Propositions-as-Types and Shared State

PEDRO ROCHA, NOVA LINCS and FCT NOVA, NOVA University Lisbon, Portugal
LUÍS CAIRES, NOVA LINCS and FCT NOVA, NOVA University Lisbon, Portugal

We develop a principled integration of shared mutable state into a proposition-as-types linear logic interpretation of a session-based concurrent programming language. While the foundation of type systems for the functional core of programming languages often builds on the proposition-as-types correspondence, automatically ensuring strong safety and liveness properties, imperative features have mostly been handled by extra-logical constructions. Our system crucially builds on the integration of nondeterminism and sharing, inspired by logical rules of differential linear logic, and ensures session fidelity, progress, confluence and normalisation, while being able to handle first-class shareable reference cells storing any persistent object. We also show how preservation and, perhaps surprisingly, progress, resiliently survive in a natural extension of our language with first-class locks. We illustrate the expressiveness of our language with examples highlighting detailed features, up to simple shareable concurrent ADTs.

CCS Concepts: • **Theory of computation → Linear logic**; **Process calculi**; **Type structures**; • **Software and its engineering → Concurrent programming languages**.

Additional Key Words and Phrases: Propositions-as-Types, Shared State, Session Types

ACM Reference Format:
Pedro Rocha and Luís Caires. 2021. Propositions-as-Types and Shared State. *Proc. ACM Program. Lang.* 5, ICFP, Article 79 (August 2021), 30 pages. https://doi.org/10.1145/3473584

1 INTRODUCTION

In this paper we introduce a propositions-as-types (PaT) approach to mutable state, defined as a conservative extension of the classical linear logic interpretation for session-typed processes [Caires et al. 2016; Wadler 2012] with first-class shareable reference cells. Modern software construction depends on imperative state, state sharing, and concurrency, with even mainstream functional programming languages such as ML variants or Haskell offering explicit typed support for mutable state. Still, safely programming with state, aliasing and concurrency (and reasoning about the combination) has always been considered a significant challenge, with the study of programming abstractions, logics and type systems for disciplining interference being a prolific research topic for decades [Ahmed et al. 2007; Caires and Seco 2013; Jung et al. 2018a; Nanevski et al. 2008; O'Hearn and Reynolds 2000; Sunshine et al. 2011].

While the design of type systems for the functional core of programming languages (e.g., the polymorphic lambda calculus) conveniently connects to the PaT correspondence [Cardelli 1991; Howard 1980; Wadler 2015]), which intrinsically ensures strong safety and liveness properties, the notion of mutable state seems to escape from the paradigm, notwithstanding the long identified deep connections between linear logic and stateful computation [Wadler 1990a], and recent advances such as [Balzer and Pfenning 2017]. Imperative state is then usually expressed by diverging from

Authors' addresses: Pedro Rocha, NOVA LINCS and FCT NOVA, NOVA University Lisbon, Portugal, pms.rocha@campus.fct.unl.pt; Luís Caires, NOVA LINCS and FCT NOVA, NOVA University Lisbon, Portugal, lcaires@fct.unl.pt.

2475-1421/2021/8-ART79
https://doi.org/10.1145/3473584

logical principles; a consequence is that many important properties offered “for free" by PaT, such as type preservation, progress / deadlock freedom, equational reasoning on program behaviour via proof equivalence, or (strong) normalisation, do not automatically result from the basic typing discipline, and need in general additional extra-logical techniques to be achieved.

We thus believe that computational models for mutable state subscribing to the full ideology of PaT (proofs as programs, formulas as types, evaluation as proof simplification) are still lacking.

A main challenge is that while in PaT computation corresponds to proof reduction, that is, the observable behaviour of a program term does not change under cut reduction / elimination, and the normal form summarises the program / proof denotation, the execution of programs with shared state and concurrency typically induces non-deterministic behaviour, e.g, as in concurrent writes to the same memory cell. Non-determinism in computations is often mentioned as an obstacle for PaT, since if may lead to loss of confluence in a (candidate) notion of proof simplification. Moreover, the existence of proof normal forms may be also compromised in the presence of shared state, since non-termination may be expressed with a higher-order store (e.g., Landin’s knot).

In the seminal PaT correspondences between session processes and linear logic [Caires and Pfenning 2010; Wadler 2012], there is no “real" non-determinism. Programs exhibit parallelism and a degree of concurrency, in the sense that many sessions may run simultaneously and independently, but the computation is nevertheless confluent. The key reason is that no races occur in communications within a linear session or even between different linear sessions. Also, processes may concurrently invoke the same replicated server, however, the typing discipline induced by the promotion rule for $!A$ ensures that the server behaviour is identical for each invocation (cf. uniform receptiveness of shared names [Sangiorgi 1999]). In such a setting, the overall computation remains essentially “functional" [Toninho and Yoshida 2021], which is not surprising given the well known interpretations of linear logic as linear lambda calculi.

The question then arises about wether and how mutable shared state might be faithfully expressed in a PaT model. In this work we approach the issue by exploring the Curry-Howard correspondence from a programming language perspective, building on a conservative extension of standard classical linear logic. As suggested by Cardelli [Cardelli 1991], “one can extrapolate [the] correspondence and turn it into a predictive tool: if a concept is present in type theory but absent in programming, or vice-versa, it can be very fruitful to both areas to investigate and see what the corresponding concept might be in the other context".

Our system fully complies with the PaT ideology (proofs as programs, formulas as types, evaluation as proof simplification). Its core ingredients are: (a) the introduction of two dual "exponential" modalities and the interpretation of their proof rules as language constructs for manipulating reference cells and (b) a principled integration of the well-known interplay between concurrency and non-determinism in algebraic concurrency theory [Hennessy and Milner 1985] now emerging from our proof theoretic analysis. Our typing rules for the imperative fragment are inspired by those for the exponentials and sum connectives of differential linear logic (DiLL) [Ehrhard 2018; Ehrhard and Regnier 2006], allowing us to internalise non-determinism equationally (an idea also explored by Beffara in the context of process algebra [Beffara 2008]), thus allowing us to capture concurrent interactions on shared state equationally, by cut-reduction / proof simplification.

Propositions-as-types pays back: programs in our basic language naturally enjoy type preservation, progress, confluence and normalisation. We also show how the key properties of our system, including progress / deadlock freedom, resiliently survive to a principled extension of our language with first-class locks. The intrinsic modularity of the PaT approach naturally ensures its compatibility with all constructs in related work that as well follows PaT, including parametric polymorphism [Caires et al. 2013; Wadler 2012], dependent types [Toninho et al. 2011], code mobility [Toninho et al. 2013], or multiparty protocols [Caires and Pérez 2016; Carbone et al. 2016]. As an example, we illustrate how the inclusion of second-order quantifiers [Caires et al. 2013; Wadler 2012] allows us to elegantly express concurrent and shareable stateful ADTs within our basic PaT framework.

1.1 Overview

We provide a high-level overview of πSS, which is a session-typed π-calculus with shared state, anticipating the technical presentation of the next sections. Our basic session discipline (see [Honda 1993; Honda et al. 1998]) applies to concurrent processes which communicate through private channels connecting two partners and is related to Linear Logic (LL) via a Curry-Howard correspondence: types correspond to propositions and are assigned to session channels, processes to proofs, and process congruence/reduction to proof simplification. In fact, πSS is a conservative extension of the fundamental session-based interpretations of LL [Caires and Pfenning 2010; Wadler 2012].

Within a linear session two parties communicating on each endpoint of the session channel comply with a certain protocol, characterised by a session type. Duality on types, $(A/\overline{A})$ where A and $\overline{A}$ are dual types, corresponds to linear logic negation and captures symmetry in interaction: when one process closes, the other waits for the session to be closed $(\mathbf{1}/\bot)$; when one sends, the other receives $(A \otimes B/\overline{A} \parr \overline{B})$; when one offers a menu, the other chooses an option from the menu $(A \mathbin{\&} B/\overline{A} \oplus \overline{B})$, and so on. The exponentials $(!A/?\overline{A})$ type replicated servers and their invocation by clients. Composition of two processes P, Q interacting on a session x of type A is represented by the cut rule, written $P \;\;|x : A|\;\; Q$, where P uses channel x at type A and process Q uses x at type $\overline{A}$. Let us start by considering the following simple session process.

$$\begin{aligned}\mathsf{menu}\ m \triangleq\quad & \mathsf{case}\ m\ \{|\mathsf{inl} : \mathsf{recv}\ m(b); \mathsf{send}\ m(b'.\mathsf{not}\ b\ b'); \mathsf{close}\ m\\ &\qquad\qquad\quad |\mathsf{inr} : \mathsf{recv}\ m(b_1); \mathsf{recv}\ m(b_2); \mathsf{send}\ m(b'.\mathsf{and}\ b_1\ b_2\ b'); \mathsf{close}\ m\}\end{aligned}$$

Process menu m offers a menu with two choices: inl or inr. In case the choice is inl, it receives a boolean b, computes the negation of b (to b') and sends back the result b'. In case the choice is inr, it receives two booleans b_1 and b_2, computes their conjunction (to b') and sends back the result b'. Both branches finalize the session m by closing it, no further interactions will take place. Process menu m offers on m a protocol of type $\mathsf{Menu} \triangleq (\overline{\mathsf{Bool}} \parr (\mathsf{Bool} \otimes \mathbf{1})) \mathbin{\&} (\overline{\mathsf{Bool}} \parr (\overline{\mathsf{Bool}} \parr (\mathsf{Bool} \otimes \mathbf{1})))$. Process client m is a client of menu m, providing at m the dual type $\overline{\mathsf{Menu}}$

$$\mathsf{client}\ m \triangleq m.\mathsf{inr}; \mathsf{send}\ m(b_1.\mathsf{ltrue}\ b_1); \mathsf{send}\ m(b_2.\mathsf{lfalse}\ b_2); \mathsf{recv}\ m(b'); \mathsf{wait}\ m; \mathsf{print}\ b'$$

It chooses the option inr (conjunction), sends two linear booleans, receives a result b' on m, waits for the session to be closed and then prints b'. Since menu m and client m have dual types on session m they can be composed with a cut

$$\mathsf{system}_0 \triangleq \mathsf{menu}\ m \;\;|m : \mathsf{Menu}|\;\; \mathsf{client}\ \ m$$

Execution of system$_0$ will eventually result in the boolean value false to be printed out. Process menu m executes and terminates after interacting with the client. Then, one may define a replicated server on name s, by server $s \triangleq\ !s(m); \mathsf{menu}\ m$. It is typed by $!\mathsf{Menu}$ and persistently offers the boolean menu, spawning a linear session on a fresh m (of type Menu) on each invocation.

Our basic πCLL language corresponds exactly to second-order Classical Linear Logic with mix. The main novelty in this paper results from the addition of two duality-related modalities to type mutable state: $\boxed{?}A$ (read *box why not*) which type reference cells, and $\boxed{!}\overline{A}$ (read *box bang*) which type cell usage, and which are governed by logical principles that take inspiration in DiLL [Ehrhard 2018]. The associated basic constructs are processes (proofs) that represent first class reference cells storing persistent values, and cell usage operations read, write and free. For example, consider

$$\mathsf{system}_1 \triangleq \mathsf{cell}\ c(b.\mathsf{true}\ b) \;\;|c : \boxed{?}\mathsf{Bool}|\;\; \mathsf{read}\ c(b_r); \mathsf{wrt}\ c(b'.\mathsf{false}\ b'); \mathsf{free}\ c; P(b_r)$$

In system_1 a reference cell at c, initialised with the persistent boolean value true and typed by $c : \boxed{?}\text{Bool}$, is composed (by cut) with a process that uses c at type $\overline{\boxed{!}\text{Bool}}$. It first reads the cell, then writes the boolean false, frees the usage, and continues as process $P(b)$. Reference cells and usages are handled linearly, therefore cannot be implicitly discarded nor copied. Releasing a cell usage is expressed by the free operation (which corresponds to co-weakening). In example system_1, after the cell usage is released, cell deallocation occurs since there is no other usage of it; notice that c does not occur in $P(b)$, as enforced by typing of free. Now, the dynamic semantics of our language is defined from a structural congruence ($\equiv$) and reduction ($\to$) relations (Definitions 6.3, 6.4), where reduction is closed under structural congruence; and we have the reductions

$$\begin{aligned}\text{system}_1 &\to \text{true } b_r \ |b_r : !\text{Bool}|\ (\text{cell } c(b.\text{true } b)\ |c : \boxed{?}\text{Bool}|\ \text{wrt } c(b'.\text{false } b'); \text{free } c; P(b_r))\\ &\to \text{true } b_r \ |b_r : !\text{Bool}|\ (\text{cell } c(b'.\text{false } b')\ |c : \boxed{?}\text{Bool}|\ \text{free } c; P(b_r))\\ &\to \text{true } b_r \ |b_r : !\text{Bool}|\ P(b_r)\end{aligned}$$

We now turn to sharing: our language supports sharing of cell usages via the share c operation which corresponds to duplication of a cell usage c (typing rule for share corresponds to co-contraction). Consider the example

$$\text{system}_2 \triangleq \text{cell } c(b.\text{true } b)\ |c : \boxed{?}\text{Bool}|\ \text{share } c\ \{\text{wrt } c(b_1.\text{true } b_1); P_1 \mid\mid \text{wrt } c(b_2.\text{false } b_2); P_2\}$$

system_2 illustrates two concurrent threads sharing a memory cell: one thread writes the boolean true and continues as P_1, whereas the other thread writes false and continues as P_2.

We look in detail how system_2 executes. First, system_2 is expanded into a sum $(P + Q)$ in which the two concurrent write operations are interleaved and brought to evidence, using the fundamental structural conversion law $\equiv$ [AA] which relates concurrency with nondeterminism. This yields

$$\begin{aligned}\text{system}_2 \equiv \text{cell } c(b.\text{true } b)\ |c : \boxed{?}\text{Bool}|\ (&\text{wrt } c(b_1.\text{true } b_1); \text{wrt } c(b_2.\text{false } b_2); \text{share } c\ \{P_1 \mid\mid P_2\} +\\ &\text{wrt } c(b_1.\text{false } b_1); \text{wrt } c(b_2.\text{true } b_2); \text{share } c\ \{P_1 \mid\mid P_2\})\end{aligned}$$

A process of the form $P + Q$ represents a non-deterministic choice between alternatives P and Q, each offering the same interface and typing, cf. sum in process algebra. Sums are also present in DiLL [Ehrhard 2018] where cut elimination need to genarate sums of proofs. In our model, sum satisfy the expected axioms of nondeterministic sums of process algebras [Hennessy and Milner 1985], like commutativity, associativity and idempotency, remarkably modelled as logical proof conversions. Now, further laws of $\equiv$ allows cut to be distributed over sum, so that

$$\begin{aligned}\text{system}_2 \equiv\ &(\text{cell } c(b.\text{true } b)\ |c : \boxed{?}\text{Bool}|\ \text{wrt } c(b_1.\text{true } b_1); \text{wrt } c(b_2.\text{false } b_2); \text{share } c\ \{P_1 \mid\mid P_2\})\\ &+ (\text{cell } c(b.\text{true } b)\ |c : \boxed{?}\text{Bool}|\ \text{wrt } c(b_2.\text{false } b_2); \text{wrt } c(b_1.\text{true } b_1); \text{share } c\ \{P_1 \mid\mid P_2\})\end{aligned}$$

Using reduction on each summand to execute the write operations, last writer wins, and we obtain

$$\text{system}_2 \quad \overset{*}{\to} \quad (\text{cell } c(b_2.\text{false } b_2) + \text{cell } c(b_1.\text{true } b_1))\ |c : \boxed{?}\text{Bool}|\ \text{share } c\ \{P_1 \mid\mid P_2\}$$

Notice how sums internalise nondeterminism in processes / proof terms, allowing us to define confluent proof simplification. In this example, all possible reduction sequences from system_2 reach a unique process up to structural congruence after performing the four write operations.

As mentioned above, at some point in computation, all cell clients must release their (shared) cell usages. It is interesting to give an intuitive understanding about how our semantics handles release of shared cell usages, that lazily leads to ultimate cell deallocation when no usage remains active; when this happens cells get "deallocated". For example, consider the process

$$\text{system}_3 \triangleq \text{cell } c(b.\text{true } b)\ |c : \boxed{?}\text{Bool}|\ \text{share } c\ \{\text{wrt } c(b'.\text{false } b'); \text{free } c; P \mid\mid \text{free } c; Q\}$$

In this case, when a read or write operation competes with a free operation, the free operation gets postponed, this is captured by a structural conversion ≡ law [FA].

$$\begin{aligned} \text{system}_3 \quad & \equiv \mathsf{cell}\ c(b.\mathsf{true}\ b)\ \ |c : \boxed{?}\mathsf{Bool}|\ \ \mathsf{wrt}\ c(b'.\mathsf{false}\ b'); \mathsf{share}\ c\ \{\mathsf{free}\ c; P \mid\mid \mathsf{free}\ c; Q\} \\ & \to \mathsf{cell}\ c(b'.\mathsf{false}\ b')\ \ |c : \boxed{?}\mathsf{Bool}|\ \ \mathsf{share}\ c\ \{\mathsf{free}\ c; P \mid\mid \mathsf{free}\ c; Q\} \end{aligned}$$

Then, by ≡ law [FF], the two shared free operations are collapsed into a single one that is then propagated up in the sharing tree. All free operations executed by threads sharing c eventually collapse into a *single* free operation, which then causes the cell to be erased.

$$\begin{aligned} & \mathsf{cell}\ c(b'.\mathsf{false}\ b')\ \ |c : \boxed{?}\mathsf{Bool}|\ \ \mathsf{share}\ c\ \{\mathsf{free}\ c; P \mid\mid \mathsf{free}\ c; Q\} \\ & \equiv \mathsf{cell}\ c(b'.\mathsf{false}\ b')\ \ |c : \boxed{?}\mathsf{Bool}|\ \ \mathsf{free}\ c; (P \mid\mid Q) \quad \to \quad P \mid\mid Q \end{aligned}$$

In this example, after freeing up c, the thread continuations P and Q will continue running in parallel (via mix) but of course they are no longer accessing the cell.

