Safe Session-Based Concurrency with Shared Linear State

Pedro Rocha(✉) and Luís Caires

NOVA LINCS, NOVA University of Lisbon, Portugal
pms.rocha@campus.fct.unl.pt lcaires@fct.unl.pt

We introduce CLASS, a session-typed, higher-order, core language that supports concurrent computation with shared linear state. We believe that CLASS is the first proposal for a foundational language able to flexibly express realistic concurrent programming idioms, with a type system ensuring all the following three key properties: CLASS programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate. CLASS owes these strong properties to a propositions-as-types foundation based on Linear Logic, which we conservatively extend with logically motivated constructs for shareable affine state. We illustrate CLASS expressiveness with several examples involving memory-efficient linked data structures, sharing of resources with linear usage protocols, and sophisticated thread synchronisation, which may be type-checked with a perhaps surprisingly light type annotation burden.

1 Introduction

Stateful programming involving concurrency and shared state plays a prominent role in modern software development, but, in practice, getting concurrent code right is still quite hard for common developers. Typical sources of "bugs" include resource leaks (forgetting to release unused memory or close a socket), violation of resource state preconditions (writing to a closed file or sending out-of-order messages), races (data invariant breaking, erratic sharing of resources), deadlocks (indefinite wait for lock release or incoming messages), livelocks, and even general non-termination. Fifty years ago Hoare noted [40]: "Parallel programs are particularly prone to time-dependent errors, which either cannot be detected by program testing nor by run-time checks. It is therefore very important that a high-level language designed for this purpose should provide complete security against time-dependent errors by means of a *compile-time* check". It does not come as a surprise that finding ways to approximate such certainly very ambitious goal is still today the object of exciting intense research.

In this paper, we approach this challenge by leveraging the propositions-as-types (PaT) paradigm towards the realm of concurrency and shared state. PaT is known to offer a unifying framework connecting logic, computation, and programming languages. Since the seminal work of Curry and Howard [42], it is a prolific structuring concept for designing and reasoning about programming languages (see [82]). Remarkably, languages derived within PaT intrinsically satisfy crucial properties: *type preservation* (since reduction corresponds to cut-reduction), *confluence* (since computation corresponds to proof simplification),

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 421–450, 2023.
https://doi.org/10.1007/978-3-031-30044-8_16

deadlock freedom (as a consequence of cut-elimination) and *livelock freedom / termination* (as a consequence of strong normalisation).

Although PaT has a traditional focus on functional computation, the emergence of linear logic has progressively motivated interpretations of stateful/resourceful computation [78,1,14,2,12], eventually leading to the discovery of tight correspondences between session types and linear logic [22,27,81]. These systems already capture aspects of state change, namely in the sequential execution of session protocols, thus raising the question of whether such approaches could be extended to express notions of shared mutable state, subject to interference, as found in typical imperative and concurrent programs. Recently, such challenge was addressed by several works [9,64,67]. In particular, [67] developed a first basic shared state model enjoying all the aforementioned strong properties of PaT. However, although [67] supports higher-order shareable store for pure values of replicated type, it forbids linear objects, such as stateful processes or data structures with update in-place, to be stored and shared as in languages like Java, Rust, and in the CLASS core language we introduce herein.

In this work, we develop a novel, more fundamental approach to shared state and PaT, and introduce CLASS, a typed, higher-order, session based core language that supports general concurrent computation with dynamically allocated shared linear (more precisely, affine) state. We believe that CLASS is the first proposal for a foundational language. able to flexibly express realistic concurrent programming idioms, while ensuring all the following three key properties by static typing: CLASS programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate.

Despite the strength of its type system, CLASS expressiveness and effectiveness substantially overcomes limitations of related works, as we show with compelling program examples that can be algorithmically typed for memory safety, dead- and live-lock freedom with a perhaps surprisingly light type annotation burden. CLASS owes these strong properties to is PaT foundation based on Second-Order Linear Logic, already known to capture the polymorphic session calculus and the linear System F [74], but which we conservatively extend with novel logically motivated constructs for shareable affine state, also based on DiLL co-exponentials [35,67], but to which we give here a different, more general and fundamental interpretation.

1.1 Overview

A main novelty and source of CLASS's expressiveness, flexibility and strong meta-theoretical properties resides in its mechanism for shared state composition. It is interesting to overview such mechanism in the context of the basic composition and interaction principles of the fundamental linear logic interpretations [22,27,81]. Our computational model is structured around processes that interact via binary sessions, the basic composition rules being mix and cut.

$$\frac{P \vdash \Delta_1; \Gamma \quad Q \vdash \Delta_2; \Gamma}{P \parallel Q \vdash \Delta_1, \Delta_2; \Gamma}\ [\text{Tmix}] \qquad \frac{P \vdash \Delta_1, x : A; \Gamma \quad Q \vdash \Delta_2, x : \overline{A}; \Gamma}{P \mid x \mid Q \vdash \Delta_1, \Delta_2; \Gamma}\ [\text{Tcut}]$$

The mix rule types the independent composition of processes P and Q, which do not share any free names and run side-by-side without interacting. This is captured by the implicit disjointness of their linear typing contexts Δ_1 and Δ_2, declaring the types of their interaction channels. Interactive composition is expressed by the cut rule, which connects exactly two processes P and Q through a *single* linear session x with *dual typed* endpoints ($x : A$ and $x : \overline{A}$), following Abramsky's idea of "cut as interactive composition" [1].

Intuitively, duality of endpoint (session) types ensures that all interactions between P and Q on x always matches: when P sends, Q receives; when Q offers, P chooses; and likewise for all types. Notice that sharing a single channel x between the threads P and Q is important to ensure acyclicity of proof structures, and cut-elimination/deadlock absence. But P, Q may use an arbitrary number of linear channels, in Δ_1, Δ_2, to also compose with other processes.

Shared composition in session types is available for *replicated* "server" objects $!x(y); P$, typed by the linear logic exponential type bang $!A$. Contraction of the dual exponential type why-not $?\overline{A}$ allows an unbounded number of usages of such replicated server object to be introduced in client processes. In the dyadic presentation of linear logic (cf. [5,11]), contraction is expressed by moving ?-typed names into the unrestricted context Γ, with the [T?] rule.

$$\frac{!x(y); P \vdash x : !A; \Gamma \qquad \dfrac{Q \vdash \Delta; \Gamma, x : \overline{A}}{?x; Q \vdash \Delta, x : ?\overline{A}; \Gamma}\ [\mathrm{T?}]}{!x(y); P \mid x \mid ?x; Q \vdash \Delta; \Gamma} \qquad \frac{\dfrac{\vdots}{R \vdash \Delta, y : \overline{A}; \Gamma, x{:}\overline{A}}}{\mathsf{call}\ x(y); R \vdash \Delta; \Gamma, x{:}\overline{A}}\ [\mathrm{Tcall}]$$

Names in Γ may be used unrestrictedly; each call (typed by [Tcall]) spawns a fresh copy of the server body at type $y : A$, to be used by the client at type $y : \overline{A}$, in a linear binary session. By the typing rule for !A (promotion) such copy does not depend on linear resources. Thus, interaction with replicated objects as captured by the exponentials $!A$ and $?A$ implements a copy semantics where each call obtains a new private *stateless* copy of the same object.

In this work, we introduce a third composition mechanism, allowing processes to concurrently share mutex memory cells, storing *linear state*. Mutex memory cells and their usages are typed respectively by a pair of dual modalities $\mathsf{S}_\bullet A$ and $\mathsf{U}_\bullet A$, whose logical rules are motivated by Differential Linear Logic (DiLL) [35], in particular *cocontraction*, expressed by the type rule [Tsh].

$$\frac{P \vdash \Delta, x : \mathsf{U}_\bullet A; \Gamma \qquad Q \vdash \Delta', x : \mathsf{U}_\bullet A; \Gamma}{\mathsf{share}\ x\ \{P \parallel Q\} \vdash \Delta, \Delta', x : \mathsf{U}_\bullet A; \Gamma}\ [\mathrm{Tsh}]$$

While sharing of replicated objects corresponds to contraction of $?A$ types, shared usage of mutex cells corresponds to cocontraction of $\mathsf{U}_\bullet A$ types. Apart from the explicit use of [Tsh], the type system ensures that memory cells are always used linearly. The shared usage $x : \mathsf{U}_\bullet A$ *is free* in the conclusion of the typing rule, therefore a memory cell may be shared by an arbitrary number of processes, by nested iterated use of cocontraction.

Moreover, cocontraction also ensures that concurrent processes may share a single mutex cell (just like [Tcut] w.r.t. binary sessions). This constraint comes from the linear logic discipline, and it is important to ensure deadlock freedom. As discussed in Concluding Remarks, this does not hinder CLASS expressiveness - e.g., a single mutex cell may act as a gateway to further bundles of shared state, organised in resource hierarchies, as our examples illustrate - and even suggests convenient concurrent programming structuring techniques.

To access a mutex memory cell in its (unlocked) full state, typed by $\mathsf{U}_{\bullet}A$, the client uses a *take* operation. Take waits for acquiring the cell lock and reads its contents. The cell then transitions to the (locked) empty state, typed by $\mathsf{U}_{\circ}A$. The taking client becomes the sole responsible for filling back the cell contents, using a *put* operation. This will restore the cell to the full state, releasing its lock, and making it accessible to other concurrent threads waiting to take it. Our mutex memory cell object is thus akin to a behaviourally typed incarnation of Concurrent Haskell MVars [45] or Rust std::sync::Mutex objects [46].

To ensure safe releasing of a memory cell, its contents are required to be of affine type $\wedge A$. Affine objects are well-behaved disposable values, that when discarded, safely dispose all resources they hereditarily refer to, this being ensured by the linear logic typing.

We illustrate the introduced concepts with a simple example, where two concurrent threads compete to set *on* an initially *off* flag, but only one may win. The flag iteratively announces its state to the client with either #Off or #On. If the state is *off*, the client must select #turnOn, if the state is *on*, it will remain *on*. Process $\mathsf{flag}(f)$ implements the flag (at name f) in the *off* state, and process $\mathsf{on}(f)$ in the *on* state, defined thus

$$\begin{aligned}\mathsf{flag}(f) &= \#\mathsf{Off}\ f; \mathsf{case}\ f\{\ |\ \#\mathsf{turnOn} : \mathsf{affine}\ f; \mathsf{on}(f)\ \}\\ \mathsf{on}(f) &= \#\mathsf{On}\ f; \mathsf{affine}\ f; \mathsf{on}(f)\end{aligned}$$

The flag object is typed with the (linear) usage protocol defined by the coinductive type Flag below, such that $\mathsf{flag}(f) \vdash f : \mathsf{Flag}$ and $\mathsf{on}(f) \vdash f : \mathsf{Flag}$

$$\mathsf{type\ corec\ Flag} = \oplus\{\ |\#\mathsf{Off} : \&\{\ |\#\mathsf{turnOn} : \wedge\mathsf{Flag}\},\ |\#\mathsf{On} : \wedge\mathsf{Flag}\}$$

We now consider a scenario where a flag object is shared via a mutex memory cell c initially storing a *off* flag of type $\wedge$Flag among two concurrent clients.

```
client(c, id) ⊢ c : U•Flag; id : int
client(c, id) =
  take c(f);
  case f {
    |#Off : println id + ": wins.';
            #turnOn f;
            put c(f); release c
    |#On : println id + ": loses.';
           put c(f); release c
  }

main() ⊢ ∅
main() =
  cut { cell c(f.affine f; flag(f))
        |c : U•Flag|
        share c {
          client(c, 1)
          ||
          client(c, 2)
        }
      }
```

When running main() exactly one of the threads (executing the same code, just with a different id) will turn the flag *on* and win, the other will loose. Notice that all threads drop usage of the memory cell c using release, which corresponds to DiLL coweakening ([35]).

