A Compositional Theory of Linearizability

ARTHUR OLIVEIRA VALE, Yale University, USA
ZHONG SHAO, Yale University, USA
YIXUAN CHEN, Yale University, USA

Compositionality is at the core of programming languages research and has become an important goal toward scalable verification of large systems. Despite that, there is no compositional account of *linearizability*, the gold standard of correctness for concurrent objects.

In this paper, we develop a compositional semantics for linearizable concurrent objects. We start by showcasing a common issue, which is independent of linearizability, in the construction of compositional models of concurrent computation: interaction with the neutral element for composition can lead to emergent behaviors, a hindrance to compositionality. Category theory provides a solution for the issue in the form of the Karoubi envelope. Surprisingly, and this is the main discovery of our work, this abstract construction is deeply related to linearizability and leads to a novel formulation of it. Notably, this new formulation neither relies on atomicity nor directly upon happens-before ordering and is only possible *because* of compositionality, revealing that linearizability and compositionality are intrinsically related to each other.

We use this new, and compositional, understanding of linearizability to revisit much of the theory of linearizability, providing novel, simple, algebraic proofs of the *locality* property and of an analogue of the equivalence with *observational refinement*. We show our techniques can be used in practice by connecting our semantics with a simple program logic that is nonetheless sound concerning this generalized linearizability.

CCS Concepts: • **Theory of computation** → **Parallel computing models**; **Denotational semantics**; **Categorical semantics**; **Program verification**; **Program specifications**; • **Software and its engineering** → **Correctness**.

Additional Key Words and Phrases: linearizability, game semantics, concurrency, program logic

ACM Reference Format:
Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. 2023. A Compositional Theory of Linearizability. *Proc. ACM Program. Lang.* 7, POPL, Article 38 (January 2023), 32 pages. https://doi.org/10.1145/3571231

1 INTRODUCTION

Linearizability is a notion of correctness for concurrent objects introduced in the 90s by Herlihy and Wing [1990]. Since then, it has become the gold standard for correctness of concurrent objects: it is taught in university courses, known by programmers in industry, and commonly used in academia. Its success can be justified by a myriad of factors: it is a safety property in a variety of settings [Guerraoui and Ruppert 2014]; it appears to capture a large class of useful concurrent objects; it allows for linearizable concurrent objects to be horizontally composed together while preserving linearizability, what Herlihy and Wing [1990] call locality; it aids in the derivation of other safety properties [Herlihy and Wing 1990]; it is intuitive: a linearizable concurrent object essentially behaves as if their operations happened atomically under any concurrent execution, a

Authors' addresses: Arthur Oliveira Vale, Yale University, New Haven, CT, USA, arthur.oliveiravale@yale.edu; Zhong Shao, Yale University, New Haven, CT, USA, zhong.shao@yale.edu; Yixuan Chen, Yale University, New Haven, CT, USA, yixuan.chen@yale.edu.

2475-1421/2023/1-ART38
https://doi.org/10.1145/3571231

property that has been formalized by the notion of linearization point by Herlihy and Wing [1990], and by an observational refinement property by Filipovic et al. [2010].

1.1 The State of the Theory of Linearizability

Linearizability is commonly used to define correctness of concurrent objects and to aid in verification of concurrent code. We believe that the current theory of linearizability suffers from a few biases.

Atomicity: Because the classic definition of linearizability is based on linearizing to an atomic specification, most of the subsequent work on it has focused on atomicity. Even though Filipovic et al. [2010] have noticed that the insight of linearizability lies not in atomicity, but rather in preservation of happens-before order, most of the subsequent work still focuses on atomicity. This is true even though many useful concurrent objects do not linearize, leading to numerous variations on the theme [Castañeda et al. 2015; Goubault et al. 2018; Haas et al. 2016; Neiger 1994].

Compositionality: The typical approach to assembling verified concurrent objects into a larger system relies on a refinement property in the style of Filipovic et al. [2010]. Usually, there is a syntactically defined programming language for expressing concurrent code and often specifications as well. The code is verified by linking a library L'_B with an implementation $N = N_1 \parallel \ldots \parallel N_k$, specified in the programming language, to form a syntactic term $\mathsf{Link}\ L'_B; N$. A trace semantics $[\![-]\!]$ allows one to obtain the traces for the resulting interface $[\![\mathsf{Link}\ L'_B; N]\!]$, and an observational refinement property allows to consider instead a linearized library L_B linked with N to reason about the linearizability of the library that N implements. Now, suppose one is given an implementation M relying on a library L'_A, that is $\mathsf{Link}\ L'_A; M$, to implement L'_B. There is *no* obvious way to compose M and N so to re-use their proofs of linearizability to obtain a linearizable object $\mathsf{Link}\ L'_A; (N \circ M)$. At best, one has to either syntactically link them together, and re-do the proofs, or inline M in N and re-verify the code obtained through this process.

$$\begin{array}{c} N \\ \hline L'_B \\ \hline M \\ \hline L'_A \end{array}$$

Syntax: As outlined in *Compositionality*, there is also a bias towards syntax, even in Filipovic et al. [2010], one of the foundational papers on linearizability. This becomes an issue when different components are modeled by different computational models but need to be connected nonetheless (such as when one wants to model both hardware and software components, or when components are written in different programming languages). This situation occurs in real systems. For instance, Gu et al. [2015, 2016, 2018]'s verified OS contains components in both C and Asm. The way they manage to make the two interact is by only composing components after compiling C code into Asm using CompCert [Leroy 2009], a solution which is yet again reliant on syntactic linking. Less optimistically, there would be no compiler to aid with this. In this context, an entire metatheory for the interaction between the two languages would need to be developed, together with a theory of observational refinement across programming languages. In a large heterogeneous system this becomes unwieldy, as there could be several computational models involved. Meanwhile, a compositional abstract model could embed each heterogeneous component and reason about them at a more coarse-grained level.

Theory: Overall, the theory of linearizability is rather underdeveloped. There are essentially two characterizations: the original happens-before order one from Herlihy and Wing [1990], and the observational refinement one from Filipovic et al. [2010]. Guerraoui and Ruppert [2014] addressed the folklore that linearizability is a safety property, while Goubault et al. [2018] gave a novel formulation of linearizability in terms of local rewrite rules and showed that linearizability may be seen as an approximation operation by proving a certain Galois connection. Otherwise, there isn't a clean theory that addresses the semantic and computational content of linearizability, providing foundations for properties such as locality and observational refinement. As a side-effect of this, the proofs of these properties are rather complicated. A more general and abstract theory of linearizability could not only simplify these issues, but also be more easily adapted to novel settings where there is no obvious happens-before ordering.

Verification: These issues are even more relevant in formal verification, especially when targeting large heterogeneous systems. A recent line of work [Koenig and Shao 2020; Oliveira Vale et al. 2022] maintains that compositional semantics is essential for the scalable verification of such systems. The idea is that individual components are verified in domain specific semantic models targeting fine-grained aspects of computation, and appropriate for the verification task. This is necessary as semantic models for verification are tailored to make the verification task tractable. But then, these components are embedded into a general compositional model, shifting the granularity of computation to the coarse-grained behavior of components. This general model acts as the compositional glue connecting the system together. As linearizability is the main correctness criterion for concurrent objects, a compositional model of linearizable objects is necessary to provide that glue for large, heterogeneous, potentially distributed, concurrent systems.

1.2 Summary and Main Contributions

- In this paper, we develop a compositional model of linearizable concurrent objects. We first construct a concurrent game semantics model (§3). For the sake of clarity, we strive for the simplest game model expressive enough to discuss linearizability: a bare-bones sequential game model interleaved to form a sequentially consistent model of concurrent computation.
- As with other models of concurrent computation, the model in §3 fails to have a *neutral* (or *identity*) element for composition. We remedy this in §4 by using a category-theoretical construction called the Karoubi envelope. We argue that this construction comes with two transformations $K_{\mathsf{Conc}}-$ and $\mathsf{Emb}_{\mathsf{Conc}}-$ converting between the models from §3 and §4.
- Surprisingly, the process of constructing the model in §4 reveals that linearizability is at the heart of compositionality, and in particular we do not need to define linearizability: it emerges out of the abstract construction of a concurrent model of computation, as we discuss in §5. We show this by giving a generalized definition of linearizability and then by showing its tight connection to $K_{\mathsf{Conc}}-$, leading to a novel abstract definition of linearizability.
- We then give a computational interpretation of linearizability in §5.3 by showing that proofs of linearizability correspond to traces of a certain program ccopy.
- Simultaneously, these new foundations reveal that compositionality is also at the heart of linearizability. In §5.4 we give an analogue of the usual contextual refinement result around linearizability which admits an extremely simple proof because of our formalism.
- In §5.5 we revisit Herlihy-Wing's locality result and provide a novel proof of locality based on our computational interpretation and abstract formulation of linearizability, leading to a more structured and algebraic proof of a generalized locality property.
- In §6 we revisit our construction from the point of view of category theory, showing that it can be generalized to other settings with similar structure.
- In §7 we showcase that our model is practical by connecting our semantics with a concrete program logic, and showing how the theory can be used to compose concurrent objects and their implementations together to build larger objects.

We cover some background, motivation, and main results informally in §2. A sequence of appendices, available in a technical report [Oliveira Vale et al. 2022], provides a novel characterization of atomicity, an object-based semantics of linearizable concurrent objects and their implementations, and a more detailed account of our program logic. There, we use this characterization of atomicity to show how our constructions specialize to the corresponding results surrounding classical linearizability and how they compare with interval-sequential linearizability.

2 BACKGROUND AND OVERVIEW

2.1 Background

2.1.1 Game Semantics. Since Herlihy and Wing [1990] was published, many techniques have been developed by the programming languages and the distributed systems communities to model concurrent computation. One technique that has risen to prominence, mostly due to its success in proving full abstraction results for a variety of programming languages, is game semantics [Abramsky et al. 2000; Blass 1992; Hyland and Ong 2000]. Its essence lies in adding more structure to traces, which are called *plays* in the paradigm. These plays describe well-formed interactions between two parties, historically called Proponent (P) and Opponent (O). A game A (or B) provides the rules of the game by describing which plays are valid; types are interpreted as games. As one typically takes the point of view of the Proponent, and models the environment as Opponent, programs of type $A \multimap B$ (a linear program that produces a play from B by interacting with A) are interpreted as *strategies* $\sigma : A \multimap B$ for the Proponent to "play" this game against the Opponent. A strategy is essentially a description of how the Proponent reacts to any move by the Opponent in any context that may arise in their interaction. The standard way of composing strategies informally goes by the motto of "interaction + hiding": given strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ the strategy $\sigma; \tau : A \multimap C$ is constructed by letting σ and τ interact through their common game B, obtaining a well-formed interaction across A, B and C, and then hiding the interaction in B to obtain an interaction that appears to happen only in A and C.

2.1.2 A Surprising Coincidence. Ghica and Murawski [2004] constructed a concurrent game semantics to give a fully abstract model of Idealized Concurrent Algol (ICA). In attempting to construct their model of ICA, they faced a problem: the naive definition of concurrent strategy does not construct a category for lack of an identity strategy. In other words, there is no strategy $\mathbf{id}_A : A \multimap A$ such that $\sigma; \mathbf{id}_A = \sigma$ holds for any strategy $\sigma : A$, a basic property of a compositional model. Their solution was to consider strategies that are "saturated" under a certain rewrite system.

Interestingly, the same rewrite system appears in Goubault et al. [2018]'s work on linearizability. There, they gave an alternative definition of linearizability based on a certain string rewrite system over traces. The key rule of this rewrite system is:

$$h \cdot \boldsymbol{\alpha}{:}m\ \boldsymbol{\alpha'}{:}m' \cdot h' \rightsquigarrow h \cdot \boldsymbol{\alpha'}{:}m'\ \boldsymbol{\alpha}{:}m \cdot h'$$

if and only if $\alpha \neq \alpha'$ and m is an invocation or m' is a return. That is, two events $\boldsymbol{\alpha}{:}m$ and $\boldsymbol{\alpha'}{:}m'$ in a trace $h \cdot \boldsymbol{\alpha}{:}m\ \boldsymbol{\alpha'}{:}m' \cdot h'$ may be swapped when they are events by different threads, α and α', and the swap makes an invocation occur later or a return occur earlier. These swaps precisely encode happens-before order preservation.