1.2 Contributions and Structure of the Paper

In Section 2 we review the polymorphic session-typed calculus πCLL, which exactly corresponds to second-order Classical Linear Logic with mix. In Section 3 we introduce our process model and type system πSS, which extends πCLL with shared state, sums, and two novel dual modalities inspired by principles of differential linear logic. Considerations about the structure of proof reductions motivate a simple extension πSSL of our basic system πSS with locking primitives, which we present in Section 4. These languages are conservative extensions of classical linear logic according to $\pi\text{CLL} \subset \pi\text{SS} \subset \pi\text{SSL}$. In Section 5 we showcase the expressiveness of our language πSSL by implementing a mutable shareable stack and queue objects. This also illustrates how the presence of standard existential and universal type quantifiers [Cardelli and Wegner 1985; Mitchell and Plotkin 1988] harmoniously combine with the basic stateful framework of πSSL, allowing us to express mutable ADTs. Throughout the paper several other examples are provided. In Section 6 we formalise the reduction semantics of πSSL and prove that, as a consequence of the proof-theoretic approach, the language enjoys type preservation (Theorem 6.6) and deadlock-freedom (Theorem 6.14). In the following Sections, we prove key meta-theoretic properties of our system. In Section 7 we establish that our system πSSL, internalising non-determinism with sum processes, enjoys confluence (Theorem 7.8), thus proof reductions and conversions represent proof identities or behavioural equivalences. As a consequence, we also reveal a new connection between logic interpretations of programming languages, concurrency and non-determinism. In Section 8 we study proof normalisation / cut-elimination (Theorem 8.4) for the sub-calculus of πSS without quantifiers. A consequence of the sub-formula property of cut-free proofs (Proposition 8.3) is that any system that implements standard linear logic based session types (even if it manipulates state) normalises to a (non-deterministic) cut-free process that does not manipulate state (Corollary 8.5). In Section 9 we give a brief description of out prototype type checker and interpreter, which was submitted as an artifact for evaluation [Rocha and Caires 2021b], as companion to this paper. Finally, in Section 10 we discuss related work, Section 11 offers closing remarks and discusses future work. Complete definitions and detailed proofs are in the technical report [Rocha and Caires 2021a].

2 THE BASIC LANGUAGE πCLL

Our starting point is the typed language πCLL for session-based interacting processes, akin to the session π-calculus, related to classical linear logic (with mix) via a Curry-Howard correspondence [Caires et al. 2013; Caires and Pfenning 2010; Caires et al. 2016; Wadler 2012]: (session) types correspond to propositions (and types are assigned to session channels), processes correspond to proofs, and process congruence/reduction corresponds to proof simplification.

$$\frac{}{\mathbf{0} \vdash \emptyset;\Gamma}\,[\mathrm{T}\mathbf{0}] \qquad \frac{P \vdash \Delta';\Gamma \quad Q \vdash \Delta;\Gamma}{P \mid\mid Q \ \vdash \Delta',\Delta;\Gamma}\,[\mathrm{Tpar}]$$

$$\frac{}{\mathsf{fwd}\ x\ y\ \vdash x:\overline{A}, y:A;\Gamma}\,[\mathrm{Tfwd}] \qquad \frac{P \vdash \Delta', x:A;\Gamma \quad Q \vdash \Delta, x:\overline{A};\Gamma}{P\ \ |x:A|\ \ Q\ \vdash \Delta',\Delta;\Gamma}\,[\mathrm{Tcut}]$$

$$\frac{}{\mathsf{close}\ x \vdash x:\mathbf{1};\Gamma}\,[\mathrm{T}\mathbf{1}] \qquad \frac{Q \vdash \Delta;\Gamma}{\mathsf{wait}\ x; Q\ \vdash \Delta, x:\bot;\Gamma}\,[\mathrm{T}\bot]$$

$$\frac{P_1 \vdash \Delta, x:A;\Gamma \quad P_2 \vdash \Delta, x:B;\Gamma}{\mathsf{case}\ x\ \{|\mathsf{inl}:P_1\ |\mathsf{inr}:P_2\}\ \vdash \Delta, x:A \mathbin{\&} B;\Gamma}\,[\mathrm{T}\&]$$

$$\frac{Q_1 \vdash \Delta', x:\overline{A};\Gamma}{x.\mathsf{inl}; Q_1\ \vdash \Delta', x:\overline{A}\oplus\overline{B};\Gamma}\,[\mathrm{T}\oplus_l] \qquad \frac{Q_2 \vdash \Delta', x:\overline{B};\Gamma}{x.\mathsf{inr}; Q_2\ \vdash \Delta', x:\overline{A}\oplus\overline{B};\Gamma}\,[\mathrm{T}\oplus_r]$$

$$\frac{P_1 \vdash \Delta_1, y:A;\Gamma \quad P_2 \vdash \Delta_2, x:B;\Gamma}{\mathsf{send}\ x(y.P_1); P_2\ \vdash \Delta_1,\Delta_2, x:A\otimes B;\Gamma}\,[\mathrm{T}\otimes] \qquad \frac{Q \vdash \Delta, z:\overline{A}, x:\overline{B};\Gamma}{\mathsf{recv}\ x(z); Q \vdash \Delta, x:\overline{A} \parr \overline{B};\Gamma}\,[\mathrm{T}\parr]$$

$$\frac{P \vdash y:A;\Gamma}{!x(y); P\ \vdash x:!A;\Gamma}\,[\mathrm{T}!] \qquad \frac{Q \vdash \Delta;\Gamma, x:\overline{A}}{?x; Q\ \vdash \Delta, x:?\overline{A};\Gamma}\,[\mathrm{T}?]$$

$$\frac{P \vdash y:A;\Gamma \quad Q \vdash \Delta;\Gamma, x:\overline{A}}{y.P\ |!x:A|\ Q\ \vdash \Delta;\Gamma}\,[\mathrm{Tcut!}] \qquad \frac{Q \vdash \Delta, z:\overline{A};\Gamma, x:\overline{A}}{\mathsf{call}\ x(z); Q\ \vdash \Delta;\Gamma, x:\overline{A}}\,[\mathrm{Tcall}]$$

$$\frac{P \vdash \Delta, x:\{B/X\}A;\Gamma}{\mathsf{send}\ x\ B; P\ \vdash \Delta, x:\exists X.A;\Gamma}\,[\mathrm{T}\exists] \qquad \frac{Q \vdash \Delta, x:\overline{A};\Gamma}{\mathsf{recv}\ x(X); Q \vdash \Delta, x:\forall X.\overline{A};\Gamma}\,[\mathrm{T}\forall]$$

$$\frac{P \vdash y:!A;\Gamma}{\mathsf{cell}\ x(y.P) \vdash x:\boxed{?}A;\Gamma}\,[\mathrm{Tcell}] \qquad \frac{Q \vdash \Delta;\Gamma}{\mathsf{free}\ x; Q \vdash \Delta, x:\boxed{!}\overline{A};\Gamma}\,[\mathrm{Tfree}]$$

$$\frac{Q \vdash \Delta, z:?\overline{A}, x:\boxed{!}\overline{A};\Gamma}{\mathsf{read}\ x(z); Q \vdash \Delta, x:\boxed{!}\overline{A};\Gamma}\,[\mathrm{Tread}] \qquad \frac{Q_1 \vdash z:!A;\Gamma \quad Q_2 \vdash \Delta, x:\boxed{!}\overline{A};\Gamma}{\mathsf{wrt}\ x(z.Q_1); Q_2\ \vdash \Delta, x:\boxed{!}\overline{A};\Gamma}\,[\mathrm{Twrite}]$$

$$\frac{P \vdash \Delta', x:\boxed{!}\overline{A};\Gamma \quad Q \vdash \Delta, x:\boxed{!}\overline{A};\Gamma}{\mathsf{share}\ x\ \{P \mid\mid Q\}\ \vdash \Delta',\Delta, x:\boxed{!}\overline{A};\Gamma}\,[\mathrm{Tshare}] \qquad \frac{P \vdash \Delta;\Gamma \quad Q \vdash \Delta;\Gamma}{P + Q \vdash \Delta;\Gamma}\,[\mathrm{Tsum}]$$

Fig. 1. Typing Rules $P \vdash \Delta;\Gamma$ for πSS.

Typing Judgements. Typing judgments are of the form

$$P \vdash \underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Delta}; \underbrace{y_1 : B_1, \ldots, y_m : B_m}_{\Gamma}$$

and mean that a process P, seen as a black box, offers a set of channel endpoints $x_1, \ldots, x_n, y_1, \ldots, y_m$ that follow a protocol specified by their respective types $A_1, \ldots, A_n, B_1, \ldots, B_m$. Later, we will see how to connect two channel endpoints x and y that have dual protocols $x : A$ and $y : \overline{A}$.

Since we are interpreting a dyadic formulation of CLL, typing contexts are separated (with a semi-colon) into two parts: a linear part denoted by Δ and an unrestricted (or exponential) part denoted by Γ. Δ is linearly handled, which means that we cannot copy nor discard channels, whereas Γ allows weakening and contraction. This dyadic formulation was introduced in [Andreoli 1992] and adopted in the session-based interpretations of [Caires and Pfenning 2010; Caires et al. 2016].

We assume that channel names in Δ and Γ are pairwise distinct. The empty context is written $\emptyset$. If Γ is empty we write just $P \vdash \Delta$ instead of $P \vdash \Delta; \emptyset$. We write Δ, Δ' (two comma-separated contexts) for the disjoint union of Δ and Δ'. All the typing rules of πSS are displayed in Figure 1.

We now go through each of the connectives of πCLL, presenting their associated typing rules (static semantics) and principal cut-reductions (dynamic semantics) in detail.

Inaction, Cut, Parallel. Inaction and composition are typed by the rules

$$\frac{}{\mathbf{0} \vdash \emptyset; \Gamma}\,[\mathrm{T}\mathbf{0}] \qquad \frac{P \vdash \Delta', x : A; \Gamma \quad Q \vdash \Delta, x : \overline{A}; \Gamma}{P \;\; |x : A| \;\; Q \;\vdash \Delta', \Delta; \Gamma}\,[\mathrm{Tcut}] \qquad \frac{P \vdash \Delta'; \Gamma \quad Q \vdash \Delta; \Gamma}{P \mid\mid Q \;\vdash \Delta', \Delta; \Gamma}\,[\mathrm{Tpar}]$$

Rule [T$\mathbf{0}$] is computationally interpreted by the inaction $\mathbf{0}$ process, which types with the empty linear typing context and an arbitrary unrestricted context Γ, and operationally it does nothing.

The logical cut rule [Tcut] is interpreted by interactive composition. Processes P and Q run concurrently, interacting on a single private linear session x. Process P provides a behaviour of type A along x, whereas Q offers on x a dual behaviour of type $\overline{A}$. Cut is annotated with the type of its left argument, but we sometimes omit this annotation.

Finally, rule [Tpar] corresponds to linear logic mix rule and types the independent composition of two processes P and Q, which run in parallel without ever interfering with each other, but offering together a common interface. Since the linear typing context is linearly handled, we assume that Δ and Δ' are disjoint. The same principle applies to [Tcut].

Session Termination. The dual types $\mathbf{1}$ (one) and $\bot$ (bottom) type session termination

$$\frac{}{\mathsf{close}\; x \vdash x : \mathbf{1}; \Gamma}\,[\mathrm{T}\mathbf{1}] \qquad \frac{Q \vdash \Delta; \Gamma}{\mathsf{wait}\; x; Q \;\vdash \Delta, x : \bot; \Gamma}\,[\mathrm{T}\bot]$$

Process close x closes a session x and wait $x; Q$ waits for the session to be closed, continuing as Q. The interaction is modelled by the reduction rule

$$\mathsf{close}\; x \;\; |x : \mathbf{1}| \;\; \mathsf{wait}\; x; Q \quad \to \quad Q \qquad [\mathbf{1}\bot]$$

where, after the interaction, the session on x disappears. As already highlighted, process reductions in our language correspond to adequate proof conversions, and are intrinsically type preserving (subject reduction holds (Theorem 6.6)). We illustrate this correspondence with rule [$\mathbf{1}\bot$]:

$$\frac{\dfrac{}{\mathsf{close}\; x \vdash x : \mathbf{1}; \Gamma}\,[\mathrm{T}\mathbf{1}] \qquad \dfrac{Q \vdash \Delta; \Gamma}{\mathsf{wait}\; x; Q \;\vdash \Delta, x : \bot; \Gamma}\,[\mathrm{T}\bot]}{\mathsf{close}\; x \;\; |x| \;\; \mathsf{wait}\; x; Q \vdash \Delta; \Gamma}\,[\mathrm{Tcut}] \quad \to \quad Q \vdash \Delta; \Gamma$$

Forwarding. Process fwd $x\, y$ interprets the identity axiom (on linear names)

$$\frac{}{\mathsf{fwd}\; x\; y \;\vdash x : \overline{A}, y : A; \Gamma}\,[\mathrm{Tfwd}]$$

Operationally, it acts as a link redirecting all interactions between channels x and y. The associated reduction is implemented by name substitution [Caires et al. 2012]

$$\mathsf{fwd}\; x\; y \;\; |y : A| \;\; P \quad \to \quad \{x/y\}P \qquad [\mathrm{fwd}]$$

Send and Receive. The dual types $A \otimes B$ (tensor) and $\overline{A} \parr \overline{B}$ (par) type session communication

$$\frac{P_1 \vdash \Delta_1, y : A; \Gamma \quad P_2 \vdash \Delta_2, x : B; \Gamma}{\mathsf{send}\ x(y.P_1); P_2\ \vdash \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\ [\mathrm{T}\otimes] \qquad \frac{Q \vdash \Delta, z : \overline{A}, x : \overline{B}; \Gamma}{\mathsf{recv}\ x(z); Q \vdash \Delta, x : \overline{A} \parr \overline{B}; \Gamma}\ [\mathrm{T}\parr]$$

We abbreviate $A \multimap B \triangleq \overline{A} \parr B$. Process $\mathsf{send}\ x(y.P_1); P_2$ sends a fresh session channel y on session x. Dually, process $\mathsf{recv}\ x(z); Q$ receives a name z on session x. Name y is bound in P_1 and z is bound in Q. The send process is composed of two independent parts: a term P_1 which implements the session on (fresh) channel $y : A$ to be sent, and a term P_2 which provides the continuation session behaviour on $x : B$. The associated reduction is expressed by

$$\mathsf{send}\ x(y.P_1); P_2\ \ |x : A \otimes B|\ \ \mathsf{recv}\ x(z); Q \quad \to \quad P_2\ \ |x : B|\ \ (P_1\ \ |y : A|\ \ \{y/z\}Q) \qquad [\otimes\parr]$$

Notice that a cut on a session $A \otimes B$ gives origin to two *lower rank* cuts on sessions A and B. The inner cut on y connects the continuation Q of the receiver with the provider P_1 of the sent channel, whereas the outer cut on x connects Q to the continuation P_2 of the sending process. Observe that the type associated with session x evolves from $A \otimes B$ to B upon communication.

As in [Caires and Pfenning 2010; Wadler 2012], only fresh (bound) names are sent in communication, following the *internal* mobility discipline of Boreale [Boreale 1996], as opposed to *external* mobility in which free names can be transmitted. However, the free output of (free) linear name y can be encoded as $\mathsf{send}\ x\ y; P\ \triangleq\ \mathsf{send}\ x(z.\ \mathsf{fwd}\ y\ z); P$. The following typing rule is then admissible

$$\frac{P \vdash \Delta, x : B; \Gamma}{\mathsf{send}\ x\ y; P \vdash \Delta, y : \overline{A}, x : A \otimes B; \Gamma}\ [\mathrm{T}\otimes_f]$$

Offer and Choice. The dual types $A \& B$ (with) and $\overline{A} \oplus \overline{B}$ (plus) type offer and choice

$$\frac{P_1 \vdash \Delta, x : A; \Gamma \quad P_2 \vdash \Delta, x : B; \Gamma}{\mathsf{case}\ x\ \{|\mathsf{inl} : P_1\ |\mathsf{inr} : P_2\}\ \vdash \Delta, x : A \& B; \Gamma}\ [\mathrm{T}\&]$$

$$\frac{Q_1 \vdash \Delta', x : \overline{A}; \Gamma}{x.\mathsf{inl}; Q_1\ \vdash \Delta', x : \overline{A} \oplus \overline{B}; \Gamma}\ [\mathrm{T}\oplus_l] \qquad \frac{Q_2 \vdash \Delta', x : \overline{B}; \Gamma}{x.\mathsf{inr}; Q_2\ \vdash \Delta', x : \overline{A} \oplus \overline{B}; \Gamma}\ [\mathrm{T}\oplus_r]$$

Process $\mathsf{case}\ x\ \{|\mathsf{inl} : P_1\ |\mathsf{inr} : P_2\}$ offers a menu of two options: left P_1 and right P_2, whereas $x.\mathsf{inl}; Q_1$ and $x.\mathsf{inr}; Q_2$ choose left and right, respectively. The two associated reduction rules are

$$\begin{array}{llll}\mathsf{case}\ x\ \{|\mathsf{inl} : P_1\ |\mathsf{inr} : P_2\}\ \ |x : A \& B|\ \ x.\mathsf{inl}; Q_1 & \to & P_1\ \ |x : A|\ \ Q_1 & [\&\oplus_l] \\ \mathsf{case}\ x\ \{|\mathsf{inl} : P_1\ |\mathsf{inr} : P_2\}\ \ |x : A \& B|\ \ x.\mathsf{inr}; Q_2 & \to & P_2\ \ |x : B|\ \ Q_2 & [\&\oplus_r]\end{array}$$

Example 2.1 (Maybe). We define $\mathsf{Maybe}\ A \triangleq \mathbf{1} \oplus A$. Processes $\mathsf{nothing}\ x \vdash x : \mathsf{Maybe}\ A$ and $\mathsf{just}\ a\ x \vdash a : \overline{A}, x : \mathsf{Maybe}\ A$ are encoded as $\mathsf{nothing}\ x \triangleq x.\mathsf{inl}; \mathsf{close}\ x$ and $\mathsf{just}\ a\ x \triangleq x.\mathsf{inr}; \mathsf{fwd}\ a\ x$.