When considering a new language, in particular with a static typing discipline, it is necessary to argue about its expressiveness, and aim for a better perception of how natural programs get past its typing rules, and of how types help in structuring programs. In this paper, we approach these concerns by showcasing many interesting examples that challenge the expressiveness of the CLASS language and type system on realistic concurrent programming scenarios. We have developed many more examples, distributed with our implementation [68], combining imperative, higher-order functional, and session-based programming styles. For all these programs, strong guarantees of memory safety, deadlock-freedom, termination, and absence of "dynamic bugs", even in the presence of blocking primitives and higher-order state, are compositionally certified by our lightweight type discipline based on Propositions-as-Types and Linear Logic.

1.2 Outline and Contributions

We believe that CLASS is the first proposal for a foundational language able to flexibly express realistic concurrent programming idioms while ensuring by typing three key properties: CLASS programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate.

In Section 2 we formally present the core language CLASS, its type system and operational semantics. Our model builds on the propositions-as-types approach to session-based concurrency [22,27,80], extending Second-Order Classical Linear Logic with inductive/coinductive types, affine types, and novel primitives for shareable first-class mutex reference cells for linear state.

In Section 3 we state and prove type preservation (Theorem 1), progress (Theorem 2) which implies deadlock-freedom, and strong normalisation (Theorem 3), which also implies livelock absence. Our proof uses a logical relations argument, extended with an interesting technique to handle shared state interference, which we believe is exploited here for the first time.

Given the strong properties of its type system, it is of course very important to substantiate our claims about CLASS expressiveness. In Section 4 we illustrate the expressiveness of CLASS language and type system by going through a series of compelling examples. Namely, we discuss a general technique for sharing linear protocols, a shareable linked list with update in-place, a shareable buffered channel, using a linked list with pointers to tail and head nodes, and executing send and receive operations in O(1) time; the dining philosophers, illustrating techniques that rely on our type structure to encode resource acquisition hierarchies; a generic barrier for n threads; and a Hoare style monitor with await/notify conditions, where our implementation of the condition's process queue is supported by a dynamic linked data structure, as in real systems code.

Section 5 discusses related work. Section 6 offers concluding remarks and suggests further research. Complete definitions and detailed proofs to all results are provided in [65].

2 The Core Language and its Type System

We present the core language, type system, and operational semantics of CLASS. The language is based on a PaT correspondence with Linear Logic, so terms of the language correspond to proof rules. We start by types and duality.

Definition 1 (Types). *Types A, B of* CLASS *are defined by*

$$\begin{array}{lllllll} A,B ::= X & \mid \mathbf{1} & \mid \bot & \mid A \,\&\, B & \mid A \oplus B & \mid A \parr B & \mid A \otimes B \\ & \mid\, !A & \mid\, ?A & \mid \exists X.A \mid & \mid \forall X.A & \mid \mu X.\ A & \mid \nu X.\ A \\ & \mid \wedge A & \mid \vee A & \mid \mathsf{S}_\bullet A & \mid \mathsf{S}_\circ A & \mid \mathsf{U}_\bullet A & \mid \mathsf{U}_\circ A \end{array}$$

Types in the first two rows correspond to Second-Order Classical Linear Logic, extended with inductive/coinductive types (μ, ν). Types comprise variables (X), units ($\mathbf{1}$, $\bot$), multiplicatives ($\otimes$, $\parr$), additives ($\oplus$, $\&$), exponentials (!, ?) and quantifiers ($\exists$, $\forall$). The third row extends basic types with affine ($\wedge, \vee$) and new modalities ($\mathsf{S}_\bullet, \mathsf{U}_\bullet, \mathsf{S}_\circ, \mathsf{U}_\circ$) to type shared *affine state*. Duality is the involution operation $A \mapsto \overline{A}$ on types, corresponding to Linear Logic negation, defined by

$$\begin{array}{lll} \overline{\mathbf{1}} = \bot & \overline{A \otimes B} = \overline{A} \parr \overline{B} & \overline{A \oplus B} = \overline{A} \,\&\, \overline{B} \\ \overline{!A} = ?\overline{B} & \overline{\exists X.A} = \forall X.\overline{A} & \overline{\mu X.\ A} = \nu X.\ \{\overline{X}/X\}(\overline{A}) \\ \overline{\wedge A} = \vee \overline{A} & \overline{\mathsf{S}_\bullet A} = \mathsf{U}_\bullet \overline{A} & \overline{\mathsf{S}_\circ A} = \mathsf{U}_\circ \overline{A} \end{array}$$

Duality captures symmetry in process interaction, as manifest in the cut rule. In our system, typing judgements have the form $P \vdash_\eta \Delta; \Gamma$. The typing context $\Delta; \Gamma$ is dyadic [4,15,63,22], where Δ is handled linearly and Γ is unrestricted; both Δ and Γ assign types to names. The index η is a finite map that holds coinduction hypothesis to type corecursive processes, as detailed later.

Definition 2. *The typing rules of* CLASS *are presented in Figs. 1 to 5.*

The type system corresponds, via propositions-as-types [22,27,80], to Second-Order Classical Linear Logic (Fig. 1) with inductive/coinductive types (Fig. 2), affinity (Fig. 3) and extended with constructs for shared mutable state (Figs. 4 - 5). The basic composition rules are [Tmix] and [Tcut], which correspond to mix and cut of Linear Logic, respectively. [Tmix] types a parallel composition $P \parallel Q$, where P and Q run in parallel without interfering. On the other hand, [Tcut] types linear interactive composition $P \,|x : A|\, Q$: processes P and Q run concurrently and communicate through a private linear session x, session endpoints being typed by dual types $A/\overline{A}$. When the cut type annotation does not play any role, we may omit it and write $P \,|x|\, Q$. In examples, for readability, we use cut $\{P \,|x|\, Q\}$ and par $\{P \parallel Q\}$ instead of $P \,|x|\, Q$ and $P \parallel Q$, respectively.

For the basic process constructs [22,27,80,19], $\otimes/\parr$ type send and receive, $\oplus/\&$ type choice and offer (in examples we use labelled choice) , !/? type

$$\frac{}{0 \vdash_\eta \emptyset; \Gamma}\ [\mathrm{T0}] \qquad \frac{P \vdash_\eta \Delta'; \Gamma \quad Q \vdash_\eta \Delta; \Gamma}{P \parallel Q \ \vdash_\eta \Delta', \Delta; \Gamma}\ [\mathrm{Tmix}]$$

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

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

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

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

$$\frac{P_1 \vdash_\eta \Delta_1, y : A; \Gamma \quad P_2 \vdash_\eta \Delta_2, x : B; \Gamma}{\mathsf{send}\ x(y.P_1); P_2 \ \vdash_\eta \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\ [\mathrm{T}\otimes]$$

$$\frac{Q \vdash_\eta \Delta, z : A, x : B; \Gamma}{\mathsf{recv}\ x(z); Q \vdash_\eta \Delta, x : A \parr B; \Gamma}\ [\mathrm{T}\parr]$$

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

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

$$\frac{P \vdash_\eta \Delta, x : \{B/X\}A; \Gamma}{\mathsf{sendty}\ x(B); P \ \vdash_\eta \Delta, x : \exists X.A; \Gamma}\ [\mathrm{T}\exists] \qquad \frac{Q \vdash_\eta \Delta, x : A; \Gamma}{\mathsf{recvty}\ x(X); Q \vdash_\eta \Delta, x : \forall X.A; \Gamma}\ [\mathrm{T}\forall]$$

Fig. 1: Typing Rules I: Second-Order CLL.

$$\frac{P \vdash_{\eta'} \Delta, z : A; \Gamma \quad \eta' = \eta, X(z, \boldsymbol{w}) \mapsto \Delta, z : Y; \Gamma}{\mathsf{corec}\ X(z, \boldsymbol{w}); P\ [x, \boldsymbol{y}] \vdash_\eta \{\boldsymbol{y}/\boldsymbol{w}\}\Delta, x : \nu Y.\ A; \{\boldsymbol{y}/\boldsymbol{w}\}\Gamma}\ [\mathrm{Tcorec}]$$

$$\frac{\eta = \eta', X(x, \boldsymbol{y}) \ \mapsto \Delta, x : Y; \Gamma}{X(z, \boldsymbol{w}) \vdash_\eta \{\boldsymbol{w}/\boldsymbol{y}\}\Delta, z : Y; \{\boldsymbol{w}/\boldsymbol{y}\}\Gamma}\ [\mathrm{Tvar}]$$

$$\frac{P \vdash_\eta \Delta, x : \{\mu X.\ A/X\}A; \Gamma}{\mathsf{unfold}_\mu\ x; P \vdash_\eta \Delta, x : \mu X.\ A; \Gamma}\ [\mathrm{T}\mu] \qquad \frac{P \vdash_\eta \Delta, x : \{\nu X.\ A/X\}A; \Gamma}{\mathsf{unfold}_\nu\ x; P \vdash_\eta \Delta, x : \nu X.\ A; \Gamma}\ [\mathrm{T}\nu]$$

Fig. 2: Typing Rules II: Induction and Coinduction.

$$\frac{P \vdash_\eta a : A, \boldsymbol{b} : \vee \boldsymbol{B}, \boldsymbol{c} : \mathsf{U}_\bullet \boldsymbol{C}; \Gamma}{\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P \ \vdash_\eta a : \wedge A, \boldsymbol{b} : \vee \boldsymbol{B}, \boldsymbol{c} : \mathsf{U}_\bullet \boldsymbol{C}; \Gamma}\ [\text{Taffine}]$$

$$\frac{}{\mathsf{discard}\ a \vdash_\eta a : \vee A; \Gamma}\ [\text{Tdiscard}] \qquad \frac{Q \vdash_\eta \Delta, a : A; \Gamma}{\mathsf{use}\ a; Q \vdash_\eta \Delta, a : \vee A; \Gamma}\ [\text{Tuse}]$$

Fig. 3: Typing Rules III: Affinity.

replicated servers and their invocation, $\forall/\exists$ type receive and send of types, implementing polymorphic processes.

Coinductive types are introduced by rule [Tcorec]. It types corecursive processes $\mathsf{corec}\ X(z, \boldsymbol{w}); P\ [x, \boldsymbol{y}]$, with parameters $z, \boldsymbol{w}$ bound in P, that are instantiated with the arguments $x, \boldsymbol{y}$ (free in the process term). By convention, the coinductive behaviour, of type $\nu Y.\ A$, of a corecursive process is always offered in the first argument z. According to [Tcorec], to type the body P of a corecursive process, the map η is extended with a coinductive hypothesis binding the process variable X to the typing context $\Delta, z : Y; \Gamma$, so that when typing the body P of the corecursion we can appeal to X, which intuitively stands for P itself, and recover its typing invariant. Crucially, the type variable Y is free only in $z : A$. This causes corecursive calls to be always applied to names z' that hereditarily descend from the initial corecursive argument z, a necessary condition for strong normalisation (Theorem 3), and morally corresponds to only allowing corecursive calls on "smaller" argument sessions (of inductive type).

Rule [Tvar] types a corecursive call $X(z, \boldsymbol{w})$ by looking up in η for the corresponding binding and renaming the parameters with the arguments of the call. Inductive and coinductive types are explicitly unfolded with [Tμ] and [Tν].

To simplify the presentation in program examples, we omit explicit unfolding actions, and write inductive and coinductive type definitions with equations of the form $\mathsf{rec}\ A = f(A)$ and $\mathsf{corec}\ B = f(B)$ instead of $A = \mu X.\ f(X)$ and $B = \nu X.\ f(X)$, respectively. Similarly, we write corecursive process definitions as $Q(x, \boldsymbol{y}) = f(Q(-))$ instead of $Q(x, \boldsymbol{y}) = \mathsf{corec}\ X(z, \boldsymbol{w}); f(X(-))\ [x, \boldsymbol{y}]$, while of course respecting the constraints imposed by typing rules [Tvar] and [Tcorec].

Affinity Affinity is important to model discardable linear resources, and plays an important role in CLASS. An affine session can either be used as a linear session or discarded. The typing rules for the affine modalities are in Fig. 3. Affine sessions are introduced by rule [Taffine] that promotes a linear $a : A$ to an affine session $a : \wedge A$. It types $\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P$, which provides an affine session at a and continues as P, and follows the structure of a standard promotion rule.

