Manifest Sharing with Session Types

STEPHANIE BALZER, Carnegie Mellon University, USA
FRANK PFENNING, Carnegie Mellon University, USA

Session-typed languages building on the Curry-Howard isomorphism between linear logic and session-typed communication guarantee session fidelity and deadlock freedom. Unfortunately, these strong guarantees exclude many naturally occurring programming patterns pertaining to shared resources. In this paper, we introduce sharing into a session-typed language where types are stratified into linear and shared layers with modal operators connecting the layers. The resulting language retains session fidelity but not the absence of deadlocks, which can arise from contention for shared processes. We illustrate our language on various examples, such as the dining philosophers problem, and provide a translation of the untyped asynchronous π-calculus into our language.

CCS Concepts: • **Theory of computation → Linear logic**; *Logic and verification*; *Type theory*; • **Software and its engineering → Concurrent programming languages**;

Additional Key Words and Phrases: session types, sharing, Curry-Howard isomorphism

ACM Reference Format:
Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 37 (September 2017), 29 pages.
https://doi.org/10.1145/3110281

1 INTRODUCTION

Session types [Honda 1993; Honda et al. 1998, 2008] prescribe the communication protocols that arise in concurrent programming. Session types and session type libraries have found their ways into various practical programming languages [Dezani-Ciancaglini et al. 2006; Hu et al. 2008; Jespersen et al. 2015; Neykova and Yoshida 2014; Scalas and Yoshida 2016] to express such protocols and ensure their adherence at compile-time. Recently, message-passing concurrency has been put onto a firm logical foundation by exhibiting a Curry-Howard isomorphism between linear logic and session-typed communication [Caires and Pfenning 2010; Caires et al. 2016; Toninho 2015; Wadler 2012]. Programming languages [Griffith and Pfenning 2015; Toninho et al. 2013] based on this isomorphism not only guarantee *session fidelity* (preservation) but also a form of *global progress*, since the process graph forms a tree and is acyclic by construction.

Unfortunately, these strong guarantees preclude programming scenarios that naturally demand sharing, such as shared databases or output devices, or implementations that make use of sharing for performance considerations. The shared channels available through the exponential modality in linear logic have a copying semantics [Caires and Pfenning 2010; Wadler 2012] and therefore do not provide the correct tools in such applications. In this paper, *shared channels* and *shared processes* always refer to mutable resources.

2475-1421/2017/9-ART37
https://doi.org/10.1145/3110281

In this paper, we contribute a session-typed programming language for message-passing concurrency that seamlessly integrates *linear* and *shared* processes. The language allows multiple *aliases* to a shared process to exist, but makes sure that any state-altering communication with such a process only happens once exclusive access to the process has been obtained. At this point, the process becomes linear and can become shared again once it is released, resulting in renunciation of exclusive access. The resulting language retains session fidelity but not the absence of deadlocks, which can arise from contention for shared processes.

A key novelty of our work is to go beyond supporting *acquire-release* as a mere language primitive, but to enrich the type system so that a session type prescribes at which points in the protocol acquisition and release must happen. We generalize the idea of type *stratification* introduced in [Pfenning and Griffith 2015], based on Benton's LNL [1994] and Reed's adjoint logic [2009], and stratify session types into a linear and shared layer and support two *modalities* going back and forth between them. We then interpret the modal operator shifting *down* from the shared to the linear layer as a *release* and the operator shifting *up* from the linear to the shared layer as an *acquire*. As a result, we obtain a type system where any form of synchronization, including the acquisition and release of a shared process, is *manifest* in the session type.

Now that types prescribe the acquisition and release points of shared processes, it is only a small step to making sure that the assumptions by a client attempting to acquire a shared process are actually met. When there is contention for a shared process and one client obtains access at type A and then releases the shared process again, the release must happen at the same type A. This is necessary since the acquire/release cycle is invisible to all other clients. To capture this constraint statically we introduce the notion of an *equi-synchronizing* session type. A session type is equi-synchronizing if it satisfies the invariant that any release restores the session to the same type at which a preceding acquire occurred.

We illustrate our language on various examples, such as producer-consumer queues and dining philosophers, and also demonstrate how nondeterministic choice can be emulated in the resulting language thanks to shared processes. Moreover, we provide an encoding of the untyped asynchronous π-calculus into our language, suggesting that manifest sharing can reclaim the computational power of the untyped π-calculus for session-typed, message-passing concurrency. We plan to confirm this hypothesis as part of future work.

An interesting question is what the meta-theoretic consequences of the introduction of sharing are. The correspondence between linear logic and session-typed communication [Caires and Pfenning 2010; Caires et al. 2016; Toninho 2015; Wadler 2012] established for purely linear session-typed languages seems no longer to hold in its original form. Under this interpretation proofs correspond to processes and cut reduction to communication. With the introduction of sharing, on the other hand, shared channels upon which a process depends may not always be available. Such a computation state corresponds to an *incomplete proof*. Overall, computation is then an interleaving of *proof construction* (acquiring a resource), *proof reduction* (communication), and *proof deconstruction* (releasing a resource). The fact that computation may deadlock is always a failure of proof construction, never communication.

The principal contributions of this paper are:

- the introduction of sharing into session-typed, message-passing, concurrent programming such that sharing is manifest in the type structure via adjoint modalities;
- its elaboration in the programming language SILL_S, resulting in type system, synchronous operational semantics, and proofs of session fidelity (preservation) and a modified form of progress that characterizes possible deadlocks;

- the notion of an equi-synchronizing session type to guarantee session fidelity without the need for run-time type checking when acquiring a process;
- an illustration of the concepts on various examples, including an encoding of the untyped asynchronous π-calculus into our language;
- an extension of the formal system to accommodate an asynchronous dynamics, using a novel transformation derived from logic;
- a prototype implementation of manifest sharing in Concurrent C0.

This paper is structured as follows: Section 2 provides a brief introduction to linear session types. Section 3 introduces manifest sharing. Section 4 illustrates manifest sharing on various examples. Section 5 details the semantics of SILL_S, including preservation and progress. Section 6 gives the encoding of the untyped asynchronous π-calculus into SILL_S. Section 7 provides a brief overview of our implementation. Section 8 summarizes the related work, and Section 9 concludes the paper with a discussion and some remarks about future work. Detailed proofs as well as further examples are available in an extended technical report [Balzer and Pfenning 2017].

2 BACKGROUND

In this section, we provide a short introduction to linear session-typed message-passing concurrency based on the functional language SILL [Griffith and Pfenning 2015; Pfenning and Griffith 2015; Toninho et al. 2013] built on the Curry-Howard isomorphism between intuitionistic linear logic and session-typed concurrency. SILL incorporates processes into a functional core via a linear contextual monad that isolates session-typed concurrency. In this introduction we focus on the linear process layer of SILL, which we extend with manifest sharing in Section 3.

Linear logic [Girard 1987] is a substructural logic that restricts the structural rules of weakening and contraction to propositions of the form $!A$, where $!$ is a so-called exponential modality. As a result, purely linear propositions (that is, propositions without an exponential modality) can be viewed as resources that must be used *exactly once* in a proof. We adopt the intuitionistic version of linear logic, which yields the following sequent [Chang et al. 2003]

$$A_1, \ldots, A_n \vdash A$$

where $A_1, \ldots, A_n$ are linear antecedents and A is the succedent.

Under the Curry-Howard isomorphism for intuitionistic linear logic, propositions are related to session types, proofs to processes, and cut reduction in proofs to communication. Appealing to this correspondence, we assign a process term P to the above judgment and label each hypothesis as well as the conclusion with a *channel*:

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (x : A)$$

The resulting judgment states that process P *provides* a service of session type A along channel x, *using* the services of session types $A_1, \ldots, A_n$ provided along channels $x_1, \ldots, x_n$. The assignment of a channel to the conclusion is convenient because, unlike functions, processes do not evaluate to a value but continue to communicate along their providing channel once they have been created. For the judgment to be well-formed, all the channel names have to be distinct. In particular, the channel name to the right of the turnstile cannot appear to its left. This intuitionistic interpretation of linear logic avoids the need for explicit dualization [Honda 1993; Honda et al. 1998; Wadler 2012] of a session type. Whether a session type is used or provided is determined by its positioning to the left or right, respectively, of the turnstile.

The balance between providing and using a session is established by the two fundamental rules of the sequent calculus that are independent of all logical connectives: cut and identity. Cut states that if P provides service A along channel x, then Q can use the service along the same channel at

the same type. Identity states that, if we are a client of a service A we can always directly provide A.

$$\frac{\Delta \vdash P_x :: (x : A) \quad \Delta', x : A \vdash Q_x :: (z : C)}{\Delta, \Delta' \vdash x \leftarrow P_x \,;\, Q_x :: (z : C)}\ (\text{T-Cut}) \qquad \frac{}{y : A \vdash \mathsf{fwd}\ x\ y :: (x : A)}\ (\text{T-Id})$$

Operationally, the process $x \leftarrow P_x \,;\, Q_x$ creates a globally fresh channel c, spawns a new process $[c/x]P_x$ providing along c, and continues as $[c/x]Q_x$. Conversely, the process $\mathsf{fwd}\ c\ d$ terminates after directly identifying channels c and d. Here, we have adopted the convention to use x, y, and z for channel *variables* and c and d for *channels*. Channels are created at run-time and substituted for channel variables in process terms.

The Curry-Howard correspondence gives each connective of linear logic an interpretation as a session type. This session type prescribes the kind of message that must be sent or received along a channel of this type and at which type the session continues after the exchange. Table 1 provides an overview of the session types arising from linear logic and their operational meaning. We generalize internal $A \oplus B$ and external choice $A \,\&\, B$ to n-ary labeled choices $\oplus\{\overline{l : A}\}$ and $\&\{\overline{l : A}\}$, respectively, where we use the overline-notation to denote a sequence, as is usual. We require external and internal choice to comprise at least one label. Otherwise, there would exist a linear channel without observable interaction along it, which is computationally uninteresting and would also complicate our proofs. Because we adopt the intuitionistic version of linear logic, session types are expressed from the point of view of the provider. Table 1 provides the point of view of the *provider* in the first line of each connective and the one of the *client* in the second line. 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. The process typing rules for the connectives shown in Table 1 can be found in Figure 3. We defer the discussion of the process typing judgment to Section 3.2.

Table 1. Overview of linear session types together with their operational meaning.

Session type		**Process term**		
current	**continuation**	**current**	**continuation**	**Description**
$c : \oplus\{\overline{l : A}\}$	$c : A_h$	$c.l_h \,;\, P$	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
$c : \&\{\overline{l : A}\}$	$c : A_h$	$\mathsf{case}\ c\ \mathsf{of}\ \overline{l \Rightarrow P}$	P_h	provider receives label l_h along c
		$c.l_h \,;\, Q$	Q	client sends label l_h along c
$c : A \otimes B$	$c : B$	$\mathsf{send}\ c\ d \,;\, P$	P	provider sends channel $d : A$ along c
		$y \leftarrow \mathsf{recv}\ c \,;\, Q_y$	$[d/y]\,Q_y$	client receives channel $d : A$ along c
$c : A \multimap B$	$c : B$	$y \leftarrow \mathsf{recv}\ c \,;\, P_y$	$[d/y]\,P_y$	provider receives channel $d : A$ along c
		$\mathsf{send}\ c\ d \,;\, Q$	Q	client sends channel $d : A$ along c
$c : \mathbf{1}$	-	$\mathsf{close}\ c$	-	provider sends "end" along c
		$\mathsf{wait}\ c \,;\, Q$	Q	provider receives "end" along c

As an illustration, we consider a protocol on how to interact with a provider of a queue data structure that contains elements of some variable type A[1]. The protocol is defined by the session type below; we will see variations of it throughout this paper.

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

The session type prescribes that 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 channel, sends it to the client, and recurs. We adopt an *equi-recursive* [Crary et al. 1999] interpretation for recursive session types, which requires recursive session types to be *contractive* [Gay and Hole 2005]. This interpretation guarantees that there are no messages associated with the unfolding of a recursive type.

Figure 1 shows two process definitions *empty* and *elem* implementing the session type queue A. In SILL, 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 a process term satisfying $y_1 : A_1, \ldots, y_n : A_n \vdash P :: (x : A)$. A new process X providing along 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. The channels $y_1, \ldots, y_n$ are passed to X and hence no longer available to Q_x.

```
elem : {queue A ← A, queue A}
q ← elem ← x, t =
  case q of
  | enq → y ← recv q ;          % x : A,  t : qu A,  y : A ⊢ q : qu A
          t.enq ; send t y ;    % x : A,  t : qu A ⊢ q : qu A
          q ← elem ← x, t
  | deq → q.some ;              % x : A,  t : qu A ⊢ q : A ⊗ qu A
          send q x ;            % t : qu A ⊢ q : qu A
          fwd q t
```

```
empty : {queue A}
q ← empty =
  case q of
  | enq → x ← recv q ;          % x : A ⊢ q : qu A
          e ← empty ;           % x : A,  e : qu A ⊢ q : qu A
          q ← elem ← x, e
  | deq → q.none ;              % ⊢ q : 1
          close q
```

Fig. 1. Processes implementing linear session type queue A.

The queue in Figure 1 is implemented as a sequence of *elem* processes, ending in an *empty* process. The recursive process *elem* provides a queue along channel q and uses a channel $x : A$ (the element in front of the queue) as well as a channel $t : \mathsf{queue}\ A$ (the tail of the queue). If it receives an enq label and then a channel y, it simply enqueues y in the tail t. If it receives a deq label it responds with some, followed by the channel x it holds, and then forwards all future communication along q to the tail t. The implementation is highly parallel; in particular, many enqueueing operations can be in flight at the same time. Process *empty*, on the other hand, builds a singleton queue from an element received to be enqueued and returns none and terminates when asked to dequeue. Perhaps the most unusual aspect of writing session-typed programs is that the type of a channel changes during interactions, as already indicated in Table 1. To make this explicit we annotate the code in Figure 1 with the types of all channels at the various points in a process definition. We abbreviate queue A to qu A in those annotations.