Example 2.2 (Linear Booleans). We encode the linear booleans as sessions of type $\mathsf{Bool} \triangleq \mathbf{1} \oplus \mathbf{1}$

$$\mathsf{lfalse}\ b \vdash b : \mathsf{Bool} \qquad \mathsf{ltrue}\ b \vdash b : \mathsf{Bool} \qquad \mathsf{lfalse}\ b \triangleq b.\mathsf{inl}; \mathsf{close}\ b \qquad \mathsf{ltrue}\ b \triangleq b.\mathsf{inr}; \mathsf{close}\ b$$

Consider processes $\mathsf{discard}\ x \vdash x : \overline{\mathsf{Bool}}$, $\mathsf{not}\ x\ y \vdash x : \overline{\mathsf{Bool}}, y : \mathsf{Bool}$ and $\mathsf{and}\ x\ y\ z \vdash x : \overline{\mathsf{Bool}}, y : \overline{\mathsf{Bool}}, z : \mathsf{Bool}$ which operate on linear booleans and are defined by

$$\begin{aligned}\mathsf{discard}\ x &\triangleq \mathsf{case}\ x\ \{|\mathsf{inl} : \mathsf{wait}\ x; \mathbf{0}\ |\mathsf{inr} : \mathsf{wait}\ x; \mathbf{0}\} \\ \mathsf{not}\ x\ y &\triangleq \mathsf{case}\ x\ \{|\mathsf{inl} : \mathsf{wait}\ x; \mathsf{true}\ y\ |\mathsf{inr} : \mathsf{wait}\ x; \mathsf{false}\ \ y\} \\ \mathsf{and}\ x\ y\ z &\triangleq \mathsf{case}\ x\ \{|\mathsf{inl} : \mathsf{wait}\ x; (\mathsf{discard}\ y\ \|\ \mathsf{false}\ z)\ |\mathsf{inr} : \mathsf{wait}\ x; \mathsf{fwd}\ y\ z\}\end{aligned}$$

Process discard x consumes a linear boolean on x, not $x\ y$ consumes a linear boolean on x and produces its negated value on y and and $x\ y\ z$ consumes booleans on channels x and y and produces their logical conjunction on channel z. Process and $x\ y\ z$ starts by performing case analysis on x. If x chooses left (false), then it waits for the session to be closed and then, in parallel, discards y (the output is false regardless of the input on y) and produces the boolean false on z. On the other hand, if x chooses right (true), then it waits for the session to be closed and then links the boolean y to z.

Servers. The dual types $!A$ (bang) and $?\overline{A}$ (why not) type server definition and client invocation

$$\frac{P \vdash y : A; \Gamma}{!x(y); P \ \vdash x : !A; \Gamma}\ [\text{T!}] \qquad \frac{Q \vdash \Delta; \Gamma, x : \overline{A}}{?x; Q \ \vdash \Delta, x : ?\overline{A}; \Gamma}\ [\text{T?}]$$

$$\frac{P \vdash y : A; \Gamma \quad Q \vdash \Delta; \Gamma, x : \overline{A}}{y.P\ |!x : A|\ Q \ \vdash \Delta; \Gamma}\ [\text{Tcut!}] \qquad \frac{Q \vdash \Delta, z : \overline{A}; \Gamma, x : \overline{A}}{\text{call}\ x(z); Q \ \vdash \Delta; \Gamma, x : \overline{A}}\ [\text{Tcall}]$$

The term $!x(y); P$ denotes a process that persistently offers on x a service of type A provided by P (the server body): it corresponds to the well-known replicated server process in the π-calculus. Process $?x; Q$ moves a session $x : ?\overline{A}$ from the linear to the unrestricted context (to be typed as $x : \overline{A}$) and proceeds as Q. Whereas [Tcut] acts on the linear context, [Tcut!] acts on the unrestricted context. The term $y.P\ |!x : A|\ Q$ composes a server body P with a pool of clients represented by Q, and which can request a service an unbounded (possibly zero) number of times. Process call $x(z); Q$ calls a service on the server x, to be processed on the fresh name z, and continues as Q. The interaction between [T!] and [T?] is expressed by rule [!?]

$$!x(y); P\ \ |x : !A|\ \ ?x; Q \quad \rightarrow \quad y.P\ |!x : A|\ Q \qquad [!?]$$

where a linear cut on $x : !A$ reduces to an unrestricted cut on $x : A$. This cut-reduction is standard in dyadic presentations of linear logic [Pfenning 1995] and it may be computationally understood as activation of the linear server code in a thread pool, for shared invocation by clients. Then, server invocation by clients is modelled by rule [call]

$$y.P\ |!x : A|\ \text{call}\ x(z); Q \quad \rightarrow \quad \{z/y\}P\ \ |z : A|\ \ (y.P\ |!x : A|\ Q) \qquad [\text{call}]$$

A linear replica of the server body is instantiated on the fresh name z as $\{z/y\}P$. Notice that the server body is still available to the continuation Q for (possibly) further requests.

The identity axiom over the unrestricted context is interpreted by the copycat process

$$\text{fwd}_!\ w\ y \triangleq !w(z); \text{call}\ y(z'); \text{fwd}\ z\ z' \vdash w : !A; \Gamma, y : \overline{A}$$

Free output on unrestricted names y is defined by send $x\ y; P \ \triangleq$ send $x(w.\text{fwd}_!\ w\ y); P$. Notice that given $P \vdash \Delta, x : B; \Gamma, y : \overline{A}$ we have send $x\ y; P \vdash \Delta, x : !A \otimes B; \Gamma, y : \overline{A}$.

Example 2.3. Consider the boolean-negation server $\text{not}_!\ s \vdash s : !(\text{Bool} \multimap \text{Bool} \otimes \mathbf{1})$, defined by

$$\text{not}_!\ s \triangleq !s(c); \text{recv}\ c(b); \text{send}\ c(b'.\ \text{not}\ b\ b'); \text{close}\ c$$

that persistently, on each call on s, spawns a process that receives a boolean b on channel c, sends back its negated value b' and closes c. The following process

$$?s; (C_1\ s\ ||\ C_2\ s) \vdash s : ?(\text{Bool} \otimes \text{Bool} \multimap \bot),\ \text{ where}$$
$$C_1 \triangleq \text{call}\ s(c); \text{send}\ c(b.\ \text{true}\ b); \text{recv}\ c(b'); \text{wait}\ c; C_1'$$
$$C_1 \triangleq \text{call}\ s(c); \text{send}\ c(b.\ \text{false}\ b); \text{recv}\ c(b'); \text{wait}\ c; C_1'$$

activates the server on s, which is then shared between two clients that run in parallel. Client C_1 calls a service on s which is spawned on c. Then, on c, the client sends true, receives the computed boolean b', waits for the server to close the session and continues as C_1'. Client C_2 behaves as C_1 but sends the boolean false instead. Both continuations $C_1', C_2' \vdash b' : \overline{\mathsf{Bool}}; s : \mathsf{Bool} \otimes \mathsf{Bool} \multimap \bot$ consume the received boolean b' and still have access to the boolean server s for further calls.

Polymorphism. The dual pair $\exists X.A$ (exists) and $\forall X.\overline{A}$ (for all) of existential and universal quantifiers implement type abstraction

$$\frac{P \vdash \Delta, x : \{B/X\}A; \Gamma}{\mathsf{send}\ x\ B; P\ \vdash \Delta, x : \exists X.A; \Gamma}\ [\mathrm{T}\exists] \qquad \frac{Q \vdash \Delta, x : \overline{A}; \Gamma}{\mathsf{recv}\ x(X); Q \vdash \Delta, x : \forall X.\overline{A}; \Gamma}\ [\mathrm{T}\forall]$$

In rule [T∀], X does not occur free in Δ, Γ. Process $\mathsf{send}\ x\ B; P$ sends the representation type B along x, and then continues as P. Dually, process $\mathsf{recv}\ x(X); Q$ receives on x a representation and continues as Q. The associated reduction is

$$\mathsf{send}\ x\ B; P\ \ |x : \exists X.\ A|\ \ \mathsf{recv}\ x(X); Q \quad \rightarrow \quad P\ \ |x : \{B/X\}A|\ \ \{B/X\}Q \qquad [\exists\forall]$$

The presence of existential quantifiers allow us to hide the representation datatype (cf. [Mitchell and Plotkin 1988]) of our concurrent shareable stateful ADTs, as we will show when introducing the Stack and Queue examples (Section 5). On the other hand, with universal quantifiers we can express inductive datatypes such as naturals and lists (see [Toninho and Yoshida 2021; Wadler 1990b]), as we illustrate in the following example, by encoding the Church numerals.

Example 2.4 (Inductive types). We illustrate the usage of type quantifiers to encode inductive types, by implementing the naturals with polymorphic sessions (cf. [Girard et al. 1989])

$$\begin{aligned}
\mathsf{Nat} &\triangleq \forall X.X \multimap !(X \multimap X) \multimap X \\
\mathsf{zero}\ n &\triangleq \mathsf{recv}\ n(X); \mathsf{recv}\ n(z); \mathsf{recv}\ n(s); ?s; \mathsf{fwd}\ z\ n \\
\mathsf{succ}\ n\ m &\triangleq \mathsf{recv}\ m(X); \mathsf{recv}\ m(z); \mathsf{recv}\ m(s); ?s; \\
&\qquad \mathsf{send}\ n\ X; \mathsf{send}\ n\ z; \mathsf{send}\ n\ s; \mathsf{call}\ s(c); \mathsf{send}\ c\ n; \mathsf{fwd}\ c\ m
\end{aligned}$$

Terms of type Nat receive a type variable X, a value $z : X$ and a server $s :!(X \multimap X)$ and call the server s a finite (possibly zero) number of times on z to return a value $n : X$. Notice that $\mathsf{zero}\ n \vdash n : \mathsf{Nat}$ simply forwards z on n without calling the server s. On the other hand, $\mathsf{succ}\ n\ m \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}$ calls the server one more time on the output produced by the calls of n. The encoding of the naturals allows for the definition of recursive operations. For example, the predicate $\mathsf{zero?}\ n\ b \vdash n : \overline{\mathsf{Nat}}, b : \mathsf{Bool}$ that consumes a natural n and produces the boolean true if n is zero and false otherwise is defined by

$$\mathsf{zero?}\ n\ b \triangleq \mathsf{send}\ n\ \mathsf{Bool}; \mathsf{send}\ n(z.\ \mathsf{true}\ z); \mathsf{send}\ n(s.\ \mathsf{kfalse}\ s); \mathsf{fwd}\ n\ b$$

where $\mathsf{kfalse}\ s \vdash s :!(\mathsf{Bool} \multimap \mathsf{Bool})$ is a server that outputs false regardless of the input.

3 πSS: A LANGUAGE WITH SHARED STATE

In this section we describe πSS, the main novelty of this paper, which is a PaT conservative extension of πCLL with shared state. We present several examples that illustrate the main features of πSS.

Reference Cells. The modalities $\boxed{?}$ and $\boxed{!}$ introduce reference cells and discipline their usage

$$\frac{P \vdash y :!A; \Gamma}{\mathsf{cell}\ x(y.P) \vdash x : \boxed{?}A; \Gamma}\ [\mathrm{Tcell}] \qquad \frac{Q \vdash \Delta; \Gamma}{\mathsf{free}\ x; Q \vdash \Delta, x : \boxed{!}\overline{A}; \Gamma}\ [\mathrm{Tfree}]$$

$$\frac{Q \vdash \Delta, z :?\overline{A}, x : \boxed{!}\overline{A}; \Gamma}{\mathsf{read}\ x(z); Q \vdash \Delta, x : \boxed{!}\overline{A}; \Gamma}\ [\mathrm{Tread}] \qquad \frac{Q_1 \vdash z :!A; \Gamma Q_2 \vdash \Delta, x : \boxed{!}\overline{A}; \Gamma}{\mathsf{wrt}\ x(z.Q_1); Q_2\ \vdash \Delta, x : \boxed{!}\overline{A}; \Gamma}\ [\mathrm{Twrite}]$$

The cell $x(y.P)$ construct denotes a reference cell at x storing a process P. P defines some replicated session behaviour at y (y binding in P). A thread releases its usage of a cell with free $x; Q$, typed by [Tfree]. When a cell is not being shared with any other thread, free will cause the cell to be "deallocated", as modelled by the reduction rule [$\boxed{?}\boxed{!}$f]

$$\text{cell } x(y.P) \ \ |x : \boxed{?}A| \ \ \text{free } x; Q \quad \rightarrow \quad Q \qquad [\boxed{?}\boxed{!}\text{f}]$$

Processes read and write reference cells with constructs read $x(y); Q$ and wrt $x(y.Q_1); Q_2$. Reading is modelled by the reduction rule [$\boxed{?}\boxed{!}$r]

$$\text{cell } x(y.P) \ \ |x : \boxed{?}A| \ \ \text{read } x(z); Q \quad \rightarrow \quad \{z/y\}P \ \ |z :!A| \ \ (\text{cell } x(y.P) \ \ |x : \boxed{?}A| \ \ Q) \qquad [\boxed{?}\boxed{!}\text{r}]$$

The contents P are atomically read, copied (it is typed $y :!A$) and received by the continuation Q at parameter z, by composition with cut. The system reduces to a process with two cuts. The inner cut connects the reference cell to the continuation Q, whereas the outer cut connects Q to a spawned copy $\{z/y\}P$ of the cell contents. The write operation is expressed by rule [$\boxed{?}\boxed{!}$w]

$$\text{cell } x(y.P) \ \ |x : \boxed{?}A| \ \ \text{wrt } x(z.Q_1); Q_2 \quad \rightarrow \quad \text{cell } x(z.Q_1) \ \ |x : \boxed{?}A| \ \ Q_2 \qquad [\boxed{?}\boxed{!}\text{w}]$$

Here, the contents of cell x are atomically and destructively updated by the write operation. The cell changes its contents from $y.P$ to $z.Q_1$, but its protocol $\boxed{?}A$ is left unchanged.

A cell is a persistent object in the sense that it may be sequentially read and written an arbitrary number of times, until freed by all clients. The basic cell protocol $\boxed{?}\ A$ may be expressed by the recursive equation $\boxed{?}\ A = \&\{\text{free} : \mathbf{1},\ \text{read} :!A \otimes \boxed{?}\ A,\ \text{write} :!A \multimap \boxed{?}\ A\}$. A cell usage name is a linear object (it lies in the linear context) and therefore cannot be implicitly copied and discarded (as opposed to unrestricted names). However, cell usages are explicitly released with the free operation and cell aliases can be created via an explicit share operation, which shall be introduced shortly.

A (replicated) value stored in a cell cannot depend on free linear names (see the premise of [Tcell]), in particular, it cannot depend on free cell references, so our language cannot express circular data structures. Particularly, we cannot express the Landin's knot, which is sensible, since otherwise this would break normalisation. However, a (replicated) value stored in a cell may of course allocate and use reference cells locally.

Our rules for reference cells take inspiration in the proof rules for the DiLL [Ehrhard 2018] modalities $?A$ and $!A$ (which are operations with a very different semantics from the usual linear logic exponentials). In general, our $\boxed{?}A$ and $\boxed{!}A$ modalities follow principles of DiLL $?A$ and $!A$ modalities respectively. In this sense, [Tcell] corresponds to $\boxed{?}$-dereliction and [Tfree] corresponds to $\boxed{!}$-co-weakening. Typing rules [Tread] and [Twrite] are new, they incorporate $\boxed{!}$-co-dereliction, but also follow the general structure of the typing rules for receive ([T$\parr$]) and send ([T$\otimes$]).

