILL_S}$ processes of type $\mathcal{U}_{\mathsf{S}}$, the linear $\mathsf{SILL_S}$ processes representing π-calculus processes merely become clients of those processes, without providing any behavior outright. In our earlier encoding [3], we have translated π-calculus processes into linear $\mathsf{SILL_S}$ processes of type $\mathbf{1}$, since the notion of a non-providing linear process is not present in $\mathsf{SILL_S}$. Our current encoding avoids the spurious exchange of wait messages required by type $\mathbf{1}$ and constitutes a return to the original interpretation of linear logic [8], where processes terminate silently. In the above typing judgment, we moreover subdivide the context Γ into three parts, to keep track of the *free* ($\Gamma_{\mathcal{F}}$) and *bound* ($\Gamma_{\mathcal{B}}$) π-calculus channels as well as of channels that are only used *internally* to the encoding ($\Gamma_{\mathcal{I}}$). When an encoded process reduces, new linear channels may be generated, for example, the providing channel of process *nd_choice*, which are all internal to the encoding.

The encoding of an output $[\![\overline{c}\langle a\rangle]\!]$ is implemented by spawning a new linear $\mathsf{SILL_S}$ process *snd* of type $\Pi x{:}\mathcal{U}_\mathsf{S}.\,\mathbf{1}$ with access to the buffer implementing channel c. The encoding then sends the channel a to the spawned process *snd*, waiting for *snd* to acquire the buffer c, insert a, and terminate. The encoding of an input $[\![c(x).P]\!]$ is implemented by spawning a new linear $\mathsf{SILL_S}$ process *poll_rcv* of type $\exists x{:}\mathcal{U}_\mathsf{S}.\,\mathbf{1}$ with access to the buffer implementing channel c. The encoding then waits for the spawned process *poll_rcv* to send back a channel and terminate, after which it continues at P, substituting the received channel for x. Process *poll_rcv* repeatedly checks, in a potentially infinite loop, if the buffer c contains an element. If so, it deletes it from the buffer, passes it on, and terminates. New name creation ($[\![\nu x\, P]\!]$) simply spawns a new buffer, offering on some fresh name x. Parallel ($[\![P_1 \mid P_2]\!]$) composition is embodied by a spawning of the processes P_1 in parallel with the executing process P_2. Finally, replication ($[\![!P]\!]$) is implemented by a loop that spawns copies of the replicated process.

To make our encoding more tangible, we derive the initial $\mathsf{SILL_S}$ configuration obtained from translating the process $[\![\overline{c}\langle a\rangle \mid c(x).\mathsf{0}]\!]$ according to the rules in Figure 4:

$$\begin{array}{ll} a_\mathsf{S}, c_\mathsf{S}\,;\,\cdot\,;\,\cdot\,;\,\cdot \vDash_\Sigma & \mathsf{proc}(\_,\ y_\mathsf{L} \leftarrow \mathit{poll\_rcv} \leftarrow c_\mathsf{S};\ z_\mathsf{S} \leftarrow \mathsf{recv}\ y_\mathsf{L};\ \mathsf{wait}\ y_\mathsf{L};\ \cdot), \\ & \mathsf{proc}(\_,\ y_\mathsf{L} \leftarrow \mathit{snd} \leftarrow c_\mathsf{S};\ \mathsf{send}\ y_\mathsf{L}\ a_\mathsf{S};\ \mathsf{wait}\ y_\mathsf{L};\ \cdot), \\ & \mathsf{buf}(a_\mathsf{S} \mid y_\mathsf{L} \leftarrow \mathsf{accept}\ a_\mathsf{S};\ P_{y_\mathsf{L}}),\ \mathsf{buf}(c_\mathsf{S} \mid y_\mathsf{L} \leftarrow \mathsf{accept}\ c_\mathsf{S};\ P_{y_\mathsf{L}}) \end{array}$$

To the left, we list the contents of the contexts $\Gamma_\mathcal{F}$; $\Gamma_\mathcal{B}$; $\Gamma_\mathcal{I}$; Δ, to the right the process configuration. For readability we use the short-form $\mathsf{buf}(a \mid P_a)$ to represent a sequence of *empty*-terminated *elem* processes denoting an entire buffer, with P_a standing for the next statements to be executed. The above configuration will reduce, according to the semantics of $\mathsf{SILL_S}$, until it halts in a state that consists of buffers representing the π-calculus channels, *coin_head* processes for any nondeterministic choices made, and $\mathsf{unavail}$ predicates for any shared channels that are not available. On the other hand, any linearly spawned processes that are internal to the encoding and not part of a buffer will have terminated. The interested reader can find a sample reduction of the above configuration in Appendix A.2.

Asynchrony of π-calculus outputs is achieved in our encoding by the introduction of the buffers, which temporarily store outputs until there is a process that is willing to receive. As a matter of fact, our buffers can be thought of manifestations of the "ether" to which asynchronous outputs are sent in the untyped asynchronous π-calculus! Our encoding is thus reminiscent of the encoding of the untyped asynchronous π-calculus into an untyped synchronous π-calculus with bags [5]. In fact, unlike the π-calculus where synchronous and asynchronous calculi have different expressive power [33], in the session-typed setting we can easily and selectively implement one in the other either by using double shifts to force acknowledgments [35] or by spawning single-message processes to achieve asynchrony [3]. The only significant point in $\mathsf{SILL_S}$ is that acquire/accept interactions must be a synchronization point. As we discuss in Section 3.3, crucial to the correctness of our encoding is also the removal of buffer elements non-deterministically. This guarantees that at no point in a reduction is the order between outputs determined. The use of nondeterministic deletion is another improvement over our earlier encoding [3], which uses non-deterministic insertion.

3.2 Operational Correspondence

We now develop an operational correspondence result for our encoding of the untyped asynchronous π-calculus. Operational correspondence results are standard *desiderata* for encodings of process calculi [18], showing that the computational features of the source language are preserved by the encoding in a precise sense. Following the terminology of [18], we aim to establish operational *completeness* (i.e., that π-calculus reductions are mimicked by the encoding) and *soundness* (i.e., that computations of encoded processes can be mapped back to those of the source terms) of our encoding.

As is the case in most encodings, some of the computation steps in the image of our encoding are purely administrative artifacts, and thus may not have a counterpart in the source. Specifically, the encoding of π-calculus channels as buffers introduces quite a few such "spurious" steps. Rather than relating source and image of the encoding at every step [5, 18], we introduce the notion of an *administrative* transition, and then state operational correspondence modulo such administrative transitions.

Given the nature of the asynchronous π-calculus, in which outputs are sent into the "ether" and synchronization only happens upon receipt, we deem the interactions leading to the insertion into a buffer as administrative and only the removal itself *relevant*. This treatment is consistent with the existing literature. In the encoding of the untyped asynchronous π-calculus into an untyped synchronous π-calculus with bags [5], output prefixes are equated with one-element bags, and synchronization amounts to directly reading from these bags. We define relevant and administrative transitions in the image of our encoding as follows:

▶ **Definition 1** (Relevant and Administrative Transitions of Encoding)**.** We say that a relevant transition, written $\longrightarrow_r$, is a standard transition between SILL configurations such that: $\Omega,\ \mathsf{proc}(d_{\mathsf{L}},\ x_{\mathsf{S}} \leftarrow \mathsf{recv}\ c_{\mathsf{L}}\ ;\ Q_{x_{\mathsf{S}}}) \longrightarrow \Omega',\ \mathsf{proc}(d_{\mathsf{L}},\ [a_{\mathsf{S}}/x_{\mathsf{S}}]\ Q_{x_{\mathsf{S}}})$, for some $\Omega, \Gamma_{\mathcal{F}}, \Gamma_{\mathcal{B}}, \Gamma_{\mathcal{I}}, a_{\mathsf{S}}, c_{\mathsf{S}}$, and d_{L} such that $a_{\mathsf{S}} \in \Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{B}}$, $c_{\mathsf{S}} \in \Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{B}}$, and $d_{\mathsf{S}} \in \Gamma_{\mathcal{I}}$.

An administrative transition, written $\longrightarrow_a$, is a transition defined by the standard transition relation between SILL configurations, but excluding a relevant transition. We write $\Longrightarrow_a$ for the reflexive transitive closure of $\longrightarrow_a$, and write $\Longrightarrow_r$ for $\Longrightarrow_a\longrightarrow_r\Longrightarrow_a$.

Inspecting our encoding (Figure 3 and Figure 4), we can see that a relevant transition amounts to the receive action in the some branch in process *poll_rcv*, which synchronizes with the buffer to receive a channel. The parameters of the above definition uniquely identify this synchronization point: process *poll_rcv* is a linear process providing along a linear channel d_{L} that is internal to the encoding ($d_{\mathsf{L}} \in \Gamma_{\mathcal{I}}$), and both the received channel a_{S} and the offering channel c_{S} of the buffer are either free our bound names of the original π-calculus process ($a_{\mathsf{S}} \in \Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{B}}$ and $c_{\mathsf{S}} \in \Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{B}}$).

Equipped with these two notions of transition, we can establish operational soundness and completeness. Their statements rely on the definition $[\![fn(P)]\!]$, which stand for a *configuration* of *empty* buffer processes of the form $\mathsf{buf}(c_{\mathsf{S}_1}\ |\ y_{\mathsf{L}} \leftarrow \mathsf{accept}\ c_{\mathsf{S}_1};\ Q_{y_{\mathsf{L}}}), \ldots, \mathsf{buf}(c_{\mathsf{S}_n}\ |\ y_{\mathsf{L}} \leftarrow \mathsf{accept}\ c_{\mathsf{S}_n};\ Q_{y_{\mathsf{L}}})$, where $fn(P) = \{c_1, \ldots, c_n\}$ denotes the set of free names in P. The definition allows us to compose an encoded π-calculus process with the appropriate buffer representations for all its free channel names.