[1]Polymorphism is orthogonal to the investigation of this paper, so we adopt it for the examples without formal treatment, which can be found in the literature [Griffith 2016; Pérez et al. 2014].

3 MANIFEST SHARING

In this section, we extend the linear process language of the previous section with the capability to share a process among several clients. The shared channels introduced previously [Caires and Pfenning 2010; Wadler 2012] via the exponential modality in linear logic have a copying semantics and therefore do not allow sharing of mutable resources as pursued in this paper. We first approach the support of shared processes programmatically, by introducing acquire-release as a primitive to our language. We then derive those primitives as modalities from logic in a stratified system of session types. Lastly, we develop the notion of an equi-synchronizing session type.

3.1 A Programming Perspective

In the intuitionistic linear setting of Section 2, processes form a tree at run-time, guaranteeing that a client of a process is the only client of that process. With the introduction of *shared processes* this invariant no longer holds because there may exist multiple clients that refer to the process by a *shared channel*. To uphold session fidelity, communication along a shared channel must only be possible once exclusive access to the process providing along that channel has been obtained. To this end, we impose 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. This view of a process undergoing phases requires an identification of a process with a thread of control, which is extremely natural in intuitionistic linear logic since we can identify a process with the channel along which it provides a service.

We illustrate the programmatic working of the acquire-release primitives on a schematic producer-consumer scenario in Figure 2. For now, we assume for both processes that the shared channel q is provided by a shared process of session type queue A that stores shared elements x of type A. In program code, we typeset shared channels as well as shared session types in red and **bold** font to make them distinguishable from linear channels and session types, which we typeset in black and regular font. Moreover, we assume that the session type queue A recurs rather than terminates upon dequeueing, if the queue is empty, which is more appropriate for a producer-consumer context. In Section 3.2 we clarify how to change the type specification to accommodate these assumptions.

Processes *produce* and *consume* in Figure 2 attempt to communicate with the queue by issuing corresponding acquire and release statements. Process *produce*, for example, issues the statement $q' \leftarrow \text{acquire } q$, which, if successful, yields the queue's linear channel q' along which the process can enqueue the element. Before the process recurs, it releases the now linear queue process providing along q' by issuing $q \leftarrow \text{release } q'$. This yields the queue's shared channel q and gives turn to another producer or consumer.

3.2 A Logic Perspective

Like send and receipt of a message, acquire and release denote synchronization points in the communication between processes. If we were to introduce acquire and release as operational primitives only, session types would no longer accurately prescribe the protocols of communication. To restore the descriptive power of session types, we enrich the type system so that the type of a process dictates at which points in the communication acquire and release must happen.

The key idea in pursuit of this goal is to generalize the notion of type *stratification* introduced in Pfenning and Griffith [2015], based on Benton's LNL [1994] and Reed's adjoint logic [2009], and to stratify session types into a *linear* and *shared* layer. We then connect these layers with *modalities* that go back and forth between them. In Pfenning and Griffith [2015] the modes are U, F,

```
produce : {1 ← A, queue A}
c ← produce ← x, q =
  q′ ← acquire q ;
  q′.enq ;
  send q′ x ;
  q ← release q′ ;
  c ← produce ← x, q
```

```
consume : {1 ← queue A}
c ← consume ← q =
  q′ ← acquire q ;
  q′.deq ;
  case q′ of
  | some → x ← recv q′ ;
           q ← release q′ ;
           c ← consume ← q
  | none → q ← release q′ ;
           c ← consume ← q
```

Fig. 2. Acquire-release primitives illustrated on producer-consumer, programmatically. Shared channels are typeset in red and **bold** font, linear channels in black and regular font. See Section 3.2 for definition of shared session type queue A.

and L for unrestricted, affine, and linear session types, respectively. In this paper, we focus on the interplay between the modes S and L, pertaining to shared and linear session types, respectively. An integration with the remaining modes U and F is straightforward.

The stratification arises from a difference in structural properties that exist for session types at a mode — or propositions at a mode, when viewed through the lens of the Curry-Howard correspondence. For example, shared propositions can be weakened, contracted, and exchanged, whereas linear propositions can only be exchanged. The difference in structural properties entails a hierarchy between modes such that a mode with fewer structural properties is at the bottom. The hierarchy for the modes S and L is:

$$\mathsf{S} > \mathsf{L}$$

The *independence principle* for modes states that proofs of a proposition of a stronger mode (with more structural properties) may not depend on hypotheses of a strictly weaker mode (with fewer structural properties). This is because a client of a stronger proposition may, for example, reuse the proposition, which would implicitly reuse the weaker proposition on which it depends. More technically, on the logical side, cut elimination would fail without this restriction. As a result, we get separate[2] hypothetical judgments for shared and linear processes which, by definition, obey the independence principle:

$$\Gamma \vdash_\Sigma P :: (x_\mathsf{S} : A_\mathsf{S})$$

$$\Gamma;\ \Delta \vdash_\Sigma P :: (x_\mathsf{L} : A_\mathsf{L})$$

The subscripts denote the respective mode of a channel or session type, and the contexts Γ and Δ consist of hypotheses on the typing of shared and linear channels, respectively. The judgments depend on a signature Σ that is populated with all process definitions prior to type-checking, allowing for recursive process definitions.

Given the two layers, we can now define the modality $\downarrow^{\mathsf{S}}_{\mathsf{L}} A_\mathsf{S}$, which shifts a shared proposition (session type) to a linear one, and the modality $\uparrow^{\mathsf{S}}_{\mathsf{L}} A_\mathsf{L}$, which shifts a linear proposition (session type) to a shared one. The resulting strata restricted to session types (propositions) at the modes S and L are:[3]

[2]We could have chosen an combined judgment with a combined context and corresponding projections onto each mode, as employed in [Pfenning and Griffith 2015] for a richer structure of modes. For this paper, we have chosen separate judgments and contexts for clarity of presentation.

[3]Shared counterparts of all the linear connectives can be defined at the shared level as well, but for the purposes of this paper we will keep the shared layer as simple as possible.

$$
\begin{array}{lcl}
A_S & \triangleq & \uparrow^S_L A_L \\
A_L, B_L & \triangleq & A_L \otimes B_L \mid \mathbf{1} \mid \oplus\{\overline{l : A_L}\} \mid \exists x{:}A_S.\, B_L \mid A_L \multimap B_L \mid \Pi x{:}A_S.\, B_L \mid \&\{\overline{l : A_L}\} \mid \downarrow^S_L A_S
\end{array}
$$

We review the new connectives and their operational meaning in Table 2. Together with Table 1, this table defines the connectives supported in SILL_S. Besides the new connectives to accommodate acquire-release, which we discuss in more detail below, we introduce the connectives $\Pi x{:}A_S.\, B_L$ and $\exists x{:}A_S.\, B_L$ to support shared channel input and output, respectively. These connectives of mixed mode are based on the dependent connectives introduced in [Cervesato et al. 2002; Watkins et al. 2002]. Even though at the present stage our language does not make use of the potentially dependent nature of these connectives, we keep the quantifier notation to avoid possible confusion with closely related connectives with a different semantics (e.g., $\supset$ and $\wedge$ in [Griffith and Pfenning 2015; Toninho et al. 2013]). The process typing rules for the connectives of SILL_S, excluding the acquire-release connectives, which we discuss below, can be found in Figure 3.

Table 2. Overview of shared session types together with their operational meaning. See Table 1 for linear connectives.

Session type		**Process term**		
current	**continuation**	**current**	**continuation**	**Description**
$c_L : \exists x{:}A_S.\, B_L$	$c_L : B_L$	send $c_L\ d_S\ ; P$	P	provider sends channel $d_S : A_S$ along c_L
		$y_S \leftarrow$ recv $c_L\ ; Q_{y_S}$	$[d_S/y_S]\, Q_{y_S}$	client receives channel $d_S : A_S$ along c_L
$c_L : \Pi x{:}A_S.\, B_L$	$c_L : B_L$	$y_S \leftarrow$ recv $c_L\ ; P_{y_S}$	$[d_S/y_S]\, P_{y_S}$	provider receives channel $d_S : A_S$ along c_L
		send $c_L\ d_S\ ; Q$	Q	client sends channel $d_S : A_S$ along c_L
$c_L : \downarrow^S_L A_S$	$c_S : A_S$	$c_S \leftarrow$ detach $c_L\ ; P_{x_S}$	$[c_S/x_S]\, P_{x_S}$	provider sends "detach c_S" along c_L
		$x_S \leftarrow$ release $c_L\ ; Q_{x_S}$	$[c_S/x_S]\, Q_{x_S}$	client receives "detach c_S" along c_L
$c_S : \uparrow^S_L A_L$	$c_L : A_L$	$c_L \leftarrow$ acquire $c_S\ ; Q_{x_L}$	$[c_L/x_L]\, Q_{x_L}$	client sends "acquire c_L" along c_S
		$x_L \leftarrow$ accept $c_S\ ; P_{x_L}$	$[c_L/x_L]\, P_{x_L}$	provider receives "acquire c_L" along c_S

We are now in a position to define the typing of the acquire-release discipline outlined in the previous section. In particular, we must determine what the types of the channels should be to which acquire and release are applied. Observing that an acquire transforms a shared channel into a linear one, the natural choice is to type the shared channel of an acquire with the modality $\uparrow^S_L A_L$. Analogously, the linear channel of a release should be typed with the modality $\downarrow^S_L A_S$ as it transforms a linear channel into a shared one. Because we adopt an intuitionistic formulation, which avoids the need for explicit dualization of a session type, we get both a left and right rule for each primitive. The notions of acquire and release are naturally formulated from the point of view of a client, so we use those terms in the left rules. For the right rules, we use the terms *accept* and *detach* with the meaning that an accept accepts an acquire and a detach initiates a release. We review each pair of rules in turn, along with their operational semantics:

The typing of the pair acquire-accept is defined by the following rules:

$$
\frac{\Gamma, x_S : \uparrow^S_L A_L;\ \Delta, x_L : A_L \vdash_\Sigma Q_{x_L} :: (z_L : C_L)}{\Gamma, x_S : \uparrow^S_L A_L;\ \Delta \vdash_\Sigma x_L \leftarrow \text{acquire } x_S\ ; Q_{x_L} :: (z_L : C_L)}\ (\text{T-}\uparrow^S_L\text{L}) \qquad \frac{\Gamma;\ \cdot \vdash_\Sigma P_{x_L} :: (x_L : A_L)}{\Gamma \vdash_\Sigma x_L \leftarrow \text{accept } x_S\ ; P_{x_L} :: (x_S : \uparrow^S_L A_L)}\ (\text{T-}\uparrow^S_L\text{R})
$$

An acquire is applied to the shared channel x_S along which the shared process offers and yields a linear channel x_L, when successful. The shared channel x_S is still available to the continuation Q_{x_L}. By accepting an acquire request by a client along its shared channel x_S, a shared process transitions to a linear process, now offering along a linear channel x_L. Since the independence principle forbids a shared process to depend on linear channels, the now linear process starts out with an empty linear context.

$$\frac{}{\Gamma;\, y_{\mathsf{L}} : A_{\mathsf{L}} \vdash_\Sigma \mathsf{fwd}\ x_{\mathsf{L}}\ y_{\mathsf{L}} :: (x_{\mathsf{L}} : A_{\mathsf{L}})}\ (\text{T-ID}_{\mathsf{L}}) \qquad \frac{\hat{A} \leq A_{\mathsf{S}}}{\Gamma,\, y_{\mathsf{S}} : \hat{A} \vdash_\Sigma \mathsf{fwd}\ x_{\mathsf{S}}\ y_{\mathsf{S}} :: (x_{\mathsf{S}} : A_{\mathsf{S}})}\ (\text{T-ID}_{\mathsf{S}})$$

$$\frac{\Gamma = \overline{w_{\mathsf{S}} : \hat{B}} \quad \overline{\hat{B}} \leq \overline{B_{\mathsf{S}}} \quad \Delta = \overline{y_{\mathsf{L}} : B_{\mathsf{L}}} \quad (x'_{\mathsf{L}} : A_{\mathsf{L}} \leftarrow X_{\mathsf{L}} \leftarrow \overline{y_{\mathsf{L}}' : B_{\mathsf{L}}},\, \overline{w_{\mathsf{S}}' : B_{\mathsf{S}}} = P_{x'_{\mathsf{L}}, \overline{y_{\mathsf{L}}'}, \overline{w_{\mathsf{S}}'}}) \in \Sigma \quad \Gamma, \Gamma';\, \Delta', x_{\mathsf{L}} : A_{\mathsf{L}} \vdash_\Sigma Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma, \Gamma';\, \Delta, \Delta' \vdash_\Sigma x_{\mathsf{L}} \leftarrow X_{\mathsf{L}} \leftarrow \overline{y_{\mathsf{L}}},\, \overline{w_{\mathsf{S}}};\ Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-SPAWN}_{\mathsf{LL}})$$

$$\frac{\Gamma = \overline{y_{\mathsf{S}} : \hat{B}} \qquad \overline{\hat{B}} \leq \overline{B} \qquad (x'_{\mathsf{S}} : A_{\mathsf{S}} \leftarrow X_{\mathsf{S}} \leftarrow \overline{y_{\mathsf{S}}' : B} = P_{x'_{\mathsf{S}}, \overline{y_{\mathsf{S}}'}}) \in \Sigma \qquad \Gamma, \Gamma', x_{\mathsf{S}} : A_{\mathsf{S}};\, \Delta \vdash_\Sigma Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma, \Gamma';\, \Delta \vdash_\Sigma x_{\mathsf{S}} \leftarrow X_{\mathsf{S}} \leftarrow \overline{y_{\mathsf{S}}};\ Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-SPAWN}_{\mathsf{LS}})$$