A session a may be promoted to affine if it only depends on resources that can be disposed, i.e. resources that satisfy some form of weakening capability, namely: coaffine sessions b_i of type $\vee B_i$, that can be discarded; full cell usages c_i of type with $\mathsf{U}_\bullet C_i$, that can be released; and unrestricted sessions in Γ, which are implicitly ?-typed. The dependencies of an affine object on coaffine or full

$$\frac{P \vdash_\eta \Delta, a : \wedge A; \Gamma}{\mathsf{cell}\ c(a.P) \vdash_\eta \Delta, c : \mathsf{S}_\bullet A; \Gamma}\ [\text{Tcell}] \qquad \frac{}{\mathsf{release}\ c \vdash_\eta c : \mathsf{U}_\bullet A; \Gamma}\ [\text{Trelease}]$$

$$\frac{}{\mathsf{empty}\ c \vdash_\eta c : \mathsf{S}_\circ A; \Gamma}\ [\text{Tempty}] \qquad \frac{Q \vdash_\eta \Delta, a : \vee A, c : \mathsf{U}_\circ A; \Gamma}{\mathsf{take}\ c(a); Q \vdash_\eta \Delta, c : \mathsf{U}_\bullet A; \Gamma}\ [\text{Ttake}]$$

$$\frac{Q_1 \vdash_\eta \Delta_1, a : \wedge \overline{A}; \Gamma \qquad Q_2 \vdash_\eta \Delta_2, c : \mathsf{U}_\bullet A; \Gamma}{\mathsf{put}\ c(a.Q_1); Q_2 \ \vdash_\eta \Delta_1, \Delta_2, c : \mathsf{U}_\circ A; \Gamma}\ [\text{Tput}]$$

Fig. 4: Typing Rules IV: Reference Cells.

cell objects are explicitly annotated as $\boldsymbol{b}, \boldsymbol{c}$ in the process term, to instrument the operational semantics, but we often omit them and simply write $\mathsf{affine}\ a; P$.

The coaffine endpoint $\vee A$ of an affine session, dual of $\wedge\overline{A}$, has two operations: use and discard. Rule [Tuse] types a process $\mathsf{use}\ a; Q$ that uses a coaffine session a and continues as Q, it is a dereliction rule. [Tdiscard] types the process $\mathsf{discard}\ a$ that discards a coaffine session a, it is a weakening rule.

Shared Mutable State Shared state is introduced in CLASS by typed constructs that model mutex memory cells, and associated cell operations allowing its use by client code, defined by the tying rules in Fig. 4.

At any moment a cell may be either *full* or *empty*, akin to the MVars of Concurrent Haskell [45]. A full cell on c, written $\mathsf{cell}\ c(a.P)$, is typed $\mathsf{S}_\bullet A$ by rule [Tcell]. Such cell stores an *affine* session of type $\wedge A$, implemented at a by P. All objects stored in cells are required to be affine, so that memory cells may always be safely disposed without causing memory leaks. An empty cell on c, of type $\mathsf{S}_\circ A$, and written $\mathsf{empty}\ c$, is typed by rule [Tempty].

Client processes manipulate cells via *take*, *put* and *release* operations. These operations apply to names of cell usage types - $\mathsf{U}_\bullet A$ (full cell usage) and $\mathsf{U}_\circ A$ (empty cell usage) - which are dual types of $\mathsf{S}_\bullet\overline{A}$ and $\mathsf{S}_\circ\overline{A}$, respectively. At any given moment, a client thread owning a $\mathsf{U}_\bullet A$-typed usage to a cell may execute a *take* operation, typed by rule [Ttake]. The *take* operation $\mathsf{take}\ c(a); Q$ waits to acquire the cell mutex c, and reads its contents into parameter a, the linear (actually coaffine, of type $\vee A$) usage for the object stored in the cell; the cell becomes empty, and execution continues as Q.

It is responsibility of the taking thread to put some value back in the empty cell, thus releasing the lock, causing the cell to transition to the full state. The *put* operation $\mathsf{put}\ c(a.Q_1); Q_2$ is typed by [Tput], the stored object a, implemented by Q_1, is required to be affine, as specified in the premise $a : \wedge\overline{A}$.

Hence a cell flips from full to empty and back; [Ttake] uses the cell c at $\mathsf{U}_\bullet A$ type, and its continuation (in the premise) at $\mathsf{U}_\circ A$ type, symmetrically [Tput] uses the cell c at $\mathsf{U}_\circ A$ type, and its continuation (in the premise) at $\mathsf{U}_\bullet A$ type.

The $\mathsf{release}\ c$ operation allows a thread to manifestly drop its cell usage c. Release is typed by [Trelease] (cf. coweakening [35]); a usage may only be released

$$\frac{P \vdash_\eta \Delta', c : \mathsf{U}_\bullet A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathsf{U}_\bullet A; \Gamma}{\mathsf{share}\ c\ \{P \parallel Q\} \vdash_\eta \Delta', \Delta, c : \mathsf{U}_\bullet A; \Gamma}\ [\text{Tsh}]$$

$$\frac{P \vdash_\eta \Delta', c : \mathsf{U}_\circ A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathsf{U}_\bullet A; \Gamma}{\mathsf{share}\ c\ \{P \parallel Q\} \vdash_\eta \Delta', \Delta, c : \mathsf{U}_\circ A; \Gamma}\ [\text{TshL}]$$

$$\frac{P \vdash_\eta \Delta', c : \mathsf{U}_\bullet A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathsf{U}_\circ A; \Gamma}{\mathsf{share}\ c\ \{P \parallel Q\} \vdash_\eta \Delta', \Delta, c : \mathsf{U}_\circ A; \Gamma}\ [\text{TshR}]$$

Fig. 5: Typing Rules V: State Sharing.

in the unlocked state $\mathsf{U}_\bullet A$. When, for some cell c, all the owning threads release their usages, which eventually happens in well-typed programs, the cell c gets disposed, and its (affine) contents safely discarded.

Our memory cells cells are linear objects, with a linear mutable payload, which are never duplicated by reduction or conversion rules. However, in CLASS, multiple cell usages may be shared between concurrent threads, which compete to take and use it in interleaved critical sections. Such aliased usages be passed around and duplicated dynamically, changing the sharing topology at runtime.

Sharing of cell usages is logically expressed in our system by the typing rules in Fig. 5. Co-contraction, introduced in Differential Linear Logic DiLL [35], allows finite multisets of linear resources to safely interact in cut-reduction, resolving concurrent sharing into nondeterminism, as required here to soundly model memory cells and their linear concurrent usages. Rule [Tsh] interprets cocontraction with the construct $\mathsf{share}\ c\ \{P \parallel Q\}$, and types sharing of the cell usage $c : \mathsf{U}_\bullet A$ between the concurrent threads P and Q.

Contrary to cut, $\mathsf{share}\ c\ \{P \parallel Q\}$ is *not* a binding operator for c. The shared usage $c : \mathsf{U}_\bullet A$ is *free* in the conclusion of the typing rule, permitting c to be shared among an arbitrary number of threads, by nested iterated use of [Tsh]. In [Tsh], P and Q only share the single mutex cell c, since the linear context is split multiplicatively, just like [Tcut] wrt. binary sessions. This condition comes from the DiLL typing discipline, and is important to ensure deadlock freedom.

While [Tsh] types sharing of a full (unlocked) cell usage of type $\mathsf{U}_\bullet A$, the symmetric rules [TshR] and [TshR] type sharing of an empty (locked) cell usage of type $\mathsf{U}_\circ A$. We may verify that for every cell c in a well-typed process, at most one unguarded operation to c may be using type $\mathsf{U}_\circ A$, all the remaining unguarded operations to c must be using type $\mathsf{U}_\bullet A$. This implies that, at runtime, only one thread may own the lock for a given (necessarily empty) cell, and execute a *put* to it, which will bring the cell back to full and release its lock, other threads must be either attempting to *take*, or *release* the reference.

Working together, the sharing typing rules ensure that in any well-typed cell sharing tree, at most one single thread at any time may be actively using a cell (in the locked empty state) and put to it, thus guaranteeing mutual exclusion,

while satisfying Progress (Theorem 2) which in turn ensures deadlock absence, even in the presence of the crucially blocking behaviour of the take operation.

2.1 Operational Semantics

We now define CLASS operational semantics, which is given by a structural precongruence relation $\leq$ that captures static relations on processes, essentially rearranging them, and a reduction relation $\rightarrow$ that captures process interaction.

Definition 3 ($P \equiv Q$ and $P \leq Q$). *Structural congruence $\equiv$ is the least congruence on processes closed under α-conversion and the $\equiv$-rules in Fig. 6. Structural precongruence $\leq$ is the least precongruence on processes including $\equiv$ and closed under α-conversion and the $\leq$-rules in Fig. 6.*

The basic rules of $\equiv$ essentially reflect the expected static laws, along the lines of the structural congruences / conversions in [22,80]. The binary operators forwarder, cut and share are commutative ([comm]). The set of processes modulo $\equiv$ is a commutative monoid with binary operation given by parallel composition and identity given by inaction 0 ([par]). Any two static constructs commute, as expressed by the laws [CM]-[ShC!]. Furthermore, we can distribute the unrestricted cut over all the static constructs as expressed by law [D-C!X], where $*$ stands for either a mix, linear or unrestricted cut or a share.

The commuting conversions [ShTake] and [ShPut] allows take and put operations on cell usages to commute with a share construct. Rule [ShTake] picks the take that occurs on the left argument, however since share is commutative, a right-biased version of [ShTake] is admissible. Using [ShTake], any of the two possible interleavings for two concurrent takes may be nondeterministically picked via $\leq$. Indeed, we express $\leq$ as a precongruence because it introduces nondeterminism, and does not express a behavioural equivalence as $\equiv$ does. N.B.: Although one could easily formulate a confluent version of CLASS semantics, using explicit sums as in [13,66,35,65], we prefer in this paper to focus on the expressiveness of CLASS as a programming language and on its deadlock and livelock absence properties, adopting a nondeterministic reduction relation.

In [ShPut] only a put, in the $\mathsf{U}_\circ A$-typed premise of [TshL], may be propagated up and eventually update the cell, causing it to transit back to the full state. Hence, take operations originating the $\mathsf{U}_\bullet A$ typed premise of [TshR] will be blocked, waiting until such (unique) put propagation occurs. Algebraically, rule [ShRel] expresses that the release operation is the identity for share composition, we orient it as a precongruence, to ensure type preservation.