▶ **Theorem 2** (Operational Correspondence)**.**

Completeness: *For all $P \longrightarrow P'$, there exists Ω_1, Ω_2 such that $[\![fn(P)]\!],\ \mathsf{proc}(\_,\ [\![P]\!]) \Longrightarrow_r \Omega_1, \Omega_2$ or $[\![fn(P)]\!],\ \mathsf{proc}(\_,\ [\![P]\!]) \Longrightarrow_a \Omega_1, \Omega_2$, with $[\![fn(P')]\!],\ \mathsf{proc}(\_,\ [\![P']\!]) \Longrightarrow_a \Omega_2$.*

Soundness: *For all P and $[\![fn(P)]\!],\ \mathsf{proc}(\_,\ [\![P]\!]) \Longrightarrow_r \Omega$, there exists a P', Ω_1, Ω_2 such that $P \longrightarrow P'$ and $\Omega = \Omega_1, \Omega_2$ and $[\![fn(P')]\!],\ \mathsf{proc}(\_,\ [\![P']\!]) \Longrightarrow_a \Omega_2$.*

For operational completeness, we identify each individual π-calculus reduction with either one relevant transition (possibly preceded or followed by several administrative transitions), or, for the π-calculus reduction corresponding to forking a parallel replica (i.e., $!P \longrightarrow P\ |!P$), with one administrative transition. For operational soundness, we match relevant transitions of encoded processes with one process reduction. In both settings we identify the artifacts of the encoding (coin processes and $\mathsf{unavail}$ channels) through the configuration Ω_1.

We note that the encodings of continuations eventually "catch up" (via administrative transitions) with the configuration that results from the relevant transition, instead of having a more immediate identification through the encoding. This treatment is due to the distinction between processes (static entities) and configurations (runtime entities) in $\mathsf{SILL_S}$, a distinction not present in the π-calculus, where processes *are* the runtime entities. For instance, parallel composition in $\mathsf{SILL_S}$ is achieved via an explicit spawning construct, whose semantics is to administratively *transition* to a configuration with the spawned process executing in parallel.

3.3 Observational Correspondence

In the previous section we have established that our encoding preserves reductions in the π-calculus in a strong sense, by identifying precisely the transitions in the operational semantics of $\mathsf{SILL_S}$ that correspond to reductions in the π-calculus processes in a way that is consistent with standard results on the nature of asynchrony of the untyped asynchronous π-calculus.

We now go further and relate *observable actions* (i.e., labelled transitions) in the π-calculus with their corresponding observables in $\mathsf{SILL_S}$ configuration rewrites. The key challenge here is to identify what those observables in $\mathsf{SILL_S}$ are because of the significant differences between the semantic frameworks of the π-calculus and $\mathsf{SILL_S}$. Whereas the π-calculus adopts an *open-world* view of observable actions with an unspecified environment (the "ether"), $\mathsf{SILL_S}$ adopts a *closed-world* view of a configuration of processes that are composed to form a complete program that can be run.

To clarify, consider the π-calculus process $\overline{c}\langle a\rangle \mid c(x).P$, where both c and a are free names. This process can interact with the environment through its free names by taking any of the following three observable actions: the output along c, the input along c, or the τ-action, corresponding to the synchronization between these dual actions. Now consider the $\mathsf{SILL_S}$ encoding of $[\![\overline{c}\langle a\rangle \mid c(x).P]\!]$. It results in a complete configuration consisting of the encoding of the process together with an *explicit* encoding of the free names c and a in terms of the buffers offering along c and a. Given this setup, any potential action on the π-calculus side will result in a series of actual computational steps on the $\mathsf{SILL_S}$ side, affecting the buffers as prescribed by the protocol of type $\mathcal{U}_{\mathsf{S}}$. In such a closed-world setting, trying to exactly mimic potential actions seems unnatural, if not impossible.

However, it is still the case that we want to relate π-calculus behavior with $\mathsf{SILL_S}$ behavior in a precise sense. To reconcile the open-world view of a labelled transition semantics with the closed-world view of computational steps, we note that the encoding already accounts for this issue by essentially *implementing* "the environment" through the channel encodings that must be composed with the processes at the top-level. Thus, what we deem to be *observable* when we consider a configuration made up of encoded π-calculus processes and corresponding channel encodings are precisely the inputs and outputs to and from buffers. Conversely, any steps in a $\mathsf{SILL_S}$ configuration that do not involve any inputs or outputs to and from buffers, we deem to be *unobservable.*

▶ **Definition 3** (Unobservable Transitions of Configuration)**.** Given a configuration Ω we say that there is an unobservable transition from Ω to Ω', written $\Omega \longrightarrow_{un} \Omega'$, iff $\Omega \longrightarrow \Omega'$ where the transition does *not* involve any of the two reductions below:

$$\begin{array}{rcl}\Omega_0,\, \mathsf{proc}(d_{\mathsf{L}},\, x_{\mathsf{S}} \leftarrow \mathsf{recv}\ c_{\mathsf{L}}\ ;\, P_{x_{\mathsf{S}}}) & \longrightarrow & \Omega_0',\, \mathsf{proc}(d_{\mathsf{L}},\, [a_{\mathsf{S}}/x_{\mathsf{S}}]\, P_{x_{\mathsf{S}}}) \\ \Omega_0,\, \mathsf{proc}(d_{\mathsf{L}},\, \mathsf{send}\ c_{\mathsf{L}}\ e_{\mathsf{S}}\ ;\, P) & \longrightarrow & \Omega_0',\, \mathsf{proc}(d_{\mathsf{L}},\, P)\end{array}$$

for some $\Omega_0, \Omega_0', \Gamma_{\mathcal{F}}, \Gamma_{\mathcal{B}}, \Gamma_{\mathcal{I}}, a_{\mathsf{S}}, c_{\mathsf{S}}, d_{\mathsf{L}}$ and e_{S} such that $a_{\mathsf{S}}, e_{\mathsf{S}}, c_{\mathsf{S}} \in \Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{B}}$, and $d_{\mathsf{S}} \in \Gamma_{\mathcal{I}}$. We write $\Omega \Longrightarrow_{un} \Omega'$ to stand for the reflexive transitive closure of $\longrightarrow_{un}$.

▶ **Definition 4** (Observable Transitions of Configuration). Given a configuration Ω we define a notion of an observable transition $\Omega \xrightarrow{\alpha} \Omega'$, stating that configuration Ω performs action α and transitions to configuration Ω', with $\alpha ::= \overline{c}\langle a\rangle \mid c(a) \mid (\nu a)\overline{c}\langle a\rangle \mid \tau$ as follows:

- $\Omega \xrightarrow{\overline{c}\langle a\rangle} \Omega'$ if $c, a \in \Gamma_{\mathcal{F}}$, $\Omega = \Omega_1, \mathsf{proc}(d_{\mathsf{L}}, \mathsf{send}\ c\ a; P), \Omega_2$, for some Ω_1, P, Ω_2 and $d_{\mathsf{S}} \in \Gamma_{\mathcal{I}}$ and $\Omega \longrightarrow \Omega'$ with $\Omega' = \Omega_1', \mathsf{proc}(d_{\mathsf{L}},\ P), \Omega_2'$, for some Ω_1', Ω_2'.
- $\Omega \xrightarrow{(\nu a)\overline{c}\langle a\rangle} \Omega'$ if $c \in \Gamma_{\mathcal{F}}$, $a \in \Gamma_{\mathcal{B}}$, $\Omega = \Omega_1, \mathsf{proc}(d_{\mathsf{L}}, \mathsf{send}\ c\ a; P), \Omega_2$, for some Ω_1, P, Ω_2 and $d_{\mathsf{S}} \in \Gamma_{\mathcal{I}}$ and $\Omega \longrightarrow \Omega'$ with $\Omega' = \Omega_1', \mathsf{proc}(d_{\mathsf{L}},\ P), \Omega_2'$, for some Ω_1', Ω_2'.
- $\Omega \xrightarrow{c(a)} \Omega'$ if $c \in \Gamma_{\mathcal{F}}$, $a \in \Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{B}}$, $\Omega = \Omega_1, \mathsf{proc}(d_{\mathsf{L}},\ x \leftarrow \mathsf{recv}\ c; P_x), \Omega_2$, for some Ω_1, P, Ω_2 and d, with $d_{\mathsf{S}} \in \Gamma_{\mathcal{I}}$ and $\Omega \longrightarrow \Omega'$ with $\Omega' = \Omega_1', \mathsf{proc}(d_{\mathsf{L}},\ [a/x]\, P_x), \Omega_2'$, for some Ω_1', Ω_2'.
- $\Omega \xrightarrow{\tau} \Omega'$ if all of the following:
  1. $c \in \Gamma_{\mathcal{B}}$, $a \in \Gamma_{\mathcal{F}} \cup \Gamma_{\mathcal{B}}$, $\Omega = \Omega_1, \mathsf{proc}(d_{\mathsf{L}}, \mathsf{send}\ c\ a; P), \Omega_2$, for some Ω_1, P, Ω_2 and $d_{\mathsf{S}} \in \Gamma_{\mathcal{I}}$ and $\Omega \longrightarrow \Omega''$ with $\Omega'' = \Omega_1', \mathsf{proc}(d_{\mathsf{L}},\ P), \Omega_2'$, for some $\Omega_1', \Omega_2', \Omega''$;
  2. $\Omega'' \Longrightarrow_{un} \Omega'''$ with $\Omega''' = \Omega_1'', \mathsf{proc}(d_{\mathsf{L}},\ x \leftarrow \mathsf{recv}\ c;\ Q_x), \Omega_2''$, for some $\Omega_1'', Q, \Omega_2''$ and d, with $d_{\mathsf{S}} \in \Gamma_{\mathcal{I}}$ and $\Omega''' \longrightarrow \Omega'$ with $\Omega' = \Omega_1''', \mathsf{proc}(d_{\mathsf{L}},\ [a/x]\ Q_x), \Omega_2'''$, for some Ω_1''', Ω_2'''.