$$\frac{\Gamma = \overline{y_{\mathsf{S}} : \hat{B}} \qquad \overline{\hat{B}} \leq \overline{B} \qquad (x'_{\mathsf{S}} : A_{\mathsf{S}} \leftarrow X_{\mathsf{S}} \leftarrow \overline{y_{\mathsf{S}}' : B} = P_{x'_{\mathsf{S}}, \overline{y_{\mathsf{S}}'}}) \in \Sigma \qquad \Gamma, \Gamma', x_{\mathsf{S}} : A_{\mathsf{S}} \vdash_\Sigma Q_{x_{\mathsf{S}}} :: (z_{\mathsf{S}} : C_{\mathsf{S}})}{\Gamma, \Gamma' \vdash_\Sigma x_{\mathsf{S}} \leftarrow X_{\mathsf{S}} \leftarrow \overline{y_{\mathsf{S}}};\ Q_{x_{\mathsf{S}}} :: (z_{\mathsf{S}} : C_{\mathsf{S}})}\ (\text{T-SPAWN}_{\mathsf{SS}})$$

$$\frac{\Gamma;\, \Delta \vdash_\Sigma Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma;\, \Delta, x_{\mathsf{L}} : \mathbf{1} \vdash_\Sigma \mathsf{wait}\ x_{\mathsf{L}}\,;\, Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-}\mathbf{1}_{\mathsf{L}}) \qquad \frac{}{\Gamma;\, \cdot \vdash_\Sigma \mathsf{close}\ x_{\mathsf{L}} :: (x_{\mathsf{L}} : \mathbf{1})}\ (\text{T-}\mathbf{1}_{\mathsf{R}})$$

$$\frac{\Gamma;\, \Delta, x_{\mathsf{L}} : B_{\mathsf{L}}, y_{\mathsf{L}} : A_{\mathsf{L}} \vdash_\Sigma Q_{y_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma;\, \Delta, x_{\mathsf{L}} : A_{\mathsf{L}} \otimes B_{\mathsf{L}} \vdash_\Sigma y_{\mathsf{L}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\,;\, Q_{y_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-}\otimes_{\mathsf{L}}) \qquad \frac{\Gamma;\, \Delta \vdash_\Sigma P :: (x_{\mathsf{L}} : B_{\mathsf{L}})}{\Gamma;\, \Delta, y_{\mathsf{L}} : A_{\mathsf{L}} \vdash_\Sigma \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{L}}\,;\, P :: (x_{\mathsf{L}} : A_{\mathsf{L}} \otimes B_{\mathsf{L}})}\ (\text{T-}\otimes_{\mathsf{R}})$$

$$\frac{\Gamma, y_{\mathsf{S}} : A_{\mathsf{S}};\, \Delta, x_{\mathsf{L}} : B_{\mathsf{L}} \vdash_\Sigma Q_{y_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma;\, \Delta, x_{\mathsf{L}} : (\exists x{:}A_{\mathsf{S}}.\, B_{\mathsf{L}}) \vdash_\Sigma y_{\mathsf{S}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\,;\, Q_{y_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-}\exists_{\mathsf{L}}) \qquad \frac{\hat{A} \leq A_{\mathsf{S}} \qquad \Gamma, y_{\mathsf{S}} : \hat{A};\, \Delta \vdash_\Sigma P :: (x_{\mathsf{L}} : B_{\mathsf{L}})}{\Gamma, y_{\mathsf{S}} : \hat{A};\, \Delta \vdash_\Sigma \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{S}}\,;\, P :: (x_{\mathsf{L}} : (\exists x{:}A_{\mathsf{S}}.\, B_{\mathsf{L}}))}\ (\text{T-}\exists_{\mathsf{R}})$$

$$\frac{\Gamma;\, \Delta, x_{\mathsf{L}} : B_{\mathsf{L}} \vdash_\Sigma Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma;\, \Delta, x_{\mathsf{L}} : A_{\mathsf{L}} \multimap B_{\mathsf{L}}, y_{\mathsf{L}} : A_{\mathsf{L}} \vdash_\Sigma \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{L}}\,;\, Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-}\multimap_{\mathsf{L}}) \qquad \frac{\Gamma;\, \Delta, y_{\mathsf{L}} : A_{\mathsf{L}} \vdash_\Sigma P_{y_{\mathsf{L}}} :: (x_{\mathsf{L}} : B_{\mathsf{L}})}{\Gamma;\, \Delta \vdash_\Sigma y_{\mathsf{L}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\,;\, P_{y_{\mathsf{L}}} :: (x_{\mathsf{L}} : A_{\mathsf{L}} \multimap B_{\mathsf{L}})}\ (\text{T-}\multimap_{\mathsf{R}})$$

$$\frac{\hat{A} \leq A_{\mathsf{S}} \qquad \Gamma, y_{\mathsf{S}} : \hat{A};\, \Delta, x_{\mathsf{L}} : B_{\mathsf{L}} \vdash_\Sigma Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma, y_{\mathsf{S}} : \hat{A};\, \Delta, x_{\mathsf{L}} : (\Pi x{:}A_{\mathsf{S}}.\, B_{\mathsf{L}}) \vdash_\Sigma \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{S}}\,;\, Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-}\Pi_{\mathsf{L}}) \qquad \frac{\Gamma, y_{\mathsf{S}} : A_{\mathsf{S}};\, \Delta \vdash_\Sigma P_{y_{\mathsf{S}}} :: (x_{\mathsf{L}} : B_{\mathsf{L}})}{\Gamma;\, \Delta \vdash_\Sigma y_{\mathsf{S}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\,;\, P_{y_{\mathsf{S}}} :: (x_{\mathsf{L}} : (\Pi x{:}A_{\mathsf{S}}.\, B_{\mathsf{L}}))}\ (\text{T-}\Pi_{\mathsf{R}})$$

$$\frac{(\forall i)\ \ \Gamma;\, \Delta, x_{\mathsf{L}} : A_{\mathsf{L}_i} \vdash_\Sigma Q_i :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma;\, \Delta, x_{\mathsf{L}} : \oplus\{\overline{l : A_{\mathsf{L}}}\} \vdash_\Sigma \mathsf{case}\ x_{\mathsf{L}}\ \mathsf{of}\ \overline{l \Rightarrow Q} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-}\oplus_{\mathsf{L}}) \qquad \frac{\Gamma;\, \Delta \vdash_\Sigma P :: (x_{\mathsf{L}} : A_{\mathsf{L}\,h})}{\Gamma;\, \Delta \vdash_\Sigma x_{\mathsf{L}}.l_h\,;\, P :: (x_{\mathsf{L}} : \oplus\{\overline{l : A_{\mathsf{L}}}\})}\ (\text{T-}\oplus_{\mathsf{R}})$$

$$\frac{\Gamma;\, \Delta, x_{\mathsf{L}} : A_{\mathsf{L}\,h} \vdash_\Sigma Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma;\, \Delta, x_{\mathsf{L}} : \&\{\overline{l : A_{\mathsf{L}}}\} \vdash_\Sigma x_{\mathsf{L}}.l_h\,;\, Q :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ (\text{T-}\&_{\mathsf{L}}) \qquad \frac{(\forall i)\ \ \Gamma;\, \Delta \vdash_\Sigma P_i :: (x_{\mathsf{L}} : A_{\mathsf{L}_i})}{\Gamma;\, \Delta \vdash_\Sigma \mathsf{case}\ x_{\mathsf{L}}\ \mathsf{of}\ \overline{l \Rightarrow P} :: (x_{\mathsf{L}} : \&\{\overline{l : A_{\mathsf{L}}}\})}\ (\text{T-}\&_{\mathsf{R}})$$

Fig. 3. Remaining process typing rules not shown inline. For the meaning of $\hat{A}$ and $\hat{B}$ see Section 3.3.

Operationally, we capture the dynamics of SILL$_{\mathsf{S}}$ by *multiset rewriting rules* [Cervesato and Scedrov 2009]. A multiset rewriting rule is generally of the form $S_1, \ldots, S_n \longrightarrow T_1, \ldots, T_m$ and denotes a transition from $S_1, \ldots, S_n$ to $T_1, \ldots, T_m$ where each S_i and T_j is a formula capturing some aspect of the current state of the computation. In our setting, we use the rules to capture a

$$
\begin{array}{ll}
(\text{D-Id}_\text{L}) & \text{proc}(a_\text{L}, \text{fwd } a_\text{L}\ b_\text{L}) \longrightarrow a_\text{L} = b_\text{L},\ a_\text{S} = b_\text{S} \\
(\text{D-Id}_\text{S}) & \text{proc}(a_\text{S}, \text{fwd } a_\text{S}\ b_\text{S}) \longrightarrow \text{unavail}(a_\text{S}),\ a_\text{S} = b_\text{S} \\
(\text{D-Spawn}_\text{LL}) & \text{proc}(a_\text{L}, x_\text{L} \leftarrow X_\text{L} \leftarrow \overline{b};\ Q_{x_\text{L}}) \longrightarrow \text{proc}(a_\text{L}, [b_\text{L}/x_\text{L}]Q_{x_\text{L}}),\ \text{proc}(b_\text{L}, [b_\text{L}/x'_\text{L},\ \overline{b}/\overline{y}]P_{x'_\text{L},\overline{y}}), \\
 & \qquad \text{unavail}(b_\text{S}) \\
 & \qquad\qquad \text{for } x'_\text{L} : A_\text{L} \leftarrow X_\text{L} \leftarrow \overline{y : B} = P_{x'_\text{L},\overline{y}} \in \Sigma \text{ and } b \textit{ fresh} \\
(\text{D-Spawn}_\text{LS}) & \text{proc}(a_\text{L}, x_\text{S} \leftarrow X_\text{S} \leftarrow \overline{b};\ Q_{x_\text{S}}) \longrightarrow \text{proc}(a_\text{L}, [b_\text{S}/x_\text{S}]Q_{x_\text{S}}),\ \text{proc}(b_\text{S}, [b_\text{S}/x'_\text{S},\ \overline{b}/\overline{y}]P_{x'_\text{S},\overline{y}}) \\
 & \qquad\qquad \text{for } x'_\text{S} : A_\text{S} \leftarrow X_\text{S} \leftarrow \overline{y : B} = P_{x'_\text{S},\overline{y}} \in \Sigma \text{ and } b \textit{ fresh} \\
(\text{D-Spawn}_\text{SS}) & \text{proc}(a_\text{S}, x_\text{S} \leftarrow X_\text{S} \leftarrow \overline{b};\ Q_{x_\text{S}}) \longrightarrow \text{proc}(a_\text{S}, [b_\text{S}/x_\text{S}]Q_{x_\text{S}}),\ \text{proc}(b_\text{S}, [b_\text{S}/x'_\text{S},\ \overline{b}/\overline{y}]P_{x'_\text{S},\overline{y}}) \\
 & \qquad\qquad \text{for } x'_\text{S} : A_\text{S} \leftarrow X_\text{S} \leftarrow \overline{y : B} = P_{x'_\text{S},\overline{y}} \in \Sigma \text{ and } b \textit{ fresh} \\
(\text{D-}\mathbf{1}) & \text{proc}(c_\text{L}, \text{wait } a_\text{L}\,; Q),\ \text{proc}(a_\text{L}, \text{close } a_\text{L}) \longrightarrow \text{proc}(c_\text{L}, Q) \\
(\text{D-}\otimes/\exists) & \text{proc}(c_\text{L}, y \leftarrow \text{recv } a_\text{L}\,; Q_y),\ \text{proc}(a_\text{L}, \text{send } a_\text{L}\ b\,; P) \longrightarrow \text{proc}(c_\text{L}, [b/y]\,Q_y),\ \text{proc}(a_\text{L}, P) \\
(\text{D-}\multimap/\Pi) & \text{proc}(c_\text{L}, \text{send } a_\text{L}\ b\,; Q),\ \text{proc}(a_\text{L}, y \leftarrow \text{recv } a_\text{L}\,; P_y) \longrightarrow \text{proc}(c_\text{L}, Q),\ \text{proc}(a_\text{L}, [b/y]P_y) \\
(\text{D-}\oplus) & \text{proc}(c_\text{L}, \text{case } a_\text{L} \text{ of } \overline{l \Rightarrow Q}),\ \text{proc}(a_\text{L}, a_\text{L}.l_h\,; P) \longrightarrow \text{proc}(c_\text{L}, Q_h),\ \text{proc}(a_\text{L}, P) \\
(\text{D-}\&) & \text{proc}(c_\text{L}, a_\text{L}.l_h\,; Q),\ \text{proc}(a_\text{L}, \text{case } a_\text{L} \text{ of } \overline{l \Rightarrow P}) \longrightarrow \text{proc}(c_\text{L}, Q),\ \text{proc}(a_\text{L}, P_h)
\end{array}
$$

Fig. 4. Remaining multiset rewriting rules not shown inline.

transition in the configuration of processes that arise from a program. As we discuss in Section 5.1, we use the predicates $\text{proc}(c_m,\ P)$ and $\text{unavail}(a_\text{S})$ to define the states of a configuration. The former denotes a process with process term P that provides along channel c_m at mode m, the latter acts as a placeholder for a shared process providing along channel a_S that is currently not available. Multiset rewriting rules are local in that they only mention the parts of a configuration they rewrite. The synchronous dynamics of the pair acquire-accept is given by the following rule:

$$
\begin{array}{c}
\text{proc}(c_\text{L}, x_\text{L} \leftarrow \text{acquire } a_\text{S}\,; Q_{x_\text{L}}),\ \text{proc}(a_\text{S}, x_\text{L} \leftarrow \text{accept } a_\text{S}\,; P_{x_\text{L}}) \\
\longrightarrow \text{proc}(c_\text{L}, [a_\text{L}/x_\text{L}]\,Q_{x_\text{L}}),\ \text{proc}(a_\text{L}, [a_\text{L}/x_\text{L}]\,P_{x_\text{L}}),\ \text{unavail}(a_\text{S})
\end{array}
\qquad (\text{D-}\uparrow^\text{S}_\text{L})
$$

The above rule exploits the invariant that a process's 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 and along a_S, while the process is shared. 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 within process terms. Since variables are subject to α-conversion, the typing rules (T-$\uparrow^\text{S}_\text{L}$L) and (T-$\uparrow^\text{S}_\text{L}$R) bind a fresh variable x_L, for which the already existing channel a at mode L will be substituted at run-time.