Example 3.1 (First-Class Cells). Our references cells are first-class (as in e.g., ML). In particular, cell references may be passed along session channels as in the following code

$$(\text{cell } c(v.V) \ \ |c : \boxed{?}A| \ \ \text{send } x\ c; S) \ \ |x : \boxed{?}A \otimes B| \ \ \text{recv } x(c'); \text{free } c'; R \quad \xrightarrow{+} \quad S \ \ |x : B| \ \ R$$

In this example, an usage to cell reference c is sent along channel x to the receiver, that frees it upon reception on the input parameter c'. Typing information is shown in detail below

$$\begin{array}{ll}
(1)\ V \vdash v :!A;\Gamma \quad (2)\ S \vdash \Delta_1, x : B;\Gamma \quad (3)\ R \vdash \Delta_2, x : \overline{B};\Gamma & \\
(4)\ \mathsf{cell}\ c(v.V)\ \vdash c : \boxed{?}A;\Gamma & ([\text{Tcell}],\ (1)) \\
(5)\ \mathsf{send}\ x\ c;S\ \vdash \Delta_1, c : \boxed{!}\overline{A}, x : \boxed{?}A \otimes B;\Gamma & ([\text{T}\otimes_f],\ (2)) \\
(6)\ \mathsf{free}\ c';R\ \vdash \Delta_2, c'\boxed{!}\overline{A}, x : \overline{B};\Gamma & ([\text{Tfree}],\ (3)) \\
(7)\ \mathsf{recv}\ x(c');\mathsf{free}\ c';R \vdash \Delta_2, x : \boxed{?}A \multimap \overline{B};\Gamma & ([\text{T}⅋],\ (6)) \\
(8)\ \mathsf{cell}\ c(v.V)\ \ |c|\ \ \mathsf{send}\ x\ c;S \vdash \Delta_1, z : \boxed{?}A \otimes B;\Gamma & ([\text{Tcut}],\ (4),\ (5)) \\
(9)\ (\mathsf{cell}\ c(v.V)\ \ |c|\ \ \mathsf{send}\ x\ c;S)\ \ |x|\ \ \mathsf{recv}\ x(c');\mathsf{free}\ c';R \vdash \Delta_1, \Delta_2;\Gamma & ([\text{Tcut}],\ (7),\ (8))
\end{array}$$

Notice that the sending process loses access to the reference cell c as shown by typing judgement (2): the name c is not available in its continuation S.

Example 3.2 (Higher-Order Store). We illustrate the use of higher-order store by giving an (inefficient) array implementation [Pierce 2002] in πSS. Let Array $\triangleq \boxed{?}(!\mathsf{Nat} \multimap \mathsf{Nat})$. An array is a reference cell to a *function* from naturals to naturals. Let

$$\mathsf{init}\ \ a \triangleq \mathsf{cell}\ a(s.!s(f);\mathsf{recv}\ f(n);?n;\mathsf{zero}\ \ f)$$

$$\mathsf{lookup}\ \ a\ n\ m\ a' \triangleq \mathsf{read}\ a(s);?s;\mathsf{call}\ s(f);\mathsf{send}\ f\ n;(\mathsf{fwd}\ f\ m\ ||\ \mathsf{fwd}\ a\ a')$$

Process init $a \vdash a$: Array allocates a reference cell a with a function f that outputs the natural zero on each inputed index n. Process lookup $a\ n\ m\ a' \vdash a : \overline{\mathsf{Array}}, n : \overline{!\mathsf{Nat}}, m : \mathsf{Nat}, a : \mathsf{Array}$ computes the n-th entry of the array a on m and forwards the unmodified array on a'. Code for the operation update $\ a\ n\ m\ a' \vdash a : \overline{\mathsf{Array}}, n : \overline{!\mathsf{Nat}}, m : \overline{!\mathsf{Nat}}, a : \mathsf{Array}$, that updates the n-th entry of the array a with m, producing the updated array on a', is given in the technical report [Rocha and Caires 2021a].

Sharing and Nondeterminism. Share and sum are typed by the rules

$$\frac{P \vdash \Delta', x : \boxed{!}\overline{A};\Gamma \quad Q \vdash \Delta, x : \boxed{!}\overline{A};\Gamma}{\mathsf{share}\ x\ \{P\ ||\ Q\}\ \vdash \Delta', \Delta, x : \boxed{!}\overline{A};\Gamma}[\text{Tshare}] \qquad \frac{P \vdash \Delta;\Gamma \quad Q \vdash \Delta;\Gamma}{P + Q \vdash \Delta;\Gamma}[\text{Tsum}]$$

The share x $\{P\ ||\ Q\}$ construct allows a reference cell at x to be aliased and manipulated by multiple threads, as the following example involving concurrent writers illustrates

$$\mathsf{cell}\ x(y.V)\ \ |x|\ \ \mathsf{share}\ x\ \{\mathsf{wrt}\ x(y.V_1);P\ ||\ \mathsf{wrt}\ x(y.V_2);Q\}$$

Here the cell at x is being accessed by concurrent threads wrt $x(y.V_1);P$ and wrt $x(y.V_2);Q$. Intuitively, the final state of the cell depends on the order of writes. In our PaT interpretation the process above reduces (via cut-reduction) to a sum that shows the alternative states of the cell after both writes (last writer wins), and corresponds to non-deterministic sum of process algebras.

$$(\mathsf{cell}\ x(y.V_1) + \mathsf{cell}\ x(y.V_2))\ \ |x|\ \ \mathsf{share}\ x\ \{P\ ||\ Q\}$$

The typing rule for share precisely corresponds to $\boxed{!}$-co-contraction (cf. DiLL). Co-contraction enforces that processes P and Q may linearly interact at the shared reference x, but not on other linear objects - the linear context is handled multiplicatively. This condition is important for deadlock freedom, and related with the relevance of acyclicity for cut-elimination in linear logic proofs (e.g., as in the session type systems [Caires and Pfenning 2010; Wadler 2012], in which interaction is expressed by the cut, different threads cannot share more than one communication channel). However, notice that (1) a single shared cell may store all the state shared by the two threads (e.g. in a resource bundle, a sequence of values) and (2) a reference cell can be used by any number of client threads, by iterated use of the share construct.

Example 3.3 (Dynamic Sharing). Notwithstanding the seemingly static character of the share construct (which appears as a special form of parallel composition where two threads are allowed to share a single mutable cell), notice that the topology of the sharing trees arising from nested share blocks may actually change dynamically during computation, because of the extrusion of cell references along session channels to outside the share block. Consider the following reduction

$$\begin{array}{l} \mathsf{share}\ x\ \{R \mid\mid \mathsf{send}\ z\ x; P\}\ \ |z|\ \ \mathsf{recv}\ z(y); \mathsf{share}\ y\ \{Q \mid\mid S\} \\ \xrightarrow{+} P\ \ |z|\ \ \mathsf{share}\ x\ \{R \mid\mid \mathsf{share}\ x\ \{\{x/y\}Q \mid\mid \{x/y\}S\}\} \end{array}$$

Here, a shared alias of x is sent along z to a partner receive process, that inputs the alias and further shares it between threads Q and S. Hence, access to x, initially only shared between the two threads R and $\mathsf{send}\ z\ x; P$, ends up being shared among the three threads R, $\{x/y\}Q$ and $\{x/y\}S$. The thread $\mathsf{send}\ z\ x; P$ transfers ownership of x on output – references aliases are linear values, whose visibility may only be duplicated by the share construct. Indeed, linear typing of session send ensures that P must lose access to x. We show typings for components of the above reduction

(1) $R \vdash \Delta_1, x : \boxed{!}\overline{A}; \Gamma$ (2) $P \vdash \Delta_2, z : B; \Gamma$ (3) $Q \vdash \Delta_3, y : \boxed{!}\overline{A}, z : \overline{B}; \Gamma$ (4) $S \vdash \Delta_4, y : \boxed{!}\overline{A}; \Gamma$
(5) $\mathsf{send}\ z\ x; P \vdash \Delta_2, x : \boxed{!}\overline{A}, z : \boxed{?}A \otimes B; \Gamma$ ([T$\otimes_f$], (2))
(6) $\mathsf{share}\ x\ \{R \mid\mid \mathsf{send}\ z\ x; P\} \vdash \Delta_1, \Delta_2, x : \boxed{!}\overline{A}, z : \boxed{?}A \otimes B; \Gamma$ ([Tshare], (1), (5))
(7) $\mathsf{share}\ y\ \{Q \mid\mid S\} \vdash \Delta_3, \Delta_4, y : \boxed{!}\overline{A}, z : \overline{B}; \Gamma$ ([Tshare], (3), (4))
(8) $\mathsf{recv}\ z(y); \mathsf{share}\ y\ \{Q \mid\mid S\} \vdash \Delta_3, \Delta_4, z : \boxed{?}A \multimap \overline{B}; \Gamma$ ([T$\bindnasrepma$], (7))
(9) $\mathsf{share}\ x\ \{R \mid\mid \mathsf{send}\ z\ x; P\}\ \ |z|\ \ \mathsf{recv}\ z(y); \mathsf{share}\ y\ \{Q \mid\mid S\} \vdash \Delta_1, \Delta_2, \Delta_3, \Delta_4, x : \boxed{!}\overline{A}; \Gamma$
([Tcut], (6), (8))

We revisit this example in Section 6, providing a step-by-step derivation for the reduction above.

4 πSSL: A LANGUAGE WITH SHARED STATE AND LOCKS

In this Section, we extend our basic stateful language πSS with typed locking primitives, without breaking the key property of deadlock freedom. Although one could suspect that locking behaviour may break the progress property, the linear logic type discipline resiliently preserves the deadlock freedom property, where typing ensures acyclicity in communication, herein in locking. The key insight is that reductions in the components of $\mathsf{share}\ x\ \{P \mid\mid Q\}$ are independent (apart from x, P and Q are typed in disjoint linear contexts, as in e.g., $P \mid\mid Q$), and concurrent operations on x are always converted to a sum via interleaving, before they interact with the cell at x. In particular, reductions are always available in any component of the resulting sum, and of course, progress in one component of a sum does not depend on progress of the other branch. The presence of locks will drop some summands that otherwise would break atomicity of the critical sections, but, intuitively, dropping a summand will never break progress.

To the collection of types we thus add a pair of dual modalities $\boxed{\cancel{?}}A$ (locked box why not) and $\boxed{\cancel{!}}\overline{A}$ (locked box bang) that represent a locked state and that act as "mirror" of the basic modalities $\boxed{?}A$, $\boxed{!}\overline{A}$, which in turn represent the unlocked state. A key property enforced by the locked usage modality $\boxed{\cancel{!}}\overline{A}$ is that only a single thread usage may be actively interacting with the (locked) memory cell. Lock and unlock operations essentially alternate between the two states

$$\frac{Q \vdash \Delta, x : \boxed{\cancel{!}}\overline{A}; \Gamma}{\mathsf{lock}\ x; Q \vdash \Delta, x : \boxed{!}\overline{A}; \Gamma}\ [\text{Tlock}] \qquad \frac{Q \vdash \Delta, x : \boxed{!}\overline{A}; \Gamma}{\mathsf{unlock}\ x; Q \vdash \Delta, x : \boxed{\cancel{!}}\overline{A}; \Gamma}\ [\text{Tunlock}]$$

The associated reductions are

$$\begin{array}{lcll} \mathsf{cell}\ x(y.P)\ \ |x : \boxed{?}A|\ \ \mathsf{lock}\ x; Q & \to & \mathsf{cell}\ x(y.P)\ \ |x : \boxed{\cancel{?}}A|\ \ Q & [\boxed{?}\boxed{!}\text{l}] \\ \mathsf{cell}\ x(y.P)\ \ |x : \boxed{\cancel{?}}A|\ \ \mathsf{unlock}\ x; Q & \to & \mathsf{cell}\ x(y.P)\ \ |x : \boxed{?}A|\ \ Q & [\boxed{\cancel{?}}\boxed{\cancel{!}}\text{u}] \end{array}$$

The previously introduced typing rules [Tcell], [Tread] and [Twrite] have now a mirrored version, corresponding to (unique) locked usage.

$$\frac{P \vdash y :!A; \Gamma}{\text{cell } x(y.P) \vdash x : \cancel{\boxed{?}}A; \Gamma} \quad \frac{Q \vdash \Delta, y :?\overline{A}, x : \cancel{\boxed{!}}\overline{A}; \Gamma}{\text{read } x(y); Q \vdash \Delta, x : \cancel{\boxed{!}}\overline{A}; \Gamma} \quad \frac{P \vdash y :!A; \Gamma \qquad Q \vdash \Delta, x : \cancel{\boxed{!}}\overline{A}; \Gamma}{\text{wrt } x(y.P); Q \ \vdash \Delta, x : \cancel{\boxed{!}}\overline{A}; \Gamma}$$

Similarly, the principal cut conversions [$\boxed{?}\,\boxed{!}$r] and [$\boxed{?}\,\boxed{!}$w] have a mirrored version [$\cancel{\boxed{?}}\,\cancel{\boxed{!}}$r] and [$\cancel{\boxed{?}}\,\cancel{\boxed{!}}$w], respectively. Rule [$\cancel{\boxed{?}}\,\cancel{\boxed{!}}$r] (resp., [$\cancel{\boxed{?}}\,\cancel{\boxed{!}}$w]) is written as [$\boxed{?}\,\boxed{!}$r] (resp., [$\boxed{?}\,\boxed{!}$w]) but with the type annotation $\boxed{?}A$ replaced by $\cancel{\boxed{?}}A$. Therefore, the behavioural 1-state protocol of cells is extended to a 2-state protocol, which we may write as

$$\boxed{?}A = \&\{\text{free} : \mathbf{1},\ \text{read} :!A \otimes \boxed{?}A,\ \text{write} :!A \multimap \boxed{?}A,\ \text{lock} : \cancel{\boxed{?}}A\}$$
$$\cancel{\boxed{?}}A = \&\{\text{read} :!A \otimes \cancel{\boxed{?}}A,\ \text{write} :!A \multimap \cancel{\boxed{?}}A,\ \text{unlock} : \boxed{?}A\}$$

Besides [Tshare], two additional co-contraction rules are now added for taming locked usage:

$$\frac{P \vdash \Delta', x : \cancel{\boxed{!}}\overline{A}; \Gamma \qquad Q \vdash \Delta, x : \boxed{!}\overline{A}; \Gamma}{\text{shareL } x\ \{P \mid\mid Q\} \vdash \Delta', \Delta, x : \cancel{\boxed{!}}\overline{A}; \Gamma}\ [\text{TshareL}] \quad \frac{P \vdash \Delta', x : \boxed{!}\overline{A}; \Gamma \qquad Q \vdash \Delta, x : \cancel{\boxed{!}}\overline{A}; \Gamma}{\text{shareR } x\ \{P \mid\mid Q\} \vdash \Delta', \Delta, x : \cancel{\boxed{!}}\overline{A}; \Gamma}\ [\text{TshareR}]$$

These constructs are slight variations of [Tshare] keeping track of the component currently holding the lock, left (shareL x $\{P \mid\mid Q\}$), and right (shareR x $\{P \mid\mid Q\}$), and particular congruence rules apply to them (see Fig. 5). For example, consider system

$$\text{cell } c(b.\text{true } b)\ \ |c : \boxed{?}\text{Bool}|\ \ \text{share } c\ \{\text{lock } c; \text{wrt } c(b'.\text{false } b'); \text{unlock } c; P \mid\mid \text{lock } c; Q\}$$

Here, two concurrent lock actions are competing for exclusive access of the boolean cell. By ≡ law [LL] this process is rewritten into an ≡-equivalent nondeterministic sum $S_1 + S_2$, where the two lock instructions get interleaved. Each summand S_i is then

$$\begin{array}{lcl} S_1 & \triangleq & \text{cell } c(b.\text{true } b)\ \ |c : \boxed{?}\text{Bool}|\ \ \text{lock } c; \text{shareL } c\ \{\text{wrt } c(b'.\text{false } b'); \text{unlock } c; P \mid\mid \text{lock } c; Q\} \\ S_2 & \triangleq & \text{cell } c(b.\text{true } b)\ \ |c : \boxed{?}\text{Bool}|\ \ \text{lock } c; \text{shareR } c\ \{\text{lock } c; \text{wrt } c(b'.\text{false } b'); \text{unlock } c; P \mid\mid Q\} \end{array}$$

Notice that reductions in summands S_1, S_2 of process $S_1 + S_2$ are now completely independent. We will then continue by illustrating reduction for S_1, the case of S_2 being similar. We have

$$\begin{array}{ll} S_1 \rightarrow & \text{cell } c(b.\text{true } b)\ \ |c : \cancel{\boxed{?}}\text{Bool}|\ \ \text{shareL } c\ \{\text{wrt } c(b'.\text{false } b'); \text{unlock } c; P \mid\mid \text{lock } c; Q\}\ (\rightarrow [\boxed{?}\,\boxed{!}\text{l}]) \\ \quad\ \equiv & \text{cell } c(b.\text{true } b)\ \ |c : \cancel{\boxed{?}}\text{Bool}|\ \ \text{wrt } c(b'.\text{false } b'); \text{shareL } c\ \{\text{unlock } c; P \mid\mid \text{lock } c; Q\}\ (\equiv [\text{leftA}]) \\ \quad\ \rightarrow & \text{cell } c(b'.\text{false } b')\ \ |c : \cancel{\boxed{?}}\text{Bool}|\ \ \text{shareL } c\ \{\text{unlock } c; P \mid\mid \text{lock } c; Q\}\ (\rightarrow [\cancel{\boxed{?}}\,\cancel{\boxed{!}}\text{w}]) \\ \quad\ \equiv & \text{cell } c(b'.\text{false } b')\ \ |c : \cancel{\boxed{?}}\text{Bool}|\ \ \text{unlock } c; \text{share } c\ \{P \mid\mid \text{lock } c; Q\}\ (\equiv [\text{leftU}]) \\ \quad\ \rightarrow & \text{cell } c(b'.\text{false } b')\ \ |c : \boxed{?}\text{Bool}|\ \ \text{share } c\ \{P \mid\mid \text{lock } c; Q\}\ (\rightarrow [\cancel{\boxed{?}}\,\cancel{\boxed{!}}\text{u}]) \end{array}$$

First, the process locks and gives exclusive cell access to the left component of shareL. The congruence rules only allows cell operations on the left component to commute to the front of the shareL, as expressed by ≡ law [leftA]. Therefore, the write operation takes precedence over the pending lock operation of the right component. When the left component unlocks, the shareL is converted back to a share via ≡ rule [leftU]. Afterwards, process P loses exclusive access to the cell and now all operations on c coming from P (except for free) compete with the lock operation of the right component (≡ rules [LA] and [LL]).