We write $\Omega \overset{\alpha}{\Longrightarrow} \Omega'$ for $\Omega \Longrightarrow_{un} \Omega'' \xrightarrow{\alpha} \Omega''' \Longrightarrow_{un} \Omega'$.

The several observable transitions mirror the π-calculus labelled transitions, but where the role of the environment is replaced with the respective channel implementations. The first three cases define, respectively, output of a free name, output of a bound name, and input of a name (using the techniques of Definition 1 to track names). To account for synchronizations (τ-actions) in the π-calculus, we model the three steps that are required to perform a full communication in the encoding: an output action of a free or bound name to a buffer, followed by some sequence of unobservable transitions (needed to complete the several intermediate stages of the encoding), and an input action from the same buffer. With the right definition of observable in place, we define the natural notion of (weak) bisimulation between a π-calculus process and a $\mathsf{SILL_S}$ configuration.

▶ **Definition 5** (Weak Bisimulation). A relation $\mathcal{R}$ between asynchronous π-calculus processes and $\mathsf{SILL_S}$ configurations is a weak bisimulation if and only if, whenever $P\mathcal{R}\Omega$:

- If $P \xrightarrow{\alpha} P'$ and $\alpha \neq \tau$ then $\Omega \overset{\alpha}{\Longrightarrow} \Omega'$ and $P'\mathcal{R}\Omega'$
- If $P \xrightarrow{\tau} P'$ then $\Omega \Longrightarrow_{un} \Omega'$ or $\Omega \overset{\tau}{\Longrightarrow} \Omega'$ and $P'\mathcal{R}\Omega'$

plus the symmetric cases. We say that P is weakly bisimilar to Ω, written $P \approx \Omega$ iff there exists a weak bisimulation $\mathcal{R}$ such that $P\mathcal{R}\Omega$.

▶ **Theorem 6** (Observational Correspondence). *Let P be an asynchronous π-calculus process. We have that $P \approx \Omega$, $\mathsf{proc}(\_, [\![P]\!])$, where Ω is a configuration made up of process encodings for the free names of P, with (non-empty)* arbitrary *contents.*

The expert reader may wonder how our use of a weak bisimulation captures asynchrony in the appropriate way, noting that a weak *asynchronous* bisimulation is necessary to accurately relate the asynchronous π-calculus and synchronous π-calculus with bags [5]. Would it then not be the case that we could use queues or stacks as buffers and replicate our bisimulation argument? Our argument holds *precisely* because of the non-deterministic (i.e., bag-like) nature of our buffer implementations. Otherwise, out-of-order message reception – a defining characteristic of asynchrony – would *not* be simulated correctly by our encoding. In this sense, our bisimulation is implicitly asynchronous by implementing the environment in terms of buffers that enforce non-deterministic removals.

4 Simulating Shared Session Types in the π-calculus

In this section, we close the loop and provide an encoding of $\mathsf{SILL_S}$ process terms into the asynchronous *polyadic* π-calculus. The extension to the polyadic π-calculus is necessary to send along with the actual channel a fresh continuation channel that must be used for the next exchange in the protocol. This continuation-passing-style encoding (similar to that of Dardha et al. [13]) ensures that messages are received in the order specified by the protocol.

The resulting encoding is shown in Figure 5, providing a prototypical excerpt, with the complete listing shown in Appendix C. To simplify our encoding, we use a type-directed expansion of forwarding corresponding to the standard identity expansion in the sequent calculus. The resulting programs no longer use forwarding as a primitive, but implement it by processes that forward messages from client to provider and vice versa. Observational correctness of this expansion has been shown for the linear fragment [7] and with recursive types [17]. The strong logical underpinnings lead us to conjecture that observational correctness extends to sharing as well.

The general pattern of the encoding is to translate a positive type [35] to an output and a negative type [35] to an input with matching bindings. In case of a linear output or input, a fresh continuation channel is provided in addition to the actual channel to be sent or received, respectively. This channel is then used in the process continuation (in parallel) in place of the original channel, guaranteeing that the session discipline is not disturbed by out-of-order messages. To encode the acquire-release discipline of $\mathsf{SILL_S}$, we must preserve the shared mode of a channel throughout the translation. To this end, we indicate a linear $\mathsf{SILL_S}$ channel by a pair $\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner$, where the left and right projections yield the linear mode x_{L} and shared mode x_{S}, respectively. A release then restores the session to the shared channel. To ensure a *blocking* semantics for an acquire, the encoding of an acquire and accept forces synchronization via the channel w. The encoding of choice makes use of a selection channel per choice, used to indicate the choice outcome and unlock the appropriate continuation. For simplicity, and without loss of generality, we limit the encoding to binary internal and external choice. Process definitions are encoded as top-level replicated processes:

For each $(x_{\mathsf{L}} \leftarrow p \leftarrow \overline{y_{\mathsf{L}_i}}, \overline{w_{\mathsf{S}_j}} = P_{x_{\mathsf{L}}, \overline{y_{\mathsf{L}_i}}, \overline{w_{\mathsf{S}_j}}}) \in \Sigma$:

$$!(p(\overline{y_{\mathsf{L}_i}}, \overline{y_{\mathsf{S}_i}}, \overline{w_{\mathsf{S}_j}}, z).\nu x_{\mathsf{L}}, x_{\mathsf{S}}\, (\overline{z}\langle x_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid [\![[\overline{\ulcorner y_{\mathsf{L}_i}, y_{\mathsf{S}_i} \urcorner}/\overline{y_{\mathsf{L}_i}}, \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner / x_{\mathsf{L}}] P_{x_{\mathsf{L}}, \overline{y_{\mathsf{L}_i}}, \overline{w_{\mathsf{S}_j}}}]\!]))$$

For each $(x_{\mathsf{S}} \leftarrow p \leftarrow \overline{y_{\mathsf{S}_i}} = P_{x_{\mathsf{S}}, \overline{y_{\mathsf{S}_i}}}) \in \Sigma$: $!(p(\overline{y_{\mathsf{S}_i}}, z).\nu x_{\mathsf{S}}\, (\overline{z}\langle x_{\mathsf{S}}\rangle \mid [\![P_{x_{\mathsf{S}}, \overline{y_{\mathsf{S}_i}}}]\!]))$

The name of the definition is used as a channel that the encoding of the spawn construct uses to access new instances of the definition (generated via replication). The process receives the sessions that are needed to execute the definition and a channel z, used to send back the pair of (fresh) channels x_{S} and x_{L} used by the encoding of the definition body.

Operational Correspondence. To establish the operational correctness of our encoding, we consider an *asynchronous* semantics for $\mathsf{SILL_S}$. While operational completeness would not be affected by a synchronous semantics, soundness would require reasoning up-to observational equivalence. Since the expressiveness of $\mathsf{SILL_S}$ has been shown to be orthogonal to the choice of synchrony or asynchrony, we opt for the latter for the sake of simplicity. The semantics spawns single-message outputting processes using a continuation-passing style to achieve type-safe asynchrony [3].

Recalling that in $\mathsf{SILL_S}$ static entities are distinct from runtime entities, we lift the encoding to configurations, where the channels along which processes offer their session behavior are represented as bound names:

$$[\![\cdot]\!] = \mathbf{0} \qquad [\![\mathsf{proc}(c,\ P),\ \Omega]\!] = (\nu c_{\mathsf{S}}, c_{\mathsf{L}})([\![P]\!] \mid [\![\Omega]\!]) \qquad [\![\mathsf{unavail}(c_{\mathsf{S}}),\ \Omega]\!] = [\![\Omega]\!]$$

$$
\begin{array}{lcl}
[\![x_{\mathsf{L}} \leftarrow p \leftarrow \overline{\ulcorner y_{\mathsf{L}_i}, y_{\mathsf{S}_i} \urcorner}, \overline{w_{\mathsf{S}_j}};\ Q_{x_{\mathsf{L}}}]\!]^{\textsc{Spawn}_{\mathrm{LL}}} & = & \nu z\, (\overline{p}\langle \overline{y_{\mathsf{L}_i}}, \overline{y_{\mathsf{S}_i}}, \overline{w_{\mathsf{S}_j}}, z\rangle \mid z(x_{\mathsf{L}}, x_{\mathsf{S}}).[\![[\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner / x_{\mathsf{L}}]\, Q_{x_{\mathsf{L}}}]\!]) \\
[\![y_{\mathsf{L}} \leftarrow \mathsf{acquire}\ x_{\mathsf{S}};\ Q_{y_{\mathsf{L}}}]\!]^{\uparrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{L}} & = & \nu y_{\mathsf{L}}, w\, (\overline{x_{\mathsf{S}}}\langle y_{\mathsf{L}}, x_{\mathsf{S}}, w\rangle \mid w(\ ).[\![[\ulcorner y_{\mathsf{L}}, x_{\mathsf{S}} \urcorner / y_{\mathsf{L}}]\, Q_{y_{\mathsf{L}}}]\!]) \\
[\![y_{\mathsf{L}} \leftarrow \mathsf{accept}\ x_{\mathsf{S}};\ P_{y_{\mathsf{L}}}]\!]^{\uparrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{R}} & = & x_{\mathsf{S}}(y_{\mathsf{L}}, y_{\mathsf{S}}, w).(\overline{w}\langle\ \rangle \mid [\ulcorner y_{\mathsf{L}}, y_{\mathsf{S}} \urcorner / y_{\mathsf{L}}]\, [\![P_{y_{\mathsf{L}}}]\!]) \\
[\![y_{\mathsf{S}} \leftarrow \mathsf{release}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner;\ Q_{y_{\mathsf{S}}}]\!]^{\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{L}} & = & x_{\mathsf{L}}(y_{\mathsf{S}}).[\![Q_{y_{\mathsf{S}}}]\!] \\
[\![x_{\mathsf{S}} \leftarrow \mathsf{detach}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner;\ P]\!]^{\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{R}} & = & \overline{x_{\mathsf{L}}}\langle x_{\mathsf{S}}\rangle \mid [\![P]\!] \\
[\![y_{\mathsf{S}} \leftarrow \mathsf{recv}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner;\ P_{y_{\mathsf{S}}}]\!]^{\exists_{\mathrm{L}}/\Pi_{\mathrm{R}}} & = & x_{\mathsf{L}}(y_{\mathsf{S}}, z_{\mathsf{L}}, z_{\mathsf{S}}).[\![[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}} \urcorner / \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner]\, P_{y_{\mathsf{S}}}]\!] \\
[\![\mathsf{send}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner\ y_{\mathsf{S}};\ P]\!]^{\exists_{\mathrm{R}}/\Pi_{\mathrm{L}}} & = & \nu z_{\mathsf{L}}\, (\overline{x_{\mathsf{L}}}\langle y_{\mathsf{S}}, z_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid [\![[\ulcorner z_{\mathsf{L}}, x_{\mathsf{S}} \urcorner / \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner]\, P]\!]) \\
[\![\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner.\mathsf{case}(P, Q)]\!]^{\oplus_{\mathrm{L}}/\&_{\mathrm{R}}} & = & \nu y_{\mathsf{inl}}, y_{\mathsf{inr}}\, (\overline{x_{\mathsf{L}}}\langle y_{\mathsf{inl}}, y_{\mathsf{inr}}\rangle \mid y_{\mathsf{inl}}(z_{\mathsf{L}}, z_{\mathsf{S}}).[\![[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}} \urcorner / \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner]\, P]\!] \mid \\
 & & \quad y_{\mathsf{inr}}(z_{\mathsf{L}}, z_{\mathsf{S}}).[\![[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}} \urcorner / \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner]\, Q]\!]) \\
[\![\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner.\mathsf{inr};\ Q]\!]^{\oplus_{\mathrm{R}_2}/\&_{\mathrm{L}_2}} & = & \nu z_{\mathsf{L}}\, (x_{\mathsf{L}}(y_{\mathsf{inl}}, y_{\mathsf{inr}}).\overline{y_{\mathsf{inr}}}\langle z_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid [\![[\ulcorner z_{\mathsf{L}}, x_{\mathsf{S}} \urcorner / \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}} \urcorner]\, Q]\!])
\end{array}
$$

