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 $\lambda$-calculus we can recover the full range of expressiveness of the untyped $\lambda$-calculus solely by adding a single recursive type $U = U \rightarrow U$. In contrast, in the session-typed $\pi$-calculus, recursion alone is insufficient to recover the untyped $\pi$-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 SILL developed in prior work) are enough to embed the untyped asynchronous $\pi$-calculus via a universal shared session type $U_S$. We show that our encoding of the asynchronous $\pi$-calculus satisfies operational correspondence and preserves observable actions (i.e., processes are weakly bisimilar to their encoding). Moreover, we clarify the expressiveness of SILL by developing an operationally correct encoding of SILL in the asynchronous $\pi$-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, $\pi$-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 $\pi$-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 $\pi$-calculus in expressiveness and the question of whether the full expressiveness of the untyped asynchronous $\pi$-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 $\pi$-calculus. Given our language SILL [3, 4] that supports manifest
sharing, we provide an encoding of the untyped asynchronous π-calculus into SILL$_S$, showing that our encoding satisfies operational correspondence and that π-calculus processes are weakly bisimilar to their SILL$_S$ encodings. To clarify the expressiveness of SILL$_S$, we moreover develop an operationally correct encoding in the other direction, which embeds SILL$_S$ into the asynchronous (polyadic) π-calculus.

Key to our encoding of the untyped asynchronous π-calculus into SILL$_S$ is the representation of a π-calculus channel as a recursive shared session type $U_S$, reminiscent of the encoding of the untyped λ-calculus into the simply-typed λ-calculus via the type $U = U \to U$. 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 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 SILL$_S$;
- Evidence of the instrumental role shared channels take in the expressiveness of session-typed process calculi;
- An operationally correct encoding of SILL$_S$ into the untyped asynchronous polyadic π-calculus.

Paper Structure. Section 2 provides the necessary background on SILL$_S$. Section 3 introduces the encoding of the untyped asynchronous π-calculus into SILL$_S$ and states and proves operational and observational correspondence. Section 4 develops an operationally correct encoding of 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 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

$$\text{queue } A = \&\{\text{enq} : A \to \text{queue } A, \text{deq} : \oplus\{\text{none} : 1, \text{some} : A \otimes \text{queue } A\}\}$$

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 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 SILL$_S$ has been published in [3] for illustration purposes, but without proof.
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 continuation) 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 queue A specified above, we have the following protocol: a process providing a service of type 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.

Linearly restricts session type 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:

\[
\text{queue } A = \uparrow S \& \{\text{enq : } \Pi x : A \rightarrow S \downarrow \text{queue } A, \text{deq : } \uplus \{\text{none : } I \downarrow \text{queue } A, \text{some : } \exists x : A \rightarrow S \downarrow \text{queue } A\}\}
\]

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 queue A 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, SILL\(_5\) 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 queue A, according to Figure 1, assuming that q is of type queue A and x is of type A.

The typing judgments forth and back between them:

\[
A_5 \triangleq \updownarrow^S_A \quad A_6, B_k \triangleq A_6 \otimes B_k | 1 | \oplus \{T-A\} | \exists x : A_5.B_k | A_6 \rightarrow B_k | \Pi x : A_5.B_k | \& \{T-A\} | \downarrow^S_A
\]

The modal operator \(\updownarrow^S_a\) shifting down from the shared to the linear layer is then interpreted as a release (and, dually, detach) and the operator \(\uparrow^S_a\) 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 queue A, 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 dequeuing 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:

\[
\frac{\Gamma, x_k : \uparrow^S_a A_k; \Delta, x_l : A_k \vdash \Sigma Q_{A_k} :: (z_i : C_l)}{(T-\uparrow^S_a L)} \quad \frac{\Gamma \vdash \Sigma P_{A_k} :: (x_k : A_k)}{(T-\downarrow^S_a R)}
\]

The typing judgments \(\Gamma \vdash \Sigma P :: (x_k : A_k)\) and \(\Delta \vdash \Sigma P :: (x_l : A_k)\) 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 \(\Gamma\) (and \(\Delta\)). \(\Gamma\) and \(\Delta\) consist of hypotheses on the typing of shared and linear channels, respectively, where \(\Gamma\) is a structural and \(\Delta\) a linear context. To allow for recursive process definitions, the typing judgment depends on a signature \(\Sigma\) 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 SILL$\Sigma$ 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.

\[
\text{proc}(a, x \leftarrow \text{acquire } a_s; Q_{a_s}), \text{proc}(a_s, x \leftarrow \text{accept } a_s; P_{a_s}) \\
\rightarrow \text{proc}(a, [a/a] Q_{a_s}), \text{proc}(a_s, [a_s/a] P_{a_s}), \text{unavail}(a_s)
\]

\[
\text{proc}(a, x \leftarrow \text{release } a_s; Q_{a_s}), \text{proc}(a_s, x \leftarrow \text{detach } a_s; P_{a_s}), \text{unavail}(a_s) \\
\rightarrow \text{proc}(a, [a_s/x] Q_{a_s}), \text{proc}(a_s, [a/x] P_{a_s})
\]

\textbf{Figure 2}

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 $\pi$-calculus in SILL$\Sigma$

We now detail our encoding of the asynchronous $\pi$-calculus into SILL$\Sigma$, show that it satisfies operational correspondence and that processes are weakly bisimilar to their SILL$\Sigma$ encodings.

#### 3.1 Encoding the Untyped Asynchronous $\pi$-calculus in SILL$\Sigma$

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 $\pi$-calculus they may connect multiple sending and receiving processes, giving rise to non-determinism. For example, the $\pi$-calculus process $c(x).P \mid \tau(a) \mid c(y).Q$, 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 $[a/x] P \mid c(y).Q \text{ or } [a/x] P \mid [c(y)] Q$.