Typing rules [Tshare], [TshareL] and [TshareR] ensure that at most one process is holding the lock in a share topology. Interestingly, lock ownership may be freely communicated.

Example 4.1 (Counter). Let Nat be the type of the natural numbers of Example 2.4 and consider persistent versions of the previously defined processes zero n and succ $n\ m$

$$\mathsf{zero}_!\ n \triangleq !n(n0); \mathsf{zero}\ n0 \vdash n :!\mathsf{Nat}$$

$$\mathsf{succ}_!\ n\ m \triangleq !m(m0); \mathsf{call}\ n(n0); \mathsf{succ}\ n0\ m0 \vdash m :!\mathsf{Nat}; n : \overline{\mathsf{Nat}}$$

Define COUNTER = $\boxed{?}\mathsf{Nat}$ and the following processes

$$\mathsf{c\_zero}\ c \triangleq \mathsf{cell}\ c(n.\ \mathsf{zero}_!\ n) \vdash c : \mathsf{COUNTER}$$

$$\mathsf{client}_1\ c \triangleq \mathsf{read}\ c(n); ?n; \mathsf{wrt}\ c(m.\ \mathsf{succ}_!\ n\ m); \mathsf{free}\ c; \mathbf{0} \vdash c : \overline{\mathsf{COUNTER}}$$

$$\mathsf{client}_2\ c \triangleq \mathsf{lock}\ c; \mathsf{read}\ c(n); ?n; \mathsf{wrt}\ c(m.(\mathsf{succ}_!\ n\ m)); \mathsf{unlock}\ c; \mathsf{free}\ c; \mathbf{0} \vdash c : \overline{\mathsf{COUNTER}}$$

$$\mathsf{system_i} \triangleq \mathsf{c\_zero}\ c\ \ |c|\ \ \mathsf{share}\ c\ \{\mathsf{client}_i\ c \ ||\ \mathsf{client}_i\ c\},\ i \in \{1, 2\}$$

Process c_zero is a counter set to zero and client$_1$ c and client$_2$ c are two clients. client$_1$ c reads the counter, writes back its incremented value, frees its usage and continues as the inaction process. client$_2$ c is a variant of client$_1$ c in which the read-and-increment sequence is within a critical section. system$_i$ is the result of sharing the counter with two concurrent threads, each being client$_\mathsf{i}$.

After the read-and-increment sequences being executed, system$_1$ leaves the cell in a nondeterministic superposition of two states $\mathsf{cell}\ c(n.\mathsf{one}_!\ n) + \mathsf{cell}\ c(n.\mathsf{two}_!\ n)$, where $\mathsf{one}_!\ n$ and $\mathsf{two}_!\ n$ offer persistently the naturals one and two on channel n. On the other hand, system$_2$ evolves to a single state $\mathsf{cell}\ c(n.\mathsf{two}_!\ n)$. This is because the interleavings that break the intended atomicity of read-and-increment are dropped in the presence of locks.

5 MUTABLE SHAREABLE ADTS

We illustrate how shared mutable state fits together with πCLL by presenting two ADTs: a shared stack and a queue object. The representation type of both ADTs is hidden from the client through existential types. All the client gets is a menu of operations to act on the ADT, among which, as we shall see for the stack, lies the capability to share the representation with another client.

5.1 A Shared Stack

Code for the Stack, together with type definitions, is provided in Figure 2. An object of type Stack A sends some representation type X, then outputs a persistent server of type $X \multimap \mathsf{Menu}(X)$ and continues as X. The type expression $\mathsf{Menu}(X)$, which depends on the representation type X,offers a menu with four options: push, pop, share and free. In this example, we use a labelled version of offer and choice, which is a standard generalisation, (see e.g. [Caires and Pérez 2017]).

Process stack s is an object of type s : Stack A whose representation type is List $!A$ (lists of elements of type $!A$) and whose persistent server body is given my menu $c \vdash c : \boxed{?}\mathsf{List}\ A \multimap \mathsf{Menu}(\boxed{?}\mathsf{List}\ A)$. Note that the stack is initially empty ($\mathsf{cell}\ s(l.\mathsf{nil}_!\ l)$).

In the companion artifact [Rocha and Caires 2021b] we encode the list datatype List A, together with the following operations: $\mathsf{nil}_! l \vdash l :!\mathsf{List}\ !A$ offers persistently the empty list, $\mathsf{cons}_!\ a\ l\ l' \vdash l' : !\mathsf{List}\ !A; a : \overline{A}, l : \overline{\mathsf{List}\ !A}$ computes on l' the result of prepending a to the head of l, $\mathsf{head}_!\ l\ h \vdash h : \mathsf{Maybe}\ !A; l : \overline{\mathsf{List}\ !A}$ produces on x the head of the list l (which might be none if the list is empty) and $\mathsf{tail}_!\ l\ l' \vdash l' :!\mathsf{List}\ !A; \overline{\mathsf{List}\ !A}$ produces on l' the tail of list l.

Method mpush $c\ s \vdash c :!A \multimap \boxed{?}\mathsf{List}\ !A \otimes \mathbf{1}, s : \boxed{!}\overline{\mathsf{List}\ !A}$ locks the stack s and then gets the current state in l :!List $!A$, through a read operation. Then receives on channel c an element a :!A to be inserted on top of the stack. The persistent names a and l are then activated before being used. The stack is updated with a new list l' that is the result of prepending a to l. The updated stack is then unlocked and sent back to the client, after which the communication session c is closed.

```
$\mathsf{Stack}\ A \triangleq \exists X.!(X \multimap \mathsf{Menu}(X)) \otimes X$
$\mathsf{Menu}(X) \triangleq \&\{$
  $\mathsf{push} : !A \multimap X \otimes \mathbf{1},$
  $\mathsf{pop} : \mathsf{Maybe}\ !A \otimes X \otimes \mathbf{1},$
  $\mathsf{share} : X \parr X \parr \bot,$
  $\mathsf{free} : \mathbf{1}\}$

$\mathsf{stack}\ s \triangleq$
  send $s\ (\boxed{?}\mathsf{List}\ !A);$
  send $s(m.\ !m(c);\ \mathsf{menu}\ c);$
  cell $s(l.\ \mathsf{nil}_!\ l)$

$\mathsf{menu}\ c \triangleq$
  recv $c(s);$
  case $c\{$
     $|\mathsf{push}\ \ : \mathsf{mpush}\ c\ s$
     $|\mathsf{pop}\ \ \ \ : \mathsf{mpop}\ c\ s$
     $|\mathsf{share} : \mathsf{mshare}\ c\ s$
     $|\mathsf{free}\ \ \ : \mathsf{mfree}\ c\ s\}$
```

```
$\mathsf{mpush}\ c\ s \triangleq$
  lock $s;$
  read $s(l);$
  recv $c(a);$
  $?a; ?l;$
  wrt $s(l'.\ \mathsf{cons}_!\ a\ l\ l');$
  unlock $s;$
  send $c\ s;$
  close $c$

$\mathsf{mpop}\ c\ s \triangleq$
  lock $s;$
  read $s(l);$
  $?l;$
  send $c(h.\ \mathsf{head}_!\ l\ h);$
  wrt $s(l'.\ \mathsf{tail}_!\ l\ l');$
  unlock $s;$
  send $c\ s;$
  close $c$
```

```
$\mathsf{mshare}\ c\ s \triangleq$
  recv $c(s_1);$
  recv $c(s_2);$
  wait $c;$
  share $s\ \{$
       fwd $s\ s_1$ ||
       fwd $s\ s_2\}$

$\mathsf{mfree}\ c\ s \triangleq$
  free $s;$
  close $c$
```

Fig. 2. A Shared Stack - Server.

Method mpop c s accesses the contents of the stack s on l and sends its head element h : $\oplus\{\mathsf{Nothing} : \mathbf{1},\ \mathsf{Just} : !A\}$. The head of the list l might be Nothing if it is empty or Just an element $x : !A$. Then, the stack is updated with the tail of list l. Notice that, like in method mpush, all the stack's critical accesses are protected within lock-unlock sections.

Stack clients will not be able to tamper with the representation reference, being forced to manipulate it via the provided methods. Thus, the process operations that share and free a cell are exported with the ADT methods mshare c s and mfree c s. Process mfree c s frees its usage to the stack s and closes the session on c. Method mshare $c\ s \vdash c : X \parr X \parr \bot, s : \overline{X}$ (where $X = \boxed{?}\mathsf{List}\ !A$) receives two stack handlers s_1 and s_2, waits for the client to terminate the session and proceeds as an operation that shares the stack s between s_1 and s_2. We find it quite satisfying how the type interface of mshare expresses the co-contraction principle in an abstract and extremely clean way.

Code defining client operations is in Figure 3. Process unpack $s \vdash s : \overline{\mathsf{Stack}\ A}$ receives the opaque abstract type X and a communication channel m to invoke the stack menu, continuing as $\mathsf{client}_m\ s$. Process $\mathsf{client}_m\ s$ can be defined composing the available protocols: cpush_m, cpop_m, cshare_m and cfree_m. $\mathsf{cpush}_m\ s\ a; (P\ s')$ pushes the element a on stack s and continues as $P\ s'$ (we use s' to denote the returned updated stack after a method call). $\mathsf{cpop}_m\ s; [P\ s', Q\ s'\ a]$: pops an element from the stack s and continues as $P\ s'$ if the stack is empty and as $Q\ s'\ a$ if the stack's top element is a. $\mathsf{cshare}_m\ s\ \mathsf{in}[P_1\ s_1, P_2\ s_2]$ shares the stack s between handlers s_1 and s_2 which are implemented by processes P_1 and P_2 (respectively). $\mathsf{cfree}_m s; P$: frees the handler s to the stack and continue as P.

Client protocol $\mathsf{push}_m\ s\ a; (P\ s')$ starts by requesting a service on the stack's menu server, which will be handled on c. Then, sends the stack, selects the operation push and sends the element a to be inserted. Then, it receives the updated stack s', waits for the server to terminate the communication on c and continues as $P\ s'$. The other client protocols follow a similar pattern.

```
unpack s ≜
  recv s(X);
  recv s(m);
  ?m;
  client_m s

cshare_m s  in  [P1 s1, P2 s2] ≜
  call m(c);
  send c s;
  c.share;
  send c(s1. P1 s1);
  send c(s2. P2 s2);
  close c
```

```
cfree_m s; P ≜
  call m(c);
  send c s;
  c.free;
  wait c;
  P

cpush_m s a; (P s') ≜
  call m(c);
  send c s;
  c.push;
  send c a;
  recv c(s');
  wait c;
  P s'
```

```
cpop_m s; [P s', Q s' a] ≜
  call m(c);
  send c s;
  c.pop;
  recv c(h);
  case  h{
   Nothing :  recv c(s');
              wait c;
              P s',
   Just :     recv c(a);
              recv c(s');
              wait c;
              Q s' a}
```

Fig. 3. A Shared Stack - Client Operations.

We now consider two sample systems system^1 and system^2 which are the result of composing the stack with two alternative client codes client^1_m and client^2_m

$$\begin{aligned}
\text{system}^i &\triangleq \text{stack } s \ \ |s| \ \ \text{recv } s(X); \text{recv } s(m); ?m; \text{client}^i_m \ s, \ \ i \in \{1, 2\} \\
\text{client}^1_m &\triangleq \text{cpush}_m \ s \ a; \text{cpop}_m s; \ [P \ s', Q \ s' \ b] \\
\text{client}^2_m &\triangleq \text{cshare}_m \ s \text{ in } [\text{cpush}_m \ s \ a; (P \ s'), \ \text{cpop}_m \ s; [Q \ s', R \ s' \ b]]
\end{aligned}$$

client^1_m defines a thread that calls methods push and pop sequentially, whereas client^2_m call those methods concurrently. As expected, the computation of system^1 is deterministic: after the interactions the client continues as $Q\ s'\ b$ where $b = a$ and s' is an empty stack. On the other hand, system^2 evolves to a nondeterministic sum of two systems. In one of these systems, the pop operation occurred after the push, and so client^2_m continues as cshare$_m$ s' in $[P\ s', R\ s'\ b]$, where s' is an empty stack and $a = b$. On the hand, if push occurs after pop, then client^2_m evolves to cshare$_m$ s' in $[P\ s', Q\ s']$ where s' is a stack with one element a.

5.2 Queue

We now implement an efficient queue, which uses a pair of lists as its representation type (see [Okasaki 1998]). One of the lists (say l_e) is used for enqueueing and the other (say l_d) for dequeueing. Elements are inserted in the front of l_e and removed from the front of l_d. When l_d is empty it is updated with the reverse of l_e, and then l_e becomes empty. Let Queue ≜ Eview ⅋ Dview. A queue is an object with two views: one for enqueueing (Eview) and another for dequeueing (Dview). Similarly to the stack implementation, we use existential types to encode each view

$$\begin{aligned}
\text{Eview} \quad &\triangleq \exists X.!(X \multimap E(X)) \otimes X, \text{ where } E(X) \triangleq !A \multimap X \otimes \mathbf{1} \\
\text{Dview} \quad &\triangleq \exists X.!(X \multimap D(X)) \otimes X, \text{ where } D(X) \triangleq \text{Maybe } !A \otimes X \otimes \mathbf{1}
\end{aligned}$$

The signatures of $E(X)$ and $D(X)$ are same as for methods push and pop (respectively) from the previous stack example. Code for queue(d) ⊢ d : Queue, eview(e, c_e) ⊢ e : Eview, c_e : $\overline{⍰\text{List } !A}$ and

$$
\begin{array}{ll}
\mathcal{N} ::= & x, y, z, \ldots \ \textit{(names)} \\
P, Q ::= & \mathbf{0}\ \textit{(inaction)} \mid \mathsf{fwd}\ x\ y\ \textit{(forwarder)} \mid P \mid\mid Q\ \textit{(par)} \mid \\
& P\ \ |x : A|\ \ Q\ \textit{(cut)}\ \mid y.P\ |!x : A|\ Q\ \textit{(cut!)} \mid \mathsf{share}\ x\ \{P \mid\mid Q\}\ \textit{(share)} \mid \\
& P + Q\ \textit{(sum)}\ \mid \mathsf{shareL}\ x\ \{P \mid\mid Q\}\ \textit{(share-left)} \mid \mathsf{shareR}\ x\ \{P \mid\mid Q\}\ \textit{(share-right)} \\
(\mathcal{A}, \mathcal{B} ::=) & \mathsf{close}\ x\ \textit{(close)} \mid \mathsf{wait}\ x; P\ \textit{(co-close)} \mid \\
& x.\mathsf{inl}; P\ \textit{(choose left)}\ \mid x.\mathsf{inr}; P\ \textit{(choose right)} \mid \mathsf{case}\ x\ \{|\mathsf{inl} : P\ |\mathsf{inr} : Q\}\ \textit{(offer)} \mid \\
& \mathsf{send}\ x(y.P); Q\ \textit{(send)} \mid \mathsf{recv}\ x(y); P\ \textit{(receive)} \mid \\
& !x(y); P\ \textit{(server)} \mid ?x; P\ \textit{(activation)}\ \mid \mathsf{call}\ x(y); P\ \textit{(call)} \mid \\
& \mathsf{send}\ x\ A; P\ \textit{(type send)} \mid \mathsf{recv}\ x(X); P\ \textit{(type receive)} \mid \\
& \mathsf{cell}\ x(y.P)\ \textit{(cell)} \mid \mathsf{free}\ x; P\ \textit{(free)} \mid \mathsf{read}\ x(y); P\ \textit{(read)} \mid \\
& \ \mathsf{wrt}\ x(y.P); Q\ \textit{(write)} \mid \mathsf{lock}\ x; P\ \textit{(lock)} \mid \mathsf{unlock}\ x; P\ \textit{(unlock)}
\end{array}
$$

Fig. 4. Processes P.