Figure 5 Translation of $\mathsf{SILL_S}$ process terms into the asynchronous, polyadic π-calculus (Excerpt – See Appendix C for the complete definition).

We can now show that $\mathsf{SILL_S}$ transitions are always matched by a synchronization in the π-calculus (and vice-versa) rather straightforwardly, given the direct nature of the encoding.

▶ **Theorem 7** (Operational Correspondence). *Let $\longrightarrow^+$ be the transitive closure of $\longrightarrow$:*

Completeness: *If P is a well-typed, forwarding-free $\mathsf{SILL_S}$ process and $\mathsf{proc}(a, P) \longrightarrow^+ \Omega$ then $[\![P]\!] \longrightarrow^+ [\![\Omega]\!]$.*

Soundness: *For all well-typed, forwarding-free $\mathsf{SILL_S}$ configurations Ω such that $[\![\Omega]\!] \longrightarrow^+ Q$, there exists a configuration Ω' such that $\Omega \longrightarrow^+ \Omega'$ and $Q \Longrightarrow [\![\Omega']\!]$.*

5 Related Work

Encodings of Asynchrony. Encodability results are a standard benchmark for expressiveness of π-calculi [18]. For the asynchronous π-calculus [21], encodings into various formulations of synchronous π-calculi exist [5], as well as impossibility results [33] regarding the ability to adequately encode certain forms of choice in an asynchronous setting.

Our encoding of the asynchronous π-calculus is reminiscent of the encoding of the asynchronous π-calculus in a π-calculus with bags by Beauxis et al. [5], shown to be in tight correspondence via an asynchronous bisimilarity. Their framework considers buffers as primitives in the target calculus, whereas we encode the bag-like behavior of buffers explicitly as $\mathsf{SILL_S}$ processes that adhere to a particularly typed protocol, making our encoding more primitive, but adding several administrative reductions to encoded processes due to the sharing discipline and the implementation of nondeterminism when reading from a buffer. This fact, combined with the restrictive (typed) usage of buffers in our setting allows us to reason using a weak bisimilarity rather than a more involved *asynchronous* bisimilarity. Beauxis et al. also consider an encoding of their calculus with bags in the asynchronous π-calculus. The general structure of the encoding is similar to our encoding of $\mathsf{SILL_S}$ in the asynchronous π-calculus, modulo the richer syntax of $\mathsf{SILL_S}$, which introduces more communication actions in the image of the encoding. We note that our encoding is greatly simplified by linearity and by the fact that $\mathsf{SILL_S}$ does not employ *mixed* choice [31].

Linear Logic and Session Types. The propositions-as-types correspondence between linear logic and session types introduced by Caires and Pfenning [8, 9] initiated an ongoing line of research exploring the logical reading of sessions along various axes [42, 24, 34, 35, 2].

Starting with [8], which translates the linear session language into a π-calculus (which is more expressiveness than the source language), various works on encodings in this logical setting have been proposed [39, 41, 29, 28]. These study encodings between session-typed processes and functional languages, since the considered session languages are not powerful enough to express general π-calculus behaviors. Recent works [2, 12] attempt to address these limitations in expressiveness by allowing composed processes to share more than one linear channel, but still do not allow for the sharing available in $\mathsf{SILL_S}$, crucial to our encoding. We also highlight the work of Dardha and Pérez [14] comparing session-typed processes arising from linear logic and those from the Kobayashi-style typings [26, 25, 32] for the π-calculus. They observe that the *degree of sharing* determines an expressiveness hierarchy for typed processes and develop encodings from the latter into the former (not preserving the degree of sharing). In this sense, our encoding of asynchronous π-calculus completely preserves the sharing of channels, at the cost of allowing deadlocks when acquiring shared channels.

Session-Typed Behavioral Theory. The behavioral theory of session-typed processes has been studied in both the multiparty [27] and the linear logic settings [7, 34, 1]. Our notion of observation is related to the observed communication semantics of Atkey [1], which must also address the challenge of observing actions within a “closed-world” framework. However, their system is based on classical linear logic and does not have sharing, making the precise relationship with our formulation of observable on shared names unclear.

Substructural Logical Reasoning. The work of Deng et al. [15] studies a natural notion of logical preorder between linear logic contexts using process calculi techniques such as simulation preorders. While the study of the relationship between contexts can be seen as a study of multiset rewriting of configurations, the process calculus induced by their reading of linear logic is a fairly different formalism from $\mathsf{SILL_S}$. For instance, their labelled transition system cannot be reasonably used as a labelled transition system for $\mathsf{SILL_S}$ since it cannot represent the equivalent of channel passing, nor does it make use of the deep inspection of multiset rewriting terms needed for our semantics and reasoning.

6 Concluding Remarks

In this paper, we gave an encoding of the untyped asynchronous π-calculus into $\mathsf{SILL_S}$ via a universal shared session type $\mathcal{U}_\mathsf{s}$, proving its operational and observational correctness. This result shows that the full expressiveness of the untyped asynchronous π-calculus can be recovered in session-typed process calculi. We also provide an operationally correct encoding in the other direction to simulate shared session types in the π-calculus. Given their universality, session-typed calculi with manifest sharing become strong competitors over traditional approaches since they not only guarantee protocol compliance in the presence of non-determinism but also make sharing explicit in the type structure. For future work, we wish to investigate a general behavioral theory of manifest sharing, as well as study techniques to establish deadlock-freedom in the presence of shared channels.

References

1. Robert Atkey. Observed communication semantics for classical processes. In *European Symposium on Programming (ESOP)*, pages 56–82, 2017.
2. Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation confers concurrency. In S. Lindley et al., editor, *Wadler Festschrift*, pages 32–55. Springer LNCS 9600, 2016.
3. Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP):37:1–37:29, 2017.

4 Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. Technical Report CMU-CS-17-106, Carnegie Mellon University, March 2017.

5 Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. On the asynchronous nature of the asynchronous π-calculus. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 473–492. Springer, 2008.

6 P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *8th International Workshop on Computer Science Logic (CSL)*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1994. An extended version appeared as Technical Report UCAM-CL-TR-352, University of Cambridge.

7 Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *European Symposium on Programming (ESOP)*, pages 330–349. Springer, 2013.

8 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR)*, pages 222–236. Springer, 2010.

9 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

10 Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, 2009.

11 Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, 1999.

12 Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 91–109, 2018.

13 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Principles and Practice of Declarative Programming (PPDP)*, pages 139–150, 2012.

14 Ornela Dardha and Jorge A. Pérez. Comparing deadlock-free session typed processes. In *EXPRESS/SOS*, pages 1–15, 2015.

15 Yuxin Deng, Robert J. Simmons, and Iliano Cervesato. Relating reasoning methodologies in linear logic and process algebra. *Mathematical Structure in Computer Science*, 26(5):868–906, January 2016.

16 Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

17 Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. In A. Ahmed, editor, *European Symposium on Programming (ESOP'18)*, pages 771–798, Thessaloniki, Greece, April 2018. Springer LNCS 10801.

18 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010.

19 Dennis Griffith and Frank Pfenning. SILL. `https://github.com/ISANobody/sill`, 2015.

20 Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR)*, pages 509–523. Springer, 1993.

21 Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 133–147. Springer, 1991.

22 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP)*, pages 122–138. Springer, 1998.

23 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008.

24 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 582–594, 2016.

25 Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.

26 Naoki Kobayashi. A new type system for deadlock-free processes. In *International Conference on Concurrency Theory (CONCUR)*, pages 233–247, 2006.

27 Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. *Logical Methods in Computer Science*, 10(4), 2014.

28 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *European Symposium On Programming (ESOP)*, pages 560–584, 2015.

29 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In *International Colloquium on Functional Progrmaming (ICFP)*, pages 434–447, 2016.