Surprisingly, this rewrite relation is an instance of that appearing in Ghica and Murawski [2004]. The coincidence is unexpected, Ghica and Murawski [2004] are simply attempting to construct a compositional model of concurrent computation, without regard for linearizability. They make their model compositional by considering only strategies saturated under a rewrite relation which happens to encode preservation of happens-before order. So why should this rewrite system appear as a result of obtaining an identity for strategy composition?

2.2 An Example on Compositionality

Compositionality is not only important for providing semantics to programming languages, but also for the sake of scalability in formal verification. We now provide a few examples of how compositionality helps profitably organize a verification effort.

2.2.1 Coarse-Grained Locking. We model an object Lock with acq and rel operations which take no arguments and return the value ok. We can encapsulate this information as the signature:

$$\mathsf{Lock} := \{\mathsf{acq} : \mathbf{1}, \mathsf{rel} : \mathbf{1}\}$$

where $\mathbf{1} = \{\mathsf{ok}\}$. We denote by †Lock the type of traces using operations of Lock and by $P_{\dagger\mathsf{Lock}}$ the set of those traces. An example of a concurrent trace in $P_{\dagger\mathsf{Lock}}$ is

$$\begin{tikzcd} \boldsymbol{\alpha_1}\mathbf{:}\mathsf{acq} & \boldsymbol{\alpha_2}\mathbf{:}\mathsf{acq} \arrow[r] & \boldsymbol{\alpha_2}\mathbf{:}\mathsf{ok} \arrow[rr, bend left] & \boldsymbol{\alpha_3}\mathbf{:}\mathsf{acq} \arrow[rr, bend right] & \boldsymbol{\alpha_2}\mathbf{:}\mathsf{acq} \arrow[rr, bend left] & \boldsymbol{\alpha_3}\mathbf{:}\mathsf{ok} & \boldsymbol{\alpha_2}\mathbf{:}\mathsf{ok} \end{tikzcd}$$

this trace linearizes to the following atomic trace, also in $P_{\dagger\mathsf{Lock}}$, called atomic because every invocation immediately receives its response

$$\boldsymbol{\alpha_2}\mathbf{:}\mathsf{acq} \rightarrow \boldsymbol{\alpha_2}\mathbf{:}\mathsf{ok} \rightarrow \boldsymbol{\alpha_2}\mathbf{:}\mathsf{acq} \rightarrow \boldsymbol{\alpha_2}\mathbf{:}\mathsf{ok} \qquad \boldsymbol{\alpha_3}\mathbf{:}\mathsf{acq} \rightarrow \boldsymbol{\alpha_3}\mathbf{:}\mathsf{ok}$$

As usual, concurrent objects are specified by sets of traces. In this way, a concurrent lock object is specified as a prefix-closed set of traces $v'_{\mathrm{lock}} \subseteq P_{\dagger\mathsf{Lock}}$. To be correct this specification v'_{lock} should linearize to the atomic specification $v_{\mathrm{lock}} \subseteq P_{\dagger\mathsf{Lock}}$ given by the set of traces $s \in P_{\dagger\mathsf{Lock}}$ such that:

$$\text{if} \qquad s = s_1 \cdot \boldsymbol{\alpha_1}\mathbf{:}m_1 \cdot \boldsymbol{\alpha_2}\mathbf{:}m_2 \cdot \boldsymbol{\alpha_3}\mathbf{:}m_3 \cdot \boldsymbol{\alpha_4}\mathbf{:}m_4 \cdot s_2 \qquad \text{then}$$

- If $m_1 = \mathsf{acq}$ then $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4$ and $m_2 = m_4 = \mathsf{ok}$ and $m_3 = \mathsf{rel}$;
- If $m_1 = \mathsf{rel}$ then $\alpha_1 = \alpha_2$, $\alpha_3 = \alpha_4$, $m_3 = \mathsf{acq}$ and $m_2 = m_4 = \mathsf{ok}$;

and moreover, if s is non-empty, then its first invocation is acq. We take the convention that a primed specification is more *concurrent* than its un-primed counterpart.

A typical application of a lock is synchronizing accesses to a resource shared by several asynchronous computational agents. For instance, suppose we have a sequential queue with signature:

$$\mathsf{Queue} := \{\mathsf{enq} : \mathbb{N} \rightarrow \mathbf{1}, \mathsf{deq} : \mathbb{N} + \{\varnothing\}\}$$

Its concurrent specification v'_{queue} can be specified as the largest set of traces $s \in P_{\dagger\mathsf{Queue}}$ such that if $s = p \cdot \boldsymbol{\alpha}\mathbf{:}\mathsf{deq} \cdot \boldsymbol{\alpha}\mathbf{:}k \cdot s'$ and p is atomic then either $\mathrm{qstate}(p) = k :: q'$ or $\mathrm{qstate}(p) = [\ ]$ and $k = \varnothing$, where qstate is an inductively defined function taking an atomic trace p and returning the state $\mathrm{qstate}(p)$ of the queue after executing the trace p from the empty queue $[\ ]$. Note that as soon as any non-atomic interleaving happens in a trace of v'_{queue} the behaviors of enq and deq are unspecified and therefore completely non-deterministic. This reflects the assumption that this Queue object is a sequential implementation that is not resilient to concurrent execution.

Such a Queue object can be shared across several agents by locking around all the operations of Queue, as demonstrated in the following implementation $M_{\mathrm{squeue}} : \mathsf{Lock} \otimes \mathsf{Queue} \multimap \mathsf{Queue}$ implementing a shared queue using a lock and a sequential queue implementation (see Fig. 1). Note that when several independent objects must be used together, we use the linear logic tensor product $- \otimes -$ to compose them horizontally into a new object, such as in the source type of M_{squeue}.

The queue object v'_{squeue} implemented by M_{squeue} is linearizable to the usual atomic specification v_{squeue} of a Queue. But observe that v'_{queue} is not linearizable to v_{squeue}. This means that the composition of v'_{lock} and v'_{queue} into an object of type $\mathsf{Lock} \otimes \mathsf{Queue}$ specified as $v'_{\mathrm{lock}} \otimes v'_{\mathrm{queue}}$ (the set of all sequentially consistent interleavings of v'_{lock} and v'_{queue}) is also not linearizable to an atomic specification. This is enough for approaches which are over-reliant on atomicity to be unable to handle this situation cleanly. A solution there is to remove the dependence on the non-linearizable queue by inlining its implementation in terms of programming language primitives. This solution is unfortunate, as intuitively what M_{squeue} does is turn a non-linearizable queue into a linearizable one. Inlining its implementation removes the connection between the sequential implementation and the code implementing this sharing pattern. Instead, what one would like to do is to be able to use off-the-shelf sequential components freely, just like in the code in Fig. 1. Meanwhile, by divorcing

```
Msqueue :
Import Q:Queue
Import L:Lock

enq(k) {              deq() {
  L.acq();                 L.acq();
  r <- Q.enq(k);           r <- Q.deq();
  L.rel();                 L.rel();
  ret r                  ret r
}                     }
```

```
Mlock :
Import F:FAI
Import C:Counter
Import Y:Yield

acq() {                                 rel() {
  my_tick <- F.fai();                     C.inc();
  while (cur_tick ≠ my_tick) {            ret ok
        Y.yield();                      }
        cur_tick <- C:get()
  }
  ret ok
}
```

Fig. 1. Shared Queue implementation (left), and Lock implementation (right)

linearizability from atomicity, we will still have that $v'_{\text{lock}} \otimes v'_{\text{queue}}$ is linearizable to $v_{\text{lock}} \otimes v'_{\text{queue}}$ according to a generalized notion of linearizability. We connect our model with a program logic to show that the code in Fig. 1 does implement a linearizable Queue object correctly.

2.2.2 Implementing a Lock. A typical implementation for Lock is the ticket lock implementation (see Fig. 1), relying on a sequential counter and a fetch-and-increment object with signatures

$$\mathsf{Counter} := \{\mathsf{inc} : \mathsf{ok}, \mathsf{get} : \mathbb{N}\} \qquad\qquad \mathsf{FAI} := \{\mathsf{fai} : \mathbb{N}\}$$

The FAI object comprises a single operation fai which both returns the current value of the fetch-and-increment object and increments it. It is well known that the concurrent v'_{fai} object specification is linearizable to an atomic one v_{fai}.

The Counter object v'_{counter} has a subtler specification. It models a semi-racy sequential counter implementation similarly to the queue from §2.2.1. But different from the racy queue, the counter must be slightly more defined, as the lock implementation requires that the sequential implementation be resilient to concurrent get calls, and with respect to concurrent get and inc calls. However, if inc calls happen concurrently, the behavior is undefined. This is not an issue for the lock implementation because it never happens in a valid execution of a lock. We model this by assuming that the concurrent specification of the Counter, v'_{counter}, is *linearizable* (in our generalized sense) with respect to a *less* concurrent one, v_{counter}, given by the largest set of traces $s \in P_{\dagger\mathsf{Counter}}$ satisfying:

If $s = p \cdot \boldsymbol{\alpha}{:}\mathsf{get} \cdot m \cdot s'$ then $m = \boldsymbol{\alpha}{:}k$ and if moreover $p\restriction_{\{\mathsf{inc:ok}\}}$ is atomic and even-length then $k = \#\mathsf{inc}(p)$,

where $\#\mathsf{inc}(-)$ is an inductively defined function returning the number $\#\mathsf{inc}(p)$ of inc calls in p. Note that we do not bother defining what v'_{counter} actually is, as our proofs, using a refinement property *à la* Filipovic et al. [2010], will only rely on the linearized strategy v_{counter}.

Occasionally, one implements the ticket lock so that it yields while spinning so to let other agents get access to the underlying computational resource (such as processor time). For some purposes, this is crucial to obtain better liveness properties. For this, we define a signature

$$\mathsf{Yield} := \{\mathsf{yield} : \mathbf{1}\}$$

with concurrent specification v'_{yield} given by

$$v'_{\text{yield}} := \{s \in P_{\dagger\mathsf{Yield}} \mid s = s_1 \cdot \boldsymbol{\alpha}{:}\mathsf{yield} \cdot s_2 \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot s_3 \Rightarrow \text{there is a pending yield in } s_1 \cdot s_2\}$$

that is to say, a call by α to yield is only allowed to return if another agent calls yield concurrently with α. A typical trace of v'_{yield} looks like:

$$\begin{tikzcd}
\boldsymbol{\alpha_1}{:}\mathsf{yield} \arrow[rrrr, bend right] & \boldsymbol{\alpha_2}{:}\mathsf{yield} \arrow[r] & \boldsymbol{\alpha_2}{:}\mathsf{ok} \arrow[rrr, bend left] & \boldsymbol{\alpha_3}{:}\mathsf{yield} \arrow[rrr, bend left] & \boldsymbol{\alpha_1}{:}\mathsf{ok} & \boldsymbol{\alpha_2}{:}\mathsf{yield} & \boldsymbol{\alpha_3}{:}\mathsf{ok}
\end{tikzcd}$$

Now, observe that by definition, v'_{yield} contains no atomic traces, as yield only returns if another yield happens concurrently with it. That means that no *atomic* linearized specification for v'_{yield} will be faithful to its actual behaviors. Despite that, its traces can always be simplified, while preserving happens-before-order, so that between a yield invocation and its return ok the only events that appear are the ok for the agent who took over the computational resource and the yield call for the agent who yielded, like so:

$$\begin{tikzcd}
\alpha_2\text{:yield} \arrow[rr, bend right] & \alpha_1\text{:yield} \arrow[rrr, bend left] & \alpha_2\text{:ok} \arrow[rrr, bend right] & \alpha_3\text{:yield} \arrow[rrr, bend left] & \alpha_1\text{:ok} & \alpha_2\text{:yield} & \alpha_3\text{:ok}
\end{tikzcd}$$

That is to say, Yield is linearizable (in our sense) to a non-atomic specification, and we can still use our observational refinement property to simplify the reasoning on the side of the client of Yield. With the Yield object at hand, we verify that the implementation in Fig. 1 for the ticket lock is linearizable using a program logic. Once M_{lock} and M_{squeue} are individually verified, we can use a vertical composition operation $-;-$ to compose them into a program implementing the shared Queue directly on top of FAI, Counter and Yield while preserving the fact that this composed implementation implements a linearizable Queue object. We depict this example in Fig. 2.