$\mathsf{dview}(d, c_e) \vdash d : \mathsf{Dview}, c_e : \overline{\boxed{?}\mathsf{List}\ !A}$ is given below

$$
\begin{array}{ll}
\mathsf{queue}(d) & \triangleq \mathsf{recv}\ d(e); \mathsf{cell}\ c_e(l_e.\ \mathsf{nil}_!\ l_e)\ \ |c_e|\ \ \mathsf{share}\ c_e\ \{\mathsf{eview}(e, c_e) \mid\mid \mathsf{dview}(d, c_e)\} \\
\mathsf{eview}(e, c_e) & \triangleq \mathsf{send}\ e\ (\boxed{?}\mathsf{List}\ !A); \mathsf{send}\ e(c_!.\ !c_!(c); \mathsf{menq}\ c); \mathsf{fwd}\ e\ c_e \\
\mathsf{dview}(d, c_e) & \triangleq \mathsf{send}\ d\ (\boxed{?}\mathsf{List}\ !A \otimes \boxed{?}\mathsf{List}\ !A); \mathsf{send}\ d(c_!.\ !c_!(c); \mathsf{mdeq}\ c); \\
& \qquad\qquad\qquad\qquad \mathsf{send}\ d(c_d.\ \mathsf{cell}\ c_d(l_d.\ \mathsf{nil}_!\ l_d)); \mathsf{fwd}\ d\ c_e
\end{array}
$$

Queue receives on channel d name e and continues as a configuration in which the reference c_e, to the initially empty list l_e for enqueueing, is being shared between the two views. $\mathsf{eview}(e, c_e)$ sends on e its representation type (a reference to a list), then sends the persistent server for enqueueing and continues as process that forwards c_e. $\mathsf{dview}(d, c_e)$ sends its representation type (a tensor of of two cells), sends the server for dequeueing, sends c_d and continues as process that forwards c_e.

Notice that the conjunction that puts together the two views to form the Queue protocol is a par ($\mathbin{⅋}$) and not a tensor ($\otimes$) since the two views share a reference cell and therefore are not orthogonal. Nevertheless, when l_d is nonempty, the operations of enqueueing and dequeueing can be performed in parallel. The complete definition is provided in the technical report [Rocha and Caires 2021a].

6 SEMANTICS, TYPE PRESERVATION AND PROGRESS

We present the syntax and the operational semantics of πSSL, which is defined from structural congruence $\equiv$ and reduction $\longrightarrow$. We then prove that the semantics is type preserving and deadlock-free. Since the process syntax depends on types due to polymorphism, types are introduced first.

Definition 6.1 (Types). Types are defined by

$$A, B ::= \mathbf{1} \mid \bot \mid A \oplus B \mid A \mathbin{\&} B \mid A \otimes B \mid A \mathbin{⅋} B \mid !A \mid ?A \mid \exists X.A \mid \forall X.A \mid \boxed{?}A \mid \boxed{!}A \mid \boxed{\cancel{?}}A \mid \boxed{\cancel{!}}A$$

where X denotes a type variable. Duality $\overline{A}$ is the involution on types defined by

$$\overline{\mathbf{1}} \triangleq \bot \quad \overline{A \otimes B} \triangleq \overline{A} \mathbin{⅋} \overline{B} \quad \overline{A \oplus B} \triangleq \overline{A} \mathbin{\&} \overline{B} \quad \overline{!A} \triangleq ?\overline{B} \quad \overline{\exists X.A} \triangleq \forall X.\overline{A} \quad \overline{\boxed{?}A} \triangleq \boxed{!}\overline{A} \quad \overline{\boxed{\cancel{?}}A} \triangleq \boxed{\cancel{!}}\overline{A}$$

Definition 6.2 (Processes). The syntax of process terms is defined in Fig. 4.

The static part of the syntax comprises inaction, par, cut, cut!, share, sum, share-left and share-right; the dynamic part includes actions $\mathcal{A}, \mathcal{B}$, and forwarder. An action is typically a process $\alpha; P$, where α is an action-prefix and P is the continuation. The subject $s(\mathcal{A})$ of an action $\mathcal{A}$ is the leftmost name occurrence of $\mathcal{A}$. For example, the subject of the action send $x(y.P); Q$ is x.

The expression $P\ |x|\ Q$ binds the name x on processes P and Q. $y.P\ |!x|\ Q$ binds y in P and x in Q. Actions send $x(y.P); Q$, recv $x(y); P$, $!x(y); P$, call $x(y); P$, cell $x(y.P)$, read $x(y); P$, wrt $x(y.P); Q$ bind y on P. All other name occurrences are free. The set of free names of P is denoted by $fn(P)$; if $fn(P) = \emptyset$, we say P is closed. The expressions recv $x(X); P$, $\exists X.A$ and $\forall X.A$ bind the type variable X on process P and type A. Capture-avoiding substitution and α-conversion are defined as usual. We denote by $\{x/y\}P$ the process obtained by replacing the name y by x on P. Similarly, we denote by $\{A/X\}B$ (resp., $\{A/X\}P$) the type (resp., process) obtained by replacing X by A in B (resp., P).

Definition 6.3 (Structural Congruence $P \equiv Q$). Structural congruence $\equiv$ is the least congruence on processes that includes α-conversion and

- commutative laws for forwarder, par, cut, share and sum;
- commuting conversions for each pair of operations par, cut, cut!, share, share-left and sum;
- laws that distribute cut! over par, cut, cut!, share, share-left and share-right;
- laws that distribute par,cut, cut!, share, share-left and share-right over sum;
- and the conversions in Fig. 5.

N.B.:$x.\alpha$ and $x.\beta$ are read/write actions with subject x.

Basic rules of $\equiv$ essentially reflect the expected static laws, along the lines of the structural congruences / conversions in [Caires and Pfenning 2010; Wadler 2012]. For example, the commuting laws between cut and cut and between cut and share are written as

$$P\ |x|\ (Q\ |y|\ R) \equiv (P\ |x|\ Q)\ |y|\ R \ \text{[C-CC]}$$
$$P\ |x|\ (\text{share}\ y\ \{Q \mid\mid R\}) \equiv \text{share}\ y\ \{(P\ |x|\ Q) \mid\mid R\} \ \text{[C-CSh]}$$

provided $x, y \in \text{fn}(Q)$. The laws that distribute cut! over static constructors and that distribute static constructors over sum can be written in the following form

$$y.P\ |!x : A|\ (Q * R) \equiv (y.P\ |!x : A|\ Q) * (y.P\ |!x : A|\ R) \ \text{[D-Cut!]}$$
$$P * (Q + R) \equiv (P * Q) + (P * R) \ \text{[D-Sm]}$$

where $*$ stands for any of the static constructors par, cut, cut!, share, share-left and share-right.

More interesting are the share commuting conversions of Figure 5, which commute a share with a cell action, a sum of cell actions or a share. In [FF] two concurrent free actions are merged into a single free with the continuations being executed independently. [FA] commutes a share (on x) with read/write action (of subject x) on one thread and a free on the other with the read/write action (free is postponed).Similarly, [FL] postpones a free over a lock. [AA], [LA] and [LL] capture the interleaving laws for the atomic actions read, write and lock: it commutes a share with two concurrent actions with a sum that represents the two possible interleavings, thereby expressing the fundamental interplay between concurrency and nondeterminism. Notice that when a lock is commuted, the share becomes annotated with either an L or R to indicate the component that holds the lock and then only actions from that component are commuted as expressed by [leftA]. Rule [leftU] commutes a share-left with an unlock operation on the left argument with an unlock followed by a share. Finally, rule [symm] commutes a share-left with a share-right. Notice that the argument position is swapped. [symm] allow us to derive a right-version of each share-left rule and vice-versa. For example, by composing [symm] (on both ends) with [leftA] we obtain a right-version of rule [leftA]

$$\text{shareR}\ x\ \{P \mid\mid x.\alpha; Q\} \equiv \text{shareL}\ x\ \{x.\alpha; Q \mid\mid P\} \equiv x.\alpha; \text{shareL}\ x\ \{Q \mid\mid P\} \equiv x.\alpha; \text{shareR}\ x\ \{P \mid\mid Q\}$$

Share Commuting Conversions

$\mathsf{share}\ x\ \{\mathsf{free}\ x; P \mid\mid \mathsf{free}\ x; Q\}\ \equiv \mathsf{free}\ x; (P \mid\mid Q)$ [FF]

$\mathsf{share}\ x\ \{\mathsf{free}\ x; P \mid\mid x.\alpha; Q\}\ \equiv x.\alpha; \mathsf{share}\ x\ \{\mathsf{free}\ x; P \mid\mid Q\}$ [FA]

$\mathsf{share}\ x\ \{\mathsf{free}\ x; P \mid\mid \mathsf{lock}\ x; Q\} \equiv \mathsf{lock}\ x; \mathsf{shareR}\ x\ \{\mathsf{free}\ x; P \mid\mid Q\}$ [FL]

$\mathsf{share}\ x\ \{x.\alpha; P \mid\mid x.\beta; Q\} \equiv x.\alpha; \mathsf{share}\ x\ \{P \mid\mid x.\beta; Q\} + x.\beta; \mathsf{share}\ x\ \{x.\alpha; P \mid\mid Q\}$ [AA]

$\mathsf{share}\ x\ \{\mathsf{lock}\ x; P \mid\mid x.\alpha; Q\}\ \equiv \mathsf{lock}\ x; \mathsf{shareL}\ x\ \{P \mid\mid \alpha; Q\} + x.\alpha; \mathsf{share}\ x\ \{\mathsf{lock}\ x; P \mid\mid Q\}$ [LA]

$\mathsf{share}\ x\ \{\mathsf{lock}\ x; P \mid\mid \mathsf{lock}\ x; Q\}\ \equiv$
$\qquad \mathsf{lock}\ x; \mathsf{shareL}\ x\ \{P \mid\mid \mathsf{lock}\ x; Q\} + \mathsf{lock}\ x; \mathsf{shareR}\ x\ \{\mathsf{lock}\ x; P \mid\mid Q\}$ [LL]

$\mathsf{shareL}\ x\ \{x.\alpha; P \mid\mid Q\} \equiv x.\alpha; \mathsf{shareL}\ x\ \{P \mid\mid Q\}$ [leftA]

$\mathsf{shareL}\ x\ \{\mathsf{unlock}\ x; P \mid\mid Q\}\ \equiv \mathsf{unlock}\ x; \mathsf{share}\ x\ \{P \mid\mid Q\}$ [leftU]

$\mathsf{shareL}\ x\ \{P \mid\mid Q\} \equiv \mathsf{shareR}\ x\ \{Q \mid\mid P\}$ [symm]

Inaction Conversions

$P \mid\mid \mathbf{0}\ \equiv P$ [**0**M] $\quad \mathbf{0} + \mathbf{0} \equiv \mathbf{0}$ [**0**Sm]

Fig. 5. Selected Structural Congruence Rules $P \equiv Q$.

Interestingly, idempotency of inaction (Fig. 5 [**0**Sm]) w.r.t. sum is sufficient to derive idempotency of all processes w.r.t. sum, as we have $P \equiv P \mid\mid \mathbf{0} \equiv P \mid\mid (\mathbf{0} + \mathbf{0}) \equiv (P \mid\mid \mathbf{0}) + (P \mid\mid \mathbf{0}) \equiv P + P$. Before defining reduction, we introduce static contexts, which are defined by

$$\begin{array}{l} C ::= - \mid C \mid\mid P \mid P \mid\mid C \mid C\ \ |x|\ \ P \mid P\ \ |x|\ \ C \mid y.P\ |!x|\ C\ \mid \mathsf{share}\ x\ \{C \mid\mid P\} \mid \mathsf{share}\ x\ \{P \mid\mid C\} \mid \\ C + P\ \mid P + C \mid \mathsf{shareL}\ x\ \{C \mid\mid P\} \mid \mathsf{shareL}\ x\ \{P \mid\mid C\} \mid \mathsf{shareR}\ x\ \{P \mid\mid C\}\ \mid \mathsf{shareR}\ x\ \{C \mid\mid P\} \end{array}$$

A static context is a context where the hole is neither guarded by any action nor lies in the server body P of a cut! $y.P\ |!x|\ Q$. We write $-$ for the empty context and $C[P]$ for the process obtained by replacing the hole in C by P (notice that in $C[P]$ the context C may bind free names of process P).

Definition 6.4 (Reduction $P \to Q$). Reduction $\to$ is the least relation on processes defined by the previously introduced principal cut-reductions and by the rules

$$\frac{P \equiv P' \quad P' \to Q' \quad Q' \equiv Q}{P \to Q}\ [\equiv] \qquad\qquad \frac{P \to Q}{C[P] \to C[Q]}\ [\text{cong}]$$

where C is an arbitrary static context. Let $\Rightarrow$ stand for the transitive closure of $\to \cup \equiv$.

Example 6.5. We illustrate our operational semantics by deriving the reduction of Example 3.3 step-by-step

$$\begin{array}{ll}
\mathsf{share}\ x\ \{R \mid\mid \mathsf{send}\ z\ x; P\}\ \ |z|\ \ \mathsf{recv}\ z(y); \mathsf{share}\ y\ \{Q \mid\mid S\} & (\equiv \text{rule [C-CSh]}) \\
\equiv\ \mathsf{share}\ x\ \{R \mid\mid \underline{\mathsf{send}\ z\ x; P}\ \ |z|\ \ \mathsf{recv}\ z(y); \mathsf{share}\ y\ \{Q \mid\mid S\}\} & (\text{def. free ouput}) \\
= \mathsf{share}\ x\ \{R \mid\mid \underline{\mathsf{send}\ z(w.\ \mathsf{fwd}\ w\ x); P}\ \ |z|\ \ \underline{\mathsf{recv}\ z(y); \mathsf{share}\ y\ \{Q \mid\mid S\}}\} & (\to \text{rule } [\otimes ⅋]) \\
\to \mathsf{share}\ x\ \{R \mid\mid P\ \ |z|\ \ (\underline{\mathsf{fwd}\ w\ x}\ \ |w|\ \ \underline{\{w/y\}\mathsf{share}\ y\ \{Q \mid\mid S\}})\} & (\to \text{rule [fwd]}) \\
\to \mathsf{share}\ x\ \{R \mid\mid P\ \ |z|\ \ \underline{\{x/w\}\{w/y\}\mathsf{share}\ y\ \{Q \mid\mid S\}}\} & (\text{name substitution}) \\
= \mathsf{share}\ x\ \{R \mid\mid P\ \ |z|\ \ \mathsf{share}\ x\ \{\{x/y\}Q \mid\mid \{x/y\}S\}\} & (\equiv \text{rule [C-CSh]}) \\
\equiv P\ \ |z|\ \ \mathsf{share}\ x\ \{R \mid\mid \mathsf{share}\ x\ \{\{x/y\}Q \mid\mid \{x/y\}S\}\} &
\end{array}$$

6.1 Subject Reduction for πSSL

We establish the fundamental type safety properties: subject reduction (Theorem 6.6) and progress (Theorem 6.11). In the context of our session typed language these properties entail session fidelity and deadlock-freedom.

Theorem 6.6 (Subject Congruence and Subject Reduction for πSSL). *Assume $P \vdash \Delta; \Gamma$. Then (1) If $P \equiv Q$ then $Q \vdash \Delta; \Gamma$ and (2) If $P \to Q$ then $Q \vdash \Delta; \Gamma$.*

Proof. We verify that all conversions included in $\equiv$ and $\to$ rules are type-preserving, and conclude by induction on derivations of $P \equiv Q$ and $P \to Q$. □

6.2 Progress for πSS

To state and prove progress (Theorem 6.11) we introduce the notion of live process.

Definition 6.7 (Live). A process P is live if $P = C[\mathcal{A}]$ or $P = C[\mathsf{fwd}\ x\ y]$ for some static context C and action $\mathcal{A}$.

Intuitively, a process is live if it presents an unguarded action prefix (or forwarder) waiting to interact. To compositionally prove progress we need to characterise the potential interactions of (possibly open) typed processes, for which we define the following notion of interaction offering, which is akin to the π-calculus observability (cf. [Sangiorgi and Walker 2001]).

Definition 6.8 (P offers an interaction at x). $P \downarrow_x$ is defined by rules

$$\frac{}{\mathsf{fwd}\ x\ y \downarrow_x}\,[\text{fwd}] \quad \frac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_x}\,[\text{act}] \quad \frac{P \downarrow_x}{(P \mid\mid Q) \downarrow_x}\,[\text{par}] \quad \frac{P \downarrow_y \quad y \neq x}{(P\ \ |x|\ \ Q) \downarrow_y}\,[\text{cut}]$$

$$\frac{Q \downarrow_z \quad z \neq x}{(y.P\ |!x|\ Q) \downarrow_z}\,[\text{cut!}] \quad \frac{P \downarrow_y \quad y \neq x}{(\mathsf{share}\ x\ \{P \mid\mid Q\}) \downarrow_y}\,[\text{share}] \quad \frac{P \downarrow_x}{(P + Q) \downarrow_x}\,[\text{sum}] \quad \frac{P \equiv Q \quad Q \downarrow_x}{P \downarrow_x}\,[\equiv]$$

The definition of $P \downarrow_x$ is explicitly closed under $\equiv$ and propagates offers on the various operators. For example, a mix offers an interaction on a name x provided one of its arguments does. The interactions of the left argument are explicitly offered by rule [par], whereas those of the right argument are derivable through rule [$\equiv$] followed by [par], since the mix construct is commutative.

In a share $\mathsf{share}\ x\ \{P \mid\mid Q\}$, processes P and Q run in parallel, freely communicating with the external context, but sharing memory cell x. As a consequence, actions on names other than x will always be offered (rule [share]). Additionally, actions on x will be offered if they are offered by both P and Q (Lemma 6.9(1)). For example, $\mathsf{share}\ x\ \{\mathsf{free}\ x; P \mid\mid \mathsf{read}\ x(y); Q\}$ offers an action on x, since it can be written in the equivalent form $\mathsf{read}\ x(y); (\mathsf{share}\ x\ \{\mathsf{free}\ x; P \mid\mid Q\})$ by applying the $\equiv$ rule [FA] (Fig. 5). We annotate $P \downarrow_x$ with $P \downarrow_{x:\text{fwd}}$ or $P \downarrow_{x:\text{act}}$, depending on whether the interaction is introduced by a forwarder or an action, respectively.