Figure 4 gives the dynamics of the remaining connectives in SILL_S. The side condition b *fresh* indicates allocation of a globally fresh channel and the equality $a = b$ expresses that b is substituted for a in the entire configuration. Multiset rewriting rules are unordered, but for ease of reading, we write them such that a providing process appears to the right of its client.

The typing of the pair release-detach is defined by the following rules:

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

The rules are essentially inverse to the typing rules of acquire-release; we point out that rule (T-$\downarrow^\text{S}_\text{L}$R) requires the linear context to be empty, to satisfy the independence principle. Operationally, the rules have the following semantics:

$$
\begin{array}{c}
\text{proc}(c_\text{L}, x_\text{S} \leftarrow \text{release } a_\text{L}\,; Q_{x_\text{S}}),\ \text{proc}(a_\text{L}, x_\text{S} \leftarrow \text{detach } a_\text{L}\,; P_{x_\text{S}}),\ \text{unavail}(a_\text{S}) \\
\longrightarrow \text{proc}(c_\text{L}, [a_\text{S}/x_\text{S}]\,Q_{x_\text{S}}),\ \text{proc}(a_\text{S}, [a_\text{S}/x_\text{S}]\,P_{x_\text{S}})
\end{array}
\qquad (\text{D-}\downarrow^\text{S}_\text{L})
$$

This time the rules shift the process from S to L, by switching the offering channel from a_L to a_S and by substituting the channel a_S for the fresh variable x_S.

Let's now return to the producer-consumer example and work out what the type specifications have to be. The processes *produce* and *consume* in Figure 2 have been devised under the assumption that the channel q is a shared channel to a shared queue and that the shared queue process recurs rather than terminates upon dequeueing, if the queue is empty. For this to be the case, we change the session type queue from Section 2 as follows:

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

With this change, the code in Figure 2 is type-correct as it is written. The new definition of session type queue A_S uses the previously introduced dependent linear session types $\Pi x{:}A_S.\ B_L$ and $\exists x{:}A_S.\ B_L$ for shared channel input and output, respectively, and prescribes the following synchronization pattern: When a process of type queue A_S is spawned, it starts out as a shared process that first must be acquired. Any of the defined sequences of inputs and outputs then are executed while the process is linear. After such an exchange, the process recurs at type $\downarrow^S_L$queue A_S. Since queue A_S is defined as $\uparrow^S_L \&\{\dots\}$, the type $\downarrow^S_L$ queue A_S amounts to the type $\downarrow^S_L \uparrow^S_L \&\{\dots\}$. This means that in its recursion, the process will first need to be released to become a shared process of type queue A_S. Looking at the implementations of processes *produce* and *consume* in Figure 2, we can see that they comply with the acquire-release pattern dictated by the above session type. For example, after process *produce* has sent the channel x along channel q', the channel q' is of type $\downarrow^S_L$queue A_S, which is why process *produce* releases that channel before it recurs.

Having changed the specification of session type queue A_S, we must correspondingly change the implementations of processes *empty* and *elem* shown in Figure 1; the result is given in Figure 5. The code predominantly contains the matching pairs accept and detach as well as acquire and release, respectively. For example, the first statement in process *empty* accepts an acquire request from a client. Similarly, the statement $q \leftarrow \mathsf{detach}\ q'$ initiates a release by a client.

```
empty : {queue A_S}
q ← empty =
  q' ← accept q ;
  case q' of
  | enq → x ← recv q' ;
          e ← empty ;
          q ← detach q' ;
          q ← elem ← x, e
  | deq → q'.none ;
          q ← detach q' ;
          q ← empty
```

```
elem : {queue A_S ← A_S, queue A_S}
q ← elem ← x, t =
  q' ← accept q ;
  case q' of
  | enq → y ← recv q' ;
          t' ← acquire t ;
          t'.enq ; send t' y ;
          t ← release t' ;
          q ← detach q' ;
          q ← elem ← x, t
  | deq → q'.some ;
          send q' x ;
          q ← detach q' ;
          fwd q t
```

Fig. 5. Implementation of a shared queue. See Figure 1 for linear version.

Session type queue A_S pinpoints a typical pattern of shared process programming where a shared recursive session type $Y_S = \uparrow^S_L A_L$ recurs at type $\downarrow^S_L Y_S$. The benefits of this pattern are two-fold: on the one hand, it guarantees that the session type Y_S allows for perpetual acquire-release cycles and,

on the other hand, it makes sure that all acquired processes are released at recursion point because linearity forbids any linear channels to be left behind.

Comparing this shared version of session type queue with its linear version in Section 2, we note that the independence principle requires the shared queue's elements to be shared, whereas a linear queue can either store linear or shared elements. The two versions also differ in the handling of a dequeuing request in case of an empty queue. Because there is only a single client in case of a linear queue, termination is a feasible choice. In case of a shared queue, however, recursion is preferable, to prevent other clients to block when attempting to acquire the terminated queue.

3.3 Equi-Synchronizing Session Types

So far we have achieved that a client communicates with a shared process in mutual exclusion from other clients and that the acquire and release points of a shared process manifest in its session type. There remains a last threat to session fidelity that we need to address: erroneous assumptions by a client on a shared process' type. These can come about, for example, in the following scenario: two clients Q_1 and Q_2 are trying to acquire access to the same shared channel c_S at type $\uparrow^S_L A_L$. Let's assume that Q_1 succeeds and then later releases c_L to a *different* type $\uparrow^S_L B_L$. Once Q_2 finally obtains access to c_S, it will disagree with the provider on the type of the channel c_L: the provider will think that $c_L : B_L$, while Q_2 will think that $c_L : A_L$, thereby violating session fidelity.

To guarantee preservation without resorting to run-time checks, we introduce the notion of an *equi-synchronizing* session type. A session type is equi-synchronizing if it imposes the invariant on a process to be *released* to the *same* type at which the process was previously *acquired*. No constraint is imposed on channels that were never acquired. For example, our shared queue A_S from Section 3.2

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

is equi-synchronizing because, in each branch, it releases a channel back to type queue A_S, which is the type at which the channel must have been acquired.

We formally define the notion of an equi-synchronizing session type in Figure 6, giving a coinductive definition. The definition is based on the judgment

$$\vdash_\Sigma (A, \hat{D})\ \mathsf{esync}$$

where $\hat{D}$ represents a constraint on the type to which a channel of type A must be released. If $\hat{D} = \top$, then there is no constraint on a future release, if $\hat{D} = D_S$, then any release must take place to type D_S. There is a third possibility, $\hat{D} = \bot$, which means that A may never be released. This constraint is only necessary for the proof of session fidelity, as further explained in Section 5.2. We say that the type A *equi-synchronizes* to $\hat{D}$ or that $\hat{D}$ is A's equi-synchronizing constraint.

Underlying the coinductive definition of an equi-synchronizing session type is the notion of a *continuation type*. To check that a type A equi-synchronizes to the type $\uparrow^S_L D_L$, the rules in Figure 6 transitively step through A's continuation (starting from $(A, \top)$) until the first acquisition point $\uparrow^S_L B_L$ is encountered. At this point, the type $\uparrow^S_L B_L$ is set to be the equi-synchronizing constraint, and the rules transitively step through each continuation of B_L until the first release point $\downarrow^S_L C_S$ is encountered. The session type is equi-synchronizing, if $C_S = \uparrow^S_L D_L$ at each such release point.

Let's exercise the rules in Figure 6 on our shared queue A_S. We start with $\vdash_\Sigma (\mathbf{queue}\ A_S,\ \top)\ \mathsf{esync}$. Since session types are interpreted equi-recursively and are contractive [Gay and Hole 2005], we can "silently" replace queue A_S with its definition, which means we have to check

$$\vdash_\Sigma (\uparrow^S_L \&\{\mathsf{enq} : \ldots\ \ \mathsf{deq} : \ldots\},\ \top)\ \mathsf{esync}$$

$$\frac{(\forall i)\ \ \vdash_\Sigma (A_{\mathsf{L}_i}, \hat{D})\ \mathsf{esync}}{\vdash_\Sigma (\oplus\{\overline{l : A_{\mathsf{L}}}\}, \hat{D})\ \mathsf{esync}}\ (\text{T-Esync}_\oplus) \qquad \frac{(\forall i)\ \ \vdash_\Sigma (A_{\mathsf{L}_i}, \hat{D})\ \mathsf{esync}}{\vdash_\Sigma (\&\{\overline{l : A_{\mathsf{L}}}\}, \hat{D})\ \mathsf{esync}}\ (\text{T-Esync}_\&)$$

$$\frac{\vdash_\Sigma (B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}{\vdash_\Sigma (A_{\mathsf{L}} \otimes B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}\ (\text{T-Esync}_\otimes) \qquad \frac{\vdash_\Sigma (B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}{\vdash_\Sigma (A_{\mathsf{L}} \multimap B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}\ (\text{T-Esync}_{\multimap})$$

$$\frac{\vdash_\Sigma (B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}{\vdash_\Sigma (\exists x{:}A_{\mathsf{S}}.\, B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}\ (\text{T-Esync}_\exists) \qquad \frac{\vdash_\Sigma (B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}{\vdash_\Sigma (\Pi x{:}A_{\mathsf{S}}.\, B_{\mathsf{L}}, \hat{D})\ \mathsf{esync}}\ (\text{T-Esync}_\Pi) \qquad \frac{}{\vdash_\Sigma (\mathbf{1}, \hat{D})\ \mathsf{esync}}\ (\text{T-Esync}_1)$$

$$\frac{\vdash_\Sigma (A_{\mathsf{L}}, \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}})\ \mathsf{esync}}{\vdash_\Sigma (\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}, \top)\ \mathsf{esync}}\ (\text{T-Esync}_{\uparrow^{\mathsf{S}}_{\mathsf{L}}}) \qquad \frac{\vdash_\Sigma (D_{\mathsf{S}}, \top)\ \mathsf{esync}}{\vdash_\Sigma (\downarrow^{\mathsf{S}}_{\mathsf{L}} D_{\mathsf{S}}, D_{\mathsf{S}})\ \mathsf{esync}}\ (\text{T-Esync}_{\downarrow^{\mathsf{S}}_{\mathsf{L}}}\text{-1}) \qquad \frac{\vdash_\Sigma (D_{\mathsf{S}}, \top)\ \mathsf{esync}}{\vdash_\Sigma (\downarrow^{\mathsf{S}}_{\mathsf{L}} D_{\mathsf{S}}, \top)\ \mathsf{esync}}\ (\text{T-Esync}_{\downarrow^{\mathsf{S}}_{\mathsf{L}}}\text{-2})$$

Fig. 6. Equi-synchronizing session type, coinductively defined.

According to rule (T-Esync$_{\uparrow^{\mathsf{S}}_{\mathsf{L}}}$), we set the equi-synchronizing constraint to queue A_{S}, requiring us to check for the continuation that

$$\vdash_\Sigma (\&\{\mathsf{enq} : \ldots\ \ \mathsf{deq} : \ldots\},\ \ \mathsf{queue}\ A_{\mathsf{S}})\ \mathsf{esync}$$

According to rule (T-Esync$_\&$), we are required to check for each continuation that

$$\vdash_\Sigma (\Pi x{:}A_{\mathsf{S}}.\, \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{queue}\ A_{\mathsf{S}},\ \ \mathsf{queue}\ A_{\mathsf{S}})\ \mathsf{esync}$$

$$\vdash_\Sigma (\oplus\{\mathsf{none} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{queue}\ A_{\mathsf{S}},\ \ \mathsf{some} : \exists x{:}A_{\mathsf{S}}.\, \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{queue}\ A_{\mathsf{S}}\},\ \ \mathsf{queue}\ A_{\mathsf{S}})\ \mathsf{esync}$$

Let's consider the first branch. According to rule (T-Esync$_\Pi$) we must check that

$$\vdash_\Sigma (\downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{queue}\ A_{\mathsf{S}},\ \ \mathsf{queue}\ A_{\mathsf{S}})\ \mathsf{esync}$$

which, according to rule (T-Esync$_{\downarrow^{\mathsf{S}}_{\mathsf{L}}}$-1), amounts to the check

$$\vdash_\Sigma (\mathsf{queue}\ A_{\mathsf{S}},\ \ \top)\ \mathsf{esync}$$

This is the check we started out with, allowing us to succeed on this branch since our rules are interpreted coinductively. Because the same holds true for all branches, unfolding type definitions where necessary, we conclude that the session type queue A_{S} is equi-synchronizing.

Not all branches must actually release. For example, the variant

$$\begin{aligned}\mathbf{queue\ A_S} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \&\{&\mathsf{enq} : \Pi x{:}A_{\mathsf{S}}.\, \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathbf{queue\ A_S},\\ &\mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1},\ \ \mathsf{some} : \exists x{:}A_{\mathsf{S}}.\, \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathbf{queue\ A_S}\}\}\end{aligned}$$

of the shared queue above is equi-synchronizing even though the queue terminates upon dequeuing in case of an empty queue. In that case, the queue can effectively no longer be acquired.

As we will show in more detail in Section 5.2, the equi-synchronizing invariants are at the core of the preservation proof, requiring us to show that each process maintains its equi-synchronizing constraint along all possible transitions. The three possible constraints $\hat{D}$, namely $\top$, $\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$, and $\bot$, are related by the following partial order, for any A_{L}:

$$\top \geq \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}} \geq \bot$$

This relationship becomes relevant for substitutions, where we allow substituting a channel of a smaller type for variables or channels of a bigger type at the client side (see Section 5.2).

When checking the signature Σ, recursive session type definitions are checked to be both contractive and equi-synchronizing and process definitions are checked to provide an equi-synchronizing

session type. The check is initiated with $\top$ as a constraint to convey that any initial release is unconstrained. A purely linear session type A_L with neither acquire nor release points will thus satisfy the constraint $\vdash_\Sigma (A_L, \top)$ esync and also the even stronger condition $\vdash_\Sigma (A_L, \bot)$ esync.