Definition 4 (Reduction $\rightarrow$). *Reduction $\rightarrow$ is defined by the rules of Fig. 7.*

We let $\stackrel{*}{\rightarrow}$ stand for the reflexive-transitive closure of $\rightarrow$. Reduction includes the set of principal cut conversions, i.e. the redexes for each pair of interacting constructs. It is closed by structural precongruence ([$\leq$]) and in rule [cong] we consider that $\mathcal{C}$ is a static context, i.e. a process context in which the hole is covered only by the static constructs mix, cut and share.

$$\mathsf{fwd}\ x\ y \equiv \mathsf{fwd}\ y\ x \quad P \mid x \mid Q \equiv Q \mid x \mid P$$

$$\mathsf{share}\ x\ \{P \parallel Q\} \equiv \mathsf{share}\ x\ \{Q \parallel P\} \qquad [\text{comm}]$$

$$P \parallel 0 \equiv P \quad P \parallel Q \equiv Q \parallel P \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \qquad [\text{par}]$$

$$P \mid x \mid (Q \parallel R) \equiv (P \mid x \mid Q) \parallel R \qquad [\text{CM}]$$

$$P \mid x \mid (Q \mid y \mid R) \equiv (P \mid x \mid Q) \mid y \mid R \qquad [\text{CC}]$$

$$P \mid x \mid \mathsf{share}\ y\ \{Q \parallel R\} \equiv \mathsf{share}\ y\ \{P \mid x \mid Q \parallel R\} \qquad [\text{CSh}]$$

$$P \mid z \mid (y.Q \mid !x \mid R) \equiv y.Q \mid !x \mid (P \mid z \mid R) \qquad [\text{CC!}]$$

$$y.Q \mid !x \mid (P \parallel R) \equiv P \parallel (y.Q \mid !x \mid R) \qquad [\text{C!M}]$$

$$y.P \mid !x : A \mid (w.Q \mid !z : B \mid R) \equiv w.Q \mid !z : B \mid (y.P \mid !x : A \mid R) \qquad [\text{C!C!}]$$

$$\mathsf{share}\ x\ \{P \parallel (Q \parallel R)\} \equiv \mathsf{share}\ x\ \{P \parallel Q\} \parallel R \qquad [\text{ShM}]$$

$$\mathsf{share}\ x\ \{P \parallel \mathsf{share}\ y\ \{Q \parallel R\}\} \equiv \mathsf{share}\ y\ \{\mathsf{share}\ x\ \{P \parallel Q\} \parallel R\} \qquad [\text{ShSh}]$$

$$\mathsf{share}\ z\ \{P \parallel y.Q \mid !x \mid R\} \equiv y.Q \mid !x \mid \mathsf{share}\ z\ \{P \parallel R\} \qquad [\text{ShC!}]$$

$$y.P \mid !x : A \mid (Q * R) \equiv (y.P \mid !x : A \mid Q) * (y.P \mid !x : A \mid R) \qquad [\text{D-C!X}]$$

$$\mathsf{share}\ x\ \{\mathsf{release}\ x \parallel P\} \leq P \qquad [\text{ShRel}]$$

$$\mathsf{share}\ x\ \{\mathsf{put}\ x(y.P); Q \parallel R\} \leq \mathsf{put}\ x(y.P); \mathsf{share}\ x\ \{Q \parallel R\} \qquad [\text{ShPut}]$$

$$\mathsf{share}\ x\ \{\mathsf{take}\ x(y_1); P_1 \parallel \mathsf{take}\ x(y_2); P_2\} \leq \mathsf{take}\ x(y_1); \mathsf{share}\ x\ \{P_1 \parallel \mathsf{take}\ x(y_2); P_2\} \qquad [\text{ShTake}]$$

Provisos: in [CM] and [ShM], $x \in \mathsf{fn}(Q)$; in [CC], [CSh] and [ShSh], $x, y \in \mathsf{fn}(Q)$; in [CC!], [C!M] and [ShC!], $x \notin \mathsf{fn}(P)$; in [C!C!], $x \notin \mathsf{fn}(Q)$ and $z \notin \mathsf{fn}(P)$.

Fig. 6: Structural congruence $P \equiv Q$ and precongruence $P \leq Q$.

Operationally, the forwarding behaviour is implemented by name substitution [23] ([fwd]). All the other conversions apply to a principal cut between two dual actions. Reduction rules for the basic session constructs that interpret Second Order Linear Logic and recursion are the expected ones [22,27,81], along predictable lines. For readability, we omit the type declarations in the cuts, as they do not actually play any role in reduction.

We comment the rules concerning affinity. The interaction between an affine session and an use operation is defined by reduction rule [∧∨u], where a cut on $a : \wedge A$ between $\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P$ and $\mathsf{use}\ a; Q$ reduces to a cut on $a : A$ between the continuations P and Q. The reduction between an affine session and a discard operation is defined by [∧∨d]. A cut between $\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P$ and $\mathsf{discard}\ a$ reduces to a mix-composition of discards (for the coaffine sessions $\boldsymbol{b}$) and releases (for the cell usages $\boldsymbol{c}$) cf. [6,20]). In the corner case where $\boldsymbol{c}$ and $\boldsymbol{a}$ are empty, the left-hand side of [∧∨d] simply degenerates to inaction 0 (the identity of mix).

The reductions for the mutable state operations are fairly self-explanatory. In rule [$\mathsf{S}_\bullet\mathsf{U}_\bullet$r], a cut between a full mutex cell cell and a release operation reduces to a process that discards the affine cell contents, cf. rule [∧∨d]. In rule [$\mathsf{S}_\bullet\mathsf{U}_\bullet$t], a cut on $c : \mathsf{S}_\bullet A$ between a full cell and a take operation reduces to a process with

$$\mathsf{fwd}\ x\ y\ |y|\ P \to \{x/y\}P \qquad [\mathrm{fwd}]$$
$$\mathsf{close}\ x\ |x|\ \mathsf{wait}\ x; P \to P \qquad [\mathbf{1}\bot]$$
$$\mathsf{send}\ x(y.P); Q\ |x|\ \mathsf{recv}\ x(z); R \to Q\ |x|\ (P\ |y|\ \{y/z\}R) \qquad [\otimes\parr]$$
$$\mathsf{case}\ x\ \{|\mathsf{inl} : P,\ |\mathsf{inr} : Q\}\ |x|\ x.\mathsf{inl}; R \to P\ |x|\ R \qquad [\&\oplus_l]$$
$$\mathsf{case}\ x\ \{|\mathsf{inl} : P,\ |\mathsf{inr} : Q\}\ |x|\ x.\mathsf{inr}; R \to Q\ |x|\ R \qquad [\&\oplus_r]$$
$$!x(y); P\ |x|\ ?x; Q \to y.P\ |!x|\ Q \qquad [!?]$$
$$y.P\ |!x|\ \mathsf{call}\ x(z); Q \to \{z/y\}P\ |z|\ (y.P\ |!x|\ Q) \qquad [\mathrm{call}]$$
$$\mathsf{sendty}\ x(A); P\ |x|\ \mathsf{recvty}\ x(X); Q \to P\ |x|\ \{A/X\}Q \qquad [\exists\forall]$$
$$\mathsf{unfold}_\mu\ x; P\ |x|\ \mathsf{unfold}_\nu\ x; Q \to P\ |x|\ Q \qquad [\mu\nu]$$
$$\mathsf{unfold}_\mu\ x; P\ |x|\ \mathsf{corec}\ Y(z, \boldsymbol{w}); Q\ [x, \boldsymbol{y}] \to P\ |x|\ \{x/z\}\{\boldsymbol{y}/\boldsymbol{w}\}\{\mathsf{corec}\ Y(z, \boldsymbol{w}); Q/Y\}Q \qquad [\mathrm{corec}]$$
$$\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P\ |a|\ \mathsf{use}\ a; Q \to P\ |a|\ Q \qquad [\wedge\vee\mathrm{u}]$$
$$\mathsf{affine}_{\boldsymbol{b},\boldsymbol{c}}\ a; P\ |a|\ \mathsf{discard}\ a \to \mathsf{discard}\ \boldsymbol{b}\ ||\ \mathsf{release}\ \boldsymbol{c} \qquad [\wedge\vee\mathrm{d}]$$
$$\mathsf{cell}\ c(a.P)\ |c|\ \mathsf{release}\ c \to P\ |a|\ \mathsf{discard}\ a \qquad [\mathsf{S}_\bullet\mathsf{U}_\bullet\mathrm{r}]$$
$$\mathsf{cell}\ c(a.P)\ |c|\ \mathsf{take}\ c(a'); Q \to P\ |a|\ (\mathsf{empty}\ c\ |c|\ \{a/a'\}Q) \qquad [\mathsf{S}_\bullet\mathsf{U}_\bullet\mathrm{t}]$$
$$\mathsf{empty}\ c\ |c|\ \mathsf{put}\ c(a.P); Q \to \mathsf{cell}\ c(a.P)\ |c|\ Q \qquad [\mathsf{S}_\circ\mathsf{U}_\circ]$$
$$P \le P' \text{ and } P' \to Q' \text{ and } Q' \le Q\ \supset\ P \to Q \qquad [\le]$$
$$P \to Q\ \supset\ \mathcal{C}[P]\ \to \mathcal{C}[Q] \qquad [\mathrm{cong}]$$

Fig. 7: Reduction $P \to Q$.

two cuts, both composed with the continuation $\{a/a'\}Q$ of the take. The outer cut on $a : \wedge A$ composes with the stored affine session, which was successfully acquired by the take operation. The inner cut on $c : \mathsf{S}_\circ A$ composes with the reference cell c, which has became empty in the reductum. Finally, in rule $[\mathsf{S}_\circ\mathsf{U}_\circ]$, a cut on session $c : \mathsf{S}_\circ A$ between an empty cell and a put operation reduces to a cut on session $c : \mathsf{S}_\bullet A$ between a full cell, that now stores the session that was put, and the continuation of the put process. Notice that the locking/unlocking behaviour of cells is simply modelled by rewriting of the process terms, from cell to empty and back, as typical in process calculi.

3 Type Safety and Strong Normalisation

In this section we state and give proof sketches for our main results of type safety and strong normalisation. Full proofs may be found in [65].

Type Preservation The semantics of CLASS is defined by a set of precongruence $\le$ and reduction $\to$ rules on process terms. Theorem 1 shows that these relations preserve typing, and gives substance to our PaT approach, showing that every $\le$ and $\to$ rule corresponds to a conversion on type derivations/proofs.

Theorem 1 (Type Preservation). *Suppose* $P \vdash_\eta \Delta; \Gamma$. *(1) If* $P \leq Q$, *then* $Q \vdash_\eta \Delta; \Gamma$. *(2) If* $P \to Q$, *then* $Q \vdash_\eta \Delta; \Gamma$.

Proof. By induction on derivations for $P \leq Q$ (resp. $P \to Q$), we verify that all the rules of $\leq$ (Def. 3) (resp. $\to$ (Def. 4)) are type preserving.

Progress We prove the progress property for well-typed CLASS processes. The following notion of *live process* becomes useful. A process P is *live* if and only if $P = \mathcal{C}[Q]$, for some static context $\mathcal{C}$ (the hole lies within the scope of static constructs mix, cut and share) and Q is an active process (a process with a topmost action prefix, such as a receive or a take, or a forwarder). We first show that a live well-typed process either reduces or offers an interaction with its environment on a free name. The following observability predicate (cf. [70]) characterises the interactions of a process with its environment

Definition 5 ($P \downarrow_x$). *The predicate* $P \downarrow_x$ *is defined by rules of Fig. 8.*

The predicate $P \downarrow_x$ holds if P offers an immediate interaction (unguarded action) on free name x. We can observe the subject of an action (rule [act]) and x, y of a forwarder fwd x y. The definition of $P \downarrow_x$ is closed by $\leq$ and propagates observations over the various static operators. Cut bound names are not free, hence cannot be observed. Share share y $\{P \mid\mid Q\}$ propagates all the observations x for which $x \neq y$ and by applying $\leq$ rules [ShTake], [ShRel] or [ShPut] via [$\leq$], an interaction on x may be observed. We have

Lemma 1 (Liveness). *Let* $P \vdash_\emptyset \Delta; \Gamma$ *be live. Either* $P \downarrow_x$ *or* P *reduces.*

Proof. (Sketch) By induction on a derivation for $P \vdash_\emptyset \Delta; \Gamma$, along the lines of [27]. To handle case [Tcut] $P = P_1 \ |y| \ P_2$: both P_1 and P_2 are live, since both type with a nonempty linear typing context, hence we can apply the induction hypothesis (i.h.) to both premises of [Tcut]: either (i) one of P_1 and P_2 reduces or (ii) both $P_1 \downarrow_{x_1}$ and $P_2 \downarrow_{x_2}$. If (i), then P reduces. Case (ii) follows because, crucially, P_1 and P_2 synchronise through a single private session y, then either $x_1 \neq y$ or $x_2 \neq y$, in which case we can observe either x_1 or x_2; or $x_1 = x_2 = y$, in which case we can trigger a reduction, by applying $\leq$ rules to P in order to exhibit a principal cut. For case [Tsh] $P =$ share y $\{P_1 \mid\mid P_2\}$: since P_1 and P_2 are live, we apply i.h. to both premises. The interesting case occurs when $P_1 \downarrow_{x_1}$ and $P_2 \downarrow_{x_2}$. Co-contraction implies that P_1 and P_2 share the single usage y, so if $x_1 \neq y$ or $x_2 \neq y$, we have either $P_1 \downarrow_{x_1}$ or $P_1 \downarrow_{x_2}$. If both $x_1 = x_2 = y$, then we derive $P \downarrow_y$: the observation corresponds to either a take or a release operation on y, which we commute up with [ShTake] or [ShRel]. For [TshL] $P =$ share y $\{P_1 \mid\mid P_2\}$, we apply the i.h. to the premise P_1, which types with an empty usage on y. If $P_1 \downarrow_y$, then $P \downarrow_y$, the observation corresponding a put operation on y, which we commute up with [ShPut]. Symmetrically for [TshR].