Lemma 6.9. *The following properties of $P \downarrow_x$ hold*

(1) *Let $P \vdash \Delta, x : \boxed{!}A; \Gamma$, $Q \vdash \Delta', x : \boxed{!}A; \Gamma$, $P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$. Then, $\mathsf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:act}$.*

(2) *Let $P \vdash \Delta, x : \overline{A}; \Gamma$, $Q \vdash \Delta', x : A; \Gamma$,$P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$. Then, $P\ \ |x|\ \ Q$ reduces.*

(3) *Let $P \vdash \Delta, x : A; \Gamma$, $Q \vdash \Delta', x : A; \Gamma$ and $P \downarrow_{x:fwd}$. Then, $P\ \ |x|\ \ Q$ reduces.*

(4) *Let $P \vdash y : \overline{A}; \Gamma$, $Q \vdash \Delta; \Gamma, x : A$ and $Q \downarrow_x$. Then, $y.P\ |!x|\ Q$ reduces.*

(5) *Let $P \vdash \Delta, x : A; \Gamma$ and suppose that A is not of the form $\boxed{?}B$. If $P \downarrow_{x:fwd}$, then either $P \downarrow_{y:fwd}$ for some $y : \overline{A} \in \Delta$ or P reduces.*

Proof. By induction on the derivations of $P \downarrow_x$. □

Lemma 6.9(5) describes how the interactions offered by a forwarder fwd $x\ y$ propagate in P. Either name y occurs free, and P offers a forwarder interaction at y, or lies in the scope of a cut $- \ |y|\ -$, in which case a reduction can be triggered (Lemma 6.9(3)). The typing constraint $A \neq \boxed{?}B$ excludes processes like share y {fwd $x\ y$ || wait $z; Q$}, that neither reduce nor offer an interaction at y. Intuitively, the share is suspended on the availability of cell usages at name y. We will see later that this kind of processes will play a key role in defining πSS normal forms (Section 8).

Lemma 6.10. *Let $P \vdash \Delta; \Gamma$ be a live process. Then either $P \downarrow_x$ or P reduces.*

Proof. By induction on the typing derivation for $P \vdash \Delta; \Gamma$ and Lemma 6.9. □

Theorem 6.11 (Progress for πSS). *Let $P \vdash \emptyset; \emptyset$ be a live process of πSS. Then, P reduces.*

Proof. Follows from Lemma 6.10, since $\text{fn}(P) = \emptyset$. □

6.3 Progress for πSSL

We prove that our extension πSSL of πSS with locks also satisfies progress.

Definition 6.12 (P offers an interaction at x). We define $P \downarrow_x$ by extending Definition 6.8 with

$$\frac{P \downarrow_y \quad y \neq x}{(\mathsf{shareL}\ x\ \{P \parallel Q\}) \downarrow_y}\ [\text{shareL1}] \qquad \frac{Q \downarrow_y \quad y \neq x}{(\mathsf{shareL}\ x\ \{P \parallel Q\}) \downarrow_y}\ [\text{shareL2}]$$

We show the following key properties of the interaction offering relation.

Lemma 6.13. *The following properties of $P \downarrow_x$ hold*

(1) *Let $P \vdash \Delta, x : \boxed{\cancel{!}}A; \Gamma$, $Q \vdash \Delta', x : \boxed{!}A; \Gamma$ and $P \downarrow_{x:act}$. Then,* shareL *x $\{P \parallel Q\} \downarrow_{x:act}$.*

(2) *Let $P \vdash \Delta, x : A; \Gamma$ and suppose that A is not of the form $\boxed{\cancel{?}}B$. If $P \downarrow_{x:fwd}$, then either $P \downarrow_{y:fwd}$ for some $y : \overline{A} \in \Delta$ or P reduces.*

Actions on the left component of a share-left are always propagated (Lemma 6.13(1)), as expressed by $\equiv$ rules [leftA] and [leftU]. From $\equiv$ [symm] and the fact that $- \downarrow_x$ is closed by $\equiv$, we obtain that actions of the right component of a share-right are always propagated as well

(1') Let $P \vdash \Delta, x : \boxed{!}A; \Gamma$, $Q \vdash \Delta', x : \boxed{\cancel{!}}A; \Gamma$ and $Q \downarrow_{x:\text{act}}$. Then, shareR x $\{P \parallel Q\} \downarrow_{x:\text{act}}$

Theorem 6.14 (Progress for πSSL). *Let $P \vdash \emptyset; \emptyset$ be a live process of πSSL. Then, P reduces.*

7 CONFLUENCE

In this section we prove that the relation $\Rightarrow$ of language πSSL satisfies the diamond property, thereby substantiating the claim that $\Rightarrow$ can be understood as a proof equivalence relation (cf. PaT). Since $\Rightarrow \triangleq (\rightarrow \cup \equiv)^+$, a natural proof strategy would be demonstrating the diamond property for $\rightarrow \cup \equiv$, but this property fails. To see why, consider the process $P \parallel (Q_1 + Q_2 + Q_2)$ and assume that $P' \leftarrow P \rightarrow P''$. Then, we can form the $\rightarrow$-reductions

$$\begin{array}{lcl} P \parallel (Q_1 + Q_2 + Q_2) & \rightarrow & P' \parallel Q_1 + P' \parallel Q_2 + P \parallel Q_3 \triangleq R \\ P \parallel (Q_1 + Q_2 + Q_2) & \rightarrow & P \parallel Q_1 + P'' \parallel Q_2 + P'' \parallel Q_3 \triangleq S \end{array}$$

Assume as well that P' and P'' reduce to a common form P^* in one single step: $P' \rightarrow P^* \leftarrow P''$. Note, however, that we need two $\rightarrow$-reduction steps to bring processes R and S to a common term

$$R \quad \overset{2}{\rightarrow} \quad P' \parallel Q_1 + P^* \parallel Q_2 + P'' \parallel Q_3 \quad \overset{2}{\leftarrow} \quad S$$

The solution is to allow, in one single step, to perform parallel independent $\rightarrow$-reductions of sum components (cf. Definition 7.1). This is a well-known technique due to Tait and Martin-Löf and which is often employed when establishing the confluence of calculi that involve nondeterministic sums (see, for example, [Ehrhard and Regnier 2003; Pagani and Tranquilli 2009]).

Definition 7.1 (Parallel Sum Reduction $\twoheadrightarrow$). Let $\twoheadrightarrow$ be the least relation that contains $\rightarrow$ and satisfies the rule [+par] $P \twoheadrightarrow P', Q \twoheadrightarrow Q' \;\supset\; P + Q \twoheadrightarrow P' + Q'$.

Lemma 7.2. *The following (in)equalities hold: (1) $\rightarrow \subseteq \twoheadrightarrow$, (2) $\twoheadrightarrow \subseteq \xrightarrow{+}$ and (3) $\Rightarrow = (\twoheadrightarrow \cup \equiv)^+$.*

Proof. (1) follows directly from Definition 7.1. (2) is by induction on $\twoheadrightarrow$. (1) and (2) imply (3). □

Since $\Rightarrow = (\twoheadrightarrow \cup \equiv)^+$, we establish our main theorem by showing the diamond property for $\twoheadrightarrow \cup \equiv$ (Lemma 7.6). To handle the complexity of the $\equiv$-commuting conversions, we show first that we can write each process in a $\equiv$-normal form by interleaving all the concurrent cell actions and distributing the static operators over sums (Lemma 7.4). This essentially computes a normal form for the left-to-right oriented $\equiv$-rules that distribute over sum [D-Sm] and that interleave cell usages [AA], [LA] and [LL]. Each $\twoheadrightarrow$-reduction is then obtained by evaluating the summands of the normal form with a restricted form of reduction (Definition 7.3) that does not manipulate sums (Lemma 7.4) and for which a diamond property is straightforward to establish (Lemma 7.6).

Definition 7.3 (Relation $\rightarrow_d$). Let $\equiv_d$ be defined by all the rules of $\equiv$ except rules [D-Sm], [AA], [LA], [LL] and [0Sm]. Let $\rightarrow_d$ be the least relation that satisfies all the rules of $\rightarrow$ except [$\equiv$] and for which the rule [$\equiv_d$] $P \equiv_d P' \rightarrow_d Q' \equiv_d Q \;\supset\; P \rightarrow_d Q$ holds. Furthermore, in rule [cong], the hole in the context C is not guarded by a sum.

Lemma 7.4. *For each process P there is a sum of processes $\sum_{i \in \mathcal{I}} P_i$ s.t. $P \equiv \sum_{i \in \mathcal{I}} P_i$ and for which the following property holds*

(1) *If $P \twoheadrightarrow Q$, then exists $\{Q_{ij}\}_{i \in \mathcal{I}, j \in \mathcal{J}_i}$ with $Q \equiv \sum_{i \in \mathcal{I}, j \in \mathcal{J}_i} Q_{ij}$ and s.t. for all $i \in \mathcal{I}$, $j \in \mathcal{J}_i$ either $P_i \equiv_d Q_{ij}$ or $P_i \rightarrow_d Q_{ij}$.*

Proof. For each process P, we define its sum expansion $\mathcal{S}(P)$ inductively. In the following, $\mathcal{X}$ stands for $\mathbf{0}$, fwd $x\, y$ or any action. Assume $\mathcal{S}(P) = \sum_{i \in \mathcal{I}} P_i$ and $\mathcal{S}(Q) = \sum_{j \in \mathcal{J}} Q_j$ and let $\mathcal{I}_x(P_i, Q_j)$ be an auxiliary map that computes all the interleavings of process share x $\{P_i \mid\mid Q_j\}$. Then

$$\mathcal{S}(\mathcal{X}) \triangleq \mathcal{X} \quad \mathcal{S}(P \mid\mid Q) \triangleq \textstyle\sum_{i \in \mathcal{I}, j \in \mathcal{J}} P_i \mid\mid Q_j \quad \mathcal{S}(P \;|x|\; Q) \triangleq \textstyle\sum_{i \in \mathcal{I}, j \in \mathcal{J}} P_i \;|x|\; Q_j$$

$$\mathcal{S}(y.P \;|!x|\; Q) \triangleq \textstyle\sum_{j \in \mathcal{J}} y.P \;|!x|\; Q_j \quad \mathcal{S}(\mathsf{shareL}\; x\; \{P \mid\mid Q\}) \triangleq \textstyle\sum_{i \in \mathcal{I}, j \in \mathcal{J}} \mathsf{shareL}\; x\; \{P_i \mid\mid Q_j\}$$

$$\mathcal{S}(\mathsf{shareR}\; x\; \{P \mid\mid Q\}) \triangleq \textstyle\sum_{i \in \mathcal{I}, j \in \mathcal{J}} \mathsf{shareR}\; x\; \{P_i \mid\mid Q_j\}$$

$$\mathcal{S}(\mathsf{share}\; x\; \{P \mid\mid Q\}) \triangleq \textstyle\sum_{i \in \mathcal{I}, j \in \mathcal{J}} \mathcal{I}_x(P_i, Q_j) \quad \mathcal{S}(P + Q) \triangleq \textstyle\sum_{i \in \mathcal{I}} P_i + \sum_{j \in \mathcal{J}} Q_j$$

Sum expansions of $\equiv$-equivalent processes P, Q are related in the following way: for all $i \in \mathcal{I}$ there exists $j \in \mathcal{J}$ s.t. $P_i \equiv_d Q_j$. Property (1) is established by induction on $P \twoheadrightarrow Q$. □

Corollary 7.5. *Suppose $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$. There are sum of processes $\sum_{i \in \mathcal{I}} P_i \equiv P$, $\sum_{i \in \mathcal{I}} Q_i \equiv Q$ and $\sum_{i \in \mathcal{I}} R_i \equiv R$ s.t. the following property holds*

(1) *For all $i \in \mathcal{I}$, P_i $(\rightarrow_d \cup \equiv_d)$ Q_i and P_i $(\rightarrow_d \cup \equiv_d)$ R_i.*

Proof. Compute a sum expansion $\{P'_i\}_{1 \le i \le n}$ for P (lemma 7.4) and repeat the terms P'_i as many times as necessary ($\equiv$-equivalence is preserved because sum is idempotent). □

As opposed to $\rightarrow$ and $\twoheadrightarrow$, the reductions of $\rightarrow_d$ cannot partially overlap, which justifies

Lemma 7.6. *If $P \rightarrow_d Q$ and $P \rightarrow_d R$, then either $Q \equiv_d R$ or exists S s.t. $Q \rightarrow_d S$ and $R \rightarrow_d S$.*

Proof. Write P as $P \equiv_d C[R_1] \ldots [R_n]$ where $R_1, \ldots, R_n$ lists all $\rightarrow_d$-redexes of P. We then verify that all $\rightarrow_d$-reductions at the R_i commute. □

LEMMA 7.7. *Suppose $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$. Either $Q \equiv R$ or exists S s.t. $Q \twoheadrightarrow S$ and $R \twoheadrightarrow S$.*

PROOF. By applying Corollary 7.5 to $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$. □

THEOREM 7.8 (DIAMOND $\Rightarrow$). *Suppose $P \Rightarrow Q$ and $P \Rightarrow R$. There exists S s.t. $Q \Rightarrow S$ and $R \Rightarrow S$.*

PROOF. Follows from Lemma 7.7 and Lemma 7.2(3). □

8 PROOF NORMALISATION AND CUT-ELIMINATION

In this section we study normalisation and cut-elimination for open processes $P \vdash \Delta; \Gamma$ of the language πSS without quantifiers. We show that every process can be reduced to a normal form on which there are no cuts, except on open identity axioms on shared aliases: we call open cells to such trivial cuts. Open cells have a simple form $\mathsf{cell}\ x(y.P)\ \ |x|\ \ \mathsf{share}\ x\ \{\mathsf{fwd}\ x\ z \parallel Q\}$, where the shared alias x is forwarded to a free name $z : \boxed{!}A$. The share in the open cell cannot be converted to a (sum of) $\boxed{!}$ introduction forms (since $\mathsf{fwd}\ x\ z$ is not a $\boxed{!}$ introduction form, and offers no structure at z). There is no real redex in an open cell: the share is suspended on the availability of cell usages at z from the environment; justifying open cells as normal forms.

Definition 8.1 (Open Cell). An open cell is a process of the form

$$\mathsf{cell}\ x(y.P)\ \ |x|\ \ \mathsf{share}\ x\ \{\mathsf{fwd}\ x\ z \parallel Q\}$$

Definition 8.2 (Normal form). A process is a normal form if it contains no cuts except open cells.

PROPOSITION 8.3 (SUB-FORMULA PROPERTY). *If $P \vdash \Delta; \Gamma$ is normal form, then the derivation only contains sub-formulas (up to duality) of the types in Δ, Γ.*

PROOF. By induction on the typing derivation. □

To obtain normalisation we introduce a conversion relation $\approx$ on proofs, which includes structural congruence $\equiv$ and reduction $\rightarrow$, but adds a complete set of commuting conversions for linear logic along standard lines [Caires and Pfenning 2010; Caires et al. 2016; Wadler 2012], and some specific conversions for our new constructs (full definition in the technical report [Rocha and Caires 2021a]).

THEOREM 8.4 (NORMALISATION). *If $P \vdash \Delta; \Gamma$ then there exists normal Q s.t. $P \approx Q$.*

PROOF. We extend Pfenning's structural cut-elimination technique [Pfenning 1995] to our setting, and explicitly construct a normal form for each proof $P \vdash \Delta; \Gamma$. We show how a normal form $R \approx P\ \ |x|\ \ Q$ can be constructed from normal forms P and Q, by lexicographical induction on $(A, \#(P\ \ |x|\ \ Q))$, where A is the cut formula (we consider that $\boxed{!}B > !B$) and # is a measure on processes. Instrumentally, we first show that any share $\mathsf{share}\ x\ \{- \parallel -\}$ is $\approx$-equivalent to a sum of #-smaller processes or to an open cell. □

The following corollary exposes a strong conservativeness and expressiveness result about our language, which follows from the sub-formula property.

COROLLARY 8.5 (CONSERVATIVITY). *Let process $P \vdash \Delta; \Gamma$ be a sequent in classical linear logic with mix. Then there is a cut-free process Q s.t. $P \approx Q$.*

Corollary 8.5 implies that any process in our language with shared state but that implements an interface only manifesting standard propositional linear logic based session types (that is without $\boxed{?}$, $\boxed{!}$ modalities in its typing) is equivalent to a (possibly non-deterministic and larger) process that *does not use imperative constructs at all.* This normal form expresses the externally observable behaviour of the original stateful open process. For example, consider the conversion

$$\begin{array}{rl} & \mathsf{cell}\ x(b.\mathsf{true}_b)\ \ |x|\ \ \mathsf{share}\ x\ \{\mathsf{wrt}\ x(b.\mathsf{false}_b); \mathsf{free}\ x; \mathbf{0} \parallel \mathsf{read}\ x(b); \mathsf{free}\ x; \mathsf{send}\ y\ b; \mathsf{close}\ y\} \\ \approx & \mathsf{send}\ y(z.!z(b); \mathsf{true}_b); \mathsf{close}\ y + \mathsf{send}\ y(z.!z(b); \mathsf{false}_b); \mathsf{close}\ y \vdash y :!\mathsf{Bool} \otimes \mathbf{1}; \emptyset \end{array}$$

The cut-free process on the right hand side of $\approx$ is behaviourally equivalent to the imperative unreduced one on the left hand side, and summarises its behaviour as a nondeterministic “state-free" process. For the sake of simplicity we opted to prove normalisation of πSS, without polymorphism. Extending the proof with locks can be done along predictable lines. Extending the proof with the impredicative polymorphic types would require a different technique however, such as linear logical relations [Pérez et al. 2012].