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 $\pi$-calculus. In previous work we have postulated this conjecture by providing an encoding of the untyped asynchronous $\pi$-calculus into SILL$\Sigma$ [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 $\pi$-calculus process by a linear SILL$\Sigma$ process and a $\pi$-calculus channel by a shared SILL$\Sigma$ process. Reminiscent of the encoding of the untyped $\lambda$-calculus into the typed $\lambda$-calculus, we type $\pi$-calculus channels with a universal recursive shared session type $U_s$:

\[
U_s = 1^*_s \& \{ \text{ins} : \Pi x:U_s, \downarrow^*_s U_s, \text{del} : \oplus \{ \text{none} : \downarrow^*_s U_s, \text{some} : \exists x:U_s, \downarrow^*_s U_s \} \}
\]
Similar to the type queue $A_1$ of Section 2, the type $\mathcal{U}$ represents a buffer that stores elements, but with the elements being of type $\mathcal{U}$ themselves and without maintaining any order. Figure 3 shows the processes empty and elem that implement session type $\mathcal{U}$. In SILL$_S$, we declare the type of a defined process $X$ with $X : \{A \leftrightarrow A_1, \ldots, A_n\}$, 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.

\begin{center}
\begin{minipage}{0.7\textwidth}
\begin{verbatim}
empty : {\mathcal{U}}
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' ;
e ← empty

del elem : {\mathcal{U} \leftrightarrow \mathcal{U} \leftrightarrow \mathcal{U}}
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
\end{verbatim}
\end{minipage}
\end{center}

\begin{minipage}{0.3\textwidth}
\begin{verbatim}
nd_pick : {\exists x : \mathcal{U} \downarrow \mathcal{U} \leftarrow \mathcal{U} \leftrightarrow \mathcal{U}}
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 ← release d' ;
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
\end{verbatim}
\end{minipage}

\begin{quote}
\textbf{Figure 3} Processes empty and elem implementing a $\pi$-calculus channel with auxiliary processes (see Appendix A.1 for omitted process bodies).
\end{quote}

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 $x : \mathcal{U}$ (the buffer element at the current position in the sequence) as well as a channel $d : \mathcal{U}$ (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 recursing 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$_S$ 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
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 _SILL₅_ simply amount to “producers” and “consumers” of shared channels of type _Uᵦ_. Any such process can communicate along a π-calculus channel by acquiring the corresponding _SILL₅_ channel of universal type. We are now ready to give the encoding of the untyped asynchronous monadic π-calculus [30, 37] into _SILL₅_. The syntax of the asynchronous π-calculus is [5]:

\[
\begin{align*}
P & \triangleq \hspace{1cm} 0 \mid \tau(a) \mid c(x).P \mid \nu x P \mid P_1 \mid P_2 \mid !P
\end{align*}
\]

0 denotes an inactive process. _τ(a)_ 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 x P_ introduces a new channel _c_ that is bound in _P_. _P_ \mid _P_ denotes parallel composition of _P_ and _P_, 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 \rightarrow P \parallel 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 _SILL₅_, satisfying the typing judgment: _Γ; Γ_B; Γ_Σ; □ \vdash □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 _SILL₅_ processes of type _Uᵦ_, the linear _SILL₅_ 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 _SILL₅_ processes of type 1, since the notion of a non-providing linear process is not present in _SILL₅_. Our current encoding avoids the spurious exchange of _wait_ messages required by type 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 (_Γ_F_) and _bound_ (_Γ_B_) π-calculus channels as well as of channels that are only used _internally_ to the encoding (_Γ_Σ). 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 $\tau(a)$ is implemented by spawning a new linear $\text{SILL}_5$ process $\text{snd}$ of type $\Pi x:U, 1$ with access to the buffer implementing channel $c$. The encoding then sends the channel $a$ to the spawned process $\text{snd}$, waiting for $\text{snd}$ to acquire the buffer $c$, insert $a$, and terminate. The encoding of an input $\nu x.P$ is implemented by spawning a new linear $\text{SILL}_5$ process $\text{poll_recv}$ of type $\exists x:U, 1$ with access to the buffer implementing channel $c$. The encoding then waits for the spawned process $\text{poll_recv}$ to send back a channel and terminate, after which it continues at $P$, substituting the received channel for $x$. Process $\text{poll_recv}$ 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. Name creation ($\nu x P$) simply spawns a new buffer, offering on some fresh name $x$. Parallel ($\langle P_1 \mid P_2 \rangle$) composition is embodied by a spawning of the processes $P_1$ in parallel with the executing process $P_2$. Finally, replication ($\exists! P$) is implemented by a loop that spawns copies of the replicated process.

To make our encoding more tangible, we derive the initial $\text{SILL}_5$ configuration obtained from translating the process $\langle \tau(a) \mid c(x).0 \rangle$ according to the rules in Figure 4:

$$a_s, c_s; \cdots; \exists \Sigma \text{proc}(\_, y_k \leftarrow \text{poll_recv} \leftarrow c_s; z_s \leftarrow \text{recv} y_k; \text{wait} y_k; ),$$

$$\text{proc}(\_, y_k \leftarrow \text{snd} \leftarrow c_s; \text{send} y_k; a_s; \text{wait} y_k; ),$$

$$\text{buf}(a_s \mid y_k \leftarrow \text{accept} a_s; P_{y_k}), \text{buf}(c_s \mid y_k \leftarrow \text{accept} c_s; P_{y_k})$$

To the left, we list the contents of the contexts $\Gamma_x; \Gamma_y; \Gamma_z; \Delta$, to the right the process configuration. For readability we use the short-form $\text{buf}(a \mid P_a)$ to represent a sequence of empty-terminated $\text{elem}$ 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 $\text{SILL}_5$, until it halts in a state that consists of buffers representing the $\pi$-calculus channels, $\text{coin_head}$ processes for any nondeterministic choices made, and $\text{unavail}$ 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 $\pi$-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 $\pi$-calculus! Our encoding is thus reminiscent of the encoding of the untyped asynchronous $\pi$-calculus into an untyped synchronous $\pi$-calculus with bags [5]. In fact, unlike the $\pi$-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 $\text{SILL}_5$ 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 $\pi$-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 \( \rightarrow_r \), is a standard transition between SILL configurations such that: \( \Omega, \text{proc}(d_l, x_0 \leftarrow \text{recv} c_s; Q_{a_0}) \rightarrow \Omega' \), \( \text{proc}(d_l, [a_s/x_0] Q_{a_2}) \), for some \( \Omega, \Gamma_x, \Gamma_y, \Gamma_z, a_s, c_s, \) and \( d_l \) such that \( a_s \in \Gamma_x \cup \Gamma_y, c_s \in \Gamma_x \cup \Gamma_y, \) and \( d_l \in \Gamma_z \).