Theorem 2 (Progress). *Let* $P \vdash_\emptyset \emptyset; \emptyset$ *be a live process. Then,* P *reduces.*

Proof. Follows from Lemma 1 since $\mathsf{fn}(P) = \emptyset$.

$$\frac{}{\mathsf{fwd}\ x\ y\downarrow_x}\ [\text{fwd}] \quad \frac{s(\mathcal{A})=x}{\mathcal{A}\downarrow_x}\ [\mathcal{A}] \quad \frac{P\leq Q \quad Q\downarrow_x}{P\downarrow_x}\ [\leq] \quad \frac{P\downarrow_x}{(P\ ||\ Q)\downarrow_x}\ [\text{mix}]$$

$$\frac{P\downarrow_x \quad x\neq y}{(P\ |y|\ Q)\downarrow_x}\ [\text{cut}] \quad \frac{Q\downarrow_x \quad x\neq y}{(z.P\ |!y|\ Q)\downarrow_x}\ [\text{cut!}] \quad \frac{P\downarrow_x \quad x\neq y}{(\mathsf{share}\ y\ \{P\ ||\ Q\})\downarrow_x}\ [\text{share}]$$

Fig. 8: Observability Predicate $P\downarrow_x$.

Remarkably, our proof of Theorem 2 leverages deep properties of Linear Logic, in particular the structure of the linear cut and co-contraction, allowing us to prove deadlock absence, even in a language with primitives exhibiting blocking behaviour, avoiding the use of extra mechanisms [47,33,48,10,25,76,31].

Strong Normalisation Establishing strong normalisation (SN) for concurrent process calculi is usually fairly challenging, particularly in the presence of name passing, recursion and higher-order shared state [32,16,83,49,69]. For example, with reference cells one may express general recursion with Landin's knot, and, in general, circular chains of references that may lead to divergence. However, our linear type system uses primitive recursion and corecursion, and excludes cyclic dependencies through state or session based interaction, allowing strong normalisation, and therefore livelock absence, to hold. Our proof relies on defining suitable linear logical relations, cf. [62,21,72], adapted to Classical Linear Logic [38,1,8], and crucially relying on a notion of reducibility up to interference that imposes stronger properties on the interpretation of the state modalities, and which allows the inductive proof of the Fundamental Lemma 2 to go through in the usual way. To this end, we extend our basic language with auxiliary constructs $\mathsf{cell}\ c(a.S)$ and $\mathsf{empty}\ c(a.S)$, which denote memory cells subject to interference from concurrent writers, allowed to take terms from the set $S\subseteq\{P\mid P\vdash_\eta a:\wedge A\}$. The intuition is that a take on the cell may always read any object from S, due to interference. We also consider the additional reduction (nondeterministic) rules (1)-(3), where in 1 and 2 we assume $P\in S$.

$$\begin{array}{lll}
\mathsf{cell}\ c(a.S)\ |c|\ \mathsf{release}\ c & \to P\ |a|\ \mathsf{discard}\ a, & (1)\\
\mathsf{cell}\ c(a.S)\ |c|\ \mathsf{take}\ c(a');Q & \to \mathsf{empty}\ c(a.S)\ |c|\ (P\ |a|\ \{a/a'\}Q) & (2)\\
\mathsf{empty}\ c(a.S)\ |c|\ \mathsf{put}\ c(a.P);Q & \to \mathsf{cell}\ c(a.S)\ |c|\ Q & (3)
\end{array}$$

In this section, we thus consider reduction of $P\to Q$ to be the relation defined in Fig 7, extended with these rules. When a take or a release interacts with $\mathsf{cell}\ c(a.S)$, an arbitrary element P from the set S may be picked (rules (1) and (2)). In (3), a put $\mathsf{put}\ c(a.P);Q$ interacts with $\mathsf{empty}\ c(a.S)$ causing $\mathsf{empty}\ c(a.S)$ to evolve to $\mathsf{cell}\ c(a.S)$ (3). The following notion is also useful. A process P is *S-preserving on x* if $P\vdash_\eta x:\mathsf{U}_\bullet A$ or $P\vdash_\eta x:\mathsf{U}_\circ A$, and

- if $P\xrightarrow{*}\approx \mathsf{take}\ x(y);P'$ and $Q\in S$, then $Q\ |y|\ P'$ is S-preserving on x.
- if $P\xrightarrow{*}\approx \mathsf{put}\ x(y.P_1);P_2$, then $P_1\in S$ and P_2 is S-preserving on x.

A set of processes T is S-preserving on x if and only for all $P \in T$, P is S-preserving on x. Intuitively a process P that uses a cell x is S-preserving on x if it only puts values from S on cell x. The notion of S-preservation, parametric on any S, brings explicit the conditions needed for safe interaction with a memory cell, subject to interference, while ensuring a state invariant S on the cell contents. We now introduce the logical predicate.

Definition 6 (Logical Predicate $[\![x : A]\!]_\sigma$). *By induction on the type A, we define the sets $[\![x : A]\!]_\sigma$ an shown in Fig. 9, such that $[\![x : \mathsf{U}_\bullet A]\!]_\sigma$ and $[\![x : \mathsf{U}_\circ A]\!]_\sigma$ are $[\![- : \wedge\overline{A}]\!]$-preserving on x.The definition is direct for the positive types A, for negative types B is given by orthogonality.*

The definition relies on Girard's notion of orthogonality $S^\perp \triangleq \{P \mid \forall Q \in S.\ P \,|x|\, Q \text{ is SN}\}$ [37]. Duality promotes succinctness in our definition: for negative types A, $[\![x : A]\!]_\sigma$ is defined as the orthogonal of the predicate for its dual $\overline{A}$ (positive) type. To handle polymorphic and inductive types, the logical predicate is indexed by a map σ that assigns reducibility candidates $R[x : A]$ to type variables. A reducibility candidate $R[x : A]$ is any set S of processes $P \vdash_\emptyset x : A$ such that P is SN and $S = S^{\perp\perp}$. We let $\mathcal{R}[- : A]$ be the set of all reducibility candidates $R[x : A]$ for some name x. The definition relies on a congruence relation $\approx$ extending $\leq$ with a complete set of commuting conversions, along standard lines [22,27,80]. It essentially plays the role of the labelled transition system in the proof of strong normalisation given in [62].

We extend the logical predicate to typing judgements $P \vdash_\eta \Delta; \Gamma$ by universal closure over the typing context and σ.

Definition 7 (Extended Logical Predicate $\mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$). *We define $\mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$ inductively on Δ, Γ and η as the set of processes $P \vdash_\eta \Delta; \Gamma$ s.t.*

$$
\begin{array}{l}
P \in \mathcal{L}[\![\vdash_\emptyset \emptyset; \emptyset]\!]_\sigma \ \textit{iff}\ P \textit{ is SN.} \\
P \in \mathcal{L}[\![\vdash_\emptyset \Delta, x : A; \Gamma]\!]_\sigma \ \textit{iff}\ \ \forall Q \in [\![x : \overline{A}]\!]_\sigma.\ Q \,|x : \overline{A}|\, P \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma]\!]_\sigma. \\
P \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma, x : A]\!]_\sigma \ \textit{iff}\ \forall Q \in [\![y : \overline{A}]\!]_\sigma.\ y.Q \,|!x : \overline{A}|\, P \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma]\!]_\sigma. \\
P \in \mathcal{L}[\![\vdash_{\eta, X(x,\boldsymbol{y}) \mapsto \Delta', x:Y; \Gamma} \Delta; \Gamma]\!]_\sigma \ \textit{iff}\ \forall Q \in \sigma(Y).\ \{Q/X\}P \ \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma.
\end{array}
$$

We now state the Fundamental Lemma (2) from which Theorem 3 follows.

Lemma 2 (Fundamental Lemma). *If $P \vdash_\eta \Delta; \Gamma$, then $P \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$.*

Proof. (Sketch) By induction on $P \vdash_\eta \Delta; \Gamma$. For cases [Tcell] and [Tempty], we show that $\mathsf{cell}\ c(a.S)$ and $\mathsf{empty}\ c(a.S)$ respectively simulate $\mathsf{cell}\ c(a.P)$ (where $P \in S$) and $\mathsf{empty}\ c$, when composed with any S-preserving on c usages. To handle one of the most challenging cases, [Tsh] we prove, for all S, and all S-preserving on x processes P_1 and P_2, that $\mathsf{cell}\ c(a.S) \,|c|\, \mathsf{share}\ c\ \{P_1 \parallel P_2\}$ (1) is simulated by $(\mathsf{cell}\ c(a.S) \,|c|\, P_1) \parallel (\mathsf{cell}\ c(a.S) \,|c|\, P_2)$ (2). This allows us to infer that if (2) is SN, then so it is (1). When $S = [\![a : \wedge\overline{A}]\!]_\sigma$, the i.h. yields $(\mathsf{cell}\ c(a.S) \,|c|\, P_i)$ SN, hence we conclude (2) SN. Similarly for [TshL], [TshR].

Theorem 3 (Strong Normalisation). *If $P \vdash_\emptyset \emptyset; \emptyset$, then P is SN.*

$$
\begin{aligned}
[\![x : X]\!]_\sigma &\triangleq \sigma(X)[x] \\
[\![x : \mathbf{1}]\!]_\sigma &\triangleq \{P \mid P \approx \mathsf{close}\ x \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : A \otimes B]\!]_\sigma &\triangleq \{P \mid \exists P_1, P_2.\ P \approx \mathsf{send}\ x(y.P_1); P_2 \text{ and } \\
&\qquad P_1 \in [\![y : A]\!]_\sigma \text{ and } P_2 \in [\![x : B]\!]_\sigma\}^{\perp\perp} \\
[\![x : A \oplus B]\!]_\sigma &\triangleq \{P \mid \exists Q.\ P \approx x.\mathsf{inl}; Q \text{ and } Q \in [\![x : A]\!]_\sigma \text{ or } \\
&\qquad P \approx x.\mathsf{inr}; Q \text{ and } Q \in [\![x : B]\!]_\sigma\}^{\perp\perp} \\
[\![x : !A]\!]_\sigma &\triangleq \{P \mid \exists Q.\ P \approx !x(y); Q \text{ and } Q \in [\![y : A]\!]_\sigma\}^{\perp\perp} \\
[\![x : \exists X.A]\!]_\sigma &\triangleq \{P \mid \exists Q, S\ \in \mathcal{R}[- : B].\ P \approx \mathsf{sendty}\ x(B); Q \text{ and } \\
&\qquad Q \in [\![x : A]\!]_{\sigma[X \mapsto S]}\}^{\perp\perp} \\
[\![x : \mu X.\ A]\!]_\sigma &\triangleq (\bigcap\{S \in \mathcal{R}[- : \mu X.A] \mid \mathsf{unfold}_\mu\ x; [\![x : A]\!]_{\sigma[X \mapsto S]} \subseteq S\})^{\perp\perp} \\
[\![x : \wedge A]\!]_\sigma &\triangleq \{P \mid \exists Q.\ P \approx \mathsf{affine}\ x; Q \text{ and } Q \in [\![x : A]\!]_\sigma\}^{\perp\perp} \\
[\![x : \mathsf{S}_\bullet A]\!]_\sigma &\triangleq \{P \mid P \approx \mathsf{cell}\ x(y.[\![y : \wedge A]\!]_\sigma) \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : \mathsf{S}_\circ A]\!]_\sigma &\triangleq \{P \mid P \approx \mathsf{empty}\ x(y.[\![y : \wedge A]\!]_\sigma) \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : B]\!]_\sigma &\triangleq [\![x : \overline{B}]\!]_\sigma^{\perp} \quad \text{(B negative type)}
\end{aligned}
$$

Fig. 9: Logical Predicate $[\![x : A]\!]_\sigma$.