4 MORE EXAMPLES

In this section, we illustrate manifest sharing on further examples. An "imperative" style of a queue implementation that maintains a reference to the back of the queue can be found in the extended technical report [Balzer and Pfenning 2017].

4.1 Dining Philosophers

The dining philosophers problem [Dijkstra 1973] is a prime example designed to illustrate the issues of enforcing mutual exclusion in the presences of circular dependencies among processes. It's precisely because of circularity that the dining philosophers problem cannot be modelled in the purely linear language presented in Section 2. With sharing at our disposal, we are now able to model the dining philosophers problem. The result is given in Figure 7.

```
lfork = ⇓ˢ_L sfork
sfork = ↑ˢ_L lfork
phil = 1

fork_proc : {sfork}
c ← fork_proc =
  c′ ← accept c ;
  c ← detach c′ ;
  c ← fork_proc

thinking : {phil ← sfork, sfork}
c ← thinking ← left, right =
  (* thinking *)
  left′ ← acquire left ;
  right′ ← acquire right ;
  c ← eating ← left′, right′

eating : {phil ← lfork, lfork}
c ← eating ← left′, right′ =
  (* eating *)
  right ← release right′ ;
  left ← release left′ ;
  c ← thinking ← left, right
```

Fig. 7. Dining philosophers.

The implementation defines the mutually dependent session types lfork and sfork and the session type phil, representing a fork and a philosopher, respectively. In support of the spirit of the example, the former allow perpetual acquire-release cycles and are implemented by process *fork_proc*. Session type phil, on the other hand, denotes a trivial linear session, which is implemented by the processes *thinking* and *eating*. As the names suggest, process *thinking* represents a philosopher that is thinking, whereas process *eating* represents a philosopher that is eating. A thinking philosopher has shared channel references to the forks on their left and right. Once the philosopher is done thinking, they attempt to acquire their left and right fork and transition to eating, if successful. An eating philosopher, on the other hand, has linear channel references to the forks on their left and right, which they release once they are done eating and before transitioning to thinking. We can set up a table of 4 philosophers using the following lines of code:

```
f0 ← fork_proc ; f1 ← fork_proc ; f2 ← fork_proc ; f3 ← fork_proc ;
p0 ← thinking ← f0, f1 ; p1 ← thinking ← f1, f2 ; p2 ← thinking ← f2, f3 ; p3 ← thinking ← f3, f0 ;
```

The above setup faithfully matches the circular table and can lead to a deadlock, as pointed out by Dijkstra, if every philosopher picks up the fork on their left and then blocks, waiting for the fork on their right. We can avoid this deadlock by following Dijkstra's originally proposed solution to impose a partial order on the forks and acquiring the forks in ascending order. This can be achieved by reversing the order of the arguments in the last line to *p3* ← *thinking* ← *f0*, *f3*.

4.2 Atomicity

Another benefit of making the acquire and release points of a process manifest in the type structure is that *atomic* sections [Flanagan and Qadeer 2003] become explicit. Since the statements between

an up- and a downshift are executed while the process is linear, they are guaranteed to be executed without interference.

We illustrate atomicity on the example of printing to standard out from a concurrent program. To make sure that the print statements will be issued to standard out in the order that they appear in a given thread, we represent the standard output stream by a shared process that obeys the mutually recursive session types s_stdout and l_stdout in Figure 8. The protocol defined by those session types requires a client to acquire standard out before being able to print to it and then to release it upon completion. The processes p and v implement the session types s_stdout and l_stdout, respectively. We have chosen their names in reminiscence of Dijkstra's semaphore operations P and V.

```
s_stdout = ↑ˢ_L&{enter : l_stdout}
l_stdout = &{print : string ⊃ l_stdout,
             leave : ↓ˢ_L s_stdout}

p : {s_stdout}
c ← p =
  c′ ← accept c ;
  case c′ of
  | enter → c′ ← v

v : {l_stdout}
c′ ← v =
  case c′ of
  | print → x ← recv c′ ;
            print x ;
            c′ ← v
  | leave → c ← detach c′ ;
            c ← p
```

Fig. 8. Atomic standard output. The connective ⊃ denotes value input, an orthogonal concept introduced in [Griffith and Pfenning 2015; Toninho et al. 2013].

The lines of code below demonstrate how a client interacts with atomic standard out for printing, assuming the channel *out* of type s_stdout to be available as a system service:

```
out′ ← acquire out ; out′.enter ;
out′.print ; send out′ "Hello " ; out′.print ; send out′ "shared " ; out′.print ; send out′ "world!" ;
out′.leave ; out ← release out′ ;
```

In session type l_stdout, we take the liberty to use the connective ⊃, a connective introduced in [Griffith and Pfenning 2015; Toninho et al. 2013] to support value input. The type "string ⊃ l_stdout" describes as session that receives a value of type string and then continues as a session of type l_stdout. Toninho et al. [Toninho et al. 2013] show how to safely integrate a functional layer with a process layer by means of a linear contextual monad. Those results are orthogonal to sharing and generalize to our language. The statement *print* in process v, lastly, abstracts the actual print primitive on a given platform. To prevent races on this primitive, processes p and v are internal, and the only way for users to interact with standard out is via the system service *out*.

4.3 Nondeterminism

Acquire-release introduces *nondeterminism* into our language because it is unknown which client among several clients that acquire a shared process will succeed. We use this property to implement binary nondeterministic choice in our language.

Figure 9 gives the definition of session type coin and its implementing, mutually recursive processes *coin_head* and *coin_tail*. Session type coin indicates which side of the coin is currently facing up. In the implementation each interaction flips the coin to its opposite side.

Figure 10 shows the process *nd_choice* which nondeterministically sends yes or no and then terminates. Process *nd_choice* achieves nondeterminism by reading a coin that it shares with process *coin_flipper*. Since both processes try to acquire the coin concurrently and the coin switches sides

$$\mathbf{coin} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \oplus \{\mathsf{head} : \downarrow^{\mathsf{S}}_{\mathsf{L}}\mathbf{coin},\ \mathsf{tail} : \downarrow^{\mathsf{S}}_{\mathsf{L}}\mathbf{coin}\}$$

```
coin_head : {coin}
c ← coin_head =
  c' ← accept c ;
  c'.head ;
  c ← detach c' ;
  c ← coin_tail
```

```
coin_tail : {coin}
c ← coin_tail =
  c' ← accept c ;
  c'.tail ;
  c ← detach c' ;
  c ← coin_head
```

Fig. 9. Session type coin with implementing processes *coin_head* and *coin_tail*.

when read, the value read by *nd_choice* depends on the order in which the coin is acquired. For a client of this service, see Figure 11 where it is used to model nondeterminism inherent in the (untyped) asynchronous π-calculus.

```
nd_choice : {⊕{yes : 1, no : 1}}
d ← nd_choice =
  c ← coin_head ;
  f ← coin_flipper ← c ;
  c' ← acquire c ;
  case c' of
  | head → c ← release c' ; d.yes ; wait f ; close d
  | tail → c ← release c' ; d.no ; wait f ; close d
```

```
coin_flipper : {1 ← coin}
d ← coin_flipper ← c =
  c' ← acquire c ;
  case c' of
  | head → c ← release c' ; close d
  | tail → c ← release c' ; close d
```

Fig. 10. Binary nondeterministic choice.

5 SEMANTICS

In this section, we complete the discussion of the semantics of SILL_S, by giving the configuration typing rules as well as elaborating on preservation and progress. For a complete listing of SILL_S's abstract syntax, statics, and dynamics we refer to the extended technical report [Balzer and Pfenning 2017]. In the last subsection, we sketch an asynchronous dynamics for SILL_S, which relies on a novel transformation derived from logic.

5.1 Configuration Typing

At run-time, a SILL_S program evolves into a number of linear and shared processes as well as placeholders for formerly shared processes that are currently linear. To type the resulting *configuration* Ω, we divide the configuration into a *linear* part Θ and a *shared* part Λ, subject to the following well-formedness conditions:

$$\begin{array}{ll}
\Omega \triangleq \cdot \mid \Lambda; \Theta & (\forall a.\mathsf{proc}(a_\mathsf{L}, _) \in \Theta \implies \mathsf{unavail}(a_\mathsf{S}) \in \Lambda) \\
\Lambda \triangleq \cdot \mid \mathsf{proc}(a_\mathsf{S}, P_{a_\mathsf{S}}), \Lambda' \mid \mathsf{unavail}(a_\mathsf{S}), \Lambda' & (\mathsf{proc}(a_\mathsf{S}, _), \mathsf{unavail}(a_\mathsf{S}) \text{ not in } \Lambda') \\
\Theta \triangleq \cdot \mid \mathsf{proc}(a_\mathsf{L}, P_{a_\mathsf{L}}), \Theta' & (\mathsf{proc}(a_\mathsf{L}, _) \text{ not in } \Theta')
\end{array}$$

The side conditions make sure that no other process (or placeholder) exists yet in the configuration that provides along the same channel and that for every linear process there exists a placeholder at the shared mode of the channel. The division is justified by the hierarchy between modes S and L, making sure that shared processes cannot depend on linear processes. We use the following typing judgment to type a configuration:

$$\Gamma \vDash_\Sigma \Lambda; \Theta :: \Gamma; \Delta$$

The judgment expresses that the configuration $\Lambda; \Theta$ is well-formed and provides the shared channels in Γ and the linear channels in Δ. To permit cyclic dependencies along shared channels, a

configuration is type-checked relative to all shared channels, which is the reason why Γ appears to the left of the turnstile. The typing of a configuration is defined by the following rule:

$$\frac{\Gamma \vDash_\Sigma \Lambda :: \Gamma \qquad \Gamma \vDash_\Sigma \Theta :: \Delta}{\Gamma \vDash_\Sigma \Lambda; \Theta :: \Gamma; \Delta}\ (\text{T-}\Omega)$$

The rule relies on the judgment $\Gamma \vDash_\Sigma \Theta :: \Delta$ for typing Θ and the judgment $\Gamma \vDash_\Sigma \Lambda :: \Gamma$ for typing Λ. The judgment $\Gamma \vDash_\Sigma \Theta :: \Delta$ expresses that the configuration Θ provides the linear channels in Δ, using the shared channels in Γ. The typing of Θ is defined by the following two rules:

$$\frac{}{\Gamma \vDash_\Sigma (\cdot) :: (\cdot)}\ (\text{T-}\Theta_1) \qquad \frac{(a_S : \hat{B}) \in \Gamma \quad \vdash_\Sigma (A_L, \hat{B})\ \mathsf{esync} \quad \Gamma;\ \Delta' \vdash_\Sigma P_{a_L} :: (a_L : A_L) \quad \Gamma \vDash_\Sigma \Theta : \Delta, \Delta'}{\Gamma \vDash_\Sigma \mathsf{proc}(a_L,\ P_{a_L}), \Theta :: (\Delta, a_L : A_L)}\ (\text{T-}\Theta_2)$$

Rule (T-Θ_2) is of particular interest as it imposes an order on linear configurations. By requiring that all the linear channels Δ' used by $\mathsf{proc}(a_L,\ P_{a_L})$ are provided by the remaining configuration Θ, the rule "flattens" the linear process tree such that for any process the providers of the channels used by the process are to the right of the process in the configuration. We maintain this order only for typing purposes, at run-time any permutations of a well-typed configuration are permissible. The rule also enforces that a linear configuration only provides the channels that are not used internally to the configuration. For example, the channels Δ' consumed by $\mathsf{proc}(a_L,\ P_{a_L})$ are no longer provided as part of the resulting configuration $\mathsf{proc}(a_L,\ P_{a_L}), \Theta$. An initial configuration $\Lambda; \Theta$ would be typed as $\Gamma \vDash_\Sigma \Lambda; \Theta :: (\Gamma; c_L : \mathbf{1})$, where the process providing along channel c_L is the main program thread and Λ may provide some pre-defined shared system services such as *out* in Section 4.2. The premises $(a_S : \hat{B}) \in \Gamma$ and $\vdash_\Sigma (A_L, \hat{B})$ esync of rule (T-Θ_2) constrain the type to which $\mathsf{proc}(a_L,\ P_{a_L})$ must be released.

Unlike the typing rules for Θ, the typing rules for Λ do not impose any order on the shared processes. Any attempt would be futile anyway because the reference structure along shared channels may not adhere to any pattern and could, for example, be cyclic. We use the judgment $\Gamma \vDash_\Sigma \Lambda :: \Gamma'$ to type such configurations, expressing that Λ offers the shared channels in Γ', using the shared channels in Γ. The typing rules for Λ are:

$$\frac{}{\Gamma \vDash_\Sigma (\cdot) :: (\cdot)}\ (\text{T-}\Lambda_1) \qquad \frac{\vdash_\Sigma (\uparrow^S_L A_L, \top)\ \mathsf{esync} \quad \Gamma \vdash_\Sigma P_{a_S} :: (a_S : \uparrow^S_L A_L)}{\Gamma \vDash_\Sigma \mathsf{proc}(a_S,\ P_{a_S}) :: (a_S : \uparrow^S_L A_L)}\ (\text{T-}\Lambda_2)$$

$$\frac{}{\Gamma \vDash_\Sigma \mathsf{unavail}(a_S) :: (a_S : \hat{A})}\ (\text{T-}\Lambda_3) \qquad \frac{\Gamma \vDash_\Sigma \Lambda :: \Gamma' \quad \Gamma \vDash_\Sigma \Lambda' :: \Gamma''}{\Gamma \vDash_\Sigma \Lambda, \Lambda' :: \Gamma', \Gamma''}\ (\text{T-}\Lambda_4)$$

Rule (T-Λ_4) permits breaking up a configuration Λ into its subparts at any point. Rule (T-Λ_2) carries again an equi-synchronizing invariant as a premise, indicating that the type to which $\mathsf{proc}(a_S,\ P_{a_S})$ must be released is not yet significant.