An administrative transition, written \( \rightarrow_a \), is a transition defined by the standard transition relation between SILL configurations, but excluding a relevant transition. We write \( \Rightarrow_a \) for the reflexive transitive closure of \( \rightarrow_a \), and write \( \Rightarrow \) for \( \Rightarrow_a \rightarrow_r \Rightarrow_a \).

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 \( \text{poll\_recv} \), which synchronizes with the buffer to receive a channel. The parameters of the above definition uniquely identify this synchronization point: process \( \text{poll\_recv} \) is a linear process providing along a linear channel \( d_l \) that is internal to the encoding \( (d_l \in \Gamma_z) \), 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 \( (a_s \in \Gamma_x \cup \Gamma_y \) and \( c_s \in \Gamma_x \cup \Gamma_y \).

Equipped with these two notions of transition, we can establish operational soundness and completeness. Their statements rely on the definition \( \llbracket \text{fn}(P) \rrbracket \), which stand for a configuration of empty buffer processes of the form \( \text{buf}(c_s_1 \mid y_1 \leftarrow \text{accept } c_s_1; Q_{a_1}), \ldots, \text{buf}(c_s_n \mid y_n \leftarrow \text{accept } c_s_n; Q_{a_n}) \), where \( \text{fn}(P) = \{c_1, \ldots, c_n\} \) 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 \( P \rightarrow P' \), there exists \( \Omega_1, \Omega_2 \) such that \( \llbracket \text{fn}(P) \rrbracket, \text{proc}_\langle\cdot, [P]\rangle \Rightarrow_r \Omega_1, \Omega_2 \) or \( \llbracket \text{fn}(P) \rrbracket, \text{proc}_\langle\cdot, [P]\rangle \Rightarrow_a \Omega_1, \Omega_2 \), with \( \llbracket \text{fn}(P') \rrbracket, \text{proc}_\langle\cdot, [P']\rangle \Rightarrow_a \Omega_2 \).

Soundness: For all \( P \) and \( \llbracket \text{fn}(P) \rrbracket, \text{proc}_\langle\cdot, [P]\rangle \Rightarrow_r \Omega \), there exists a \( P', \Omega_1, \Omega_2 \) such that \( P \rightarrow P' \) and \( \Omega_1, \Omega_2 \) and \( \llbracket \text{fn}(P') \rrbracket, \text{proc}_\langle\cdot, [P']\rangle \Rightarrow_a \Omega_2 \).

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., \( !P \rightarrow P \parallel P \)), 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 unavail channels) through the configuration \( \Omega_1 \).

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 \( \text{SILL}_S \), a distinction not present in the \( \pi \)-calculus, where processes are the runtime entities. For instance, parallel composition in \( \text{SILL}_S \) 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 \( \pi \)-calculus in a strong sense, by identifying precisely the transitions in the operational semantics of \( \text{SILL}_S \) that correspond to reductions in the \( \pi \)-calculus processes in a way that is consistent with standard results on the nature of asynchrony of the untyped asynchronous \( \pi \)-calculus.

We now go further and relate observable actions (i.e., labelled transitions) in the \( \pi \)-calculus with their corresponding observables in \( \text{SILL}_S \) configuration rewrites. The key challenge here is to identify what those observables in \( \text{SILL}_S \) are because of the significant differences between the semantic frameworks of the \( \pi \)-calculus and \( \text{SILL}_S \). Whereas the \( \pi \)-calculus adopts an open-world view of observable actions with an unspecified environment (the “ether”), \( \text{SILL}_S \) 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 \( \pi \)-calculus process \( \tau(a) | c(x).P \), 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 \( \tau \)-action, corresponding to the synchronization between these dual actions. Now consider the \( \text{SILL}_S \) encoding of \( \llbracket \tau(a) | c(x).P \rrbracket \). 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 \( \pi \)-calculus side will result in a series of actual computational steps on the \( \text{SILL}_S \) side, affecting the buffers as prescribed by the protocol of type \( \mathcal{U} \). 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 \( \pi \)-calculus behavior with \( \text{SILL}_S \) 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 \( \pi \)-calculus processes and corresponding channel encodings are precisely the inputs and outputs to and from buffers. Conversely, any steps in a \( \text{SILL}_S \) 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 \( \Omega \) we say that there is an unobservable transition from \( \Omega \) to \( \Omega' \), written \( \Omega \rightarrow_{un} \Omega' \), iff \( \Omega \rightarrow \Omega' \) where the transition does not involve any of the two reductions below:

\[
\begin{align*}
\Omega_0, \text{proc}(d, \ x_0 \leftarrow \text{recv} \ c_0 \ : \ P_{x_0}) & \rightarrow \ \Omega'_0, \text{proc}(d, [a_0/x_0]P_{a_0}) \\
\Omega_0, \text{proc}(d, \text{send} \ a, \ e_0 \ : \ P) & \rightarrow \ \Omega'_0, \text{proc}(d, \ P)
\end{align*}
\]
We have that
\[ \Omega = \text{characteristic of asynchrony} \]
would not be the case that we could use queues or stacks as buffers and replicate our bisimulation relation the appropriate way, noting that a weak bisimulation exists a weak bisimulation between a