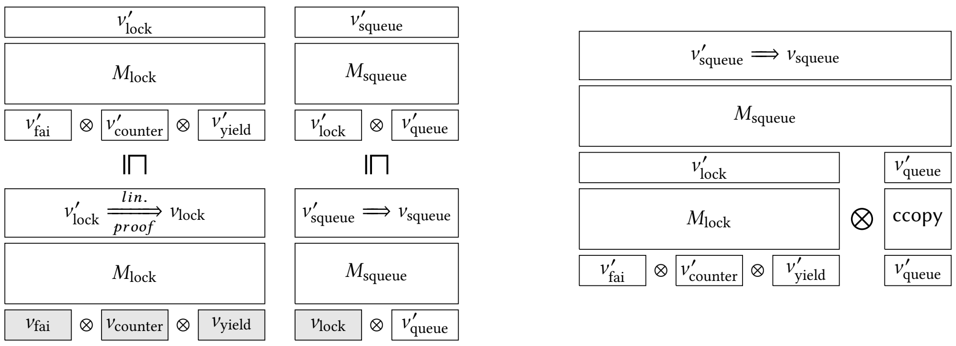
v'_{lock}
M_{lock}
$v'_{\text{fai}} \otimes v'_{\text{counter}} \otimes v'_{\text{yield}}$
$\sqsubseteq$
$v'_{\text{lock}} \xRightarrow[\text{proof}]{\text{lin.}} v_{\text{lock}}$
M_{lock}
$v_{\text{fai}} \otimes v_{\text{counter}} \otimes v_{\text{yield}}$
v'_{squeue}
M_{squeue}
$v'_{\text{lock}} \otimes v'_{\text{queue}}$
$\sqsubseteq$
$v'_{\text{squeue}} \Longrightarrow v_{\text{squeue}}$
M_{squeue}
$v_{\text{lock}} \otimes v'_{\text{queue}}$
$v'_{\text{squeue}} \Longrightarrow v_{\text{squeue}}$
M_{squeue}
v'_{lock}
v'_{queue}
M_{lock}
$\otimes$
ccopy
$v'_{\text{fai}} \otimes v'_{\text{counter}} \otimes v'_{\text{yield}}$
v'_{queue}

Fig. 2. In our compositional model, off-the-shelf components can be composed horizontally by using the linear logic tensor product $- \otimes -$. Each component's implementation is verified against its linearized specification individually (left). Refinement and generalized linearizability allow to use the simpler specifications v_{fai}, and v_{yield} to prove that v'_{lock}, implemented by M_{lock} is linearizable to v_{lock}. By assuming v'_{counter} linearizable to the specification v_{counter}, it is unnecessary to know the actual concurrent behavior of the racy counter. Vertical composition (right) allows one to compose the two implementations together to obtain a fully concurrent description of the composed system while maintaining that after the composition v'_{squeue} is still linearizable to v_{squeue}. We use ccopy to denote the neutral (or identity) element for composition, discussed in §3.2.

2.3 Overview

Our work will address the question raised at §2.1.2 by showing that linearizability is already baked in a compositional model of computation. Crucially, our goal is to show that a model of concurrent computation with enough structure naturally gives rise to its own notion of linearizability, and that linearizability is intrinsically connected to the compositional structure of the model.

For this, we define a model of sequentially consistent, potentially blocking, concurrent computation $\underline{\mathbf{Game}}_{\mathbf{Conc}}$, inspired by Ghica and Murawski [2004]. Similarly to their model, this model fails to have a neutral element for composition $-;-$. An abstract construction called the Karoubi envelope allows us to construct from $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ a compositional model $\mathbf{Game}_{\mathbf{Conc}}$ which does have neutral elements. This new model $\mathbf{Game}_{\mathbf{Conc}}$ differs from $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ in that its strategies σ of type

$\mathbf{A} \multimap \mathbf{B}$ are strategies of $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ that moreover are invariant upon composition with a certain strategy called ccopy_. This strategy corresponds to the traces of a program where each agent in the concurrent system runs the code in Fig. 3 in parallel, which implements f by importing an implementation of f itself, or alternatively to an η-redex $\lambda x.f\ x$.

```
Import F            ||     ||  Import F
                    ||     ||
f(a) {              ||     ||  f(a) {
  r <- F.f(a);      || ... ||    r <- F.f(a);
  ret r             ||     ||    ret r
}                   ||     ||  }
      ⋮             ||     ||        ⋮
```

$$\begin{tikzcd}
\underline{\mathbf{Game}}_{\mathbf{Conc}} \arrow[r, bend left, "K_{\mathrm{Conc}}"] & \mathbf{Game}_{\mathbf{Conc}} \arrow[l, bend left, "\mathrm{Emb}_{\mathrm{Conc}}"]
\end{tikzcd}$$

Fig. 3. Code corresponding to ccopy_ (left); Diagram depicting the operations K_{Conc} and $\mathrm{Emb}_{\mathrm{Conc}}$ (right)

This construction comes with some infrastructure: a saturation operation K_{Conc} and a forgetful operation $\mathrm{Emb}_{\mathrm{Conc}}$, depicted in Fig. 3. Importantly, $K_{\mathrm{Conc}}\ \sigma$ is *defined* to be $\mathrm{ccopy}_{\mathbf{A}}; \sigma; \mathrm{ccopy}_{\mathbf{B}}$ while $\mathrm{Emb}_{\mathrm{Conc}}\ \sigma$ is *by definition* just σ itself. The central but simple result of this paper is that

Proposition 2.1 (Abstract Linearizability). *A strategy $\sigma : \mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}$ is linearizable to a strategy $\tau : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ if and only if*

$$\sigma \subseteq K_{\mathrm{Conc}}\ \tau$$

By *linearizability* we mean a generalized, but *concrete*, definition of linearizability which nonetheless faithfully generalizes Herlihy-Wing linearizability when τ is an atomic strategy. It is important to emphasize that because K_{Conc} arises from the Karoubi envelope construction, not only it does not involve happens-before ordering, but also it immediately suggests an *abstract* definition of linearizability which could be sensible anywhere this abstract construction is used.

We give yet another characterization of linearizability by showing that the strategy ccopy_, corresponds to proofs of linearizability, giving a computational interpretation to proofs of linearizability (where $s \upharpoonright_{\mathbf{A}}$ denotes the projection of the trace s to events of $\mathbf{A}$). We call this a *computational* interpretation because ccopy_ is the denotation of the concrete program in Fig. 3.

Proposition 2.2 (Computational Interpretation). *s_1 linearizes to s_0, both plays of type $\mathbf{A}$, if and only if there exists a play $s \in \mathrm{ccopy}_{\mathbf{A}}$ such that*

$$s \upharpoonright_{\mathbf{A}_0} = s_0 \qquad\qquad s \upharpoonright_{\mathbf{A}_1} = s_1$$

Then, we show a property analogous to the usual contextual refinement property, admitting a very simple proof due to the abstract formalism we develop.

Proposition 2.3 (Interaction Refinement). *$\nu'_A : \mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}$ is linearizable to $\nu_A : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ if and only if for all concurrent games $\mathbf{B}$ and $\sigma : \mathbf{A} \multimap \mathbf{B}$ it holds that*

$$\nu'_A; \sigma \subseteq \nu_A; \sigma$$

After that, we define a tensor product $\mathbf{A} \otimes \mathbf{B}$ amounting to all the sequentially consistent interleavings of traces of type $\mathbf{A}$ with traces of type $\mathbf{B}$. We then use the insight given by the computational interpretation of linearizability proofs and show that for any $\mathbf{A}$ and $\mathbf{B}$:

$$\mathrm{ccopy}_{\mathbf{A} \otimes \mathbf{B}} = \mathrm{ccopy}_{\mathbf{A}} \otimes \mathrm{ccopy}_{\mathbf{B}}$$

This equation can be interpreted to say that proofs of linearizability for objects of type $\mathbf{A} \otimes \mathbf{B}$ correspond to a pair of a proof of linearizability for the $\mathbf{A}$ part and a separate proof of linearizability for the **B** part. We use this insight to give a more general account of the locality property originally appearing in Herlihy and Wing [1990], obtaining as a corollary the following locality property:

Proposition 2.4 (Locality). *Let* $v'_A : \mathbf{A}$, $v'_B : \mathbf{B}$ *and* $v_A : \mathbf{A}$, $v_B : \mathbf{B}$. *Then*

$$v' = v'_A \otimes v'_B \text{ is linearizable w.r.t. } v = v_A \otimes v_B$$
$$\text{if and only if}$$
$$v'_A \text{ is linearizable w.r.t. } v_A \text{ and } v'_B \text{ is linearizable w.r.t. } v_B$$

Perhaps more important than the property itself is the methodology we use to establish it. Rather than the usual argument using partial orders, originally from Herlihy and Wing [1990] and also appearing in a setting closer to ours in Castañeda et al. [2015], we give an algebraic proof relying on the abstract definition of linearizability from Prop. 2.1.

At this point we will have all the ingredients to compose concurrent objects into larger systems, such as in the example in Fig. 2. We showcase this by using a program logic to verify individual components. Vertical composition corresponds to strategy composition $-;-$. Horizontal composition is provided by the tensor product $- \otimes -$ which is well-behaved with respect to linearizability due to the locality property. As our model is enriched over a simple notion of refinement, we will also have that these constructions are harmonious with refinement. The interaction refinement property allows us to leverage the linearized specification of components to ease reasoning.

3 CONCURRENT GAMES

In this section, we define our model of concurrent games, built by interleaving several copies of a sequential game model. We start by defining a simple model of sequential games $\mathbf{Game_{Seq}}$ in §3.1. Then, we define the interleaved model $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ in §3.2 and observe that it defines a semicategory.

3.1 Sequential Games

Before we proceed, we briefly define a sequential game model. Similar models appear elsewhere in the literature. See, for instance Abramsky and McCusker [1999]; Hyland [1997], which we suggest for the reader who seeks a detailed treatment. Our concurrent model amounts to interleaving several sequential agents which behave as in the sequential game model we define now.

As we outlined in §2.1.2, types are interpreted as games. In the following definitions $\mathsf{Alt}(S, S')$ is the set of sequences of $S + S'$ that alternate between S and S', $\sqsubseteq$ is the prefix relation, and $\sqsubseteq_{\mathsf{even}}$ is the even-length prefix relation.

Definition 3.1. A *(sequential) game* A is a pair (M_A, P_A) of a set of polarized *moves* $M_A = M_A^O + M_A^P$ and a non-empty, prefix-closed, set of alternating sequences $P_A \subseteq \mathsf{Alt}(M_A^O, M_A^P)$ of M_A, called *plays*, such that every non-empty play $s \in P_A$ starts with a move in M_A^O.

The moves in M_A^O are the Opponent moves, and those in M_A^P the Proponent moves. Every sequential game A defines a labeling map $\lambda_A : M_A \to \{O, P\}$ by the universal property of the sum.

An example of a game is the unit game Σ in which Opponent is allowed to ask a question q which Proponent may answer with a response a. In this way, $M_\Sigma^O = \{q\}$ and $M_\Sigma^P = \{a\}$, and Σ admits exactly the following three plays:

$$P_\Sigma := \{ \quad \epsilon \quad , \quad q \quad , \quad q \longrightarrow a \quad \}$$

corresponding to the empty play, the play where Opponent has asked q and waits for a response from the Proponent, and a play where Proponent has replied.

Games can be composed together to form new games. Of particular importance for us will be the tensor $A \otimes B$ of two games A and B, and the linear implication $A \multimap B$. In the following, we denote by $s \upharpoonright_A$ the projection of s to its largest subsequence containing only moves of the game A.