30 Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

31 Uwe Nestmann. What is a "good" encoding of guarded choice? *Inf. Comput.*, 156(1-2):287–319, 2000.

32 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *Computer Science Logic – Logic in Computer Science (CSL-LICS)*, pages 72:1–72:10, 2014.

33 Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

34 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

35 Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 3–22. Springer, 2015.

36 Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, January 2009. URL: `http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf`.

37 Davide Sangiorgi and David Walker. *The π-Calculus - A Theory of Mobile Processes*. Cambridge University Press, 2001.

38 Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and New University of Lisbon, 2015.

39 Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In *15th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, pages 346–360. Springer, 2012.

40 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *22nd European Symposium on Programming (ESOP)*, pages 350–369. Springer, 2013.

41 Bernardo Toninho and Nobuko Yoshida. On polymorphic sessions and functions - A tale of two (fully abstract) encodings. In *European Symposium On Programming (ESOP)*, pages 827–855, 2018.

42 Philip Wadler. Propositions as sessions. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012.

A Proofs and Additional Definitions

A.1 Auxiliary processes omitted from Section 3.1

$$\mathit{nd\_choice} : \{\oplus\{\mathsf{yes} : \mathbf{1},\ \mathsf{no} : \mathbf{1}\}\}$$

$$\begin{array}{l}
d \leftarrow \mathit{nd_choice} = \\
\quad \mathbf{c} \leftarrow \mathit{coin_head}\ ; \\
\quad f \leftarrow \mathit{coin_flipper} \leftarrow \mathbf{c}\ ; \\
\quad c' \leftarrow \mathsf{acquire}\ \mathbf{c}\ ; \\
\quad \mathsf{case}\ c'\ \mathsf{of} \\
\quad \mid \mathsf{head} \rightarrow \mathbf{c} \leftarrow \mathsf{release}\ c'\ ; \\
\qquad\qquad\quad d.\mathsf{yes}\ ; \\
\qquad\qquad\quad \mathsf{wait}\ f\ ; \\
\qquad\qquad\quad \mathsf{close}\ d \\
\quad \mid \mathsf{tail} \rightarrow \ \ \mathbf{c} \leftarrow \mathsf{release}\ c'\ ; \\
\qquad\qquad\quad d.\mathsf{no}\ ; \\
\qquad\qquad\quad \mathsf{wait}\ f\ ; \\
\qquad\qquad\quad \mathsf{close}\ d
\end{array}$$

$$\begin{array}{l}
\mathbf{coin} = \uparrow_{\mathsf{L}}^{\mathsf{S}} \oplus \{\mathsf{head} : \downarrow_{\mathsf{L}}^{\mathsf{S}}\mathbf{coin}, \\
\qquad\qquad\quad \mathsf{tail} : \downarrow_{\mathsf{L}}^{\mathsf{S}}\mathbf{coin}\}
\end{array}$$

$$\mathit{coin\_head} : \{\mathbf{coin}\}$$

$$\begin{array}{l}
\mathbf{c} \leftarrow \mathit{coin_head} = \\
\quad c' \leftarrow \mathsf{accept}\ \mathbf{c}\ ; \\
\quad c'.\mathsf{head}\ ; \\
\quad \mathbf{c} \leftarrow \mathsf{detach}\ c'\ ; \\
\quad \mathbf{c} \leftarrow \mathit{coin_tail}
\end{array}$$

$$\mathit{coin\_tail} : \{\mathbf{coin}\}$$

$$\begin{array}{l}
\mathbf{c} \leftarrow \mathit{coin_tail} = \\
\quad c' \leftarrow \mathsf{accept}\ \mathbf{c}\ ; \\
\quad c'.\mathsf{tail}\ ; \\
\quad \mathbf{c} \leftarrow \mathsf{detach}\ c'\ ; \\
\quad \mathbf{c} \leftarrow \mathit{coin_head}
\end{array}$$

$$\mathit{coin\_flipper} : \{\mathbf{1} \leftarrow \mathbf{coin}\}$$

$$\begin{array}{l}
d \leftarrow \mathit{coin_flipper} \leftarrow \mathbf{c} = \\
\quad c' \leftarrow \mathsf{acquire}\ \mathbf{c}\ ; \\
\quad \mathsf{case}\ c'\ \mathsf{of} \\
\quad \mid \mathsf{head} \rightarrow \mathbf{c} \leftarrow \mathsf{release}\ c'\ ; \\
\qquad\qquad\quad \mathsf{close}\ d \\
\quad \mid \mathsf{tail} \rightarrow \ \ \mathbf{c} \leftarrow \mathsf{release}\ c'\ ; \\
\qquad\qquad\quad \mathsf{close}\ d
\end{array}$$

A.2 Sketch of Encoding Reductions, Complementing Section 3.1

To make our encoding more tangible, we sketch how the process $[\![\overline{c}\langle a\rangle \mid c(x).0]\!]$ reduces. Our encoding yields the following initial configuration:

$$\begin{array}{ll}
a_{\mathsf{S}}, c_{\mathsf{S}}\ ;\ \cdot\ ;\ \cdot\ ;\ \cdot \vDash_{\Sigma} & \mathsf{proc}(_,\ y_{\mathsf{L}} \leftarrow \mathit{poll_rcv} \leftarrow c_{\mathsf{S}};\ z_{\mathsf{S}} \leftarrow \mathsf{recv}\ y_{\mathsf{L}};\ \mathsf{wait}\ y_{\mathsf{L}};\ \cdot), \\
& \mathsf{proc}(_,\ y_{\mathsf{L}} \leftarrow \mathit{snd} \leftarrow c_{\mathsf{S}};\ \mathsf{send}\ y_{\mathsf{L}}\ a_{\mathsf{S}};\ \mathsf{wait}\ y_{\mathsf{L}};\ \cdot), \\
& \mathsf{buf}(a_{\mathsf{S}} \mid y_{\mathsf{L}} \leftarrow \mathsf{accept}\ a_{\mathsf{S}};\ P_{y_{\mathsf{L}}}),\ \mathsf{buf}(c_{\mathsf{S}} \mid y_{\mathsf{L}} \leftarrow \mathsf{accept}\ c_{\mathsf{S}};\ P_{y_{\mathsf{L}}})
\end{array}$$

To the left, we list the contents of the context $\Gamma_{\mathcal{F}};\ \Gamma_{\mathcal{B}};\ \Gamma_{\mathcal{I}};\ \Delta$, to the right the process configuration. For ease of reading we use the short-form $\mathsf{buf}(a \mid s_a;\ Q_a \mid b_0, c_1 \cdot \ldots \cdot b_{n-1}, c_n)$ to represent a sequence of *empty*-terminated *elem* processes denoting an entire buffer:

$$\begin{array}{lcl}
\mathsf{buf}(a \mid s_a;\ Q_a \mid b_0, c_1 \cdot \ldots \cdot b_{n-1}, c_n) & \triangleq & \mathsf{proc}(a,\ [b_0/x, c_1/d](s_a;\ Q_a))_{\mathit{elem}}, \\
& & \mathsf{proc}(c_{n-1},\ [b_{n-1}/x, c_n/d](s_{c_{n-1}};\ Q_{c_{n-1}}))_{\mathit{elem}}, \\
& & \mathsf{proc}(c_n,\ s_{c_n};\ Q_{c_n})_{\mathit{empty}} \\
\mathsf{buf}(a \mid s_a;\ Q_a) & \triangleq & \mathsf{proc}(a,\ s_a;\ Q_a)_{\mathit{empty}}
\end{array}$$

where *elem* and *empty* stand for the processes associated with the corresponding process definitions (Figure 3) and $s;\ Q$ for the next statement s with continuation Q to be executed as part of those processes, with the supplied channel arguments substituted. Analogously, we provide corresponding short-forms for internal processes, such as *snd* and *coin_head*. After a few transitions, we obtain:

$$a_{\mathsf{S}}, c_{\mathsf{S}}\,;\,\cdot\,;\,b_{\mathsf{S}}\,;\,b_{\mathsf{L}}, c_{\mathsf{L}} \vDash_\Sigma \mathsf{proc}(\_,\, y_{\mathsf{L}} \leftarrow \mathit{poll\_rcv} \leftarrow c_{\mathsf{S}};\, z_{\mathsf{S}} \leftarrow \mathsf{recv}\, y_{\mathsf{L}};\, \mathsf{wait}\, y_{\mathsf{L}};\, \cdot),\ \mathsf{proc}(\_,\, \mathsf{wait}\, b_{\mathsf{L}};\, \cdot),$$
$$\mathsf{snd}(b_{\mathsf{L}},\, \mathsf{send}\, c_{\mathsf{L}}\, a_{\mathsf{S}};\, [c_{\mathsf{S}}/z_{\mathsf{S}_1}, a_{\mathsf{S}}/z_{\mathsf{S}_2}, c_{\mathsf{L}}/y_{\mathsf{L}}]\, P_{z_{\mathsf{S}_1}, z_{\mathsf{S}_2}, y_{\mathsf{L}}}),\ \mathsf{buf}(a_{\mathsf{S}} \mid y_{\mathsf{L}} \leftarrow \mathsf{accept}\, a_{\mathsf{S}};\, P_{y_{\mathsf{L}}}),$$
$$\mathsf{buf}(c_{\mathsf{L}} \mid z_{\mathsf{S}} \leftarrow \mathsf{recv}\, c_{\mathsf{L}};\, [c_{\mathsf{L}}/y_{\mathsf{L}}]\, P_{y_{\mathsf{L}}, z_{\mathsf{S}}}),\ \mathsf{unavail}(b_{\mathsf{S}}),\ \mathsf{unavail}(c_{\mathsf{S}})$$

which pinpoints the state in which the internal *snd* process providing along channel b_{L} is about to insert ("send") the channel a_{S} into the buffer providing along channel c_{L}. After a few more transitions, the computation halts in the following configuration:

$$a_{\mathsf{S}}, c_{\mathsf{S}}\,;\,\cdot\,;\,b_{\mathsf{S}}, b'_{\mathsf{S}}, b''_{\mathsf{S}}, d_{\mathsf{S}}, d'_{\mathsf{S}}, d''_{\mathsf{S}}, d'''_{\mathsf{S}}, d''''_{\mathsf{S}}\,;\,\cdot \vDash_\Sigma \mathsf{buf}(a_{\mathsf{S}} \mid y_{\mathsf{L}} \leftarrow \mathsf{accept}\, a_{\mathsf{S}};\, P_{y_{\mathsf{L}}}),\ \mathsf{buf}(c_{\mathsf{S}} \mid y_{\mathsf{L}} \leftarrow \mathsf{accept}\, c_{\mathsf{S}};\, P_{y_{\mathsf{L}}}),$$
$$\mathsf{coin\_head}(d''''_{\mathsf{S}},\, y_{\mathsf{L}} \leftarrow \mathsf{accept}\, d''''_{\mathsf{S}};\, P_{y_{\mathsf{L}}}),\ \mathsf{unavail}(b_{\mathsf{S}}),\ \mathsf{unavail}(b'_{\mathsf{S}}),$$
$$\mathsf{unavail}(b''_{\mathsf{S}}),\ \mathsf{unavail}(d'_{\mathsf{S}}),\ \mathsf{unavail}(d''_{\mathsf{S}}),\ \mathsf{unavail}(d_{\mathsf{S}}),\ \mathsf{unavail}(d'''_{\mathsf{S}})$$