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

1. \( \Omega \xrightarrow{\pi(a)} \Omega' \) if \( c, a \in \Gamma_X, \Omega = \Omega_1, \text{proc}(d_c, \text{send } c \ a; P), \Omega_2, \) for some \( \Omega_1, P, \Omega_2 \) and \( d_c \in \Gamma_X \) and \( \Omega \xrightarrow{\pi(a)} \Omega' \) with \( \Omega' = \Omega_1', \text{proc}(d_c, P), \Omega_2', \) for some \( \Omega_1', \Omega_2' \).
2. \( \Omega \xrightarrow{(a)\pi(a)} \Omega' \) if \( c \in \Gamma_X, a \in \Gamma_B, \Omega = \Omega_1, \text{proc}(d_c, \text{recv } c; P_x), \Omega_2, \) for some \( \Omega_1, P, \Omega_2 \) and \( d_c \in \Gamma_X \) and \( \Omega \xrightarrow{\pi(a)} \Omega' \) with \( \Omega' = \Omega_1', \text{proc}(d_c, P), \Omega_2', \) for some \( \Omega_1', \Omega_2' \).
3. \( \Omega \xrightarrow{c(a)} \Omega' \) if \( c \in \Gamma_X, a \in \Gamma_B \cup \Gamma_B, \Omega = \Omega_1, \text{proc}(d_c, \text{recv } c; P_x), \Omega_2, \) for some \( \Omega_1, P, \Omega_2 \) and \( d_c \in \Gamma_X \) and \( \Omega \xrightarrow{\pi(a)} \Omega' \) with \( \Omega' = \Omega_1', \text{proc}(d_c, [a/x] P_x), \Omega_2', \) for some \( \Omega_1', \Omega_2' \).
4. \( \Omega \xrightarrow{\tau} \Omega' \) if all of the following:
   1. \( c \in \Gamma_B, a \in \Gamma_X \cup \Gamma_B, \Omega = \Omega_1, \text{proc}(d_c, \text{recv } c; P_x), \Omega_2, \) for some \( \Omega_1, P, \Omega_2 \) and \( d_c \in \Gamma_X \) and \( \Omega \xrightarrow{\pi(a)} \Omega'' \) with \( \Omega'' = \Omega_1'', \text{proc}(d_c, P), \Omega_2'' \), for some \( \Omega_1'', \Omega_2'' \).
   2. \( \Omega'' \xrightarrow{\tau} \Omega'''' \) with \( \Omega'''' = \Omega_1''''', \text{proc}(d_c, \text{recv } c; P_x), \Omega_2'''' \), for some \( \Omega_1''''', Q, \Omega_2'''' \) and \( d_c \in \Gamma_X \) and \( \Omega'' \xrightarrow{\pi(a)} \Omega'''' \) with \( \Omega'''' = \Omega_1''''''', \text{proc}(d_c, [a/x] Q_x), \Omega_2''''' \), for some \( \Omega_1''''''', \Omega_2''''' \).

We write \( \Omega \xrightarrow{\pi(a)} \Omega' \) for \( \Omega \xrightarrow{\tau} \Omega' \) and \( \Omega = \Omega' \) and \( \Omega = \Omega' \). We write \( \Omega \xrightarrow{\pi(a)} \Omega' \) for \( \Omega \xrightarrow{\tau} \Omega' \) and \( \Omega = \Omega' \) and \( \Omega = \Omega' \).

The several observable transitions mirror the \( \pi \)-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 (\( \tau \)-actions) in the \( \pi \)-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 \( \pi \)-calculus process and a SILL configuration.

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

- If \( P \xrightarrow{\alpha} P' \) and \( \alpha \neq \tau \) then \( \Omega \xrightarrow{\alpha} \Omega' \) and \( P' \mathcal{R} \Omega' \).
- If \( P \xrightarrow{\tau} P' \) then \( \Omega \xrightarrow{\tau} \Omega' \) or \( \Omega \xrightarrow{\tau} \Omega' \) and \( P' \mathcal{R} \Omega' \).

plus the symmetric cases. We say that \( P \) is weakly bisimilar to \( \Omega \), 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 \( \pi \)-calculus process. We have that \( P \approx \Omega, \text{proc}(_-,[P]) \), where \( \Omega \) 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 \( \pi \)-calculus and synchronous \( \pi \)-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.
Simulating Shared Session Types in the $\pi$-calculus

In this section, we close the loop and provide an encoding of SILL$S$ process terms into the asynchronous polyadic $\pi$-calculus. The extension to the polyadic $\pi$-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 SILL$S$, we must preserve the shared mode of a channel throughout the translation. To this end, we indicate a linear SILL$S$ channel by a pair $\tau x_i, x_s \rangle$, where the left and right projections yield the linear mode $x_i$ 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 $(x_i \leftarrow p \leftarrow \langle y_i, \bar{w}_i \rangle = P_{x_i, y_i, \bar{w}_i} ) \in \Sigma: ! (p(y_i, \bar{w}_i, z). \nu x_s.(x_i, x_s) (\tau(x_i, x_s)) [([y_i, y_i, y_i, y_s, x_i, x_s, x_s/y_i] P_{x_i, y_i, \bar{w}_i}))$

For each $(x_s \leftarrow p \leftarrow \langle y_s, \bar{w}_s \rangle = P_{x_s, y_s, \bar{w}_s} ) \in \Sigma: ! (p(y_s, z). \nu x_s.(z(x_s)) [([x_s, x_s, x_s, y_s, z] P_{x_s, y_s, \bar{w}_s}))$

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 $x_i$ used by the encoding of the definition body.

**Operational Correspondence.** To establish the operational correctness of our encoding, we consider an asynchronous semantics for 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 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 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:

$$[\cdot] = 0 \quad [\text{proc}(c, P), \Omega] = (\nu c_s, c_i)([P] \mid [\Omega]) \quad [\text{unavail}(c_s), \Omega] = [\Omega]$$
This fact, combined with the restrictive (typed) usage of buffers in our setting allows us to simplify communication actions in the image of the encoding. We note that our encoding is greatly tightened by linearity and by the fact that π-beauxis et al. also consider an encoding of their calculus with bags in the asynchronous sharing discipline and the implementation of nondeterminism when reading from a buffer. Asynchrony actions in the image of the encoding. We note that our encoding is greatly simplified by linearity and by the fact that π-beauxis et al. also consider an encoding of their calculus with bags in the asynchronous sharing discipline and the implementation of nondeterminism when reading from a buffer.