Definition 3.2. Let A and B be (sequential) games. The tensor product of A and B is the game $A \otimes B = (M_{A\otimes B}, P_{A\otimes B})$ defined by:

$$M^O_{A\otimes B} := M^O_A + M^O_B \quad M^P_{A\otimes B} := M^P_A + M^P_B \quad P_{A\otimes B} := \{s \in \mathsf{Alt}(M^O_{A\otimes B}, M^P_{A\otimes B}) \mid s{\restriction_A} \in P_A \wedge s{\restriction_B} \in P_B\}$$

The game $A \multimap B = (M_{A\multimap B}, P_{A\multimap B})$ is defined by:

$$M^O_{A\multimap B} := M^P_A + M^O_B \quad M^P_{A\multimap B} := M^O_A + M^P_B \quad P_{A\multimap B} := \{s \in \mathsf{Alt}(M^O_{A\multimap B}, M^P_{A\multimap B}) \mid s{\restriction_A} \in P_A \wedge s{\restriction_B} \in P_B\}$$

The plays of $A \otimes B$ are essentially plays of A and B interleaved in a sequential play, so that $A \otimes B$ corresponds to independent horizontal composition. The game $A \multimap B$ meanwhile corresponds to switching the roles of Opponent and Proponent in A and then taking the tensor with B.

As a matter of illustration, the maximal plays (under prefix ordering) for the games $\Sigma_0 \otimes \Sigma_1$ (the two plays on the left) and $\Sigma_0 \multimap \Sigma_1$ (the two plays on the right) are depicted below. We denote by Σ_0, Σ_1 the two components of these types, both of which are instances of the game Σ. We will similarly add an index to the moves of each component.

$$\begin{tikzcd}
\Sigma_1 & & & q_1 \arrow[r] & a_1 & q_1 \arrow[r] & a_1 \arrow[ddr] & & \\
\otimes & & & & & & & & \\
\Sigma_0 & q_0 \arrow[r] & a_0 \arrow[uur] & & & & & q_0 \arrow[r] & a_0
\end{tikzcd}
\qquad
\begin{tikzcd}
\Sigma_1 & q_1 \arrow[r] & a_1 & q_1 \arrow[ddr] & & & a_1 \\
\multimap & & & & & & \\
\Sigma_0 & & & & q_0 \arrow[r] & a_0 \arrow[uur] &
\end{tikzcd}$$

Observe that in the game $\Sigma \otimes \Sigma$ Opponent can choose to start in either component, while in the game $\Sigma \multimap \Sigma$ Opponent must start in the target component (Σ_1) due to the flip of polarity in the source component (Σ_0). In $\Sigma \otimes \Sigma$ only Opponent may switch components, while in $\Sigma \multimap \Sigma$ only Proponent may switch components because of alternation (these are typically called the switching conditions of sequential games).

Continuing along what we outlined in §2.1.2, programs are interpreted as strategies.

Definition 3.3. A (sequential) strategy σ over the game A, denoted $\sigma : A$, consists of a non-empty, prefix-closed and O-receptive set of plays in P_A, where O-receptivity is defined as:

$$\text{If } s \in \sigma, \text{ Opponent to move at } s \text{ and } s \cdot a \in P_A, \text{ then } s \cdot a \in \sigma$$

A morphism between sequential games A and B will then be defined as a strategy for the game $A \multimap B$. Strategy composition is defined as usual by "interaction + hiding". Formally,

Definition 3.4. Given games A, B, C we define the set $\mathsf{int}(A, B, C)$ of finite sequences of moves from $M_A + M_B + M_C$ as follows:

$$s \in \mathsf{int}(A, B, C) \iff s{\restriction_{A,B}} \in P_{A\multimap B} \wedge s{\restriction_{B,C}} \in P_{B\multimap C}$$

The interaction $\mathsf{int}(\sigma, \tau)$ of two strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ is given by the set

$$\mathsf{int}(\sigma, \tau) := \{s \in \mathsf{int}(A, B, C) \mid s{\restriction_{A,B}} \in \sigma \wedge s{\restriction_{B,C}} \in \tau\}$$

And finally, the composition $\sigma; \tau$ is defined as:

$$\sigma; \tau := \{s{\restriction_{A,C}} \mid s \in \mathsf{int}(\sigma, \tau)\}$$

Proposition 3.5. *Strategy composition is well-defined and associative.*

The neutral element for strategy composition is the (sequential) copycat strategy.

Definition 3.6. The (sequential) copycat strategy $\mathsf{copy}_A : A \multimap A$ is defined as

$$\mathsf{copy}_A := \{s \in P_{A\multimap A} \mid \forall p \sqsubseteq_{\mathsf{even}} s.p{\restriction_{A_1}} = p{\restriction_{A_2}}\}$$

Proposition 3.7. *The copycat strategy is the neutral element for strategy composition.*

We collect these results as the category $\mathbf{Game}_{\mathsf{Seq}}$ of sequential games defined in the following.

Definition 3.8. The category $\mathbf{Game_{Seq}}$ of sequential games and (sequential) strategies is the category whose objects are sequential games A, B, C and whose morphisms are strategies $\sigma : A \multimap B$, $\tau : B \multimap C$. Strategy composition is given by $\sigma;\tau : A \multimap C$ and the neutral elements for strategy composition are given by the copycat strategy $\mathsf{copy}_A : A \multimap A$.

A useful example of sequential games to keep in mind are games associated to effect signatures.

Definition 3.9. An effect signature is given by a collection of operations, or effects, $E = (e_i)_{i \in I}$ together with an assignment $\mathsf{ar}(-) : E \longrightarrow \mathbf{Set}$ of a set for each operation in E. This is conveniently described by the following notation:

$$E = \{e_i : \mathsf{ar}(e_i) \mid i \in I\}$$

Cursorily, we can define a game $\mathbf{Game_{Seq}}(E)$ associated with an effect signature E as the game which has as O moves the set of effects $e \in E$ and as P moves the set $\cup_{e \in E} \mathsf{ar}(e)$ of arities in E. We take the freedom of writing E for $\mathbf{Game_{Seq}}(E)$. The typical plays of E appear below in the left and consist of an invocation of an effect $e \in E$ followed by a response $v \in \mathsf{ar}(e)$.

$$E : \quad e \longrightarrow v \qquad\qquad \dagger E : \quad e_1 \rightarrow v_1 \rightarrow e_2 \rightarrow v_2 \rightarrow \dots \rightarrow e_n \rightarrow v_n$$

We can lift such a game E to a game $\dagger E$ that allows several effects of E to be invoked in sequence. Its plays, depicted above on the right, consist of sequences of invocations $e_i \in E$ alternating with their responses $v_i \in \mathsf{ar}(e_i)$. The examples in §2.2 were all specified using effect signatures. It is easy to observe that $\dagger E$ accurately captures the type of sequential traces of an object with E as interface.

For example, the game corresponding to the Counter signature defined in §2.2 has as maximal plays the plays depicted below on the left. †Counter allows for several plays of Counter to be played in sequence. Note, however, that it merely specifies the shape of the interactions with †Counter. Two plays of †Counter are displayed on the right.

$$\begin{array}{ll|l} & \mathsf{inc} \rightarrow \mathsf{ok} & \mathsf{get} \longrightarrow 3 \longrightarrow \mathsf{inc} \rightarrow \mathsf{ok} \rightarrow \mathsf{get} \rightarrow 7 \rightarrow \mathsf{get} \longrightarrow 2 \longrightarrow \mathsf{inc} \\ \forall n \in \mathbb{N}. & \mathsf{get} \longrightarrow n & \mathsf{inc} \rightarrow \mathsf{ok} \rightarrow \mathsf{get} \longrightarrow 1 \longrightarrow \mathsf{get} \rightarrow 1 \rightarrow \mathsf{inc} \rightarrow \mathsf{ok} \end{array}$$

3.2 Concurrent Games

We assume as a parameter a countable set of agent names Υ. These names will be used to distinguish different agents playing a concurrent game $\boldsymbol{A}$. We are now ready to define concurrent games.

Definition 3.10. A concurrent game $\boldsymbol{A} = (M_{\mathbf{A}}, P_{\mathbf{A}})$ is defined in terms of an underlying sequential game $A = (M_A, P_A)$ in the following way:

- Its set of moves $M_{\mathbf{A}}$ is given by the disjoint sum $M_{\mathbf{A}} := \sum_{\alpha \in \Upsilon} M_A$. That is to say, its moves are of the form $\boldsymbol{\alpha}\!:\!m \in M_{\mathbf{A}}$ for any agent $\alpha \in \Upsilon$ and move $m \in M_A$.
- Its set of plays $P_{\boldsymbol{A}}$ is the set $P_{\boldsymbol{A}} := P_A^{\Phi}$ of self-interleaving of plays of the sequential game A.

Formally, denote by $s \parallel t$ the set of interleavings of the finite sequences s and t. Given sets of finite sequences S, T, we define the set of interleavings $S \parallel T$ and the set of self-interleavings S^{Φ}:

$$S \parallel T := \bigcup_{s \in S, t \in T} s \parallel t \qquad\qquad S^{\Phi} := \bigcup_{n \in \mathbb{N}} \bigcup_{\{\alpha_1, \dots, \alpha_n\} \in \mathcal{P}^n(\Upsilon)} (\iota_{\alpha_1}(S) \parallel \dots \parallel \iota_{\alpha_n}(S))$$

where $\mathcal{P}^n(\Upsilon)$ denotes the set of subsets of Υ of size n, and $\iota_\alpha(s)$ labels every move m in s, of every sequence $s \in S$ with the label α denoted by $\boldsymbol{\alpha}\!:\!m$.

The sequential game A is the game that each agent $\alpha \in \Upsilon$ plays locally. We denote by $\pi_\alpha(s)$ the projection of a concurrent play $s \in P_{\boldsymbol{A}}$ to the local play $\pi_\alpha(s)$ by agent α. In particular, for any play $s \in P_{\mathbf{A}}$, $\pi_\alpha(s) \in P_A$. Observe that a concurrent game $\mathbf{A}$ with underlying sequential game

$A = (M_A, P_A)$ is completely determined by its underlying sequential game A per the formula $\mathbf{A} = (\sum_{\alpha \in \Upsilon} M_A, P_A^{\Phi})$. Because of this, it is convenient to write $\mathbf{A} = (M_A, P_A)$ when specifying a concurrent game, as we will do for the rest of the paper.

Along the lines of our sequential game model $\mathbf{Game_{Seq}}$ we now define the notion of a (concurrent) strategy over a (concurrent) game $\boldsymbol{A}$.

Definition 3.11. Let $\boldsymbol{A} = (M_A, P_A)$ be a concurrent game. A (concurrent) strategy σ over $\boldsymbol{A}$, denoted $\sigma : \boldsymbol{A}$, is a non-empty, prefix-closed, O-receptive subset of $P_{\mathbf{A}}$, where O-receptivity is defined by:

$$\text{If } s \in \sigma,\ o \text{ an Opponent move and } s \cdot o \in P_{\mathbf{A}}, \text{ then } s \cdot o \in \sigma.$$

The definition of a concurrent strategy is mostly analogous to that of a sequential game. In fact, $\pi_\alpha(\sigma)$ is a sequential strategy over the sequential game A for every $\alpha \in \Upsilon$. We again defined morphisms by first defining an implication game $\boldsymbol{A} \multimap \boldsymbol{B}$, which simply instantiates the underlying sequential game as the sequential implication game.

Definition 3.12. Given concurrent games $\boldsymbol{A} = (M_A, P_A)$ and $\boldsymbol{B} = (M_B, P_B)$,where $A = (M_A, P_A)$ and $B = (M_B, P_B)$ are sequential games, we define the concurrent game $\boldsymbol{A} \multimap \boldsymbol{B}$ as:

$$\boldsymbol{A} \multimap \boldsymbol{B} := (M_{A \multimap B}, P_{A \multimap B})$$

Strategy composition is defined analogously to the sequential case.