The above configuration is prototypical for what is "left behind" after computation has come to a halt: buffers representing the free and bound π-calculus channels, coin_head processes for any nondeterministic choices made, and unv predicates for any shared channels that are not available. On the other hand, any linearly spawned processes that are internal to the encoding and not part of a buffer will have terminated.

A.3 Proofs for Section 3.2

▶ **Theorem 2** (Operational Correspondence).

Completeness: *For all $P \longrightarrow P'$, there exists Ω_1, Ω_2 such that* $[\![fn(P)]\!], \mathsf{proc}(\_, [\![P]\!]) \Longrightarrow_r \Omega_1, \Omega_2$ *or* $[\![fn(P)]\!], \mathsf{proc}(\_, [\![P]\!]) \Longrightarrow_a \Omega_1, \Omega_2$*, with* $[\![fn(P')]\!], \mathsf{proc}(\_, [\![P']\!]) \Longrightarrow_a \Omega_2$.

Soundness: *For all P and* $[\![fn(P)]\!], \mathsf{proc}(\_, [\![P]\!]) \Longrightarrow_r \Omega$*, there exists a P', Ω_1, Ω_2 such that $P \longrightarrow P'$ and $\Omega = \Omega_1, \Omega_2$ and* $[\![fn(P')]\!], \mathsf{proc}(\_, [\![P']\!]) \Longrightarrow_a \Omega_2$.

Proof. By Lemma 8 and Lemma 9. ◀

▶ **Theorem 8** (Operational Completeness). *For all $P \longrightarrow P'$, there exists Ω_1, Ω_2 such that* $[\![fn(P)]\!], \mathsf{proc}(\_, [\![P]\!]) \Longrightarrow_r \Omega_1, \Omega_2$ *or* $[\![fn(P)]\!], \mathsf{proc}(\_, [\![P]\!]) \Longrightarrow_a \Omega_1, \Omega_2$*, with* $[\![fn(P')]\!], \mathsf{proc}(\_, [\![P']\!]) \Longrightarrow_a \Omega_2$.

Proof. By induction on the given π-calculus reductions. In all cases below, except that for replication, we omit the second disjunct in the inductive hypothesis since it is not necessary.

Case: $\overline{c}\langle a\rangle \mid c(x).P_0 \longrightarrow [a/x]\, P_0$

Follows by carrying out the calculation of the transitions of the encoding, as in the example of Section 3.1. The resulting configuration is of the form: Ω_1, $[\![fn(P_0) \cup \{c, a\}]\!]$, $\mathsf{proc}(\_, [\![[a/x]\, P_0]\!])$, where Ω_1 is made up of $\mathsf{unavail}(\_)$ channels and coin processes.

Case: $P_1 \mid P_2 \longrightarrow P'_1 \mid P_2$ with $P_1 \longrightarrow P'_1$

$[\![P_1 \mid P_2]\!] = \_ \leftarrow [\![P_1]\!]\,;\,[\![P_2]\!]$ by definition

$fn(P_1 \mid P_2) = fn(P_1) \cup fn(P_2)$ by definition

$[\![fn(P_1)]\!], [\![fn(P_2)]\!], \mathsf{proc}(\_, \_ \leftarrow [\![P_1]\!]\,;\,[\![P_2]\!]) \longrightarrow_a [\![fn(P_1)]\!], [\![fn(P_2)]\!], \mathsf{proc}(\_, [\![P_1]\!]), \mathsf{proc}(\_, [\![P_2]\!])$ by semantics

$[\![fn(P_1)]\!], \mathsf{proc}(\_, [\![P_1]\!]) \Longrightarrow_r \Omega_1, \Omega_2$ with $[\![fn(P'_1)]\!], \mathsf{proc}(\_, [\![P'_1]\!]) \Longrightarrow_a \Omega_2$ **(1)** by i.h.

$[\![fn(P_1)]\!], [\![fn(P_2)]\!], \mathsf{proc}(\_, [\![P_1]\!]), \mathsf{proc}(\_, [\![P_2]\!]) \Longrightarrow_r \Omega_1, \Omega_2, [\![fn(P_2) \setminus fn(P'_1)]\!], \mathsf{proc}(\_, [\![P_2]\!])$ **(2)** by semantics

$[\![P'_1 \mid P_2]\!] = \_ \leftarrow [\![P'_1]\!]\,;\,[\![P_2]\!]$ by definition

$\llbracket fn(P_1') \cup fn(P_2) \rrbracket$, $\mathsf{proc}(\_, \_ \leftarrow \llbracket P_1' \rrbracket \mathbin{;} \llbracket P_2 \rrbracket) \longrightarrow_a \llbracket fn(P_1') \rrbracket, \llbracket fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_1' \rrbracket), \mathsf{proc}(\_, \llbracket P_2 \rrbracket)$
by semantics

$\llbracket fn(P_1') \rrbracket, \llbracket fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_1' \rrbracket), \mathsf{proc}(\_, \llbracket P_2 \rrbracket) \Longrightarrow_a \Omega_2, \llbracket fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_2 \rrbracket)$
by **(1)** and **(2)**

Case: $P_1 \mid P_2 \longrightarrow P_1 \mid P_2'$ with $P_2 \longrightarrow P_2'$
As above.

Case: $!P_1 \longrightarrow P_1 \mid !P_1$

$\llbracket fn(!P_1) \rrbracket, \mathsf{proc}(\_, \llbracket !P_1 \rrbracket) = \llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \_ \leftarrow \llbracket P_1 \rrbracket \mathbin{;} Rec_{!P_1})$ by definition
$\longrightarrow_a \llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket), \mathsf{proc}(\_, Rec_{!P_1})$ by semantics
$\llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \mid !P_1 \rrbracket) = \llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \_ \leftarrow \llbracket P_1 \rrbracket \mathbin{;} \llbracket !P_1 \rrbracket)$ by definition
$\longrightarrow_a \llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket), \mathsf{proc}(\_, Rec_{!P_1})$ by semantics

◀

▶ **Theorem 9** (Operational Soundness). *For all P and $\llbracket fn(P) \rrbracket, \mathsf{proc}(\_, \llbracket P \rrbracket) \Longrightarrow_r \Omega$, there exists a P', Ω_1, Ω_2 such that $P \longrightarrow P'$ and $\Omega = \Omega_1, \Omega_2$ and $\llbracket fn(P') \rrbracket, \mathsf{proc}(\_, \llbracket P' \rrbracket) \Longrightarrow_a \Omega_2$*

Proof. By induction on the structure of P.

Case: $\overline{c}\langle a \rangle$
No relevant transition.

Case: $c(x).P_1$
No relevant transition.

Case: $\overline{c}\langle a \rangle \mid c(x).P_1$
Without loss of generality, consider the relevant transition of $\llbracket fn(\overline{c}\langle a \rangle \mid c(x).P_1) \rrbracket, \mathsf{proc}(\_, \llbracket \overline{c}\langle a \rangle \mid c(x).P_1 \rrbracket)$ leading to some $\Omega_1 = \Omega_1', \Omega_1''$ where the output on c and the input on c have synchronized, followed by some number of administrative transitions in P_1, leading to $\Omega_2 = \Omega_2', \Omega_2''$.
Let $P' = [a/x]\, P_1$
$\llbracket \{c, a\} \cup fn(P_1) \rrbracket, \mathsf{proc}(\_, [a/x]\, \llbracket P_1 \rrbracket) = \Omega_1''$
by definition, where Ω_1' are leftover unavailable and coin processes
$\Omega_1'' \Longrightarrow_a \Omega_2''$ by definition, where Ω_2'' are leftover unavailable and coin processes

Case: $P_1 \mid P_2$
We note that $\llbracket fn(P_1) \cup fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \mid P_2 \rrbracket) \Longrightarrow_a \llbracket fn(P_1) \cup fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket), \mathsf{proc}(\_, \llbracket P_2 \rrbracket)$.
We consider the following subcases: **(1)** $\llbracket fn(P_1) \cup fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket), \mathsf{proc}(\_, \llbracket P_2 \rrbracket) \Longrightarrow_r \Omega_1$, where the relevant transition arises from $\llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket)$ and $\mathsf{proc}(\_, \llbracket P_2 \rrbracket)$ has not transitioned;
(2) $\llbracket fn(P_1) \cup fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket), \mathsf{proc}(\_, \llbracket P_2 \rrbracket) \Longrightarrow_r \Omega_1$, where the relevant transition arises from $\llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket)$ and $\mathsf{proc}(\_, \llbracket P_2 \rrbracket)$ has transitioned.