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

We can now show that \( \text{SILL}_s \) 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 \( \Rightarrow^+ \) be the transitive closure of \( \Rightarrow \):

Completeness: If \( P \) is a well-typed, forwarding-free \( \text{SILL}_s \) process and \( \text{proc}(a, P) \Rightarrow^+ \Omega \) then \( \llbracket P \rrbracket \Rightarrow^+ \llbracket \Omega \rrbracket \).

Soundness: For all well-typed, forwarding-free \( \text{SILL}_s \) configurations \( \Omega \) such that \( \llbracket \Omega \rrbracket \Rightarrow^+ Q \), there exists a configuration \( \Omega' \) such that \( \Omega \Rightarrow^+ \Omega' \) and \( Q \Rightarrow \llbracket \Omega' \rrbracket \).

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 \( \text{SILL}_s \) 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 \( \text{SILL}_s \) in the asynchronous π-calculus, modulo the richer syntax of \( \text{SILL}_s \), 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 \( \text{SILL}_s \) 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 $\pi$-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 $\pi$-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 $\text{SILL}_S$, 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 $\pi$-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 $\pi$-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 $\text{SILL}_S$. For instance, their labelled transition system cannot be reasonably used as a labelled transition system for $\text{SILL}_S$ 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 $\pi$-calculus into $\text{SILL}_S$ via a universal shared session type $U_S$, proving its operational and observational correctness. This result shows that the full expressiveness of the untyped asynchronous $\pi$-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 $\pi$-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


A Proofs and Additional Definitions

A.1 Auxiliary processes omitted from Section 3.1

\[ nd\_choice : \{ \oplus\{yes : 1, no : 1\}\} \]

\[ d \leftarrow nd\_choice = \]
\[ c \leftarrow coin\_head ; \]
\[ f \leftarrow coin\_flipper \leftarrow c ; \]
\[ c' \leftarrow acquire c ; \]
\[ case c' of \]
\[ | head \rightarrow c \leftarrow release c' ; \]
\[ d.yes ; \]
\[ wait f ; \]
\[ close d \]
\[ | tail \rightarrow c \leftarrow release c' ; \]
\[ d.no ; \]
\[ wait f ; \]
\[ close d \]

\[ coin = \{ \}
\[ \oplus\{head : L, \}
\[ tail : R\} \]

\[ coin\_head : \{coin\} \]

\[ c \leftarrow coin\_head = \]
\[ c' \leftarrow accept c ; \]
\[ c'.head ; \]
\[ c \leftarrow detach c' ; \]
\[ c \leftarrow coin\_tail \]

\[ coin\_tail : \{coin\} \]

\[ c \leftarrow coin\_tail = \]
\[ c' \leftarrow accept c ; \]
\[ c'.tail ; \]
\[ c \leftarrow detach c' ; \]
\[ c \leftarrow coin\_head \]

\[ coin\_flipper : \{1 \leftarrow coin\} \]

\[ d \leftarrow coin\_flipper \leftarrow c = \]
\[ c' \leftarrow acquire c ; \]
\[ case c' of \]
\[ | head \rightarrow c \leftarrow release c' ; \]
\[ close d \]
\[ | tail \rightarrow c \leftarrow release c' ; \]
\[ close d \]

A.2 Sketch of Encoding Reductions, Complementing Section 3.1

To make our encoding more tangible, we sketch how the process \(\llbracket \pi(a) | c(x).0 \rrbracket\) reduces. Our encoding yields the following initial configuration:

\[ a_0, c_0 : \cdots : b_1 ; \]
\[ \text{proc}(., y_1 \leftarrow \text{poll}_\text{rcv} \leftarrow c_1 : z \leftarrow \text{recv} y_1 ; \text{wait} y_1 ; \cdot), \]
\[ \text{proc}(., y_1 \leftarrow \text{snd} \leftarrow c_1 ; \text{send} y_1 a_0 ; \text{wait} y_1 ; \cdot), \]
\[ \text{buf}(a_0 | y_1 \leftarrow \text{accept} a_0 ; P_{y_1}), \text{buf}(c_1 | y_1 \leftarrow \text{accept} c_1 ; P_{y_1}) \]

To the left, we list the contents of the context \(\Gamma_x; \Gamma_B; \Gamma_{\pi}; \Delta\), to the right the process configuration. For ease of reading we use the short-form \(\text{buf}(a | s_a; Q_0 | b_0, c_1 \cdots b_{n-1}, c_n)\) to represent a sequence of empty-terminated \(\text{elem}\) processes denoting an entire buffer:

\[ \text{buf}(a | s_a; Q_0 | b_0, c_1 \cdots b_{n-1}, c_n) \triangleq \text{proc}(a, [b_0/x, c_1/d](s_a; Q_a))_{\text{elem}}, \]
\[ \text{proc}(c_{n-1}, [b_{n-1}/x, c_n/d](s_{c_{n-1}}; Q_{c_{n-1}})_{\text{elem}}), \]
\[ \text{proc}(c_n; s_{c_n}; Q_{c_n})_{\text{empty}} \]

\[ \text{buf}(a | s_a; Q_0) \triangleq \text{proc}(a, s_a; Q_a)_{\text{empty}} \]

where \(\text{elem}\) and \(\text{empty}\) 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 \(\text{snd}\) and \(\text{coin\_head}\). After a few transitions, we obtain:
\(a_s, c_s; \cdots; b_i, c_i \vdash \Sigma\) \quad \text{proc}(\_, y_k \leftarrow \text{poll} \_\text{rev} \leftarrow c_i; z_k \leftarrow \text{recv} y_k; \text{wait} y_k; \_); \quad \text{proc}(\_, \text{wait} b_i; \_),

\text{snd}(b_i, \text{send} c_i; a_i; [c_i/z_k, a_i/z_k, c_i/y_k]P_{\Sigma, x_k,y_k}), \quad \text{buf}(a_i | y_k \leftarrow \text{accept} a_i; P_{\Sigma, y_k}),

\text{buf}(a_i | z_k \leftarrow \text{recv} c_i; [c_i/y_k]P_{\Sigma, y_k,z_k}), \quad \text{unavail}(b_i), \quad \text{unavail}(c_i)

which pinpoints the state in which the internal \text{snd} process providing along channel \(b_i\) is about to insert (“send”) the channel \(a_i\) into the buffer providing along channel \(c_i\). After a few more transitions, the computation halts in the following configuration:

\(a_s, c_s; \cdots; b_i, c_i, b''_i, d_s, d'_i, d''_i, d'''_i; \vdash \Sigma\)

\text{buf}(a_i | y_k \leftarrow \text{accept} a_i; P_{\Sigma, y_k}), \quad \text{buf}(c_i | y_k \leftarrow \text{accept} c_i; P_{\Sigma, y_k}),

\text{coin} \_\text{head}(d''_i, y_k \leftarrow \text{accept} d''''_i; P_{\Sigma, y_k}), \quad \text{unavail}(b_i), \quad \text{unavail}(c_i),

\text{unavail}(b''_i), \quad \text{unavail}(d'_i), \quad \text{unavail}(d''_i), \quad \text{unavail}(d'''_i)

The above configuration is prototypical for what is “left behind” after computation has come to a halt: buffers representing the free and bound \(\pi\)-calculus channels, \text{coin} \_\text{head} processes for any nondeterministic choices made, and \text{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).** For all \(P \rightarrow P'\), there exists \(\Omega_1, \Omega_2\) such that \([fn(P)], \text{proc}(\_, \lfloor P \rfloor) \Rightarrow \rightarrow_r\ \Omega_1, \Omega_2\) or \([fn(P)], \text{proc}(\_, \lfloor P \rfloor) \Rightarrow \rightarrow_a \Omega_1, \Omega_2\), with \([fn(P')], \text{proc}(\_, \lfloor P' \rfloor) \Rightarrow \rightarrow_a \Omega_1, \Omega_2\).

**Soundness:** For all \(P\) and \([fn(P)], \text{proc}(\_, \lfloor P \rfloor) \Rightarrow \rightarrow_r\ \Omega_1, \Omega_2\), there exists a \(P', \Omega_1, \Omega_2\) such that \(P \rightarrow \rightarrow P'\) and \(\Omega = \Omega_1, \Omega_2\) and \([fn(P')], \text{proc}(\_, \lfloor P' \rfloor) \Rightarrow \rightarrow_a \Omega_1, \Omega_2\).

**Proof.** By Lemma 8 and Lemma 9.

**Theorem 8 (Operational Completeness).** For all \(P \rightarrow P'\), there exists \(\Omega_1, \Omega_2\) such that \([fn(P)], \text{proc}(\_, \lfloor P \rfloor) \Rightarrow \rightarrow_r \Omega_1, \Omega_2\) or \([fn(P)], \text{proc}(\_, \lfloor P \rfloor) \Rightarrow \rightarrow_a \Omega_1, \Omega_2\), with \([fn(P')], \text{proc}(\_, \lfloor P' \rfloor) \Rightarrow \rightarrow_a \Omega_1, \Omega_2\).

**Proof.** By induction on the given \(\pi\)-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:** \(\tau(a) \mid c(x).P_0 \rightarrow [a/x]P_0\)

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:

\(\Omega_1, [fn(P_0) \cup \{c, a\}], \text{proc}(\_, \lfloor [a/x]P_0 \rfloor), \) where \(\Omega_1\) is made up of \text{unv}(\_)

channels and \text{coin} processes.

**Case:** \(P_1 \mid P_2 \rightarrow P'_1 \mid P_2\) with \(P_1 \rightarrow P'_1\)

\([P_1 \mid P_2] = \_ \leftarrow \lfloor P_1 \rfloor : \lfloor P_2 \rfloor\) by definition

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

\([fn(P_1)], \lfloor fn(P_2) \rfloor, \text{proc}(\_, \_ \leftarrow \lfloor P_1 \rfloor : \lfloor P_2 \rfloor) \Rightarrow \rightarrow_a fn(P_1), fn(P_2), \text{proc}(\_, \lfloor P_1 \rfloor), \text{proc}(\_, \lfloor P_2 \rfloor)\) by semantics

\([fn(P_1)], \text{proc}(\_, \lfloor P_1 \rfloor) \Rightarrow \rightarrow_r \Omega_1, \Omega_2\) with \([fn(P')], \text{proc}(\_, \lfloor P'_1 \rfloor) \Rightarrow \rightarrow_a \Omega_2\) (1) by i.h.

\([fn(P_1)], \lfloor fn(P_2) \rfloor, \text{proc}(\_, \lfloor P_1 \rfloor), \text{proc}(\_, \lfloor P_2 \rfloor) \Rightarrow \rightarrow_r \Omega_1, \Omega_2, [fn(P_2) \setminus fn(P')], \text{proc}(\_, \lfloor P_2 \rfloor)\) (2) by semantics

\([P'_1 \mid P_2] = \_ \leftarrow \lfloor P'_1 \rfloor : \lfloor P_2 \rfloor\) by definition
Case: \( P_1 | P_2 \rightarrow P_1 | P'_2 \) with \( P_2 \rightarrow P'_2 \)
As above.

Case: \(!P_1 \rightarrow P_1 | \!P_1\)

\[
[fn(P_1)], \ proc(_, \ ![P_1]) = [fn(P_1)], \ proc(_, \ ![P_1] \leftarrow ![P_1]) \quad \text{by definition}
\]
\[
\rightarrow_a [fn(P_1)], \ proc(_, \ ![P_1]) \quad \text{by semantics}
\]
\[
[fn(P_1)], \ proc(_, \ ![P_1]) = [fn(P_1)], \ proc(_, \ ![P_1] \leftarrow ![P_1]) \quad \text{by definition}
\]
\[
\rightarrow_a [fn(P_1)], \ proc(_, \ ![P_1]) \quad \text{by semantics}
\]

\[\uparrow\]

**Theorem 9 (Operational Soundness).** For all \( P \) and \([fn(P)]\), \( \proc(_, \ ![P]) \rightarrow_r \Omega \), there exists a \( P', \Omega_1, \Omega_2 \) such that \( P \rightarrow P' \) and \( \Omega = \Omega_1, \Omega_2 \) and \([fn(P')]\), \( \proc(_, \ ![P']) \rightarrow_a \Omega_2 \)

**Proof.** By induction on the structure of \( P \).

Case: \( \tau(a) \)
No relevant transition.