4 Typeful Concurrent Programming in CLASS

In this section, we discuss the expressiveness of CLASS's type system, going through a sequence of illustrative realistic concurrent programming idioms.

Sharing a Linear Session. Our first example illustrates how objects subject to a linear usage protocol and satisfying an invariant may be shared among multiple concurrent clients by serialising linear usages using a mutex cell, alternating ownership from the cell to clients and back at the invariant state, a commonly used discipline to implement and reason about resource sharing (see, e.g., [39,17,9]). We illustrate with a basic toggle switch with two states - On and Off - the resource invariant is the state Off, and two operations #turnOn and #turnOff that must be executed in strict linear sequence (Fig. 10). The toggle protocol, defined by type Off, offers the single option #turnOn, after which it evolves to On. Conversely, type On offers the single option #turnOff, after which it evolves to an affine Off. The toggle process at t is defined by two mutually corecursive processes $\mathsf{on}(t)$ and $\mathsf{off}(t)$, which define the expected behaviour, and comply with types On and Off.

Process main() introduces a mutex cell c storing an affine toggle object at the invariant type $\wedge$Off. It then shares it with two concurrent clients, each acquires the toggle in the invariant type and uses the linear protocol independently. After their linear interaction, they put back the toggle, the type system ensures that this can only happen when the invariant (given by the cell type) holds. When they are done, both clients release their respective usages of c, which ultimately leads to the cell being deallocated and the (affine) toggle to be discarded.

```
type corec Off = &{|#turnOn : On}
type corec On  = &{|#turnOff : ∧Off}

off(t) ⊢ t : Off
off(t) = case t {|#turnOn : on(t)}

on(t) ⊢ t : On
on(t) = case t {|#turnOff :
                  affine t; off(t)}

client1(c) ⊢ c : S•Off
client1(c) = take c(t);
             #turnOn t; #turnOff t;
             put c(t); release c

client2(c) ⊢ c : S•Off
client2(c) = take c(t);
             #turnOn t; #turnOff t;
             #turnOn t; #turnOff t;
             put c(t); release c

main()  ⊢ ∅
main()  = cut {cell c(t.affine t; off(t))
                |c|
                share c {
                  client1(c)  ||
                  client2(c) }}
```

Fig. 10: Sharing a Linear Toggle Switch

```
type rec SList(A) = S•List(A)
type rec List(A)  = ⊕{
                      |#Null : 1,
                      |#Next : ∧A ⊗ SList(A)}

nil(l) ⊢ l : ∧List(A)
nil(l) = affine l; #Null l; close l

cnext(a, c, l) ⊢ a: ∨A, c:SList(A), l: ∧List(A)
cnext(a, c, l) = affine l;
                 #Next l;
                 send l(a);
                 fwd l c

append(c, l′, c′) =
 take c(l);
 case l {
  |#Null :
    wait l; put c(l′); fwd c c′
  |#Next :
    recv l(a);
    cut {
      append(l, l′, x)
      |x|
      put c(y.cnext(a, x, y));
      fwd c c′ }}
```

Fig. 11: A Linked List with an Append In-Place Operation.

We have also developed CLASS code for a generic (polymorphic) wrapper factory that, for any affine corecursive protocol, generates a wrapper to a general invariant-based sharing interface.

Linked Lists, Update In-Place. In this example, we show how inductive/-coinductive types combine harmoniously with CLASS state modalities to type linked data structures with memory-efficient updates in-place. Specifically, we show how to code a linked list, parametric on the type A of its affine values, with update in-place append (Fig. 11). An object of type $\mathsf{SList}(A)$ is a (full) cell storing a $\mathsf{List}(A)$ object. An object of type $\mathsf{List}(A)$ is a session that either selects #Null (the list is empty), in which case it closes; or selects #Next, in which case it sends an affine session $\wedge A$ representing the head element and continues as the tail $\mathsf{SList}(A)$. Process $\mathsf{nil}(l)$ - defines an empty list at l - and process $\mathsf{cnext}(a, c, l)$ - constructs a nonempty list l with head a and tail c. For example, a list with

elements a, b stored at $c_1 : \mathsf{S}_\bullet\mathsf{List}(A)$ is represented

$$\mathsf{cut}\{\ \mathsf{cell}\ c_1(l_1.\mathsf{cnext}(a, c_2, l_1))\ |c_2|\ \mathsf{cell}\ c_2(l_2.\mathsf{cnext}(b, c_s, l_2))\ |c_s|\ \mathsf{cell}\ c_s(l_0.\mathsf{nil}(l_0))\}$$

Process $\mathsf{append}(c, l', c') \vdash c : \overline{\mathsf{SList}(A)}, l' : \overline{\mathsf{List}(A)}, c' : \mathsf{SList}(A)$ produces on c' the result of appending l (in place) to c. It takes the list l stored in c, and then performs case analysis on l. If l selects $\#\mathsf{Null}$, it simply replaces the previous null node of c by l' and forwards the updated cell c to the output c'. This corresponds to the recursion base case in which the list l is empty.

If l selects $\#\mathsf{Next}$, in which case l has at least one element, one receives at l the node element $a : \vee\overline{A}$, and corecursively call append l' to the tail $l : \overline{\mathsf{SList}(A)}$ and puts back in c element a and tail x "returned" by the call. Notice that x is exactly x (by forwarding), which was passed along linearly. Remarkably, the $\mathsf{append}(c, l', c')$ operation just defined may be safely applied concurrently to the same shared linked list, with the final result being the correct one (some serialisation of the appends), without deadlocks or livelocks. It is also interesting to see how the type system forbids a list to be appended to itself.

We have also developed many other in-place operations on linked data structures, such as insertion sort, and other kinds of linked structures such as queues and binary search trees. In the next examples we discuss a shared queue ADT with a fine-grained locking discipline and O(1) enqueue and dequeue operations.

A Concurrent Shareable Buffered Channel. We illustrate increased degrees of sharing in a mutable data structure with various references pointing to different parts of it, how the CLASS type system may express interfaces that talk about different client views for using a stateful object, and the use of polymorphism to implement information hiding ensuring that client code will never break the representation invariants of stateful ADTs, particularly challenging when aliasing and sharing are involved.

More concretely, we consider a shareable buffered channel (Fig. 12), and provide a realistic and efficient implementation [56] based on a message queue represented by a linked list with update-in-place (cf. Section 4 above) and two independent pointers: one to the head of the list, used for receiving, and another to the tail, used for sending. The operations are executed in O(1) time. Moreover we provide a typing with two separate send and receive views, which may be used by an arbitrary number of concurrent clients. In particular, when the list is nonempty, both send and receive run in true concurrency (asynchronously), without blocking each other, thanks to fine-grained locking.

The buffered channel type $\mathsf{BChan}(M)$, where M is the type of messages, offers two views: $\mathsf{SendT}(M)$ and $\mathsf{RecvT}(M)$, interfaces for sender and receiver endpoint clients. These views are exposed with a par ($\parr$), since they share an underlying resourceful structure. In fact, they could not be exported using a tensor ($\otimes$); it is interesting to notice how the type system imposes these constraints, important to ensure deadlock freedom. The representation type of both views is $Rep = \mathsf{S}_\bullet\mathsf{SList}(M)$ (see Section 4), hidden behind the SV and RV existential types [29,58]; sending clients use a cell storing a reference to the tail node of

$$\mathsf{type}\ \mathsf{BChan}(M) = \mathsf{SendT}(M) \parr \mathsf{RecvT}(M)$$
$$\mathsf{type}\ \mathsf{SendT}(M) = \exists SV.!\mathsf{MenuS}(M, SV) \otimes SV$$
$$\mathsf{type}\ \mathsf{RecvT}(M) = \exists RV.!\mathsf{MenuR}(M, RV) \otimes RV$$

$$\mathsf{type}\ \mathsf{MenuS}(M, SV) = \& \{ \\ |\#\mathsf{Send} : SV \multimap \wedge M \multimap SV, \\ |\#\mathsf{Share} : SV \multimap (SV \parr SV), \\ |\#\mathsf{Free} : SV \multimap \mathbf{1} \},$$

$$\mathsf{type}\ \mathsf{MenuR}(M, RV) = \& \{ \\ |\#\mathsf{Recv} : RV \multimap (\mathsf{Maybe}(\wedge M) \otimes RV), \\ |\#\mathsf{Share} : RV \multimap (RV \parr RV), \\ |\#\mathsf{Free} : RV \multimap \mathbf{1} \}$$

$$Rep = SV = RV = \mathsf{S}_\bullet\mathsf{SList}(M)$$

```
msend(me) =
recv me(tailptr);
recv me(a);
take tailptr(c);
take c(l);
cut {
  cell c'(l)
  |c'|
  share c' {
    put c(l'.cnext(a, c', l'));
    release c'
    ||
    put tailptr(c');
    send me(tailptr);
    close me}}
```

Fig. 12: A Concurrent Shareable Buffered Channel.

the queue; receiving clients use a cell storing a reference to the head node of the queue.

Clients use the buffer through references of abstract type SV and RV and replicated menus $!\mathsf{MenuS}(M, SV)$ and $!\mathsf{MenuR}(M, RV)$. Both menus export the options #Share and #Free to allow sharing and release of the views. To send, a client selects #Send, sends his handle (of opaque type SV), the message to send and receives the (linear) handle back. In this implementation, receive is non-blocking, so operation #Recv returns a $\mathsf{Maybe}(\wedge M)$ value: the client receives either #Nothing (if the buffer is empty) or #Just followed by a message a, otherwise. In 4 we discuss the implementation, in CLASS, of (Hoare style) monitors with conditions, which would allow a blocking receive to be implemented.

Process $\mathsf{msend}(me)$ implements the #Send "method". It first receives the sending view handle (of concrete type Rep), which is a cell with the *tailptr*, and the message a to be sent. Then, a new cell c' with nil (l) is created, the current tail of the list c is updated with a new node storing a and pointing to c'. Finally, the *tailptr* cell is updated to point to the new tail node c' of the linked list.

Dining Philosophers. A resource hierarchy solution for the dining philosophers problem [34] requires forks to be acquired in a defined order. We "encode" such order in CLASS with an explicit (necessarily) acyclic structure, which informs the type system about the code safety. This allows us to define a correct implementation that satisfies deadlock freedom by pure linear logic typing. More concretely, we organise the forks in a linked chain defined by the inductive types $\mathsf{rec}\ \mathsf{Fork} = \mathsf{S}_\bullet\mathsf{Node}$ and $\mathsf{rec}\ \mathsf{Node} = \oplus\{\#\mathsf{Null} : \mathbf{1}, \#\mathsf{Next} : \mathsf{Fork}\}$.