Definition 3.13. Given concurrent games $\mathbf{A} = (M_A, P_A), \mathbf{B} = (M_B, P_B), \mathbf{C} = (M_C, P_C)$ we define the set $\mathsf{int}(\mathbf{A}, \mathbf{B}, \mathbf{C})$ of finite sequences of moves from $M_{\mathbf{A}} + M_{\mathbf{B}} + M_{\mathbf{C}}$ as follows:

$$s \in \mathsf{int}(\mathbf{A}, \mathbf{B}, \mathbf{C}) \iff s\restriction_{\mathbf{A},\mathbf{B}} \in P_{\mathbf{A} \multimap \mathbf{B}} \wedge s\restriction_{\mathbf{B},\mathbf{C}} \in P_{\mathbf{B} \multimap \mathbf{C}}$$

Then, the parallel interaction $\mathsf{int}(\sigma, \tau)$ of two strategies $\sigma : \mathbf{A} \multimap \mathbf{B}$ and $\tau : \mathbf{B} \multimap \mathbf{C}$ is the set

$$\mathsf{int}(\sigma, \tau) := \{ s \in \mathsf{int}(\mathbf{A}, \mathbf{B}, \mathbf{C}) \mid s\restriction_{\mathbf{A},\mathbf{B}} \in \sigma \wedge s\restriction_{\mathbf{B},\mathbf{C}} \in \tau \}$$

And finally, the composition $\sigma; \tau$ is defined as:

$$\sigma; \tau := \{ s\restriction_{\mathbf{A},\mathbf{C}} \mid s \in \mathsf{int}(\sigma, \tau) \}$$

Proposition 3.14. *Strategy composition is well-defined and associative.*

Prop. 3.14 establishes a semicategorical structure to concurrent games and strategies (recall that a semicategory is a category without the requirement of neutral elements for composition).

Definition 3.15. The semicategory $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ has concurrent games $\mathbf{A}$,$\mathbf{B}$ as objects and concurrent strategies $\sigma : \mathbf{A} \multimap \mathbf{B}$ as morphisms. Composition is given by $-;-$.

We define the game $\dagger\mathbf{E}$ of concurrent traces over the signature E by first defining $\mathbf{E} := (M_E, P_E)$ and then $\dagger\mathbf{E} := (M_{\dagger E}, P_{\dagger E})$. So the game $\dagger\mathbf{E}$ has each agent playing the corresponding sequential game $\dagger E$ concurrently. This justifies all the notation used in §2.2, and in particular all the traces depicted serve as examples of plays of games $\dagger\mathbf{E}$ for the respective effect signatures. Effect signatures as games and the replay modality $\dagger-$ admit a rich theory. We treat it in more detail in our technical report [Oliveira Vale et al. 2022].

4 CONCURRENT GAMES AND SYNCHRONIZATION

In §3.2, we defined a concurrent game semantics modeling potentially blocking sequentially consistent computation and we noted that we obtain a semicategorical structure. In this section we discuss the issue with neutral elements (§4.1) and present a solution by constructing from the semicategory $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ a category $\mathbf{Game_{Conc}}$ of concurrent games (§4.2), presented abstractly, and discuss some infrastructure around it (§4.3,§4.4). We finalize by adapting a result of Ghica and Murawski [2004] which allows us to give a concrete characterization of this category (§4.5).

4.1 The Copycat Strategy

In order to appreciate the difficulty with neutral elements in concurrent models, one must first understand what such a neutral element looks like. So let's first ground the discussion on sequential computation. As we saw in §3.1, the neutral element in $\mathbf{Game}_{\mathbf{Seq}}$ is the copycat strategy $\mathrm{copy}_{\_}$. The name comes from the fact that it replicates O moves from the target component to the source component and replicates P moves from the source component to the target component. In the case of $\mathrm{copy}_\Sigma : \Sigma \multimap \Sigma$ there is only one possible interaction (displayed on the left):

$$\begin{tikzcd}
\Sigma & q \arrow[ddr, bend right] & & & a \\
\multimap & & & & \\
\Sigma & & q \arrow[r] & a \arrow[uur, bend right] &
\end{tikzcd}$$

```
Import Σ

q () {
  a <- q
  ret a
}
```

Fig. 4. Maximal play of copy_Σ (left) and corresponding pseudocode (right)

All other plays of copy_Σ are prefixes of this play. This strategy corresponds to the implementation displayed on the right of Fig. 4, for the method q using a library that already implements the method q. Suppose we compose the copycat with itself, that is, we build the strategy $\mathrm{copy}_\Sigma;\mathrm{copy}_\Sigma$, and recall the motto "interaction + hiding". The resulting interaction prior to hiding is:

$$\begin{tikzcd}
\Sigma & q \arrow[ddr, bend left] & & & & & a \\
\multimap & & & & & & \\
\Sigma & & q \arrow[ddr, bend right] & & & a \arrow[uur, bend left] & \\
\multimap & & & & & & \\
\Sigma & & & q \arrow[r] & a \arrow[uur, bend right] & &
\end{tikzcd}$$

```
Import Σ₀

q () {
  a <- (a' <- q; ret a')
  ret a
}
```

Fig. 5. Maximal play of $\mathrm{int}(\mathrm{copy}_\Sigma, \mathrm{copy}_\Sigma)$ (left) and corresponding pseudocode (right).

The middle row of the interaction is the one that is then hidden. It simultaneously plays the role of the source of the play in the top two rows, and the target in the play in the bottom two rows. The resulting interaction, after hiding, is the interaction from Fig. 4, as expected. In terms of the corresponding implementations composing the two strategies amounts to inlining the code of one into the other, as depicted in the right of Fig. 5.

In the concurrent version $\Sigma \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ of Σ each agent of Υ locally plays Σ. The obvious neutral element in this situation would be to have each agent $\alpha, \alpha' \in \Upsilon$ locally run copy_Σ, a strategy we call $\mathrm{ccopy}_\Sigma : \Sigma \multimap \Sigma$, which is akin to linking the code from Fig. 4 for each agent in Υ. ccopy_Σ therefore consists of all plays which are interleavings of copy_Σ. One such play is the play t displayed below:

$$\begin{tikzcd}
\Sigma & \boldsymbol{\alpha}{:}q \arrow[ddr, bend left] & & \boldsymbol{\alpha'}{:}q \arrow[ddr, bend left] & & & \boldsymbol{\alpha'}{:}a & \\
\multimap & & & & & & & \in \mathrm{ccopy}_\Sigma \\
\Sigma & & \boldsymbol{\alpha}{:}q & & \boldsymbol{\alpha'}{:}q \arrow[r] & \boldsymbol{\alpha'}{:}a \arrow[uur, bend left] & &
\end{tikzcd}$$

Now, consider a strategy $\sigma : \Sigma \multimap \Sigma$ consisting only of the play s below (and its prefixes):

$$\begin{tikzcd}
\Sigma & \boldsymbol{\alpha}{:}q \arrow[ddrrrrr, bend right] & \boldsymbol{\alpha'}{:}q \arrow[ddr, bend right] & & & \boldsymbol{\alpha'}{:}a & & \\
\multimap & & & & & & & \in \sigma \\
\Sigma & & & \boldsymbol{\alpha'}{:}q \arrow[r] & \boldsymbol{\alpha'}{:}a \arrow[uur, bend right] & & \boldsymbol{\alpha}{:}q &
\end{tikzcd}$$

The plays s and t can interact in the following two ways (among others) when considering the composition $\sigma; \mathrm{ccopy}_\Sigma$:

$$\begin{tikzcd}
\Sigma & \boldsymbol{\alpha}{:}q \arrow[dr, bend left] & & \boldsymbol{\alpha'}{:}q \arrow[dr, bend left] & & & & & \color{red}\boldsymbol{\alpha'}{:}a & \\
\rotatebox{90}{$\multimap$} & & & & & & & & & \\
\Sigma & & \boldsymbol{\alpha}{:}q \arrow[ddrrrrrrr, bend right] & & \boldsymbol{\alpha'}{:}q \arrow[ddr, bend right] & & & \boldsymbol{\alpha'}{:}a \arrow[uur, bend left] & & \\
\rotatebox{90}{$\multimap$} & & & & & & & & & \\
\Sigma & & & & & \boldsymbol{\alpha'}{:}q \arrow[r] & \boldsymbol{\alpha'}{:}a \arrow[uur, bend right] & & & \color{red}\boldsymbol{\alpha}{:}q
\end{tikzcd}$$

$$\begin{tikzcd}
\Sigma & \boldsymbol{\alpha}{:}q \arrow[dr, bend left] & & \boldsymbol{\alpha'}{:}q \arrow[dr, bend left] & & & & & & \color{red}\boldsymbol{\alpha'}{:}a \\
\rotatebox{90}{$\multimap$} & & & & & & & & & \\
\Sigma & & \boldsymbol{\alpha}{:}q \arrow[ddrrrrrr, bend right] & & \boldsymbol{\alpha'}{:}q \arrow[ddr, bend right] & & & \boldsymbol{\alpha'}{:}a \arrow[uurr, bend left] & & \\
\rotatebox{90}{$\multimap$} & & & & & & & & & \\
\Sigma & & & & & \boldsymbol{\alpha'}{:}q \arrow[r] & \boldsymbol{\alpha'}{:}a \arrow[uur, bend right] & & \color{red}\boldsymbol{\alpha}{:}q &
\end{tikzcd}$$

Each of these interactions results in a different ordering of the last two moves: $\boldsymbol{\alpha'}{:}a$ and $\boldsymbol{\alpha}{:}q$. Therefore, the strategy $\sigma; \mathrm{ccopy}_\Sigma$ includes both of the following plays:

$$\boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha'}{:}a \cdot {\color{red}\boldsymbol{\alpha'}{:}a} \cdot {\color{red}\boldsymbol{\alpha}{:}q} \ , \ \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha'}{:}a \cdot \cdot {\color{red}\boldsymbol{\alpha}{:}q} \cdot {\color{red}\boldsymbol{\alpha'}{:}a} \ \in \sigma; \mathrm{ccopy}_\Sigma$$

This is despite the fact that the second play is not in σ. Therefore, ccopy_Σ is not a neutral element.

This issue is not due to a bad choice of candidate for a neutral element, it turns out that there is no strategy that behaves like the neutral element for every concurrent strategy. This is the issue that Ghica and Murawski [2004] faced and is a common issue in compositional models of concurrent computation. Now, if strategies were required to be saturated under the rewrite system from §2.1.2 (where we interpret invocation as O move and return as P move), then σ would not be a valid strategy, as it must include both orderings to be saturated. While saturation solves the issue, the deeper question of why happens-before order preservation appears remains.

4.2 Concurrent Games and Saturated Strategies

We start by formally defining the concurrent copycat strategy ccopy:

Definition 4.1. The concurrent copycat strategy $\mathrm{ccopy}_{\mathbf{A}} : \mathbf{A} \multimap \mathbf{A}$ is defined as the self-interleaving of the sequential copycat strategy $\mathrm{copy}_A : A \multimap A$ defined as

$$\mathrm{ccopy}_{\mathbf{A}} := \mathrm{copy}_A^{\Phi}$$

Proposition 4.2. $\mathrm{ccopy}_{\mathbf{A}}$ *is idempotent.*

This observation is all it takes to make use of an abstract construction called the Karoubi envelope to construct a model of concurrent games where ccopy_ does act as the neutral element for strategy composition, as we will treat in detail in §6. This construction allows us to construct a category $\mathbf{Game}_{\mathbf{Conc}}$ that specializes $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ to strategies that are well-behaved upon composition with the family of idempotents ccopy_. Concretely, $\mathbf{Game}_{\mathbf{Conc}}$ is defined as follows:

Definition 4.3. The category $\mathbf{Game}_{\mathbf{Conc}}$ has as objects concurrent games $\mathbf{A}, \mathbf{B}$ and as morphisms strategies $\sigma : \mathbf{A} \multimap \mathbf{B} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ *saturated* in that

$$\mathrm{ccopy}_{\mathbf{A}}; \sigma; \mathrm{ccopy}_{\mathbf{B}} = \sigma$$

Composition is given by strategy composition $-;-$ with the concurrent copycat ccopy_ as identity.