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

Case: \( \tau(a) | c(x).P_1 \)
Without loss of generality, consider the relevant transition of \([fn(\tau(a) | c(x).P_1)]\), \( \proc(_, \ ![\tau(a) | c(x).P_1]) \)
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 \( P' = [a/x]P_1 \)
\( \Omega_1 \quad \text{by definition, where } \Omega'_1 \text{ are leftover unavailable and coin processes} \)
\( \Omega''_1 \rightarrow_a \Omega''_2 \quad \text{by definition, where } \Omega''_2 \text{ are leftover unavailable and coin processes} \)

Case: \( P_1 | P_2 \)
We note that \([fn(P_1) \cup fn(P_2)]\), \( \proc(_, \ ![P_1] \cup ![P_2]) \rightarrow_a [fn(P_1) \cup fn(P_2)], \ proc(_, \ ![P_1]), \ proc(_, \ ![P_2]) \).

We consider the following subcases: (1) \([fn(P_1) \cup fn(P_2)], \ proc(_, \ ![P_1]), \ proc(_, \ ![P_2]) \rightarrow_r \Omega_1 \)
\( \Omega_1 \) where the relevant transition arises from \([fn(P_1)], \ proc(_, \ ![P_1]) \) and \( \proc(_, \ ![P_2]) \)
has not transitioned;

(2) \([fn(P_1) \cup fn(P_2)], \ proc(_, \ ![P_1]), \ proc(_, \ ![P_2]) \rightarrow_r \Omega_1 \), where the relevant transition arises from \([fn(P_1)], \ proc(_, \ ![P_1]) \) and \( \proc(_, \ ![P_2]) \) has transitioned.

**Subcase: (1)**
\( \Omega_1 \) is of the form \([fn(P_2) \setminus fn(P_1)], \ proc(_, \ ![P_2]) \Omega'_1 \), where \([fn(P_1)], \ proc(_, \ ![P_1]) \rightarrow_r \Omega'_1 \)
\( P_1 \rightarrow P'_1 \) such that \( \Omega'_1 = \Omega'_1, \Omega''_1 \) and \([fn(P'_1)], \ proc(_, \ ![P'_1]) \rightarrow_a \Omega''_1 \) (i) by i.h.
\( P_1 | P_2 \rightarrow P'_1 | P_2 \) by the reduction semantics
\( \rightarrow_a [fn(P_2) \setminus fn(P_1)], \ proc(_, \ ![P_2]), \ Omega''_1 \) 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 \( \langle [\text{fn}(P_1)], \text{proc}(\_), [\text{!}P_1] \rangle \rightarrow_a \langle [\text{fn}(P_1)], \text{proc}(\_), \text{Rec}(P_1) \rangle \Rightarrow_r \Omega. \)

We have that \( !P_1 \rightarrow P_1 !P_1 \). Then, \( \langle [\text{fn}(P_1)], \text{proc}(\_), [P_1 \mid !P_1] \rangle \rightarrow_a \langle [\text{fn}(P_1)], \text{proc}(\_), \text{Rec}(P_1) \rangle \) which transitions via \( \Rightarrow_r \) to \( \Omega \) by assumption.

**B** Proofs for Section 3.3

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

**Proof.** Let \( \Omega_1 \) be any configuration made up of buffer processes for the free names of \( P \), with arbitrary contents, and \( \mathcal{R}_1 = \{(P, \Omega) \mid \Omega = \Omega_1, \Omega_2, \text{proc}(\_), [P]\} \), for all \( P, \Omega_2 \) where \( \Omega_2 \) is made up of an arbitrary number of unavail and coin processes that are no longer used; and \( \mathcal{R}_2 = \{(P, \Omega) \mid \Omega = \Omega_1, \Omega_2, \text{proc}(\_), [P]\} \Rightarrow_{\text{un}} \Omega_2 \}. \) 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:** 0

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

**Case:** \( \tau(a) \)

The only labelled transition in \( \tau(a) \) is \( \tau(a) \xrightarrow{\tau(a)} 0 \). By definition of the encoding of \( \tau(a) \) we have that \( \Omega, \text{proc}(\_), [\tau(a)] \rangle \xrightarrow{\tau(a)} \Omega' \) such that \( \Omega' \) satisfies the defining clause for \( \mathcal{R}_1 \) - specifically \( (0, \Omega') \in \mathcal{R}_1 \). Conversely, there are no available immediate moves from the configuration above. From configurations \( \Omega \) from \( (\tau(a), \Omega) \in \mathcal{R}_2 \) 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 \( \tau(a) \).

**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 \( \Omega, \text{proc}(\_), [c(x).P] \rangle \xrightarrow{\tau(a)} \Omega' \), since \( \Omega \) can contain a in the buffer for \( c \), such that \( \Omega' \) 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 \( \Omega \) 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 \( \pi \)-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 \( \Omega, \text{proc}(\_), [P_1 \mid P_2] \rangle \xrightarrow{\alpha} \Omega' \) where \( \Omega' = \Omega_1, \text{proc}(\_), [P_1'] \rangle, \text{proc}(\_), [P_2] \rangle \) 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 \( \Omega, \text{proc}(\_), [P_1 \mid P_2] \rangle \xrightarrow{\tau} \Omega' \) where \( \Omega' = \Omega_1, \text{proc}(\_), [P_1'] \rangle, \text{proc}(\_), [P_2] \rangle \) 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 $x \not\in \text{subj}(\alpha)$. By the definition of the encoding and of observable transitions we have that $\Omega, \text{proc}(\_ || [P_2]) \xrightarrow{\alpha} \Omega'$ with $\Omega' = \Omega_1, \text{proc}(\_ || [P_2])$, 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 $!P_1 \xrightarrow{\tau} P_1 || !P_1$. By the definition of the encoding we have that $\Omega, \text{proc}(\_ || [P_2]) \Rightarrow_{\text{un}} \Omega'$ with $\Omega' = \Omega_1, \text{proc}(\_ || [P_2])$, proc(\_, [P_2]) and $(P_1 || !P_1, \Omega') \in \mathcal{R}_2$. 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 [P_1], which are straightforwardly matched by a corresponding observable after the \tau transition above.