Any fork in the chain may be shared by an arbitrary number of philosophers, cocontraction ensures that philosophers cannot communicate between themselves via any other channel, all synchronisation must happen via the chained

```
putNull(f, f') ⊢ f : U∘Node‾, f' : Fork
putNull(f, f') ≜ put f(n.null()); fwd f f'

eat(f, f') ⊢ f : Fork‾, f' : Fork
eat(f, f') ≜
  take f(n);
  case n {
    |#Null :
    wait n; putNull(f, f')
    |#Next :
    take n(m);
    put n(m); put f(n'.next(n, n'));
    fwd f f'}
```

```
eat2(f, f') ⊢ f : Fork‾, f' : Fork
eat2(f, f') ≜
  take f(n);
  case n {
    |#Null :
    wait n; putNull(f, f')
    |#Next :
    cut {
      takeLast(n, x)
      |x|
      recv x(m); wait x;
      put f(n'.next(m, n'));
      fwd f f'}
```

Fig. 13: The Dining Philosophers.

forks. Furthermore, the chain can be resized and grow unboundedly to accommodate an arbitrary number of philosophers. If a philosopher successfully takes a fork f_i, he can then take any fork f_j, with $i < j$; crucially, he must follow the path dictated by the chain, hence cannot acquire forks f_j with $j < i$. In Fig. 13 we define the eat operation, which allows each philosopher P_i, with $0 \leq i < k-1$ to eat: it acquires two consecutive forks in the chain. And eat2, which is the specific eating operation for the symmetry breaker P_{k-1}: it acquires the first fork, and traverses the chain to acquire the last with $\mathsf{takeLast}(n, x) \vdash n : \overline{\mathsf{Fork}}, x : \mathsf{Fork} \otimes \mathbf{1}$.

A Barrier for N threads. We describe in Fig. 14 a CLASS implementation of a simple barrier, parametric on the number N of threads to synchronise. We find it interesting to model the "real" code shown in the Rust reference page for std::sync::Mutex [46]. The code uses if-then-else and primitive integers, as offered in our implementation, that could be defined as idioms in CLASS. We represent a barrier by a mutex cell storing a pair consisting of an integer n, holding the number of threads that have not yet reached the barrier, and a stack s of waiting threads, each represented by a session of *affine* type $\wedge\bot$ (so they will be safely aborted if at least one thread fails to reach the barrier).

The type Barrier of the barrier is $\mathsf{S}_\bullet\mathsf{BState}$, where $\mathsf{BState} \triangleq \mathsf{Int} \otimes \wedge\mathsf{List}(\wedge\bot)$. Initially the barrier is initialised with $n = N$ threads and an empty stack, so that the invariant $n + depth(s) = N$ holds during execution. Each $\mathsf{thread}(c; i)$ acquires the barrier c and checks if it is the last thread to reach the barrier (if $n == 1$): in this case, it awakes all the waiting threads ($\mathsf{awakeAll}(w_s)$) and resets the barrier. Otherwise, it updates the barrier by decrementing n and pushing its continuation into the stack (the continuation for thread i just prints "finished"). The following process $\mathsf{main}() \vdash \emptyset$ creates a new barrier c and spawns N threads, each labelled

```
init(w_s) ⊢ w_s : ∧BState
init(w_s) ≜
  affine w_s; send w_s(N); affine w_s; nil(w_s)

awakeAll(w_s : List(∧⊥))
awakeAll(w_s) ≜
  case w_s {
    #Nil : wait w_s; 0
    #Cons :
    recv w_s(w);
    par {close w || awakeAll(w_s)}

spawnAll(c; i, n) ⊢ c : Barrier; i : Int, n : Int
spawnAll(c; i, n) ≜
  if (n == 0) { release c}
  { share c {
      thread(c; i)
      ||
      spawnall(c; i + 1, n − 1)}}

thread(c; i) ⊢ c : Barrier; i : Int
thread(c; i) =
  println i + ": waiting.";
  take c(w_s); recv w_s(n);
  if (n == 1) {
    par {
      println i + ": finished.";
      awakeAll(w_s)
      ||
      put c(w'_s.init(w'_s));
      release c}}
  { cut {
      affine w; wait w;
      println i + ": finished."; 0
      |w| put c(w'_s.affine w'_s;
                send w'_s(n − 1);
                affine w'_s;
                cons(w, w_s, w'_s));
      release c}}
```

Fig. 14: A Barrier for N Threads

by a unique id i: main() $\triangleq$ cut { cell $c(w_s$.init$(w_s))$ $|c|$ spawnAll$(c; 0, N)$ }. Again, our type system statically ensures that the code does not deadlock or livelock.

A Hoare Style Monitor. A Hoare style monitor is a well-know powerful programming abstraction [39], allowing concurrent operations on shared data to be coordinated in a sound way, so that it always satisfy a correctness invariant. The key essential idea is that concurrent client threads use the monitor lock to access the protected state in mutual exclusion, but may also wait (via a *await* primitive) inside the monitor until the state satisfies specific (pre-)conditions, while transferring state ownership to other threads potentially responsible for establishing such conditions and announcing it (via a *notify* primitive).

We discuss a CLASS implementation of a monitor, sketching the main components and how they are typed (Fig. 15). We consider a counter with value n, with increment #Inc and decrement #Dec operations, and subject to the invariant $n \geq 0$. The type of the counter CounterI exposes two separate, coinductively defined, client interfaces DecI and IncI for decrementing and incrementing.

While the #Inc operation is synchronous, the #Dec operation is always called asynchronously by passing a continuation (of type ContDec). This allows decrementers to wait inside the monitor for condition NZ ($n > 0$) when $n = 0$. The condition NZ is represented by a wait queue of type WaitQ. The representation type of the monitor (Rep) holds the counter value and the wait queue. Each node in the wait queue stores information, of type ContDecW, for the waiting thread.

```
type corec IncI ≜ &{|#Inc : IncI, |#End : ⊥}
type corec DecI ≜
  ∨ & {|#Dec : ∨(ContDec ⊸ ⊥), #End : ⊥}
type corec ContDec ≜ ∨(DecI ⊗ 1)
type CounterI ≜ DecI ⅋ IncI

type rec Rep ≜ (!Int) ⊗ WaitQ
type rec WaitQ ≜ ∧ ⊕ {|#Null : 1, |#Next : NodeQ}
type rec NodeQ ≜ S•(ContDecW ⊗ WaitQ)
type rec ContDecW ≜ ∧(∧Rep ⊸ ∧Rep ⊗ DecI ⊸ ⊥)

awaitNZ ⊢ m : U◦Rep,
           n : !Int, w : WaitQ, cc : ContDecW
notifyNZ ⊢ m : U◦Rep, s : Rep, m' : S•Rep
incloop ⊢ iv : IncI, m : U•Rep
```

```
awaitNZ(m, n, w, cc) ≜
  put m(w'.affine v;
    send w'(n);
    consWQ(cc, w, w'));
    release m

incloop(iv, m) ≜
  case iv {
  #Inc : take m(r);
         recv r(n);
  cut {
  send s(n + 1); fwd s r
  |s| notifyNZ(m, s, m')
  |m'| incloop(iv, m') }
  #End : wait iv;
         release m}
```

Fig. 15: Implementing a Counter Monitor with Await / Notify.

Every such ContDecW objects stores (1) the pending action on the internal monitor state (of type ∧Rep ⊸ ∧Rep), to be executed after await returns, and (2) a callback to the continuation provided by the external client in the asynchronous call (of type DecI ⊸ ⊥).

The awaitNZ(m, n, w, cc) process implements the monitor wait operation, used in the #Dec operation. It receives the (empty) cell usage m to the monitor state, the integer value n (where $n = 0$), a reference w to the wait queue, and the continuation cc, it pushes a new node in the queue and puts the monitor state back, unlocking the cell m, and releases m. The incloop(iv, m) process implements the counter IncI interface. The call to notifyNZ(m, s, m') after incrementing n will cause a waiting DecI thread to be awaken (if any), and continue by applying the pending action to the Rep state s in which $n > 0$ holds, before passing the updated state m' to the incloop recursive call. Affinity plays a key role, allowing all data structures, including waiting continuations to be safely discarded, at the end of any computation. We have only shown here some code snippets, the complete code is available in the CLASS distribution.

Our examples illustrate how our system types non-trivial concurrent code, akin to real system-level code, involving higher-order state, rich sharing and ownership transfer patterns, while ensuring deadlock, livelock freedom and memory safety. Our typing of sharing imposes that only a single bundle of linear resources may be shared by two independent threads. As our examples show, code can often be structured in that way, so that bundles of many linear resources may be safely shared by monitor-like structures, exposing informative typed interfaces.

The feasibility of CLASS is corroborated by our implementation [68] of a fully-fledged type checker and interpreter, developed in Java (∼15k), and packaged

with an extensive CLASS library of code and test suites (∼10k), including all the examples in this paper. Type checking is decidable in polynomial time, using a minimal type annotation, only on cut-bound names and function parameters, the multiplicative rules are handled by lazy context splitting (cf. [41]). The type checker ensures that corecursive calls are done on a session *hereditarily descendent* from the corecursion parameter, a condition motivated by our SN result (Theorem 3). But we also support an unsafe corecursion mode, in which this check is turned off, to type programs defined by general corecursion.

The type checker supports useful type inference and reconstruction abilities. The interpreter uses java.util.concurrent.* package [53], using primitives such as fine-grained locks and condition variables to emulate the synchronous interactions of CLASS sessions and a cached thread pool to manage the life cycle of short-lived threads. Cell deallocation is implemented by reference counting, incremented on each share and decremented on each release. Forwarding redirects the clients of a shared cell through a chain of forwarding pointers (cf. [9]).

5 Related Work

Many resource-aware logics and type systems to tame shared state and interference have been proposed [3,18,57,77,44,17,60,61,24]. These systems adopt some form of linearity and/or affinity to resourceful programming [75,30] and to model failures/exceptions [28,59,20,36,52]. In CLASS, linearity allows us to control state sharing, whereas affinity is useful to ensure memory safety and to represent safely finalizable or abortable computations. The hereditary session-discarding behaviour of affine sessions, modelled by rule [∧∨d], is also present in other works, e.g. [6,59,20].

CLASS builds on top of the PaT correspondence with Linear Logic [22,27,80], the logical principles for the state modalities being inspired by DiLL [35]. Recent works [43,9,10,7,50,64,67] also address the problem of sharing and nondeterminism in the setting of session-based PaT. In [67], reference cells may only store replicated sessions (of type $!A$), thus cannot refer to linear entities such as other cells or linear sessions, hence cannot represent many realistic programming idioms that CLASS does (see Section 4). Accommodating linear state in a pure PaT approach is thus addressed in this work with a novel, more fundamental approach. Furthermore, in [67], recursion is obtained via a system-F style encoding [79], which cannot model inductive stateful structures with updates in-place as we do with CLASS native inductive/coinductive types.

The take/put operations of CLASS relate with Concurrent Haskell MVars [45] and the acquire/release operations of the manifest sharing session-typed language SILL_S [9,10]. Sharing in SILL_S is based on shift modalities to move from shared to linear mode and back, and contraction principles to alias shared sessions. In CLASS we explore DiLL modalities and cocontraction principles [35] to express sharing of linear state and put / take protocols of mutex memory cells of invariant type. The work [10] ensures deadlock-freedom by relying on programmer provided partial orders on events [55,33,26], whereas in CLASS deadlock-freedom follows the same simple and general inductive argument of

the corresponding result in e.g. [22], thanks to the logical character of the new proof rules (DiLL cocontraction, that enjoys cut-elimination). The work [64] introduces the language CSLL, by extending linear logic with coexponentials that support a notion of shared state, with a quite different approach than ours. CSLL does not claim the ability to naturally express shared linked data structures with update in-place and fine-grained locking, as CLASS does. Nevertheless, it is natural to define in CLASS sessions exporting weakening, sharing and dereliction capabilities for linear behaviours, as in our shared buffer example.

Recently, the work [43] develops λ_{lock}, a substructural-typed λ-calculus with higher-order locks, which enjoys deadlock-freedom by imposing a set of high-level principles that guarantee acyclicity of the lock-sharing topologies, and which follow in CLASS as a consequence of its logical-motivated type system and DiLL's cocontraction. This work also extends λ_{locks} with partial orders in which a resource can shared by more than two concurrent threads. None of the models in [43,9,10,64] addresses livelock absence or memory safety, as CLASS does.