4.3 Refinement for Concurrent Strategies

We endow the semicategory of concurrent strategies with an order enrichment, which also gives our notion of refinement. We order strategies $\sigma, \tau \in \underline{\mathbf{Game}}_{\mathbf{Conc}}(\mathbf{A}, \mathbf{B})$ by set containment $\sigma \subseteq \tau$. This assembles the hom-set $\underline{\mathbf{Game}}_{\mathbf{Conc}}(\mathbf{A}, \mathbf{B})$ into a join-semilattice. Joins are given by union of strategies, which are well-defined as prefix-closure, non-emptiness and receptivity are all preserved by unions. Composition is well-behaved with respect to this ordering in the following sense:

Proposition 4.4. *Strategy composition is monotonic and join-preserving.*

Refinement is a pesky issue in the context of concurrency, non-determinism, and undefined behavior. We do not purport to address this issue in this paper. Instead, we choose trace set containment to remain faithful with linearizability, where this notion of refinement is prevalent. Interestingly, strategy containment is a standard notion of refinement in game semantics as well.

4.4 The Semifunctors K_{Conc} and $\mathsf{Emb}_{\mathsf{Conc}}$

The abstract treatment in §6 will also show that the abstract construction giving rise to $\mathbf{Game}_{\mathbf{Conc}}$ comes with some infrastructure around it for free. For instance, it readily gives a forgetful semi-functor from $\mathbf{Game}_{\mathbf{Conc}}$ (seen here as a semicategory instead of a category) to $\underline{\mathbf{Game}}_{\mathbf{Conc}}$

$$\mathsf{Emb}_{\mathsf{Conc}} : \mathsf{Semi}\ \mathbf{Game}_{\mathbf{Conc}} \longrightarrow \underline{\mathbf{Game}}_{\mathbf{Conc}}$$

acting as the identity semifunctor. We will omit applications of $\mathsf{Emb}_{\mathsf{Conc}}$ when it causes no harm.

There is also a transformation which takes a *not* necessarily saturated concurrent strategy σ and constructs the smallest strategy that is saturated and contains σ, which we name

$$K_{\mathsf{Conc}} : \underline{\mathbf{Game}}_{\mathbf{Conc}} \rightarrow \mathsf{Semi}\ \mathbf{Game}_{\mathbf{Conc}}$$

as defined in §6, and explicitly given by:

$$\mathbf{A} \xmapsto{K_{\mathsf{Conc}}} \mathbf{A} \qquad\qquad \sigma : \mathbf{A} \multimap \mathbf{B} \xmapsto{K_{\mathsf{Conc}}} \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$$

unfortunately this mapping does not assemble into a semifunctor. Despite that, K_{Conc} is an oplax semifunctor, in the sense described in the following proposition.

Proposition 4.5. *For any $\sigma : \mathbf{A} \multimap \mathbf{B}$ and $\tau : \mathbf{B} \multimap \mathbf{C}$:*

$$K_{\mathsf{Conc}}(\sigma; \tau) \subseteq K_{\mathsf{Conc}}(\sigma); K_{\mathsf{Conc}}(\tau)$$

It is straight-forward to check that K_{Conc} is continuous, that is, it is monotonic and join-preserving. It is important to emphasize that while we give concrete definitions for these operations, they come from the abstract construction we describe for an arbitrary semicategory in §6.

4.5 Fine-Grained Synchronization in Concurrent Games

In §4.2, we gave a rather abstract definition for the strategies in $\mathbf{Game}_{\mathbf{Conc}}$. Ghica [2019], in a slightly different setting, observed that this abstract definition is equivalent to a concrete one, originally appearing in Ghica and Murawski [2004], involving the rewrite system we discussed in §2.1.2, which we now adapt to our setting.

Definition 4.6. Let $\mathbf{A} = (M_A, P_A)$ be a concurrent game. We define an abstract rewrite system $(P_{\mathbf{A}}, \rightsquigarrow_{\mathbf{A}})$ with local rewrite rules:

- $\forall m, m' \in M_A . \forall \alpha, \alpha' \in \Upsilon . \alpha \neq \alpha' \wedge \lambda_{\mathbf{A}}(m) = \lambda_{\mathbf{A}}(m') \Rightarrow \boldsymbol{\alpha}{:}m \cdot \boldsymbol{\alpha'}{:}m' \rightsquigarrow_{\mathbf{A}} \boldsymbol{\alpha'}{:}m' \cdot \boldsymbol{\alpha}{:}m$
- $\forall o, p \in M_A . \forall \alpha, \alpha' \in \Upsilon . \alpha \neq \alpha' \wedge \lambda_{\mathbf{A}}(o) = O \wedge \lambda_{\mathbf{A}}(p) = P \Rightarrow \boldsymbol{\alpha}{:}o \cdot \boldsymbol{\alpha'}{:}p \rightsquigarrow_{\mathbf{A}} \boldsymbol{\alpha'}{:}p \cdot \boldsymbol{\alpha}{:}o$

The main result of this section is the following alternative characterization of saturation.

Proposition 4.7. *A strategy $\sigma : \mathbf{A} \multimap \mathbf{B}$ is saturated if and only if*

$$\forall s \in \sigma . \forall t \in P_{\mathbf{A} \multimap \mathbf{B}} . t \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s \Rightarrow t \in \sigma$$

The key lemma to show this alternative characterization is the synchronization lemma, as coined by Ghica [2019], which plays a similar role to concurrent games as does the switching condition in sequential games. It essentially establishes that there is still synchronization happening under this liberal setting, all enabled by the fact that each agent is still synchronizing with itself.

It is useful to define a closure operator over sets of plays. Given a set of plays $S \subseteq P_{\mathbf{A}}$ we call $\mathrm{strat}(S) : \mathbf{A}$ the least strategy containing S, obtained as the prefix and receptive closure of S.

Proposition 4.8 (Synchronization Lemma). *Let $s = p \cdot \boldsymbol{\alpha}{:}m \cdot \boldsymbol{\alpha'}{:}m' \cdot p'$ be a play of $\mathbf{A} \multimap \mathbf{B}$. Let $\sigma = \mathrm{strat}(p \cdot \boldsymbol{\alpha}{:}m \cdot \boldsymbol{\alpha'}{:}m' \cdot p')$. Then,*

$$p \cdot \boldsymbol{\alpha'}{:}m' \cdot \boldsymbol{\alpha}{:}m \cdot p' \in \mathrm{ccopy}_{\mathbf{A}} ; \sigma ; \mathrm{ccopy}_{\mathbf{B}} \iff \boldsymbol{\alpha}{:}m \cdot \boldsymbol{\alpha'}{:}m' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} \boldsymbol{\alpha'}{:}m' \cdot \boldsymbol{\alpha}{:}m$$

The core of the proof of Prop. 4.8 lies in the dynamics of ccopy_. If we focus on an agent $\alpha \in \Upsilon$, a typical play in $\mathrm{ccopy}_{\mathbf{B}}$ behaves as displayed below on the left.

$$\begin{tikzcd}
\boldsymbol{\alpha}{:}q \arrow[dr] & & & \boldsymbol{\alpha}{:}a \\
& \boldsymbol{\alpha}{:}q \arrow[r] & \boldsymbol{\alpha}{:}a \arrow[ur] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\boldsymbol{\alpha}{:}q \arrow[dr] & \cdots & \cdots & \boldsymbol{\alpha'}{:}a & \\
\cdots \arrow[rr, bend right] & \boldsymbol{\alpha}{:}q \arrow[rrr, bend right] & \boldsymbol{\alpha'}{:}a \arrow[ur] & & \cdots
\end{tikzcd}$$

Observe that no matter what the other agents are doing it is always the case that the copy of an O move in the target appears later in the source, and a copy of a P move in the target appears earlier in the source. So if we have a play $s \in P_{\mathbf{B}}$ such that $s = p \cdot \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}a \cdot p'$ any of its interactions with $\mathrm{ccopy}_{\mathbf{B}}$, such as in $\mathrm{strat}(s) ; \mathrm{ccopy}_{\mathbf{B}}$, look something like the play displayed above on the right.

After hiding the interaction in the source, the resulting play can at most make $\boldsymbol{\alpha}{:}q$ appear earlier and $\boldsymbol{\alpha'}{:}a$ appear later, so it cannot change their order. For any of the other cases for the polarities of those two moves, there is always a case where they can appear swapped as the result of the interaction. So the proof of Prop. 4.8 is a case analysis of the polarities of $\boldsymbol{\alpha}{:}m$ and $\boldsymbol{\alpha'}{:}m'$.

5 LINEARIZABILITY

In this section, we argue that linearizability emerges from the Karoubi construction used to define $\mathbf{Game}_{\mathrm{Conc}}$ and establish several of the main results of this paper. In §5.1 we establish that K_{Conc} exactly corresponds to a general notion of linearizability which is improved in §5.2, while in §5.3 we observe that plays of ccopy_ correspond to proofs of linearizability. In §5.4 we show a property analogous to the usual observational refinement property, and in §5.5 we show the locality property.

5.1 Linearizability

We start by defining linearizability.

Definition 5.1. We say a play $s \in P_{\mathbf{A}}$ is linearizable to a play $t \in P_{\mathbf{A}}$ if there exists a sequence of Opponent moves $s_O \in (M_{\mathbf{A}}^O)^*$ and a sequence of Proponent moves $s_P \in (M_{\mathbf{A}}^P)^*$ such that

$$s \cdot s_P \rightsquigarrow_{\mathbf{A}} t \cdot s_O$$

A play $s \in P_{\mathbf{A}}$ is linearizable with respect to a strategy $\tau : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathrm{Conc}}$ if there exists t in τ such that s is linearizable to t. If every play of a strategy $\sigma : \mathbf{A}$ is linearizable with respect to $\tau : \mathbf{A}$ then we say σ is linearizable with respect to τ.

In this general definition of linearizability, s_P completes some pending O moves with a response by P while the sequence s_O plays the role of the pending invocations that are removed from s. Note that t need not be atomic and may still have pending Opponent moves. The rewrite relation $\rightsquigarrow_{\mathbf{A}}$

plays the role of preservation of happens-before order. In this sequentially consistent formulation of concurrent games, this generalized definition of linearizability is closely related to interval-sequential linearizability [Castañeda et al. 2015], what we address in more detail in our technical report [Oliveira Vale et al. 2022]. When the linearized strategy is specialized to atomic strategies only, we obtain Herlihy-Wing linearizability. In our technical report [Oliveira Vale et al. 2022] we give a thorough account of the specialization to atomic games.

The central result of this paper is a characterization of K_{Conc} in terms of linearizability.

Proposition 5.2. *For any* $\tau : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$

$$K_{\text{Conc}}\ \tau = \{s \in P_{\mathbf{A}} \mid s \textit{ is linearizable with respect to } \tau\}$$

Proof. Suppose $s \in K_{\text{Conc}}\ \tau$. By Prop. 4.7 it follows that there exists $t \in \tau$ such that $s \rightsquigarrow_{\mathbf{A}} t$ and therefore by setting $s_O = s_P = \epsilon$ we are done.

Suppose there are s_P and s_O such that $s \cdot s_P \rightsquigarrow_{\mathbf{A}} t \cdot s_O$. By receptivity $t \cdot s_O \in \tau$. By Prop. 4.7, $s \cdot s_P \in K_{\text{Conc}}\ \tau$. By prefix-closure, $s \in K_{\text{Conc}}\ \tau$, as desired. □

A lot of this proposition is taken for by Prop. 4.7. Observe that τ's receptivity explains why some Opponent moves s_O may be removed, while the fact that the play can be completed with Proponent moves s_P arises from prefix-closure. We also find it important to remind the reader that K_{Conc} is defined in terms of its role in the relationship between a semicategory and its Karoubi envelope, as will be treated in detail in §6. In this way, Prop. 5.2 shows that linearizability arises as a result of an abstract construction solving the problem of lack of neutral elements in our concurrent model of computation. An immediate corollary of Prop. 5.2 is an alternative definition of linearizability.