C Additional Definitions and Proofs for Section 4

The full definition of the translation of SILL process is given below:

\[
\begin{align*}
[x_1 \leftarrow p \leftarrow y_1; w_5; Q_{z_1}]^{\text{SILLx}} & = \nu z (p(y_1, w_5, z) | z(x_1, z)).[x_1, x_3 \leftarrow Q_{z_1}] \\
[x_3 \leftarrow p \leftarrow y_3; Q_{z_3}]^{\text{SILLx}/\text{SS}} & = \nu z (p(y_3, z) | z(x_3)).[Q_{z_3}] \\
y_1 \leftarrow \text{acquire } x_3; Q_{y_1} & = \nu y_1 w (x_3(y_1, x_3, w) | w(yy_1, x_3)).[Q_{y_1}] \\
y_1 \leftarrow \text{accept } x_3; P_{y_1} & = x_3(y_1, y_3, w).[x_1, x_3 \leftarrow Q_{y_1}] \\
y_3 \leftarrow \text{release } x_1, x_3 \leftarrow; Q_{y_3} & = x_1(y_3, y_3).[Q_{y_3}] \\
y_3 \leftarrow \text{detach } x_1, x_3 \leftarrow; P_{y_3} & = [x_1, x_3 \leftarrow P] \\
[\text{wait } x_1, x_3 \leftarrow; Q] & = \pi(x_1).[Q] \\
[\text{close } x_1, x_3 \leftarrow] & = \pi(x_1, x_3) \\
y_1 \leftarrow \text{recv } x_1, x_3 \leftarrow; P_{y_1} & = \pi(x_1, x_3).[x_1, x_3 \leftarrow P_{y_1}] \\
[\text{send } x_1, x_3 \leftarrow; Q_{y_1} & = \nu x_1 y_1 z_1 (x_1, y_1, z_1).[x_1, x_3 \leftarrow P_{y_1}] \\
y_3 \leftarrow \text{recv } x_1, x_3 \leftarrow; P_{y_3} & = \nu x_1 y_3 z_3 (x_1, y_3, z_3).[x_1, x_3 \leftarrow P_{y_3}] \\
[\text{send } x_1, x_3 \leftarrow; Q_{y_1} & = \nu x_1 y_3 z_3 (x_1, y_3, z_3).[x_1, x_3 \leftarrow P_{y_3}] \\
[\text{case } P, Q] & = \nu x_1 x_3 y_n (x_1, x_3, y_n x_1, x_3).[x_1, x_3 \leftarrow P_{y_1}] \\
[\text{inl } x_1, x_3 \leftarrow; P] & = \nu x_1 x_3 y_n (x_1, x_3, y_n x_1, x_3).[x_1, x_3 \leftarrow P_{y_1}] \\
[\text{inr } x_1, x_3 \leftarrow; Q] & = \nu x_1 x_3 y_n (x_1, x_3, y_n x_1, x_3).[x_1, x_3 \leftarrow P_{y_1}]
\end{align*}
\]

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

Completeness: If $P$ is a well-typed, forwarding-free SILLs process and $\text{proc}(a, P) \Rightarrow^+ \Omega$, then $[P] \Rightarrow^+ [\Omega]$. 

Soundness: For all well-typed, forwarding-free SILLs configurations $\Omega$ such that $[\Omega] \Rightarrow^+ Q$, there exists a configuration $\Omega'$ such that $\Omega \Rightarrow^+ \Omega'$ and $Q \Rightarrow^+ [\Omega']$. 


Proof. By Theorems 10 and 11.

**Theorem 10** (Operational Completeness). Let $P$ be a well-typed SILL process such that $\text{proc}(a, P) \rightarrow^+ \Omega$. We have that $\llbracket P \rrbracket \rightarrow^+ \llbracket \Omega \rrbracket$.

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 $\exists_L$

$\text{proc}(a, P) \rightarrow^+ \text{proc}(x, \text{send } x; P_1), \text{proc}(a, y \leftarrow \text{recv } x; P_2)$

assumption, without loss of generality

$\llbracket P \rrbracket = \nu x_1 (\pi(y_1, z, x) | [\llbracket z/x \rrbracket, x_1 \rightarrow, x_5 \rightarrow] P_1)) | x_4(y_1, z, z) | [\llbracket z/x \rrbracket, x_5 \rightarrow, x_1 \rightarrow] P_2)$

by definition

$\text{proc}(x, \text{send } x; P_1), \text{proc}(a, y \leftarrow \text{recv } x; P_2) \rightarrow \text{proc}(z, [z/x_1] P_1), \text{proc}(a, [z/x_1] P_2)$

assumption

$\llbracket P \rrbracket \rightarrow^+ \nu z_1 ([z/x_1] P_1) | [z/x_1] P_2) \equiv \llbracket \Omega \rrbracket$

by reduction semantics

Transitions that happen immediately in $P_1$ or after the transition above in $[z/x_1] P_1$ or $[z/x_1] P_2$ follow straightforwardly by induction.

**Theorem 11** (Operational Soundness). For all well-typed SILL configurations $\Omega$ such that $\llbracket \Omega \rrbracket \rightarrow^+ Q$, there exists a configuration $\Omega'$ such that $\Omega \rightarrow^+ \Omega'$ and $Q \Rightarrow \llbracket \Omega' \rrbracket$.

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

**Case:** $\Omega = \text{proc}(x, \text{send } x; P_1), \text{proc}(a, y \leftarrow \text{recv } x; P_2)$

$\llbracket \Omega \rrbracket \rightarrow \nu x_1, x_3, z, a, \llbracket [z/x_1] P_1 \rrbracket | [z/x_1] P_2) = Q$

assumption

$\Omega \rightarrow \text{proc}(z, [z/x_1] P_1), \text{proc}(a, [z/x_1] P_2) = \Omega'$

by semantics

$Q \equiv \llbracket \Omega' \rrbracket$

by definition

All subsequent reductions follow by i.h.