Unlike process expressions encountered during type checking, which have occurrences of variables only, the premises $\Gamma;\ \Delta' \vdash_\Sigma P_{a_L} :: (a_L : A_L)$ and $\Gamma \vdash_\Sigma P_{a_S} :: (a_S : \uparrow^S_L A_L)$ in rules (T-Θ_2) and (T-Λ_2), respectively, have occurrences of both variables and channels. The occurrence of channels is a result of substituting channels for variables during execution. As detailed in Section B.2 in the extended technical report [Balzer and Pfenning 2017], those process expressions satisfy slightly weaker well-formedness conditions than the ones to be met during type-checking (see Section 2).

5.2 Preservation and Progress

In this section, we state preservation and progress for SILL_S and review the key issues that had to be adressed to prove preservation and progress. For detailed proofs we refer to the extended technical report [Balzer and Pfenning 2017].

The challenges that arise from extending the linear system discussed in Section 2 with manifest sharing are twofold. For preservation, we need to make sure that clients will encounter shared processes at the type they would like to acquire them. For progress, we need to account for the possibility of deadlock due to cyclic dependencies along shared channels or for termination of a process providing a shared service, while ruling out other forms of failure of progress.

To address the first challenge, we have introduced the notion of an equi-synchronizing session type in Section 3.3, which statically imposes the invariant that each shared channel is released to the same session type at which is was acquired (if at all). The preservation proof shows that this invariant is maintained for each channel along any possible transition, as captured in the corresponding premises of rules (T-Θ_2) and (T-Λ_2). Key are the three forms of type constraints $\hat{D}$ with $\vdash_\Sigma (A, \hat{D})$ esync where A is the current type of a linear process providing along a_L:

(1) $\hat{D} = \top$, indicating that there is no constraint on a future release of a_L because a_L has never been shared. $\hat{D} = \top$ holds initially, when a linear process is spawned, and continues to hold until the process is released for the first time to become shared. Processes which remain linear throughout their lifetime will never be subject to an equi-synchronization constraint.

(2) $\hat{D} = D_\mathsf{S}$, indicating that if there is a future release of a_L to a shared channel a_S, then a_S must have type D_S. Preservation holds since we have statically checked that $\vdash_\Sigma (A, \hat{D})$ esync and this property is maintained along all continuations of A.

(3) $\hat{D} = \bot$, expressing that a_S must never be released, which means that any client attempting to acquire a_S will be blocked forever. The need for $\bot$ is subtle. Imagine we forward between two linear channels fwd a_L b_L. The forward has to identify not only a_L and b_L, but also the underlying shared channels a_S and b_S, because releasing one now amounts to releasing the other:

$$\mathsf{proc}(a_\mathsf{L}, \mathsf{fwd}\ a_\mathsf{L}\ b_\mathsf{L}) \longrightarrow a_\mathsf{L} = b_\mathsf{L}, a_\mathsf{S} = b_\mathsf{S} \qquad (\text{D-Id}_\mathsf{L})$$

While the types of a_L and b_L must be at the same A, it is possible that the constraints on the releases of a_L and b_L are $\vdash (A, D_\mathsf{S})$ esync and $\vdash (A, D'_\mathsf{S})$ esync for $D_\mathsf{S} \neq D'_\mathsf{S}$. This can come about because a_L and b_L may have different histories. Preservation still holds in this case because there cannot be a down shift in any continuation of A (shown by coinduction on the definition of esync), so neither a_L nor b_L could ever be released. Formally, this is conveniently expressed as $\vdash_\Sigma (A, \bot)$ esync.

The introduction of $\bot$ requires us to generalize all the typing rules where a process uses a shared channel. For example, we change rule (T-$\uparrow^\mathsf{S}_\mathsf{L}$L) as follows:

$$\frac{\hat{B} \leq \uparrow^\mathsf{S}_\mathsf{L} A_\mathsf{L} \qquad \Gamma, x_\mathsf{S} : \hat{B};\ \Delta, x_\mathsf{L} : A_\mathsf{L} \vdash_\Sigma Q_{x_\mathsf{L}} :: (z_\mathsf{L} : C_\mathsf{L})}{\Gamma, x_\mathsf{S} : \hat{B};\ \Delta \vdash_\Sigma x_\mathsf{L} \leftarrow \mathsf{acquire}\, x_\mathsf{S}\, ; Q_{x_\mathsf{L}} :: (z_\mathsf{L} : C_\mathsf{L})} \ (\text{T-}\uparrow^\mathsf{S}_\mathsf{L}\text{L})$$

In contrast to the rule introduced in Section 3.2 the above rule accounts for the possibility of a shared process to be of type $\bot$. In this case, a client can freely choose the type of the process to be acquired because it will never succeed in acquiring that process. As can be seen in Figure 3, the rules (T-Id$_\mathsf{S}$), (T-Spawn$_\mathsf{LL}$), (T-Spawn$_\mathsf{LS}$), (T-Spawn$_\mathsf{SS}$), (T-$\exists_\mathsf{R}$), and (T-Π_L) require analogous treatment.

We can finally state the preservation theorem. It expresses that the types of the providing linear channels are maintained along transitions and that new shared channels may be allocated.

Theorem 5.1 (Preservation). *If* $\Gamma \vDash_\Sigma \Lambda; \Theta :: \Gamma; \Delta$ *and* $\Lambda; \Theta \longrightarrow \Lambda'; \Theta'$*, then* $\Gamma' \vDash_\Sigma \Lambda'; \Theta' :: \Gamma'; \Delta$*, for some* Λ'*,* Θ'*, and* Γ'*.*

Proof. Preservation is proved by induction on the dynamics, constructing a derivation of a well-formed and well-typed configuration $\Gamma' \vDash_\Sigma \Lambda''; \Theta'' :: \Gamma'; \Delta$, where Λ'' and Θ'' are permutations of Λ' and Θ', respectively, and using a variety of substitution lemmas and inversion. Note that the linear context Δ remains the same: freshly spawned linear channels have both a provider and client and are therefore not part of the interface. The set of shared channels however can grow. □

Our progress theorem is based on the notion of a *poised* process introduced in [Pfenning and Griffith 2015]. A $\mathsf{proc}(a,\ P_a)$ is *poised* if it is communicating along its providing channel. The poised forms of processes in SILL_S are:

Receiving	*Sending*
$\mathsf{proc}(a_\mathsf{L},\ y \leftarrow \mathsf{recv}\ a_\mathsf{L}\ ;\ P_y)$	$\mathsf{proc}(a_\mathsf{L},\ \mathsf{send}\ a_\mathsf{L}\ b\ ;\ P)$
	$\mathsf{proc}(a_\mathsf{L},\ \mathsf{close}\ a_\mathsf{L})$
$\mathsf{proc}(a_\mathsf{L},\ \mathsf{case}\ a_\mathsf{L}\ \mathsf{of}\ \overline{l \Rightarrow P})$	$\mathsf{proc}(a_\mathsf{L},\ a_\mathsf{L}.l_h\ ;\ P)$
$\mathsf{proc}(a_\mathsf{S},\ x_\mathsf{L} \leftarrow \mathsf{accept}\ a_\mathsf{S}\ ;\ P_{x_\mathsf{L}})$	$\mathsf{proc}(a_\mathsf{L},\ x_\mathsf{S} \leftarrow \mathsf{detach}\ a_\mathsf{L}\ ;\ P_{x_\mathsf{S}})$

A linear configuration Θ is poised if all $\mathsf{proc}(a_\mathsf{L},\ P_{a_\mathsf{L}}) \in \Theta$ are poised and a shared configuration Λ is poised if all $\mathsf{proc}(a_\mathsf{S},\ P_{a_\mathsf{S}}) \in \Lambda$ are poised.

To account for the possibility of deadlock, we introduce the notion of a *blocked* process. We say that a process is *blocked along* a_S if it has the form $\mathsf{proc}(c_\mathsf{L},\ x_\mathsf{L} \leftarrow \mathsf{acquire}\ a_\mathsf{S}\ ;\ Q_{x_\mathsf{L}})$. We then state the progress theorem such as to express that being blocked is the *only* way the whole configuration may be stuck [Harper 2013]. Case (2-c) captures the scenario where a blocked process cannot proceed because the shared channel is unavailable. Case (2-a), on the other hand, captures a successful acquire.

Theorem 5.2 (Progress). *If* $\Gamma \vDash_\Sigma \Lambda; \Theta :: \Gamma; \Delta$*, then either*

(1) $\Lambda \longrightarrow \Lambda'$*, for some* Λ'*, or*
(2) Λ *is poised and*
  (a) $\Lambda; \Theta \longrightarrow \Lambda'; \Theta'$*, for some* Λ' *and* Θ'*, or*
  (b) Θ *is poised, or*
  (c) *some process in* Θ *is blocked along* a_S *and* $\mathsf{unavail}(a_\mathsf{S}) \in \Lambda$.

Proof. Progress is proved by induction on the typing of the configurations Λ and Θ. □

At the top level, we have $\Delta = (c_0 : \mathbf{1})$, which means that if Θ is poised then it and all subcomputations must be finished, trying to close c_0. If it cannot transition, then the remaining possibility is that some process in Θ is blocked along a shared channel. A blocked process may wait indefinitely in case of a deadlock, or because the underlying shared process has terminated, or may never be released. Dining philosophers (Figure 7), for instance, is an example leading to a classic deadlock due to a cyclic dependency along the shared forks.

5.3 Asynchronous Dynamics

The synchronous operational semantics we have provided for SILL_S is simple, but not realistic in many applications. Fortunately, we can easily model *asynchronous output* in the existing language in a logically meaningful way. In order to explain this, we reintroduce the general form of cut (spawn) which is not tied to process definitions, and remind the reader of the identity (forward)

rule. For simplicity, we restrict the presentation to the linear case; the shown technique directly generalizes to the shared case.

$$\frac{\Delta \vdash P_x :: (x : A) \quad \Delta', x : A \vdash Q_x :: (z : C)}{\Delta, \Delta' \vdash_\Sigma x \leftarrow P_x \,;\, Q_x :: (z : C)}\ (\text{T-Cut}) \qquad \frac{}{y : A \vdash_\Sigma \mathsf{fwd}\ x\ y :: (x : A)}\ (\text{T-Id})$$

To asynchronously send a channel y along x we spawn a new process which carries the message y, immediately followed by forwarding.

$$\mathsf{send}\ x\ y \,;\, P \quad \simeq \quad x' \leftarrow (\mathsf{send}\ x\ y \,;\, \mathsf{fwd}\ x'\ x) \,;\, [x'/x]P$$

Intuitively, the spawned process $(\mathsf{send}\ x\ y \,;\, \mathsf{fwd}\ x'\ x)$ represents the message y sent along x with fresh continuation channel x' [DeYoung et al. 2012]. The continuation channel is necessary so that multiple messages sent along the same channel are guaranteed to arrive in the correct order. It is easy to see that, if the synchronous form on the left is well-typed, then so is the asynchronous form on the right. Logically, we can obtain the proof of the left from the proof of the right by a commuting conversion and reduction of cut with identity.

Operationally, the single synchronous reduction

$$\begin{aligned}&\mathsf{proc}(c, \mathsf{send}\ a\ b \,;\, P), \mathsf{proc}(a, y \leftarrow \mathsf{recv}\ a \,;\, Q_y)\\ &\quad\longrightarrow \mathsf{proc}(c, P), \mathsf{proc}(a, [b/y]Q_y)\end{aligned}$$

is now decomposed into several steps, where P can proceed with its continuation before b is received.

$$\begin{aligned}&\mathsf{proc}(c, x' \leftarrow (\mathsf{send}\ a\ b \,;\, \mathsf{fwd}\ x'\ a) \,;\, [x'/a]P), \mathsf{proc}(a, y \leftarrow \mathsf{recv}\ a \,;\, Q_y)\\
&\quad\longrightarrow \mathsf{proc}(c, [a'/a]P), \mathsf{proc}(a', \mathsf{send}\ a\ b \,;\, \mathsf{fwd}\ a'\ a), \mathsf{proc}(a, y \leftarrow \mathsf{recv}\ a \,;\, Q_y) && (\text{spawn, } a' \text{ fresh})\\
&\quad\longrightarrow \mathsf{proc}(c, [a'/a]P), \mathsf{proc}(a', \mathsf{fwd}\ a'\ a), \mathsf{proc}(a, [b/y]Q_y) && (\text{receive})\\
&\quad\longrightarrow \mathsf{proc}(c, [a'/a]P), \mathsf{proc}(a', [a'/a][b/y]Q_y) && (\text{forward})\end{aligned}$$

Since a' is chosen globally fresh and a is linear, the result is an α-variant of the synchronous outcome. This technique can be applied to all send operations of the semantics. Effectively, this allows a program written in the synchronous style to be executed fully asynchronously.

The caveat is that we would not want to translate acquire in this manner even though the logical semantics dictates it must be a send operation [Pfenning and Griffith 2015]. The reason is that a process would no longer block when trying to acquire a shared channel. Instead it would continue until the corresponding linear channel is actually used to receive a message, which is not the intended meaning. In the implementation (see Section 7) all sends are asynchronous, using a more efficient message buffer instead of explicit continuation channels, except for acquire which blocks until the shared channel becomes available.

Alternatively, we could directly provide an asynchronous semantics for all the operations and use an additional acknowledgment step (a "double shift" [Pfenning and Griffith 2015]) to ensure that acquiring a shared resource is synchronous. For this paper, we have chosen the former route because it simplifies the operational semantics and therefore our theorems: without loss of expressiveness, we do not have to explicitly deal with messages or message queues.