As far as we are aware, CLASS is a first proposal integrating shared state and recursion in a language based on PaT and Linear Logic, while guaranteeing strong normalisation. Least/greatest fixed points in Linear Logic were studied in [8], which inspired the development of recursion in [54,73], our treatment of recursion draws inspiration on [73]. Several works exploit the technique of logical relations to establish strong normalisation for concurrent process calculi [1,83,69,16,62]. The work [16] proves strong normalisation for a language with higher-order store with a type and effect system that stratifies memory into regions so as to preclude circularities. Interestingly, in CLASS such stratification is implicitly guaranteed by the acyclicity inherent to Linear Logic. Linear logical relations were studied in [62,21,72,74]. In this work we recast and extend the technique to Classical Linear Logic, exploring orthogonality [38,8,1], and demonstrate, using a specially devised technique of interference-sensitive reducibility, how logical relations scale to accommodate shared state.

6 Concluding Remarks

We have introduced CLASS, a session-based language founded on a propositions-as-types interpretation of Second-Order Classical Linear Logic, extended with recursion, affine types, first-class mutex cells and shared linear state. We believe that CLASS is the first proposal of a language of its kind to provide the following three strong properties by static typing: well-typed CLASS programs enjoy progress, hence never deadlock, do not leak memory and always terminate.

CLASS metatheoretical properties are obtained in a compositional and modular way, by leveraging the key features of propositions-as-types, from which the operational semantics and type system also emerges. In CLASS, types and process have a consistent proof-theoretical behaviour: typed program constructs correspond exactly to proof rules, with a proper compositional semantics via logical relations (Section 3). Programs are composed by plugging basic constructs with the cut rule, and all interaction principles are captured by principal cut reductions that act locally in proofs/type derivations (Def. 4). We also obtain

an algebraic system based on proof simplification to reason about program (observational) equivalence, due to confluence (cf. [65]).

Besides the foundational relevance of our work, we also argued how CLASS can cleanly express realistic concurrent higher-order programming idioms, with many compelling examples. Any type system introduces conservative restrictions on its language, but we believe that CLASS offers an interesting balance between the strong properties it ensures by typing and its expressiveness. In fact, we find CLASS type system helpful to guide the development of safe concurrent idioms, with a fairly light type annotation burden. As future work, we would like to investigate several possible refinements of the CLASS type discipline, namely, allowing finer-grained resource-access policies to be expressed, and exploring the integration of dependent and refinement types [71,51], enhancing the logical expressiveness of the basic type system.

References

1. Abramsky, S.: Computational Interpretations of Linear Logic. Theoret. Comput. Sci. **111**(1–2), 3–57 (1993)
2. Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: NATO ASI DPD. pp. 35–113 (1996)
3. Ahmed, A., Fluet, M., Morrisett, G.: L^3: A linear language with locations. Fundam. Inf. **77**(4), 397–449 (Dec 2007)
4. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. J. Logic Comput. **2**(3), 197–347 (1992)
5. Andreoli, J.M.: Logic Programming with Focusing Proofs in Linear Logic. J. Log. Comput. **2**(3), 297–347 (1992)
6. Asperti, A., Roversi, L.: Intuitionistic light affine logic. ACM Transactions on Computational Logic (TOCL) **3**(1), 137–175 (2002)
7. Atkey, R., Lindley, S., Morris, J.G.: Conflation confers concurrency. In: A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 32–55. Springer (2016)
8. Baelde, D.: Least and greatest fixed points in linear logic. TOCL **13**(1) (Jan 2012)
9. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. **1**(ICFP) (Aug 2017)
10. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Caires, L. (ed.) Programming Languages and Systems. pp. 611–639. Springer International Publishing, Cham (2019)
11. Barber, A.: Dual Intuitionistic Linear Logic. Tech. Rep. LFCS-96-347, Univ. of Edinburgh (1996)
12. Beffara, E.: A Concurrent Model for Linear Logic. ENTCS **155**, 147–168 (2006)
13. Beffara, E.: An algebraic process calculus. In: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science. p. 130–141. LICS '08, IEEE Computer Society, USA (2008)
14. Bellin, G., Scott, P.: On the π-calculus and linear logic. Theoret. Comput. Sci. **135**(1), 11–65 (1994)
15. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: International Workshop on Computer Science Logic. pp. 121–135. Springer (1994)

16. Boudol, G.: Typing termination in a higher-order concurrent imperative language. Information and Computation **208**(6), 716–736 (2010)
17. Brookes, S., O'Hearn, P.W.: Concurrent Separation Logic. ACM SIGLOG News **3**(3), 47–65 (2016)
18. Caires, L.: Logical Semantics of Types for Concurrency. In: International Conference on Algebra and Coalgebra in Computer Science. pp. 16–35. CALCO'07, Springer LNCS 4624 (2007)
19. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Relational parametricity for polymorphic session types. Tech. Rep. CMU-CS-12-108, Carnegie Mellon Univ. (2012)
20. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201. p. 229–259. Springer-Verlag, Berlin, Heidelberg (2017)
21. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: Proceedings of the 22nd European Conference on Programming Languages and Systems. p. 330–349. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013)
22. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 222–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
23. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. p. 1–12. TLDI '12, Association for Computing Machinery, New York, NY, USA (2012)
24. Caires, L., Seco, J.a.C.: The type discipline of behavioral separation. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 275–286. POPL '13, Association for Computing Machinery, New York, NY, USA (2013)
25. Caires, L., Vieira, H.T.: Conversation types. In: Castagna, G. (ed.) Programming Languages and Systems. pp. 285–300. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
26. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. **411**(51-52), 4399–4440 (2010)
27. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
28. Carbone, M., Honda, K., Yoshida, N.: Structured Interactional Exceptions in Session Types. In: CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer (2008)
29. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Computing Surveys (CSUR) **17**(4), 471–523 (1985)
30. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 48–64. OOPSLA '98, Association for Computing Machinery, New York, NY, USA (1998)
31. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures. pp. 91–109. Springer International Publishing, Cham (2018)
32. Demangeon, R., Hirschkoff, D., Sangiorgi, D.: Mobile processes and termination. In: Semantics and Algebraic Specification, pp. 250–273. Springer (2009)
33. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On progress for structured communications. In: Barthe, G., Fournet, C. (eds.) Trustworthy Global Computing. pp. 257–275. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

34. Dijkstra, E.W.: Hierarchical ordering of sequential processes. In: The origin of concurrent programming, pp. 198–227. Springer (1971)
35. Ehrhard, T.: An introduction to differential linear logic: proof-nets, models and antiderivatives. Mathematical Structures in Computer Science **28**(7), 995–1060 (2018)
36. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)
37. Girard, J.Y.: Linear logic. Theoret. Comput. Sci. **50**(1), 1–102 (1987)
38. Girard, J.Y.: Linear logic. Theoretical computer science **50**(1), 1–101 (1987)
39. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
40. Hoare, C.A.R.: Towards a theory of parallel programming. In: The origin of concurrent programming, pp. 231–244. Springer (1972)
41. Hodas, J.S., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Information and computation **110**(2), 327–365 (1994)
42. Howard, W.A.: The formulae-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press (1980)
43. Jacobs, J., Balzer, S.: Higher-order leak and deadlock free locks. Proceedings of the ACM on Programming Languages **7**(POPL), 1027–1057 (2023)
44. Jacobs, J., Balzer, S., Krebbers, R.: Connectivity graphs: a method for proving deadlock freedom based on separation logic. Proc. ACM Program. Lang. **6**(POPL), 1–33 (2022)
45. Jones, S.P., Gordon, A., Finne, S.: Concurrent Haskell. In: POPL. vol. 96, pp. 295–308. Citeseer (1996)
46. Klabnik, S., Nichols, C.: The Rust Programming Language (2021)
47. Kobayashi, N.: A type system for lock-free processes. Information and Computation **177**(2), 122–159 (2002)
48. Kobayashi, N.: A new type system for deadlock-free processes. In: International Conference on Concurrency Theory. pp. 233–247. Springer (2006)
49. Kobayashi, N., Sangiorgi, D.: A hybrid type system for lock-freedom of mobile processes. ACM Transactions on Programming Languages and Systems (TOPLAS) **32**(5), 1–49 (2008)
50. Kokke, W., Morris, J.G., Wadler, P.: Towards races in linear logic. In: Riis Nielson, H., Tuosto, E. (eds.) Coordination Models and Languages. pp. 37–53. Springer International Publishing, Cham (2019)
51. Krishnaswami, N.R., Pradic, P., Benton, N.: Integrating linear and dependent types. ACM SIGPLAN Notices **50**(1), 17–30 (2015)
52. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine rust programming with multiparty session types. arXiv preprint arXiv:2204.13464 (2022)
53. Lea, D.: Concurrent programming in Java: design principles and patterns. Addison-Wesley Professional (2000)
54. Lindley, S., Morris, J.G.: Talking bananas: structural recursion for session types. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 434–447. ACM (2016)
55. Lynch, N.A.: Fast allocation of nearby resources in a distributed system. In: Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing. p. 70–81. STOC '80, Association for Computing Machinery, New York, NY, USA (1980)

56. Marlow, S.: Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming. " O'Reilly Media, Inc." (2013)
57. Militão, F., Aldrich, J., Caires, L.: Aliasing control with view-based typestate. In: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs. pp. 1–7 (2010)
58. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems (TOPLAS) **10**(3), 470–502 (1988)
59. Mostrous, D., Vasconcelos, V.T.: Affine Sessions. In: Proc. of COORDINATION 2014. LNCS, vol. 8459, pp. 115–130. Springer (2014)
60. Nanevski, A., Morrisett, J.G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5-6), 865–911 (2008)
61. O'Hearn, P.W., Reynolds, J.C.: From Algol to polymorphic linear lambda-calculus. J. ACM **47**(1), 167–223 (2000)
62. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. Information and Computation **239**, 254–302 (2014)
63. Pfenning, F.: Structural cut elimination. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science. p. 156. LICS '95, IEEE Computer Society, USA (1995)
64. Qian, Z., Kavvos, G., Birkedal, L.: Client-server sessions in linear logic. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–31 (2021)
65. Rocha, P.: A Logical Foundation for Session-Based Concurrent Computation. Ph.D. thesis, NOVA School of Science and Technology (July 2022)
66. Rocha, P., Caires, L.: A Propositions-as-Types System for Shared State. Tech. rep., NOVA Laboratory for Computer Science and Informatics (06 2021)
67. Rocha, P., Caires, L.: Propositions-as-types and shared state. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–30 (2021)
68. Rocha, P., Caires, L.: Safe ssession-based concurrency with shared linear state (artifact) (January 2023). https://doi.org/10.5281/zenodo.7506064
69. Sangiorgi, D.: Termination of processes. Math. Struct. in Comp. Sci. **16**(1), 1–39 (2006)
70. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, USA (2001)
71. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming. p. 161–172. PPDP '11, Association for Computing Machinery, New York, NY, USA (2011)
72. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: Maffei, M., Tuosto, E. (eds.) TGC 2014. Lecture Notes in Computer Science, vol. 8902, pp. 159–175. Springer (2014)
73. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: International Symposium on Trustworthy Global Computing. pp. 159–175. Springer (2014)
74. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: A tale of two (fully abstract) encodings. ACM Trans. Program. Lang. Syst. **43**(2) (Jun 2021)
75. Tov, J.A., Pucella, R.: Practical Affine Types. In: POPL 2011. pp. 447–458 (2011)
76. Vieira, H.T., Vasconcelos, V.T.: Typing progress in communication-centred systems. In: International Conference on Coordination Languages and Models. pp. 236–250. Springer (2013)

77. Voinea, A.L., Dardha, O., Gay, S.J.: Resource sharing via capability-based multiparty session types. In: International Conference on Integrated Formal Methods. pp. 437–455. Springer (2019)
78. Wadler, P.: Linear types can change the world! In: Broy, M. (ed.) Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, 1990. p. 561. North-Holland (1990)
79. Wadler, P.: Recursive types for free (1990)
80. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 273–286. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012)
81. Wadler, P.: Propositions as Sessions. Journal of Functional Programming **24**(2-3), 384–418 (2014)
82. Wadler, P.: Propositions as types. Communications of the ACM **58**(12), 75–84 (2015)
83. Yoshida, N., Berger, M., Honda, K.: Strong normalisation in the π-calculus. Information and Computation **191**(2), 145–202 (2004)