9 TYPE CHECKER AND INTERPRETER

We developed a type checker and interpreter in Java ($\sim$ 11k loc) using the JavaCC parser generator, which was submitted as a companion artifact for this paper [Rocha and Caires 2021b]. All the examples in the paper are validated by the implementation, and we also have developed many others ($\sim$ 3k loc), ranging from the definition of inductive datatypes (Naturals, Lists) using System-F style encodings, to concurrent ADTs (Counter, Stack, Queue), as well as a test suite. The supported language includes efficient pragmatic extensions (native basic datatypes int, boolean and string).

Our type checker deals with linearity and context splitting by lazy propagation of context residuals [Hodas and Miller 1994], and imperative update of the typing environments. The interpreter is a fine-grain concurrent runtime system, relying on the java.util.concurrent.* package. Our implementation exposes nondeterminism arising from concurrency as real committed nondeterminism, so that the sum operator introduced in the paper, while crucial to establish a PaT model and to characterise the semantics of our language, is not present in our practical runtime system, that is, the implementation commits to one of the summands in sums, while any of such may be nondeterministically picked. The share construct is simply implemented by aliasing. We use reference counting instrumented by the share constructs to garbage collect reference cells. Importantly, structural congruence does not play any explicit role in the implementation model. Types are not used at runtime, but to orient forwarders at state types, and handle polymorphic type variables.

When running complex examples involving the concurrent shareable ADTs, such as the Queue, where we intensively use session processes to implement everything down to basic inductive data types such as lists (along the lines of [Wadler 1990b]), our implementation spawns thousands of short-lived processes. In order to reduce the overhead associated with thread creation/destruction we have opted to manage the execution of concurrent tasks through a cached thread pool.

10 RELATED WORK

In this paper, we develop a propositions-as-types (PaT) interpretation connecting a programming language with concurrency and shared state constructs within a conservative extension of classical linear logic. The notion of PaT goes back to the functional interpretation of intuitionistic logic due to Brouwer, Heyting and Kolmogorov, but was only brought under the spotlight after the famous notes of Curry and Howard [Howard 1980]. It has been since then considered both an intriguing and prolific concept, with many instances and consequences (see [Wadler 2015]).

We base our development on the sessions-as-types interpretation of linear logic [Caires and Pfenning 2010; Caires et al. 2016; Wadler 2012], which yields an expressive typed programming language related to the session π-calculus, while ensuring "for-free“ progress (deadlock-freedom), confluence, and normalisation as a consequence of the correspondence between computation and proof reduction. Many works have explored linearity (or affinity) and mutable state, e.g., [Ahmed et al. 2007; Caires and Seco 2013; Nanevski et al. 2008; O'Hearn and Reynolds 2000; Sunshine et al. 2011; Wadler 1990a]. Recently, Rust [Jung et al. 2018a; Matsakis and Klock 2014] has introduced affine types and forms of ownership types [Clarke et al. 1998; DeLine and Fähndrich 2001] to the world of widely adopted programming languages. In our system, a simple form of ownership and ownership transfer results naturally from the underlying linear typing discipline.

We believe that our work is the first to address the challenge of defining a PaT interpretation for shared state, which we have accomplished by internalising nondeterminism in the logic using sums, thus ensuring confluence. From the proof-theoretic side, we have built on ideas from DiLL [Ehrhard 2018] to obtain a logical perspective of the interplay between concurrency and non-determinism as captured by algebraic interleaving laws [Hennessy and Milner 1985]. In the presence of interactions between competing resources, for example, such as concurrent accesses to reference cells as in our language, or matching of quantities of resources in DiLL (e.g., in cut-reduction between co-contraction and contraction), finding an adequate model in which confluence must hold has motivated the introduction of sums (or linear combinations) $P + Q$ of objects [Ehrhard 2018], to be understood as "superposition" or nondeterministic choice of "incompatible" alternatives. The idea of using formal sums to recover, in an algebraic form, confluence and normalisation for an untyped CCS-like process calculus was also explored in [Beffara 2008]. Sums allow confluence of cut-reduction to be preserved, allowing nondeterminisic proof-reductions to be understood equationally, cf. behavioural equivalence in process algebras [Hennessy and Milner 1985] or program equivalence in power-domain denotational semantics [Plotkin 1976]. It is also used crucially in our PaT interpretation, where interleaving of concurrent imperative actions leads to sum of states.

In our system, constructs for modelling state are represented by proof terms. Particularly, our “store" is represented by proof objects itself, thus approaching a pure algebraic theory where imperative computation can be reasoned about equationally, in the spirit of the syntactic store of [Felleisen and Friedman 1989], distinct of the monadic approaches [Peyton Jones and Wadler 1993].

The works [Balzer and Pfenning 2017; Balzer et al. 2019] introduced manifest sharing, the first proposal to represent shared state on top of a session types linear logic interpretation. Although the resulting system is grounded on the Curry-Howard correspondences of [Caires and Pfenning 2010; Caires et al. 2016; Wadler 2012] it departs from a *pure* PaT interpretation in its stateful extension, unlike the work presented in this paper, and explores a different route, based on a special purpose operational semantics, designed to keep track of shared channels availability to control resource acquisition and perform context switching. Therefore, their work does not address how to achieve deadlock freedom or confluence from the basic linear logic type discipline. As a consequence manifest sharing supports programs with linear stored state, such as an imperative linked-list based queue, or with cyclic dependencies, such as the the dining philosophers. These examples are out of the scope of our PaT system, which is nevertheless expressive enough to represent imperative ADTs like stacks and queues within a pure logical approach. Thus both manifest sharing and our PaT approach tackle a common theme with different contributions, directions, and merits.

We further detail our discussion. The key idea of manifest sharing is to serialise concurrent access to linear objects by two modal operators (acquire and release), which induce a stratification [Pfenning and Griffith 2015] of session types in two layers, and provide locking / unlocking behaviour at computation points where a resource invariant holds. Reduction in [Balzer and Pfenning 2017; Balzer et al. 2019] cannot solely be seen as proof simplification, as the operational semantics relies on proof construction / deconstruction steps, where the “wait" computation states are seen as an “incomplete proof", possibly introducing deadlock. Moreover, in manifest sharing, computation is not confluent, and thus cannot be seen as a proof simplification. This is unlike our system in which confluence holds due to the introduction of sums and co-contraction based sharing.

In [Balzer et al. 2019] the authors further extend the framework of [Balzer and Pfenning 2017] with an additional type system to enforce deadlock freedom, by relying on (extra-logical) partial orders on events to ensure acyclicity [Dezani-Ciancaglini et al. 2008; Lynch 1980]. Those partial orders on locks impose some restrictions on the type system, for example: every process must have released all its acquired resources before communicating along its offering channel and the linear forwarder cannot be typed. Furthermore, in manifest sharing, partial orders on locks have to be manually defined by the programmer. In our case, even in the presence of locks, deadlock freedom results directly from typing and the progress theorem as a consequence of acyclicity in linear logic proofs. Moreover, the partial order approach of [Balzer et al. 2019] does not type processes that spawn a statically undetermined number of shared objects, while in our work such systems are naturally allowed and intrinsically covered by PaT.

Earlier work established relationships between variants of the π-calculus and proof-net reduction in DiLL [Ehrhard and Laurent 2010]. Recently, [Atkey et al. 2016; Kokke et al. 2019] also studied non-determinism in the setting of logical interpretations of session types. The work [Atkey et al. 2016] suggests the introduction of concurrency on extensions of the system of Wadler using conflation, but this carries the cost of losing confluence and deadlock absence as present in the basic system. The work [Kokke et al. 2019] draws on bounded linear logic, by introducing two integer-subscripted modalities that keep track of the number of statically fixed client-server interactions. On the other hand, our reference cells can be shared by an arbitrary number of clients that evolves dynamically (see Example 3.3). Recently, [Qian et al. 2021] have introduced coexponentials to Classical Linear Logic in order to model interactions between stateful servers and several clients. This work also draws inspiration on DiLL but, as opposed to DiLL and our calculus, they do not internalise non-determinism with sums. In [Caires and Pérez 2017], the authors extend the linear logic interpretation of session types [Caires and Pfenning 2010; Caires et al. 2016; Wadler 2012] with modalities to model effects and non-determinism, following a monadic approach, that excludes the emergence of non-determinism as the effect of races in memory access as we do in this paper.

11 CONCLUDING REMARKS

We introduced a PaT approach to state, in the context of linear logic interpretations of session types. Our model handles first-class mutable and shareable reference cells, and enjoys basic properties of PaT, namely proofs as programs, formulas as types, and evaluation as proof simplification, justified by key meta-theoretical results: preservation, progress, confluence and normalisation.

Our results shed light on the relationship between concurrency and nondeterminism from a logical viewpoint, and reconcile nondeterminism with proof identity by internalising choice. Our concrete implementation collapses nondeterminism into particular choices during execution, while our model still captures all possible choices, cf. the confluence property. We presented an extension that allows critical sections to be represented using locks, while keeping faithful to PaT, and preserving the crucial progress property.

We have developed a type checker and interpreter (using multi-threading), submitted as a companion artifact [Rocha and Caires 2021b]. All the examples in the paper are validated by our implementation ranging from the definition of inductive datatypes to concurrent ADTs.

Currently, our approach only handles shared cells holding persistent values which do not depend on free linear names and, as such, we cannot express cyclic data structures. However, we have shown how our language is already quite expressive, see e.g. the implementation of shareable mutable ADTs, such as Stacks and Queues. Extending our framework to handle linear stored state is object of ongoing work and would require a more fine grained analysis of the state manipulation primitives, of the recursive behaviour of cells, and the consideration of sharing protocols [Jung et al. 2018b; Militão et al. 2016]. We would also like to explore how dependent types may express resource invariants and possibly notions of abstract separation [Dinsdale-Young et al. 2013; Jung et al. 2018b; Krishnaswami et al. 2012].

Acknowledgements. We would like to thank Stephanie Balzer, Bernardo Toninho, Nobuko Yoshida and Philip Wadler for insightful discussions. We would like also to thank the anonymous reviewers for their helpful comments and suggestions. This work is supported by NOVA LINCS (UID/CEC/04516/2019 and UIDB/04516/2020).

REFERENCES

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L<sup>3</sup>: A Linear Language with Locations. *Fundam. Inf.* 77, 4 (Dec. 2007), 397–449.

J-M. Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *J. Logic Comput.* 2, 3 (1992), 197–347.

Robert Atkey, Sam Lindley, and J. Garrett Morris. 2016. *Conflation Confers Concurrency.* Springer International Publishing, Cham, 32–55. https://doi.org/10.1007/978-3-319-30936-1_2

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

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 611–639.

Emmanuel Beffara. 2008. An Algebraic Process Calculus. In *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science (LICS '08).* IEEE Computer Society, USA, 130–141. https://doi.org/10.1109/LICS.2008.40

Michele Boreale. 1996. On the Expressiveness of Internal Mobility in Name-Passing Calculi. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96).* Springer-Verlag, Berlin, Heidelberg, 163–178.

Luís Caires and Jorge A. Pérez. 2016. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In *Formal Techniques for Distributed Objects, Components, and Systems*, Elvira Albert and Ivan Lanese (Eds.). Springer International Publishing, Cham, 74–95.

Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201.* Springer-Verlag, Berlin, Heidelberg, 229–259. https://doi.org/10.1007/978-3-662-54434-1_9

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13).* Springer-Verlag, Berlin, Heidelberg, 330–349. https://doi.org/10.1007/978-3-642-37036-6_19

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236.

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Towards Concurrent Type Theory. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(TLDI '12).* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2103786.2103788

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. https://doi.org/10.1017/S0960129514000218

Luís Caires and João C. Seco. 2013. The Type Discipline of Behavioral Separation. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13).* Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2429069.2429103

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: a logical explanation of multiparty session types. In *27 International Conference on Concurrency Theory (CONCUR'16).* Québec City, Canada. https://hal.inria.fr/hal-01336600

Luca Cardelli. 1991. Typeful Programming. *IFIP State-of-the-Art Reports: Formal Description of Programming Concepts* (1991), 431–507.

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523.

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98).* Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01).* Association for Computing Machinery, New York, NY, USA, 59–69. https://doi.org/10.1145/378795.378811

Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. 2008. On Progress for Structured Communications. In *Trustworthy Global Computing*, Gilles Barthe and Cédric Fournet (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–275.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of POPL* (proceedings of popl ed.). https://www.microsoft.com/en-us/research/publication/views-compositional-reasoning-for-concurrent-programs/

Thomas Ehrhard. 2018. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science* 28, 7 (2018), 995–1060.

Thomas Ehrhard and Olivier Laurent. 2010. Interpreting a finitary pi-calculus in differential interaction nets. *Inf. Comput.* 208, 6 (2010), 606–633.

Thomas Ehrhard and Laurent Regnier. 2003. The differential lambda-calculus. *Theoretical Computer Science* 309, 1-3 (2003), 1–41.

Thomas Ehrhard and Laurent Regnier. 2006. Differential Interaction Nets. *Theor. Comput. Sci.* 364, 2 (2006), 166–195.

Matthias Felleisen and Daniel P. Friedman. 1989. A Syntactic Theory of Sequential State. *Theor. Comput. Sci.* 69, 3 (1989), 243–287.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types.* Cambridge University Press, USA.

Matthew Hennessy and Robin Milner. 1985. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM* 32, 1 (1985), 137–161.

Joshua S Hodas and Dale Miller. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and computation* 110, 2 (1994), 327–365.

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

W. A. Howard. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley (Eds.). Academic Press, 479–490.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2 (2018), 66:1–66:34.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Wen Kokke, J. Garrett Morris, and Philip Wadler. 2019. Towards Races in Linear Logic. In *Coordination Models and Languages*, Hanne Riis Nielson and Emilio Tuosto (Eds.). Springer International Publishing, Cham, 37–53.

Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. 2012. Superficially Substructural Types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12).* Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/2364527.2364536

Nancy A. Lynch. 1980. Fast Allocation of Nearby Resources in a Distributed System. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing* (Los Angeles, California, USA) *(STOC '80).* Association for Computing Machinery, New York, NY, USA, 70–81. https://doi.org/10.1145/800141.804654

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14).* Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Filipe Militão, Jonathan Aldrich, and Luís Caires. 2016. Composing Interfering Abstract Protocols. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.16

John C Mitchell and Gordon D Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 3 (1988), 470–502.

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911.

Peter W. O'Hearn and John C. Reynolds. 2000. From Algol to polymorphic linear lambda-calculus. *J. ACM* 47, 1 (2000), 167–223.

Chris Okasaki. 1998. *Purely Functional Data Structures.* Cambridge University Press. https://doi.org/10.1017/CBO9780511530104

Michele Pagani and Paolo Tranquilli. 2009. Parallel Reduction in Resource Lambda-Calculus. In *Programming Languages and Systems*, Zhenjiang Hu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 226–242.

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear Logical Relations for Session-Based Concurrency. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 539–558.

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93).* Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

Frank Pfenning. 1995. Structural Cut Elimination. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95).* IEEE Computer Society, USA, 156.

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–22.

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Gordon D. Plotkin. 1976. A Powerdomain Construction. *SIAM J. Comput.* 5, 3 (1976), 452–487.

Zesen Qian, G. A. Kavvos, and Lars Birkedal. 2021. Client-Server Sessions in Linear Logic. 5, ICFP, Article 62 (Aug. 2021). https://doi.org/10.1145/3473567

Pedro Rocha and Luís Caires. 2021a. *A Propositions-as-Types System for Shared State.* Technical Report. NOVA Laboratory for Computer Science and Informatics. http://ctp.di.fct.unl.pt/~lcaires/papers/PTSStech-report2021.pdf

Pedro Rocha and Luís Caires. 2021b. Propositions-as-Types and Shared State (Artifact). (May 2021). https://doi.org/10.5281/zenodo.5037493

Davide Sangiorgi. 1999. The Name Discipline of Uniform Receptiveness. *Theor. Comput. Sci.* 221, 1-2 (1999), 457–493.

Davide Sangiorgi and David Walker. 2001. *PI-Calculus: A Theory of Mobile Processes.* Cambridge University Press, USA.

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-Class State Change in Plaid. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 713–732. https://doi.org/10.1145/2048066.2048122

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent Session Types via Intuitionistic Linear Type Theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming* (Odense, Denmark) *(PPDP '11)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/2003476.2003499

Bernardo Toninho, Luis Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–369.

Bernardo Toninho and Nobuko Yoshida. 2021. On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 7 (June 2021), 55 pages. https://doi.org/10.1145/3457884

Philip Wadler. 1990a. Linear Types can Change the World!. In *Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, 1990*, Manfred Broy (Ed.). North-Holland, 561.

Philip Wadler. 1990b. Recursive types for free.

Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 273–286. https://doi.org/10.1145/2364527.2364568

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.