Subcase: (1)
Ω_1 is of the form $\llbracket fn(P_2) \setminus fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_2 \rrbracket), \Omega_1'$, where $\llbracket fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_1 \rrbracket) \Longrightarrow_r \Omega_1'$
$P_1 \longrightarrow P_1'$ such that $\Omega_1' = \Omega_1'', \Omega_1'''$ and $\llbracket fn(P_1') \rrbracket, \mathsf{proc}(\_, \llbracket P_1' \rrbracket) \Longrightarrow_a \Omega_1'''$ **(i)** by i.h.
$P_1 \mid P_2 \longrightarrow P_1' \mid P_2$ by the reduction semantics
$\llbracket fn(P_1') \cup fn(P_2) \rrbracket, \mathsf{proc}(\_, \llbracket P_1' \mid P_2 \rrbracket) \longrightarrow_a \llbracket fn(P_1') \rrbracket, \llbracket fn(P_2) \setminus fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_1' \rrbracket), \mathsf{proc}(\_, \llbracket P_2 \rrbracket)$
by semantics
$\Longrightarrow_a \llbracket fn(P_2) \setminus fn(P_1) \rrbracket, \mathsf{proc}(\_, \llbracket P_2 \rrbracket), \Omega_1'''$ by **(i)**

Subcase: (2)
This subcase is as **(1)**, noting that we can always identify the transition sequence originating from the encoding of P_1 alone. We can then use the i.h. to show part

of the transition sequence of the encoding of $P_1' \mid P_2$ and then "replay" the transitions from encoded P_2.

The other subcases are symmetric or fall under the previous case.

Case: $!P_1$

Without loss of generality, consider $[\![fn(P_1)]\!], \mathsf{proc}(\_, [\![!P_1]\!]) \longrightarrow_a [\![fn(P_1)]\!]$ $, \mathsf{proc}(\_, [\![P_1]\!]), \mathsf{proc}(\_, Rec_{!P_1}) \Longrightarrow_r \Omega$.
We have that $!P_1 \longrightarrow P_1 \mid !P_1$. Then, $[\![fn(P_1)]\!], \mathsf{proc}(\_, [\![P_1 \mid !P_1]\!]) \longrightarrow_a [\![fn(P_1)]\!]$ $, \mathsf{proc}(\_, [\![P_1]\!]), \mathsf{proc}(\_, Rec_{!P_1})$ which transitions via $\Longrightarrow_r$ to Ω by assumption. ◀

B Proofs for Section 3.3

▶ **Theorem 6** (Observational Correspondence). *Let P be an asynchronous π-calculus process. We have that $P \approx \Omega, \mathsf{proc}(\_, [\![P]\!])$, where Ω is a configuration made up of process encodings for the free names of P, with (non-empty)* arbitrary *contents.*

Proof. Let Ω_1 be any configuration made up of buffer processes for the free names of P, with *arbitrary* contents, and $\mathcal{R}_1 = \{(P, \Omega) \mid \Omega = \Omega_1, \Omega_2, \mathsf{proc}(\_, [\![P]\!])\}$, for all P, Ω_2 where Ω_2 is made up of an arbitrary number of $\mathsf{unavail}$ and coin processes that are no longer used; and $\mathcal{R}_2 = \{(P, \Omega) \mid \Omega = \Omega_1, \Omega_2$ with $\mathsf{proc}(\_, [\![P]\!]) \Longrightarrow_{un} \Omega_2\}$. We show that $\mathcal{R}_1 \cup \mathcal{R}_2$ is a weak bisimulation by case analysis on P and the possible labelled transitions.

Case: $\mathbf{0}$

There are no labelled transitions in the π-calculus process and no actions in the configuration, which are trivially in the relation $\mathcal{R}_1$.

Case: $\overline{c}\langle a \rangle$

The only labelled transition in $\overline{c}\langle a \rangle$ is $\overline{c}\langle a \rangle \xrightarrow{\overline{c}\langle a \rangle} \mathbf{0}$. By definition of the encoding of $\overline{c}\langle a \rangle$ we have that $\Omega, \mathsf{proc}(\_, [\![\overline{c}\langle a \rangle]\!]) \overset{\overline{c}\langle a \rangle}{\Longrightarrow} \Omega'$ such that Ω' satisfies the defining clause for $\mathcal{R}_1$ – specifically $(\mathbf{0}, \Omega') \in \mathcal{R}_1$. Conversely, there are no available immediate moves from the configuration above. From configurations Ω from $(\overline{c}\langle a \rangle, \Omega) \in \mathcal{R}_2$ we need only consider the one which exposes the output action (all others have no moves that need to be matched), which is then matched immediately by $\overline{c}\langle a \rangle$.

Case: $c(x).P_1$

The only labelled transition in $c(x).P_1$ is $c(x).P_1 \xrightarrow{c(a)} [a/x]\, P_1$. By definition of the encoding we have that $\Omega, \mathsf{proc}(\_, [\![c(x).P]\!]) \overset{\overline{c}\langle a \rangle}{\Longrightarrow} \Omega'$, since Ω can contain a in the buffer for c, such that Ω' satisfies the defining clause for $\mathcal{R}_1$ – specifically $([a/x]\, P_1, \Omega') \in \mathcal{R}_1$. Conversely, there are no available immediate moves from the configuration above. From configurations Ω from $(c(x).P_1, \Omega) \in \mathcal{R}_2$ we need only consider the one which exposes the input action (all others have no moves that need to be matched), which is then matched immediately by $c(a)$ in the π-calculus process.

Case: $P_1 \mid P_2$

There are 3 possible labelled transitions from $P_1 \mid P_2$: $P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2$, with $P_1 \xrightarrow{\alpha} P_1'$; the symmetric case, $P_1 \mid P_2 \xrightarrow{\alpha} P_1 \mid P_2'$, with $P_2 \xrightarrow{\alpha} P_2'$; and, $P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'$.
In the first case, we have that $\Omega, \mathsf{proc}(\_, [\![P_1 \mid P_2]\!]) \overset{\alpha}{\Longrightarrow} \Omega'$ where $\Omega' = \Omega_1, \mathsf{proc}(\_, [\![P_1']\!]), \mathsf{proc}(\_, [\![P_2]\!])$ and $(P_1' \mid P_2, \Omega') \in \mathcal{R}_2$. The symmetric case is identical. The last case must arise due to a synchronisation between a top-level input in P_1 or P_2 and an output in P_2 or P_1, respectively, and so $\Omega, \mathsf{proc}(\_, [\![P_1 \mid P_2]\!]) \overset{\tau}{\Longrightarrow} \Omega'$ where $\Omega' = \Omega_1, \mathsf{proc}(\_, [\![P_1']\!]), \mathsf{proc}(\_, [\![P_2]\!])$ and $(P_1' \mid P_2', \Omega') \in \mathcal{R}_2$.

The moves from configurations are matched straightforwardly as in the previous cases.

Case: $(\nu x)P_1$

The possible transitions are of the form $(\nu x)P_1 \xrightarrow{\alpha} (\nu x)P_2$ with $x \notin \mathit{subj}(\alpha)$. By the definition of the encoding and of observable transitions we have that $\Omega, \mathsf{proc}(\_, \llbracket(\nu x)P_1\rrbracket) \stackrel{\alpha}{\Longrightarrow} \Omega'$ with $\Omega' = \Omega_1, \mathsf{proc}(\_, \llbracket P_2\rrbracket)$, noting that $((\nu x)P_2, \Omega') \in \mathcal{R}_2$. The symmetric cases involving transitions from the configurations that need to be matched in the processes are straightforward.

Case: $!P_1$

The only possible transition is $!P_1 \xrightarrow{\tau} P_1 \mid !P_1$. By the definition of the encoding we have that $\Omega, \mathsf{proc}(\_, \llbracket !P_1\rrbracket) \Longrightarrow_{un} \Omega'$ with $\Omega' = \Omega_1, \mathsf{proc}(\_, \llbracket P_1\rrbracket), \mathsf{proc}(\_, \llbracket !P_1\rrbracket)$ and $(P_1 \mid !P_1, \Omega') \in \mathcal{R}_2$. Conversely, any configuration paired with $!P_1$ in $\mathcal{R}_1 \cup \mathcal{R}_2$ either exhibits no observable actions or exhibits observables arising from $\llbracket P_1\rrbracket$, which are straightforwardly matched by a corresponding observable after the τ transition above. ◀

C Additional Definitions and Proofs for Section 4

The full definition of the translation of $\mathsf{SILL_S}$ processes is given below:

$$
\begin{array}{lcl}
\llbracket x_{\mathsf{L}} \leftarrow p \leftarrow \overline{\ulcorner y_{\mathsf{L}_i}, y_{\mathsf{S}_i}\urcorner}, \overline{w_{\mathsf{S}_j}};\ Q_{x_{\mathsf{L}}}\rrbracket^{\textsc{Spawn}_{\mathrm{LL}}} & = & \nu z\,(\overline{p}\langle \overline{y_{\mathsf{L}_i}}, \overline{y_{\mathsf{S}_i}}, \overline{w_{\mathsf{S}_j}}, z\rangle \mid z(x_{\mathsf{L}}, x_{\mathsf{S}}).\llbracket[\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner/x_{\mathsf{L}}]\,Q_{x_{\mathsf{L}}}\rrbracket) \\
\llbracket x_{\mathsf{S}} \leftarrow p \leftarrow \overline{y_{\mathsf{S}_i}};\ Q_{x_{\mathsf{S}}}\rrbracket^{\textsc{Spawn}_{\mathrm{LL/SS}}} & = & \nu z\,(\overline{p}\langle \overline{y_{\mathsf{S}_i}}, z\rangle \mid z(x_{\mathsf{S}}).\llbracket Q_{x_{\mathsf{S}}}\rrbracket) \\
\llbracket y_{\mathsf{L}} \leftarrow \mathsf{acquire}\ x_{\mathsf{S}};\ Q_{y_{\mathsf{L}}}\rrbracket^{\uparrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{L}} & = & \nu y_{\mathsf{L}}, w\,(\overline{x_{\mathsf{S}}}\langle y_{\mathsf{L}}, x_{\mathsf{S}}, w\rangle \mid w(\ ).\llbracket[\ulcorner y_{\mathsf{L}}, x_{\mathsf{S}}\urcorner/y_{\mathsf{L}}]\,Q_{y_{\mathsf{L}}}\rrbracket) \\
\llbracket y_{\mathsf{L}} \leftarrow \mathsf{accept}\ x_{\mathsf{S}};\ P_{y_{\mathsf{L}}}\rrbracket^{\uparrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{R}} & = & x_{\mathsf{S}}(y_{\mathsf{L}}, y_{\mathsf{S}}, w).(\overline{w}\langle\ \rangle \mid [\ulcorner y_{\mathsf{L}}, y_{\mathsf{S}}\urcorner/y_{\mathsf{L}}]\,\llbracket P_{y_{\mathsf{L}}}\rrbracket) \\
\llbracket y_{\mathsf{S}} \leftarrow \mathsf{release}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner;\ Q_{y_{\mathsf{S}}}\rrbracket^{\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{L}} & = & x_{\mathsf{L}}(y_{\mathsf{S}}).\llbracket Q_{y_{\mathsf{S}}}\rrbracket \\
\llbracket x_{\mathsf{S}} \leftarrow \mathsf{detach}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner;\ P\rrbracket^{\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{R}} & = & \overline{x_{\mathsf{L}}}\langle x_{\mathsf{S}}\rangle \mid \llbracket P\rrbracket \\
\llbracket \mathsf{wait}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner;\ Q\rrbracket^{\mathbf{1}\mathrm{L}} & = & x_{\mathsf{L}}(\ ).\llbracket Q\rrbracket \\
\llbracket \mathsf{close}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner\rrbracket^{\mathbf{1}\mathrm{R}} & = & \overline{x_{\mathsf{L}}}\langle\ \rangle \\
\llbracket y_{\mathsf{L}} \leftarrow \mathsf{recv}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner;\ P_{y_{\mathsf{L}}}\rrbracket^{\otimes\mathrm{L}/\multimap\mathrm{R}} & = & x_{\mathsf{L}}(y_{\mathsf{L}}, y_{\mathsf{S}}, z_{\mathsf{L}}, z_{\mathsf{S}}).\llbracket[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner, \ulcorner y_{\mathsf{L}}, y_{\mathsf{S}}\urcorner/y_{\mathsf{L}}]P_{y_{\mathsf{L}}}\rrbracket \\
\llbracket \mathsf{send}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner\ \ulcorner y_{\mathsf{L}}, y_{\mathsf{S}}\urcorner;\ P\rrbracket^{\otimes\mathrm{R}/\multimap\mathrm{L}} & = & \nu z_{\mathsf{L}}\,(\overline{x_{\mathsf{L}}}\langle y_{\mathsf{L}}, y_{\mathsf{S}}, z_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid \llbracket[\ulcorner z_{\mathsf{L}}, x_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\,P\rrbracket) \\
\llbracket y_{\mathsf{S}} \leftarrow \mathsf{recv}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner;\ P_{y_{\mathsf{S}}}\rrbracket^{\exists\mathrm{L}/\Pi\mathrm{R}} & = & x_{\mathsf{L}}(y_{\mathsf{S}}, z_{\mathsf{L}}, z_{\mathsf{S}}).\llbracket[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\,P_{y_{\mathsf{S}}}\rrbracket \\
\llbracket \mathsf{send}\ \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner\ y_{\mathsf{S}};\ P\rrbracket^{\exists\mathrm{R}/\Pi\mathrm{L}} & = & \nu z_{\mathsf{L}}\,(\overline{x_{\mathsf{L}}}\langle y_{\mathsf{S}}, z_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid \llbracket[\ulcorner z_{\mathsf{L}}, x_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\,P\rrbracket) \\
\llbracket \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner.\mathsf{case}(P, Q)\rrbracket^{\oplus\mathrm{L}/\&\mathrm{R}} & = & \nu y_{\mathsf{inl}}, y_{\mathsf{inr}}\,(\overline{x_{\mathsf{L}}}\langle y_{\mathsf{inl}}, y_{\mathsf{inr}}\rangle \mid \\
 & & \quad y_{\mathsf{inl}}(z_{\mathsf{L}}, z_{\mathsf{S}}).\llbracket[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\,P\rrbracket \mid \\
 & & \quad y_{\mathsf{inr}}(z_{\mathsf{L}}, z_{\mathsf{S}}).\llbracket[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\,Q\rrbracket) \\
\llbracket \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner.\mathsf{inl}; P\rrbracket^{\oplus\mathrm{R}_1/\&\mathrm{L}_1} & = & \nu z_{\mathsf{L}}\,(x_{\mathsf{L}}(y_{\mathsf{inl}}, y_{\mathsf{inr}}).\overline{y_{\mathsf{inl}}}\langle z_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid \llbracket[\ulcorner z_{\mathsf{L}}, x_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\,P\rrbracket) \\
\llbracket \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner.\mathsf{inr}; Q\rrbracket^{\oplus\mathrm{R}_2/\&\mathrm{L}_2} & = & \nu z_{\mathsf{L}}\,(x_{\mathsf{L}}(y_{\mathsf{inl}}, y_{\mathsf{inr}}).\overline{y_{\mathsf{inr}}}\langle z_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid \llbracket[\ulcorner z_{\mathsf{L}}, x_{\mathsf{S}}\urcorner/\ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\,Q\rrbracket)
\end{array}
$$

▶ **Theorem 7** (Operational Correspondence). *Let $\longrightarrow^+$ be the transitive closure of $\longrightarrow$:*

Completeness: *If P is a well-typed, forwarding-free $\mathsf{SILL_S}$ process and $\mathsf{proc}(a, P) \longrightarrow^+ \Omega$ then $\llbracket P\rrbracket \longrightarrow^+ \llbracket\Omega\rrbracket$.*

Soundness: *For all well-typed, forwarding-free $\mathsf{SILL_S}$ configurations Ω such that $\llbracket\Omega\rrbracket \longrightarrow^+ Q$, there exists a configuration Ω' such that $\Omega \longrightarrow^+ \Omega'$ and $Q \Longrightarrow \llbracket\Omega'\rrbracket$.*

Proof. By Theorems 10 and 11. ◀

▶ **Theorem 10** (Operational Completeness). *Let P be a well-typed* $\mathsf{SILL_S}$ *process such that* $\mathsf{proc}(a,\ P) \longrightarrow^{+} \Omega$. *We have that* $[\![P]\!] \longrightarrow^{+} [\![\Omega]\!]$.

Proof. Straightforward induction on the structure of P. We illustrate the case for communication of a shared name.

Case: P is a cut between $\exists_R$ and $\exists_L$

$\mathsf{proc}(a,\ P) \longrightarrow^{+} \mathsf{proc}(x_{\mathsf{L}},\ \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{S}}; P_1),\ \mathsf{proc}(a,\ y_{\mathsf{S}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}; P_2)$
assumption, without loss of generality

$[\![P]\!] = \nu z_{\mathsf{L}}\ (\overline{x_{\mathsf{L}}}\langle y_{\mathsf{S}}, z_{\mathsf{L}}, x_{\mathsf{S}}\rangle \mid [\![[\ulcorner z_{\mathsf{L}}, x_{\mathsf{S}}\urcorner / \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\ P_1]\!]) \mid x_{\mathsf{L}}(y_{\mathsf{S}}, z_{\mathsf{L}}, z_{\mathsf{S}}).[\![[\ulcorner z_{\mathsf{L}}, z_{\mathsf{S}}\urcorner / \ulcorner x_{\mathsf{L}}, x_{\mathsf{S}}\urcorner]\ P_2]\!]$
by definition

$\mathsf{proc}(x_{\mathsf{L}},\ \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{S}}; P_1),\ \mathsf{proc}(a,\ y_{\mathsf{S}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}; P_2) \longrightarrow \mathsf{proc}(z_{\mathsf{L}},\ [z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_1),\ \mathsf{proc}(a,\ [z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_2)$
$= \Omega$ assumption

$[\![P]\!] \longrightarrow^{+} \nu z_{\mathsf{L}}\ [\![[z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_1]\!] \mid [\![[z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_2]\!] \equiv [\![\Omega]\!]$ by reduction semantics

Transitions that happen immediately in P_1 or after the transition above in $[z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_1$ or $[z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_2$ follow straightforwardly by induction.

◀

▶ **Theorem 11** (Operational Soundness). *For all well-typed* $\mathsf{SILL_S}$ *configurations Ω such that* $[\![\Omega]\!] \longrightarrow^{+} Q$, *there exists a configuration Ω' such that $\Omega \longrightarrow^{+} \Omega'$ and $Q \Longrightarrow [\![\Omega']\!]$.*

Proof. Straightforward induction on the structure of the underlying $\mathsf{SILL_S}$ processes. We illustrate the case for communication of a shared name.

Case: $\Omega = \mathsf{proc}(x_{\mathsf{L}},\ \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{S}}; P_1),\ \mathsf{proc}(a,\ y_{\mathsf{S}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}; P_2)$

$[\![\Omega]\!] \longrightarrow \nu x_{\mathsf{L}}, x_{\mathsf{S}}, z_{\mathsf{L}}, a_{\mathsf{L}}, a_{\mathsf{S}}\ [\![[z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_1]\!] \mid [\![[z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_2]\!] = Q$ assumption

$\Omega \longrightarrow \mathsf{proc}(z_{\mathsf{L}},\ [z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_1),\ \mathsf{proc}(a,\ [z_{\mathsf{L}}/x_{\mathsf{L}}]\ P_2) = \Omega'$ by semantics

$Q \equiv [\![\Omega']\!]$ by definition

All subsequent reductions follow by i.h.

◀