6 ENCODING THE UNTYPED π-CALCULUS INTO SILL_S

When we view Howard's original isomorphism between typed λ-calculus and intuitionistic natural deduction [Howard 1969] as a type assignment system for untyped λ-terms, we lose much of the computational power of the untyped λ-calculus. For example, normalization for natural deduction implies termination of computation on well-typed λ-terms, while arbitrary λ-terms may not have a normal form. However, there is a simple way we can embed *all* untyped λ-terms if we add recursive types. In linear instances of the Curry-Howard correspondence, just adding recursion appears

insufficient to recover the computational power of the asynchronous π-calculus [Wadler 2012], and so far there has been no logically motivated and fully satisfactory way to do so.[4]

In this section, we give an encoding of the asynchronous, untyped π-calculus into SILL_S, suggesting that shared channels can recover the computational power of the untyped π-calculus. We plan to confirm this hypothesis as part of future work (see Section 9). The key points to address in the encoding are that *(i)* π-calculus channels may connect arbitrarily many processes, *(ii)* messages sent along a π-calculus channel may arrive in arbitrary order, and *(iii)* π-calculus channels are untyped. Furthermore, since the π-calculus permits deadlock, it is important here that SILL_S also admits deadlock.

The basic idea of our encoding is to translate π-calculus *processes* to *linear* SILL_S processes of type $\mathbf{1}$, and π-calculus *channels* to *shared* SILL_S processes of a universal shared type $\mathcal{U}_\mathsf{S}$. The latter are unordered buffers and obey the following protocol:

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

Type $\mathcal{U}_\mathsf{S}$ provides the choice to either send (ins) or receive (del) a channel. In the latter case, it communicates whether the buffer is empty (none) or not empty (some) and delivers a channel in the buffer, if the buffer is non-empty. Figure 11 shows the processes *empty* and elem that implement session type $\mathcal{U}_\mathsf{S}$. To guarantee that the resulting buffer is unordered, process *elem* nondeterministically inserts the received channel at an arbitrary point in the buffer, using *nd_choice* defined in Figure 10. It is also possible and slightly more complicated to postpone the nondeterministic choice to the deletion operation.

```
empty : {U_S}
c ← empty =
  c' ← accept c ;
  case c' of
  | ins → x ← recv c' ;
          e ← empty ;
          c ← detach c' ; 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' ;
          ndc ← nd_choice ;
          case ndc of
          | yes → e ← elem ← x, d ;
                  wait ndc ;
                  c ← detach c' ; c ← elem ← y, e
          | no → d' ← acquire d ;
                 d'.ins ; send d' y ;
                 d ← release d' ;
                 wait ndc ;
                 c ← detach c' ; c ← elem ← x, d
  | del → c'.some ;
          send c' x ;
          c ← detach c' ; fwd c d
```

Fig. 11. Processes *empty* and *elem* implement session type $\mathcal{U}_\mathsf{S}$, representing a π-calculus channel. To guarantee that the resulting buffer is unordered, process *elem* nondeterministically inserts the received channel at an arbitrary point in the buffer, using process *nd_choice* defined in Figure 10.

[4]Other recent work in this direction in the setting of classical linear logic and differential interaction nets by Atkey et al. [2016] and Mazza [2016], respectively, use quite different techniques from ours.

The linear SILL_S processes representing π-calculus processes now simply amount to "producers" and "consumers" of shared channels of type $\mathcal{U}_\mathsf{s}$. Any number of such processes can communicate along a π-calculus channel by acquiring the shared SILL_S channel of universal type.

We are now ready to give the encoding of processes. We first review the syntax of the *asynchronous monadic* π-calculus [Milner 1999; Sangiorgi and Walker 2001], defining the set P^π of π-calculus process terms. We follow the presentation in [Beauxis et al. 2008]:

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

$\mathtt{0}$ denotes an inactive process. $\overline{x}\langle y\rangle$ represents an asynchronous send of y along channel x. $x(y).P$ represents the receiving of a channel along channel x, after which the process continues with executing P with the received channel bound to y in P. The action prefix $x(y)$ acts as a guard, making sure that P can only become active once the input has occurred. $\nu x\, P$ introduces a new channel x that is bound in P. $P_1 \mid P_2$ denotes parallel composition of P_1 and P_2 and $!P$ replication of P.

Our translation shown in Figure 12 yields for each π-calculus process term P^π a corresponding linear process $[\![P^\pi]\!]_a$ in SILL_S, satisfying the typing judgment

$$\Gamma;\, \cdot \vdash_\Sigma [\![P^\pi]\!]_a :: (a_\mathsf{L} : \mathbf{1})$$

where Γ consists of declarations $x_S : \mathcal{U}_\mathsf{s}$ for every shared channel in the overall process configuration. We use type $\mathbf{1}$ since all communication goes though π-calculus channels, which are mapped to shared channels in Γ. This is also the reason why there are no linear channels in the context. Of course, as shared channels are acquired when send or receive operations are modeled, we communicate with the buffer along a linear channel until it is released again.

Because of the different semantic basis (asynchronous π-calculus on one hand and multiset rewriting on the other), and the question what precisely is observable about a computation, the precise nature of the correspondence between traces in the source and target is difficult to formulate and prove and left to future work (see Section 9 for further remarks).

7 IMPLEMENTATION

We briefly describe our implementation of manifest sharing in the context of a type-safe C-like imperative language with session types called Concurrent C0 [Willsey et al. 2016], which is an extension of C0 [Arnold 2010; Pfenning 2010] designed for and used in an introductory imperative programming course [Pfenning et al. 2011]. Because session-typed programming follows a monadic style, this imperative implementation is semantically adequate for exploring the expressive power and programming style of manifest sharing. Besides an occasional illustrative use of imperative language features (e.g., loops in place of recursion, or mutable arrays instead of sequences), the only significant difference is the lack of parametric polymorphism in Concurrent C0. Examples have therefore been modified to use either base types, such as `int`, or ad hoc polymorphism in the form of `void*`, which engenders tagging of values with their dynamic type to ensure type safety. The implementation uses asynchronous message passing, as described in Section 5.3. Moreover, the downshift modality $\downarrow^\mathsf{S}_\mathsf{L}$ has no explicit syntax but implicitly precedes every upshift $\uparrow^\mathsf{S}_\mathsf{L}$. This is adequate since, just as in this paper, the only constructor of shared mode is an upshift, so there is no other possible continuation.

The compiler translates C0 source to C. Each logical thread of control is implemented as an operating system thread, as provided by the `pthread` library. Message passing is implemented via shared memory. Each channel is therefore a data structure in shared memory that can progress through linear and shared phases. Figure 13 provides a schematic overview of this data structure. While linear, access is shared between a provider and a client. The channel contains a current

$$
\begin{array}{lcl}
[\![\mathtt{0}]\!]_a & = & \mathsf{close}\ a \\
[\![\overline{c}\langle b\rangle]\!]_a & = & p \leftarrow \mathit{snd}\ \mathbf{c}\ ; \\
 & & \mathsf{send}\ p\ \mathbf{b}\ ; \\
 & & \mathsf{wait}\ p\ ; \\
 & & \mathsf{close}\ a \\
[\![c(x).P]\!]_a & = & p \leftarrow \mathit{poll\_rcv} \leftarrow \mathbf{c}\ ; \\
 & & \mathbf{b} \leftarrow \mathsf{recv}\ p\ ; \\
 & & \mathsf{wait}\ p\ ; \\
 & & a \leftarrow [\mathbf{b}/x]\,[\![P]\!]_a \\
[\![\nu x\, P]\!]_a & = & \mathbf{e} \leftarrow \mathit{empty}\ ; \\
 & & a \leftarrow [\mathbf{e}/x]\,[\![P]\!]_a \\
[\![P_1 \mid P_2]\!]_a & = & b \leftarrow [\![P_1]\!]_b\ ; \\
 & & c \leftarrow [\![P_2]\!]_c\ ; \\
 & & \mathsf{wait}\ b\ ; \\
 & & \mathsf{wait}\ c\ ; \\
 & & \mathsf{close}\ a \\
[\![!P]\!]_a & = & \mathit{Rec}^a_{!P} \quad \text{where} \\
\mathit{Rec}^a_{!P} & = & b \leftarrow [\![P]\!]_b\ ; \\
 & & c \leftarrow \mathit{Rec}^c_{!P}\ ; \\
 & & \mathsf{wait}\ b\ ; \\
 & & \mathsf{wait}\ c\ ; \\
 & & \mathsf{close}\ a
\end{array}
$$

$$
\begin{array}{l}
\mathit{snd} : \{(\Pi x{:}\mathcal{U}_S.\,\mathbf{1}) \leftarrow \mathcal{U}_S\} \\
d \leftarrow \mathit{snd} \leftarrow \mathbf{c} = \\
\quad \mathbf{x} \leftarrow \mathsf{recv}\ d\ ; \\
\quad c' \leftarrow \mathsf{acquire}\ \mathbf{c}\ ; \\
\quad c'.\mathsf{ins}\ ; \\
\quad \mathsf{send}\ c'\ \mathbf{x}\ ; \\
\quad \mathbf{c} \leftarrow \mathsf{release}\ c'\ ; \\
\quad \mathsf{close}\ d \\
\\
\mathit{poll\_rcv} : \{(\exists x{:}\mathcal{U}_S.\,\mathbf{1}) \leftarrow \mathcal{U}_S\} \\
d \leftarrow \mathit{poll\_rcv} \leftarrow \mathbf{c} = \\
\quad c' \leftarrow \mathsf{acquire}\ \mathbf{c}\ ; \\
\quad c'.\mathsf{del}\ ; \\
\quad \mathsf{case}\ c'\ \mathsf{of} \\
\quad \mid \mathsf{none} \rightarrow \mathbf{c} \leftarrow \mathsf{release}\ c'\ ; \\
\qquad\qquad\quad d \leftarrow \mathit{poll\_rcv} \leftarrow \mathbf{c} \\
\quad \mid \mathsf{some} \rightarrow \mathbf{x} \leftarrow \mathsf{recv}\ c'\ ; \\
\qquad\qquad\quad \mathbf{c} \leftarrow \mathsf{release}\ c'\ ; \\
\qquad\qquad\quad \mathsf{send}\ d\ \mathbf{x}\ ; \\
\qquad\qquad\quad \mathsf{close}\ d \\
\\
\mathit{empty} : \{\mathcal{U}_S\} \text{ is defined in Figure 11}
\end{array}
$$

Fig. 12. Translation of untyped asynchronous π-calculus processes into SILL$_S$ and auxiliary processes *snd* and *poll_recv*.

direction of communication and a message queue implemented as a ring buffer whose size is calculated from the session type. Access to the buffer for send and receive operations is protected by a mutex and associated condition variable. In the shared phase, there will be zero or one provider and an arbitrary number of clients. The channel therefore contains a flag that indicates whether the channel is currently available to be acquired. This flag is turned off when the channel is acquired by one of the clients and remains off until the client has been detached and the provider is ready to accept another client. Access to this flag is protected by a separate mutex and condition variable. The operating system scheduler will then nondeterministically select one of the clients.

As might be expected from the theory, the most difficult aspect of the implementation is forwarding. For forwarding between two linear channels, fwd c d, we send a message `FWD` c along d, or `FWD` d along c, depending on the current direction of communication. Then the thread executing the forward terminates. When the `FWD` e message arrives (where e is either c or d, depending on the direction), the recipient changes its internal reference to the shared channel to e, effectively now continuing communication along e. For more details and some failed alternatives, see [Willsey et al. 2016].

Unfortunately, this strategy fails for forwarding between two shared channels, fwd c d, because there is no effective way to notify all clients of c to now communicate along d via a message. Instead, before terminating, the provider installs a forwarding pointer from c to d and marks the availability

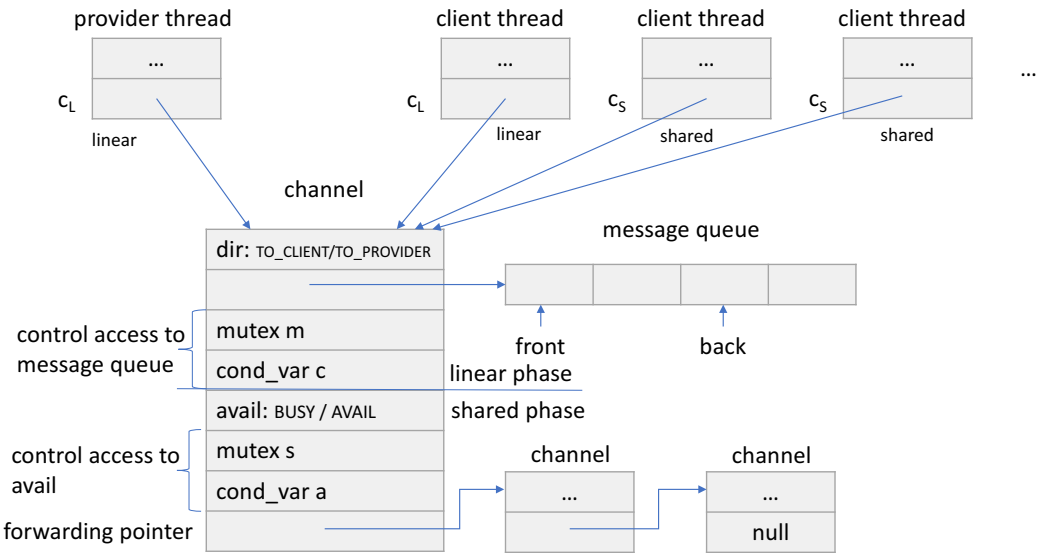

Fig. 13. Schematic overview of channel data structure internal to the Concurrent C0 compiler.

of c. Attempts to acquire c will follow the forwarding pointer to d. A potential client may have to follow a whole chain of such forwarding pointers. However, each client has to do so at most once.

Returning to a linear forward: when we execute fwd c d for linear channels c and d that where once shared, the semantics requires that we also forward between the underlying shared channels. For example, if the client replaces references to c by references to d and d is eventually released, then subsequent attempts to acquire c should obtain access to d. In order to account for this scenario, we also install the forwarding pointer from c to d upon a linear forward if the channel has ever been a shared channel with possibly multiple waiting clients.

The current implementation of Concurrent C0 does not deallocate channels that were shared at any point during the program execution. We conjecture that manifest sharing admits an effective reference counting garbage collector by transforming the typing derivation to make implicit applications of weakening and contraction explicit. This is one of the immediately planned items of future work.