Corollary 5.3 (Abstract Linearizability). *A strategy* $\sigma : \mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}$ *is linearizable to a strategy* $\tau : \mathbf{A}$ *if and only if*

$$\sigma \subseteq K_{\text{Conc}}\ \tau$$

As K_{Conc} appear as a result of an abstract construction, this alternative definition may be used even in situations where there is no candidate for a happens-before-ordering or a rewrite relation such as $- \rightsquigarrow -$. As matter of example, Ghica [2013] defines a compositional model of delay-insensitive circuits. There, the Karoubi envelope is used to turn a model of asynchronous circuits which is not physically realizable into one that is. This abstract definition of linearizability implied by Prop. 5.3 and developed in detail in §6 could be adapted to that setting to give a notion of linearizability for delay-insensitive circuits.

This abstract construction will also allow us to give a more general but simple proof of the refinement property in §5.4 and locality in §5.5.

5.2 Strong Linearizability

This alternative and abstract characterization also suggests the following variation of linearizability:

Definition 5.4. We say $\sigma : \mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}$ is strongly linearizable to $\tau : \mathbf{A}$ when σ is linearizable with respect to τ and $\tau \subseteq \sigma$.

We call this *strong* because it implies the conventional notion of linearizability as defined in 5.1. In particular, atomic strong linearizability implies Herlihy-Wing linearizability. Note that when σ is strongly linearizable with respect to τ we obtain that:

$$K_{\text{Conc}}\ \tau \subseteq K_{\text{Conc}}\ \sigma = \sigma$$

Together with Corollary 5.3 it follows that $\sigma = K_{\text{Conc}}\ \tau$ so that σ is fully characterized by its linearization. Therefore, a strongly linearizable σ is a strategy which is in the image of K_{Conc}.

Concretely, strong linearizability is what most intuitively call linearizability. Indeed, in works based on operational semantics there is always the possibility that by chance the scheduler schedules the threads in such a way that it generates an atomic execution for the system. Those atomic executions turn out to be exactly the linearization of the objects that are studied in that context.

When an object is non-strongly linearizable to a specification, it means that the specification is not accurate: it is an over-approximation. For example, it is easy to prove that every concurrent strategy is linearizable to some atomic strategy. In particular, ν'_{yield} is (Herlihy-Wing) linearizable to an atomic spec ν with plays of the form:

$$\boldsymbol{\alpha_1}{:}\text{yield} \cdot \boldsymbol{\alpha_1}{:}\text{ok} \cdot \boldsymbol{\alpha_1}{:}\text{yield} \cdot \boldsymbol{\alpha_1}{:}\text{ok} \cdot \ldots \cdot \boldsymbol{\alpha_n}{:}\text{yield} \cdot \boldsymbol{\alpha_n}{:}\text{ok}$$

But ν'_{yield} does not strongly linearize to ν. Moreover, ν does not make sense as a specification for yield. Standard linearizability does not rule out such bad specifications, while strong linearizability does. Our formalism shows exactly in which sense non-strong linearizability yields an *over-approximation*: If σ is strongly linearizable to τ then $\sigma = K_{\text{Conc}}\ \tau$, as we showed above. Meanwhile, when σ is linearizable to τ but not strongly linearizable, we have a strict containment $\sigma \subset K_{\text{Conc}}\ \tau$.

5.3 Computational Interpretation of Linearizability

We just saw that linearizability can be characterized by the transformation K_{Conc}. We now offer yet another perspective on linearizability by providing a computational interpretation of linearizability proofs. Recall that in our discussion in §4.1 we observed that ccopy_ is the denotation of a concrete program. Interestingly, the plays of ccopy_ correspond to proofs of linearizability.

Proposition 5.5. *$s_1 \in P_{\mathbf{A}}$ linearizes to $s_0 \in P_{\mathbf{A}}$ if and only if there is a play $s \in \text{ccopy}_{\mathbf{A}}$ such that*

$$s\restriction_{\mathbf{A}_0} = s_0 \qquad\qquad s\restriction_{\mathbf{A}_1} = s_1$$

Proof. For this, one first proves that every play $s \in \text{ccopy}_{\mathbf{A}}$ satisfies $s\restriction_{\mathbf{A}_1} \rightsquigarrow_{\mathbf{A}} s\restriction_{\mathbf{A}_0}$. Then, prefix-closure and receptivity of $\text{ccopy}_{\mathbf{A}}$ allow for linearizability to be used instead of $- \rightsquigarrow_{\_} -$, similarly to the the proof of Prop. 5.2. See Oliveira Vale et al. [2022] for a detailed proof. □

What Prop. 5.5 essentially establishes is that proofs of linearizability encode executions of the code in Fig. 3, and that executions of the code in Fig. 3 encode proofs of linearizability. Intuitively, the reason for this is that in a play of $\text{ccopy}_{\mathbf{A}}$ an O move followed by a P move in the target component forms an interval *around* their corresponding moves in the source component. So if we have two such pairs by different agents, one happening entirely before the other, then their corresponding moves in the source must happen in the same order. This means that happens-before order is preserved from the target component to the source component. See the figure below depicting a play of ccopy_{Σ}:

$$\begin{tikzcd}
\boldsymbol{\alpha}{:}q \arrow[dr, bend right] & & & \boldsymbol{\alpha}{:}a & \boldsymbol{\alpha'}{:}q \arrow[dr, bend right] & & & \boldsymbol{\alpha'}{:}a \\
& \boldsymbol{\alpha}{:}q \arrow[r] & \boldsymbol{\alpha}{:}a \arrow[ur, bend right] & & & \boldsymbol{\alpha'}{:}q \arrow[r] & \boldsymbol{\alpha'}{:}a \arrow[ur, bend right] &
\end{tikzcd}$$

5.4 Interaction Refinement

One is often interested in implementing an interface of type $\mathbf{B}$ making use of some other interface of type $\mathbf{A}$ by using an implementation specified as a saturated strategy of type $\sigma : \mathbf{A} \multimap \mathbf{B}$. Now, the game $\mathbf{A}$ appears in a negative position in the type $\mathbf{A} \multimap \mathbf{B}$. Because of this there is a contravariant effect to linearizability on $\multimap$ in that if $s \rightsquigarrow_{\mathbf{A}\multimap\mathbf{B}} t$ then, while $s\restriction_{\mathbf{B}}$ is "more concurrent" than $t\restriction_{\mathbf{B}}$, $s\restriction_{\mathbf{A}}$ is "less concurrent" than $t\restriction_{\mathbf{A}}$. This intuition leads to the following result analogous to the observational refinement equivalence of Filipovic et al. [2010].

Proposition 5.6 (Interaction Refinement). *$\nu'_A : \mathbf{A} \in \mathbf{Game_{Conc}}$ is linearizable to $\nu_A : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ if and only if for all concurrent games $\mathbf{B}$ and $\sigma : \mathbf{A} \multimap \mathbf{B} \in \mathbf{Game_{Conc}}$ it holds that*

$$\nu'_A ; \sigma \subseteq \nu_A ; \sigma$$

Proof. By Corollary 5.3, monotonicity of composition, and saturation of σ:

$$\nu'_A ; \sigma \subseteq K_{\mathrm{Conc}}\ \nu_A ; \sigma = (\mathsf{ccopy}_{\mathbf{1}} ; \nu_A ; \mathsf{ccopy}_{\mathbf{A}}) ; \sigma = (\mathsf{ccopy}_{\mathbf{1}} ; \nu_A) ; (\mathsf{ccopy}_{\mathbf{A}} ; \sigma) = \nu_A ; \sigma$$

For the reverse direction, simply observe that:

$$\nu'_A \subseteq \nu'_A ; \mathsf{ccopy}_{\mathbf{A}} \subseteq \nu_A ; \mathsf{ccopy}_{\mathbf{A}} = \mathsf{ccopy}_{\mathbf{1}} ; \nu_A ; \mathsf{ccopy}_{\mathbf{A}} = K_{\mathrm{Conc}}\ \nu_A$$

□

This immediately implies a stronger result under strong linearizability

Corollary 5.7. *Let $\nu'_A : \mathbf{A} \in \mathbf{Game_{Conc}}$ is* strongly *linearizable w.r.t. to $\nu_A : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ if and only for all $\mathbf{B}$ and $\sigma : \mathbf{A} \multimap \mathbf{B} \in \mathbf{Game_{Conc}}$:*

$$\nu'_A ; \sigma = \nu_A ; \sigma$$

5.5 Locality

We revisit the locality property from Herlihy and Wing [1990] by reformulating the notion of an object system with several independent objects as the linear logic tensor product $\otimes$. For this we start with a *faux* definition of tensor product.

Definition 5.8. If $\mathbf{A} = (M_A, P_A)$ and $\mathbf{B} = (M_B, P_B)$ are games in $\underline{\mathbf{Game}}_{\mathbf{Conc}}$, we define the game $\mathbf{A} \otimes \mathbf{B} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ as $\mathbf{A} \otimes \mathbf{B} = (M_{A \otimes B}, P_{A \otimes B})$. We denote by $\mathbf{1}$ the game $\mathbf{1} = (M_1, P_1)$.

Given strategies $\sigma_A : \mathbf{A}$ and $\sigma_B : \mathbf{B}$ we define the strategy $\sigma_A \otimes \sigma_B : \mathbf{A} \otimes \mathbf{B}$ as the set $(\sigma_A \parallel \sigma_B) \cap P_{\mathbf{A} \otimes \mathbf{B}}$, the set of sequentially consistent interleavings of σ_A and σ_B.

We call this a *faux* tensor product because there is no reasonable definition of a *monoidal semicategory* for lack of neutral elements with which to express the coherence conditions. Despite that, the $- \otimes -$ operation becomes a proper tensor product when specialized to $\mathbf{Game_{Conc}}$.

Proposition 5.9. *$(\mathbf{Game_{Conc}}, - \otimes -, \mathbf{1})$ assembles into a symmetric monoidal closed category.*

This structure is obtained by mapping the corresponding structural maps in $\mathbf{Game_{Seq}}$ through an interleaving functor. In particular, Prop. 5.9 says that $- \otimes -$ is a bifunctor in $\mathbf{Game_{Conc}}$, so that

Proposition 5.10. *For all concurrent games $\mathbf{A}$, $\mathbf{B}$:*

$$\mathsf{ccopy}_{\mathbf{A} \otimes \mathbf{B}} = \mathsf{ccopy}_{\mathbf{A}} \otimes \mathsf{ccopy}_{\mathbf{B}}$$

This rather simple result is rather auspicious given the computational interpretation of ccopy_ in terms of linearizability proofs seen in §5.3. This property, together with the fact that $- \otimes -$ is a bi-semifunctor, readily implies that K_{Conc} distributes over the tensor product.

Proposition 5.11. *Let $\sigma_A : \mathbf{A} \multimap \mathbf{A}'$ and $\sigma_B : \mathbf{B} \multimap \mathbf{B}'$. Then:*

$$K_{\mathrm{Conc}}\ (\sigma_A \otimes \sigma_B) = K_{\mathrm{Conc}}\ \sigma_A \otimes K_{\mathrm{Conc}}\ \sigma_B$$

Proof.

$$\begin{aligned}
K_{\mathrm{Conc}}\ (\sigma_A \otimes \sigma_B) &= \mathsf{ccopy}_{\mathbf{A} \otimes \mathbf{B}} ; (\sigma_A \otimes \sigma_B) ; \mathsf{ccopy}_{\mathbf{A}' \otimes \mathbf{B}'} && \text{(Def.)} \\
&= (\mathsf{ccopy}_{\mathbf{A}} \otimes \mathsf{ccopy}_{\mathbf{B}}) ; (\sigma_A \otimes \sigma_B) ; (\mathsf{ccopy}_{\mathbf{A}'} \otimes \mathsf{ccopy}_{\mathbf{B}'}) && \text{(Prop. 5.10)} \\
&= (\mathsf{ccopy}_{\mathbf{A}} ; \sigma_A ; \mathsf{ccopy}_{\mathbf{A}'}) \otimes (\mathsf{ccopy}_{\mathbf{B}} ; \sigma_B ; \mathsf{ccopy}_{\mathbf{B}'}) && \text{(bi-semifunctoriality of } - \otimes -) \\
&= K_{\mathrm{Conc}}\ \sigma_A \otimes K_{\mathrm{Conc}}\ \sigma_B && \text{(Def.)}
\end{aligned}$$

□

which gives as corollary a generalization of Herlihy and Wing [1990]'s locality theorem.