8 RELATED WORK

Our work is situated in the family of works on session types [Gay and Hole 2005; Honda 1993; Honda et al. 1998, 2008] among which it extends work based on the Curry-Howard isomorphism between linear logic and session-typed communication [Caires and Pfenning 2010; Caires et al. 2016; Toninho 2015; Toninho et al. 2013; Wadler 2012] with manifest sharing. We have already summarized that work in Section 2 and have pointed out that the shared channels available through the exponential modality in linear logic have a copying semantics and therefore cannot accommodate the examples presented in this paper. Perhaps most closely related is work by Atkey et al. [2016], which proceeds by conflating dual pairs of types in classical linear logic, whereas in this paper we maintain the original interpretation of propositions as session types, but provide an alternative operational semantics for a shared layer of channels separated from the linear types by a pair of adjoint modalities.

From the point of view of protocol expression, our work is related to the line of research that uses typestate [Strom and Yemini 1986] for protocol checking [Bierhoff and Aldrich 2007; DeLine

and Fähndrich 2004; Fähndrich and DeLine 2002; Militão et al. 2014] or program verification [Nistor et al. 2014], in a sequential, object-oriented context. Whereas first approaches [DeLine and Fähndrich 2004] support a rather restricted set of aliasing patterns to facilitate modular protocol checking, subsequent approaches lift some of the imposed restrictions, notably by combining aliasing information with typestate [Bierhoff and Aldrich 2007; Naden et al. 2012] or rely-guarantee-based reasoning [Militão et al. 2014]. Most closely related to our work is Fähndrich's and DeLine's work [2002] on adoption and focus for protocol checking in an object-oriented language. In the resulting language, linear and non-linear objects coexist such that every non-linear object (adoptee) has a linear adopter. Aliases are permitted to adoptees, as long as access goes through the adopter and mutating access happens in a temporary scope, called focus. While an aliased object is in focus, access to the object via another alias is disabled by capability tracking. From this aspect, a focus scope bears ressemblance to a critical section arising between acquire and release points in our system, even though adoption and focus are employed in a purely sequential setting. Whilst capabilities are treated as resources, the underlying type system is not linear, but the required semantics is achieved by threading the capabilities through program execution.

From the point of view of allowing controlled aliasing in a concurrent setting, our work is related to permission-based logics [Boyland 2003; Heule et al. 2013; Leino and Müller 2009; Smans et al. 2009] and concurrent separation logic [Brookes 2004; Jung et al. 2015; O'Hearn 2004; Turon et al. 2013; Vafeiadis 2011]. Permission-based logics maintain a distinction between read and write access to a shared memory location, allowing read access even if only a fractional permission [Boyland 2003] is held, whereas write access requires the entire permission. From a session type perspective, this distinction is less relevant because any communication, input (write) and output (read) alike, amounts to a change in protocol state and thus must be protected sufficiently. Separation logic shares with linear logic the separating conjunction to reason about resource consumption, but uses a Hoare-style reasoning approach that is extrinsic to the type system, whereas resource-awareness is intrinsic to our type system via the Curry-Howard correspondence. Moroever, both permission-based logics and concurrent separation logic target shared-memory concurrency, whereas our work is situated in the realm of message-passing concurrency, offering a different level of abstraction.

Linear types have also found various applications in systems programming. For example, Walker and Watkins [2001] combine linear types with regions [Gifford and Lucassen 1986], and Smith et al. [2000] relax the operational "use-once" semantics of linear types [Wadler 1990] to exploit pointer aliasing for destructive operations. Similar observations have been made by Castegren and Wrigstad [2017] in the context of implementing lock-free algorithms. Our work differs from these approaches in that it is based on a richer semantics of linearity derived from the Curry-Howard isomorphism between linear logic and session-typed communication. Moreover, our work employs a message-passing approach to concurrency rather than a shared-memory-based approach. From this perspective, our work has closer ties with the Rust systems programming language [MozillaResearch 2016], which supports message-passing concurrency in an affine setting. Shared data in Rust is normally immutable, but Rust also supports various abstractions (e.g., mutexes) that support the safe mutation of shared data. We have found that the programming patterns arising in SILL_S readily translate into Rust code with mutexes.

9 DISCUSSION AND FUTURE WORK

We have presented an extension of logic-based session-typed message-passing concurrency by permitting shared resources encapsulated in processes. This allows the elegant expression of examples, such as queues with multiple producers and multiple consumers, dining philosophers, shared databases, shared input and output devices, or nondeterministic choice. In fact, all of the asynchronous π-calculus can now be embedded in a statically typed framework satisfying session fidelity

by modeling π-calculus channels as shared processes maintaining a nondeterministic message buffer. We were able to maintain the view of linear propositions as session types, sequent proofs as processes, and linear proof reduction as communication. To accomodate shared processes, we had to generalize the usual Curry-Howard correspondence and allow interleaved proof construction (acquire), proof reduction (communication), and proof deconstruction (release). Proof construction may fail, which manifests operationally as deadlock. Key insights are the decomposition of the exponential modality $!A$ into $\downarrow^{\mathsf{S}}_{\mathsf{L}}\uparrow^{\mathsf{S}}_{\mathsf{L}}A_{\mathsf{L}}$, inspired by adjoint logic, and the insistence on equi-synchronizing types, which guarantee that a shared process is always released to the same type at which it was acquired. The former makes sharing manifest in the type; the latter guarantees session fidelity without runtime checking of types.

On the theory side, we plan to consider how to overlay a likely very different type system or static analysis in order to recover absence of deadlocks. Some recent promising work in this direction [Kobayashi and Laneve 2017; Lange et al. 2017] in a different context may be adaptable to our situation. We are also interested in relaxing the restriction on equi-synchronization. A first avenue to pursue is to extend our definitions to support subtyping, along the lines of Gay and Hole [2005]. Another possibility is to complement the static approach with run-time type-checking to maintain session fidelity [Jia et al. 2016], particularly in a distributed setting. On the implementation side, we would like to develop the proof-theoretic foundation of a reference counting implementation so that resources associated with shared processes that are no longer accessible can be released.

Finally, the embedding of the asynchronous π-calculus into $\mathsf{SILL}_{\mathsf{S}}$ raises the interesting question of how precise the modeling is. While we can easily relate computation traces, other traditional notions of concurrency theory such as bisimulation do not immediately apply since our semantics is given as a multiset rewriting system. We conjecture that a slightly modified interpretation with late application of nondeterministic choice describes a bisimulation, according to the definitions mapped out by Deng et al. [2016].

ACKNOWLEDGMENTS

The authors would like to thank Peter Thiemann, Philip Wadler, and other participants for comments on a preliminary talk on this work at the Dagstuhl Seminar on Theory and Applications of Behavioural Types, January 29 – February 3, 2017. The authors would also like to thank Bob Harper for discussions on process calculi and coinduction.

This material is based upon work supported by a Mozilla Research grant and partially sponsored by the National Science Foundation under Grant No. CNS-1423168: "Blameworthy Programs: Accountability via Deviance and Causal Determination". Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Mozilla Research or the National Science Foundation.

REFERENCES

Rob Arnold. 2010. *C_0, an Imperative Programming Language for Novice Computer Scientists.* Master's thesis. Department of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-10-145.

Robert Atkey, Sam Lindley, and J. Garrett Morris. 2016. Conflation Confers Concurrency. In *Wadler Festschrift*, S. Lindley et al. (Ed.). Springer LNCS 9600, 32–55.

Stephanie Balzer and Frank Pfenning. 2017. *Manifest Sharing with Session Types.* Technical Report CMU-CS-17-106. Carnegie Mellon University.

Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. 2008. On the Asynchronous Nature of the Asynchronous π-calculus. In *Concurrency, Graphs and Models (Lecture Notes in Computer Science)*, Vol. 5065. Springer, 473–492.

P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models. In *8th International Workshop on Computer Science Logic (CSL) (Lecture Notes in Computer Science)*, Vol. 933. Springer, 121–135. An extended version appeared as Technical Report UCAM-CL-TR-352, University of Cambridge.

Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. ACM, 301–320.

John Boyland. 2003. Checking Interference with Fractional Permissions. In *10th International Symposium on Static Analysis (SAS)*. 55–72.

Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *15th International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Vol. 3170. Springer, 16–34.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *21st International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Vol. 6269. Springer, 222–236.

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423.

Elias Castegren and Tobias Wrigstad. 2017. Relaxed Linear References for Lock-free Data Structures. In *31st European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 74. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 6:1–6:32.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. 2002. *A Concurrent Logical Framework II: Examples and Applications*. Technical Report CMU-CS-02-102. Computer Science Department, Carnegie Mellon University. Revised May 2003.

Iliano Cervesato and Andre Scedrov. 2009. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information and Computation* 207, 10 (2009), 1044–1077.

Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. 2003. *A Judgmental Analysis of Linear Logic*. Technical Report CMU-CS-03-131R. School of Computer Science, Carnegie Mellon University.

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

Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *18th European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 3086. Springer, 465–490.

Yuxin Deng, Robert J. Simmons, and Iliano Cervesato. 2016. Relating Reasoning Methodologies in Linear Logic and Process Algebra. *Mathematical Structure in Computer Science* 26, 5 (Jan. 2016), 868–906.

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Proceedings of the 21st Conference on Computer Science Logic (CSL 2012)*, P. Cégielski and A. Durand (Eds.). Leibniz International Proceedings in Informatics, Fontainebleau, France, 228–242.

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-Oriented Languages. In *20th European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 4067. Springer, 328–352.

Edsger W. Dijkstra. 1971–1973. Hierarchical Ordering of Sequential Processes. (1971–1973). EWD Manuscript 310.

Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 13–24.

Cormac Flanagan and Shaz Qadeer. 2003. A Type and Effect System for Atomicity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 338–349.

Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the π-Calculus. *Acta Informatica* 42, 2–3 (2005), 191–225.

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *LISP and Functional Programming*. 28–38.

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

Dennis Griffith. 2016. *Polarized Substructural Session Types*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Dennis Griffith and Frank Pfenning. 2015. SILL. https://github.com/ISANobody/sill. (2015).

Robert Harper. 2013. *Practical Foundations for Programming Languages*. Cambridge University Press.

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, Vol. 7737. Springer, 315–334.

Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Vol. 715. Springer, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 1381. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 273–284.

W. A. Howard. 1969. The Formulae-as-Types Notion of Construction. (1969). Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press

(1980).
Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *22nd European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 5142. Springer, 516–541.
Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *11th ACM SIGPLAN Workshop on Generic Programming (WGP)*.
Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. Monitors and Blame Assignment for Higher-Order Session Types. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 582–594.
Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 637–650.
Naoki Kobayashi and Cosimo Laneve. 2017. Deadlock Analysis of Unbounded Process Networks. *Information and Computation* 252 (2017), 48–70.
Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. Fencing off Go: Liveness and Safety for Channel-Based Programming. In *44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 748–761.
K. Rustan M. Leino and Peter Müller. 2009. A Basis for Verifying Multi-Threaded Programs. In *18th European Symposium on Programming (ESOP)*. 378–393.
Damiano Mazza. 2016. The True Concurrency of Differential Interaction Nets. *Mathematical Structures in Computer Science* (11 2016), 1–29.
Filipe Militão, Jonathan Aldrich, and Luís Caires. 2014. Rely-Guarantee Protocols. In *28th European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 8586. Springer, 334–359.
Robin Milner. 1999. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press.
MozillaResearch. 2016. The Rust Programming Language. https://doc.rust-lang.org/stable/book. (November 2016).
Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A Type System for Borrowing Permissions. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 557–570.
Rumyana Neykova and Nobuko Yoshida. 2014. Multiparty Session Actors. In *16th International Conference on Coordination Models and Languages (COORDINATION) (Lecture Notes in Computer Science)*, Vol. 8459. Springer, 131–146.
Ligia Nistor, Jonathan Aldrich, Stephanie Balzer, and Hannes Mehnert. 2014. Object Propositions. In *19th International Symposium on Formal Methods (FM) (Lecture Notes in Computer Science)*, Vol. 8442. Springer, 497–513.
Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *15th International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Vol. 3170. Springer, 49–67.
Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logical Relations and Observational Equivalences for Session-Based Concurrency. *Information and Computation* 239 (2014), 254–302.
Frank Pfenning. 2010. C0 Language. http://c0.typesafety.net. (2010).
Frank Pfenning, Thomas J. Cortina, and William Lovas. 2011. Teaching Imperative Programming With Contracts at the Freshmen Level. (2011). Unpublished note.
Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS) (Lecture Notes in Computer Science)*, Vol. 9034. Springer, 3–22.
Jason Reed. 2009. A Judgmental Deconstruction of Modal Logic. (January 2009). http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf Unpublished manuscript.
Davide Sangiorgi and David Walker. 2001. *The π-Calculus - A Theory of Mobile Processes.* Cambridge University Press.
Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 56. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 21:1–21:28.
Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *23rd European Conference on Object-Oriented Programming (ECOOP'09) (Lecture Notes in Computer Science)*, Vol. 5653. Springer, 148–172.
Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *9th European Symposium on Programming (ESOP)*. 366–381.
Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering (TSE)* 12, 1 (1986), 157–171.
Bernardo Toninho. 2015. *A Logical Foundation for Session-based Concurrent Computation.* Ph.D. Dissertation. Carnegie Mellon University and New University of Lisbon.
Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: a Monadic Integration. In *22nd European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 7792. Springer, 350–369.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 377–390.

Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *Electronic Notes in Theoretical Computer Science* 276 (2011), 335–351.

Philip Wadler. 1990. Linear Types Can Change the World!. In *Woking Conference on Programming Concepts and Methods*.

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

David Walker and Kevin Watkins. 2001. On Regions and Linear Types. In *6th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. ACM, 181–192.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2002. *A Concurrent Logical Framework I: Judgments and Properties*. Technical Report CMU-CS-02-101. Computer Science Department, Carnegie Mellon University. Revised May 2003.

Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2016. Design and Implementation of Concurrent C0. In *Fourth International Workshop on Linearity*.