Corollary 5.12 (Locality). *Let $v'_A : \mathbf{A}, v'_B : \mathbf{B} \in \mathbf{Game}_{\mathbf{Conc}}$ and $v_A : \mathbf{A}, v_B : \mathbf{B} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$. Then*

$$v' = v'_A \otimes v'_B \textit{ is linearizable w.r.t. } v = v_A \otimes v_B$$
$$\textit{if and only if}$$
$$v'_A \textit{ is linearizable w.r.t. } v_A \textit{ and } v'_B \textit{ is linearizable w.r.t. } v_B$$

Proof. By Prop. 5.11 and Prop. 5.2

$$v' = v'_A \otimes v'_B \subseteq K_{\mathrm{Conc}}\,(v_A \otimes v_B) = K_{\mathrm{Conc}}\; v_A \otimes K_{\mathrm{Conc}}\; v_B$$

in particular,

$$v'_A = (v'_A \otimes v'_B)\restriction_{\mathbf{A}} \subseteq (K_{\mathrm{Conc}}\; v_A \otimes K_{\mathrm{Conc}}\; v_B)\restriction_{\mathbf{A}} = K_{\mathrm{Conc}}\; v_A$$
$$v'_B = (v'_A \otimes v'_B)\restriction_{\mathbf{B}} \subseteq (K_{\mathrm{Conc}}\; v_A \otimes K_{\mathrm{Conc}}\; v_B)\restriction_{\mathbf{B}} = K_{\mathrm{Conc}}\; v_B$$

For the reverse direction, we have:

$$v' = v'_A \otimes v'_B \subseteq K_{\mathrm{Conc}}\; v_A \otimes K_{\mathrm{Conc}}\; v_B = K_{\mathrm{Conc}}\,(v_A \otimes v_B)$$

□

We would like to observe that not only our methodology yields a stronger result in Prop. 5.10 and 5.11, but also that it supports simpler, mostly algebraic proofs. Meanwhile, even in the simpler case of atomic linearizability, Herlihy and Wing [1990]'s original proof is rather ad hoc.

6 THE KAROUBI ENVELOPE AND ABSTRACT LINEARIZABILITY

In this section, we establish the main abstract tools we used to construct models of concurrent computation. The most important points of this section are the definitions of $\mathbf{C}_e$, K_e, and Emb_e, which we wrote concretely as $\mathbf{Game}_{\mathbf{Conc}}$, K_{Conc} and $\mathrm{Emb}_{\mathrm{Conc}}$ in the main development. An extended version of this section is available in Oliveira Vale et al. [2022].

Given a semicategory $\mathbf{C}$ the Karoubi envelope is the category Kar $\mathbf{C}$ which has as objects pairs:

$$(C \in \mathbf{C}, e : C \to C)$$

of an object C and an idempotent e of C. Recall that an idempotent of an object is simply an idempotent endomorphism of that object, in the sense that $e \circ e = e$. A morphism $f : (C, e) \to (C', e')$ in Kar $\mathbf{C}$ is a morphism $f : C \to C'$ of the underlying semicategory $\mathbf{C}$ that is invariant upon the idempotents, involved in the sense that:

$$e' \circ f \circ e = f$$

which we call a *saturated* morphism of $\mathbf{C}$. Observe that by construction the Karoubi envelope Kar $\mathbf{C}$ is indeed a category by defining the neutral elements by the equation $\mathbf{id}_{(C,e)} = e$.

The following is folklore in the theory of semicategories. There is a forgetful functor

$$\mathrm{Semi} : \mathbf{Cat} \to \mathbf{SemiCat}$$

which given a category $\mathbf{C}$ assigns a semicategory Semi $\mathbf{C}$ by forgetting the data about the neutral elements in $\mathbf{C}$, which also determines its action of transforming functors into semifunctors by forgetting the fact that it preserves neutral elements. Semi admits a right adjoint

$$\mathrm{Kar} : \mathbf{SemiCat} \to \mathbf{Cat}$$

which maps a semicategory $\mathbf{C}$ to its Karoubi envelope Kar $\mathbf{C}$.

When $\mathbf{C}$ has neutral elements, so that it actually assembles into a category, one obtains a fully faithful functor (of categories) into the Karoubi envelope by:

$$\mathbf{C} \longrightarrow \mathrm{Kar}\,\mathbf{C} \qquad\qquad C \longmapsto (C, \mathbf{id}_C)$$

which immediately makes any morphism $f : C \to C'$ into a morphism $f : (C, \mathbf{id}_C) \to (C', \mathbf{id}_{C'})$ due to the unital laws. Note that this functor corresponds to selecting a family $(e_C : C \to C)_{C \in \mathbf{C}}$ of idempotents e_C for each object $C \in \mathbf{C}$; in this case $e_C = \mathbf{id}_C$. The mapping of morphisms should saturate any morphism $f : C \to D$. Hence, it must be given by:

$$f \longmapsto e_D \circ f \circ e_C$$

Unfortunately, for lack of neutral elements in the semicategory case, there is no obvious choice of idempotents to construct such a functor, and in fact, there is no canonical choice of idempotents that makes it into a functor. Despite that, there is always a forgetful semifunctor:

$$\mathsf{Emb} : \mathsf{SemiKar}\ \mathbf{C} \to \mathbf{C}$$

Intuitively, the Karoubi envelope "splits" an object $C \in \mathbf{C}$ into many versions of itself: one for each idempotent e of C. Meanwhile, morphisms $f : C \to D$ are "classified" as morphisms $f : (C, e) \to (C', e')$ when they tolerate e and e' as neutral elements. So choosing an idempotent for each object of $\mathbf{C}$ really amounts to choosing a version of each object $C \in \mathbf{C}$ to obtain a category. We take the intuition we get from these remarks to define the following construction.

Let $\mathbf{C}$ be a semicategory enriched over $\mathbf{Cat}$ and let

$$e_- = \{e_C : C \to C\}_{C \in \mathbf{C}}$$

be a family of idempotents. Any such family defines a full subcategory $\mathbf{C}_e$ of the Karoubi envelope $\mathsf{Kar}\ \mathbf{C}$ of $\mathbf{C}$ by restricting the objects to precisely the idempotents in e_-. We call such a subcategory of $\mathsf{Kar}\ \mathbf{C}$ an *embeddable subcategory*. This naming is justified by the fact that the restriction

$$\mathsf{Emb}_e : \mathsf{Semi}\ \mathbf{C}_e \to \mathbf{C}$$

of the forgetful functor Emb defines an embedding. There is a *candidate* for a semifunctor

$$K_e : \mathbf{C} \to \mathsf{Semi}\ \mathbf{C}_e$$

going in the reverse direction, and given by

$$C \xmapsto{K_e} (C, e_C) \qquad\qquad f : C \to D \xmapsto{K_e} e_D \circ f \circ e_C$$

K_e often fails to be a semifunctor, as we noted. Despite that, semifunctoriality, even weakly, is not required for our purposes. Observe at this point that $\mathbf{Game}_{\mathbf{Conc}} = (\underline{\mathbf{Game}}_{\mathbf{Conc}})_{\mathrm{ccopy}}$ and that K_{Conc} is precisely the induced mapping K_{ccopy}.

We are now ready to define abstract linearizability.

Definition 6.1. Let $\mathbf{C}$ be an enriched semicategory equipped with a bi-semifunctor

$$- \otimes - : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$$

and an object $\mathbf{1}$ such that $(\mathbf{C}_e, - \otimes -, 1)$ is a symmetric monoidal category.

We say a morphism $f : \mathbf{1} \to C \in \mathbf{C}_e$ is linearizable to a morphism $g : \mathbf{1} \to C \in \mathbf{C}$ when

$$f \Rightarrow K_e\ g$$

Since our proofs of locality and interaction refinement on $\mathbf{Game}_{\mathbf{Conc}}$ were abstract, relying on Prop. 5.3, we can collect the necessary assumptions to obtain those results.

Proposition 6.2. *In the following, let* $\mathbf{C}$ *and* $\mathbf{C}_e$ *satisfy the conditions of Def. 6.1.*

Interaction Refinement *Suppose for all* $C \in \mathbf{C}$ *and* $f : \mathbf{1} \to C \in \mathbf{C}$ *it holds that*

$$f \circ e_{\mathbf{1}} = f$$

Then $f : \mathbf{1} \to C$ is linearizable to $g : \mathbf{1} \to C$ iff and only if for all $D \in \mathbf{C}$ and $h : C \to D \in \mathbf{C}_e$ it holds that

$$h \circ f \Rightarrow h \circ g$$

Locality *K_e distributes over $- \otimes -$ in the sense that for all $f : C \to C'$ and $g : D \to D'$*

$$K_e\ (f \otimes g) = K_e\ f \otimes K_e\ g$$

and if for all $C, C', D, D' \in \mathbf{C}$ it holds that

$$\mathbf{C}_e(C, C') \otimes \mathbf{C}_e(D, D') \cong \mathbf{C}_e(C, C') \times \mathbf{C}_e(D, D')$$

then $f'_C : \mathbf{1} \to C$ and $f'_D : \mathbf{1} \to D$ are linearizable to $f_C : \mathbf{1} \to C$ and $f_D : \mathbf{1} \to D$ if and only if $f'_C \otimes f'_D$ is linearizable to $f_C \otimes f_D$.

7 PRAGMATICS

Now that we have established the core results of the paper, we revisit the example in §2.2. We start by outlining a program logic for showing that certain concurrent programs implement linearizable objects, which is developed in detail in Oliveira Vale et al. [2022]. Then, we outline how the theory we develop can be used to reason about the example from §2.2. Our program logic is adapted from Khyzha et al. [2017], but contains significant modifications.

7.1 Programming Language

7.1.1 Syntax. We start by defining a language Com for commands over an effect signature E:

$$\mathsf{Prim} := x \leftarrow e(a) \mid \mathsf{assert}(\phi) \mid \mathsf{ret}\ v \qquad \mathsf{Com} := \mathsf{Prim} \mid \mathsf{Com}; \mathsf{Com} \mid \mathsf{Com} + \mathsf{Com} \mid \mathsf{Com}^* \mid \mathsf{skip}$$

Prim stands for primitive commands while Com is the grammar of commands. The most important commands work as follows:

- $x \leftarrow e(a)$ executes the effect $e \in E$ with argument a, which might contain variables defined in a local environment $\Delta \in \mathsf{Env}$.
- ret v stores in a reserved variable the value v, and may only be called once in any execution.
- assert(ϕ) takes a boolean function over the local environment and terminates computation if ϕ evaluates to False. assert($-$) can be used to implement a while loop and if conditionals in the usual way.

The remaining commands are per usual in a Kleene algebra.

An implementation $M[\alpha]$ of type $E \to F$, where E and F are effect signatures, is then given by a collection $M[\alpha] = (M[\alpha]^f)_{f \in F}$ indexed by F, so that for each $f \in F$ we have $M[\alpha]^f \in \mathsf{Com}$; we denote the set of implementations by Mod.

Meanwhile, a concurrent module $M[A]$ is given by a collection of implementations $M[A] = (M[\alpha])_{\alpha \in A}$ indexed by a set $A \subseteq \Upsilon$ of active agents, so that $M[\alpha] \in \mathsf{Mod}$ is an implementation for each active agent $\alpha \in A$; we denote the set of concurrent modules by CMod.

7.1.2 Operational Semantics. Each primitive command B receives an interpretation as a state transformer $[\![B]\!]_\alpha : \mathsf{UndState} \to \mathcal{P}(\mathsf{UndState})$ over a set of states $\mathsf{UndState} := \mathsf{Env} \times P_{\dagger\mathbf{E}}$ and returning a new set of states. A state $(\Delta, s) \in \mathsf{UndState}$ contains a local environment $\Delta \in \mathsf{Env}$ and a state represented canonically as a play of $s \in \dagger\mathbf{E}$. Concretely, s is the history of operations on the underlying object. The state transformer $[\![B]\!]_\alpha$ depends on α only in that it tags the events it adds to the underlying state with an identifier for α.

We lift this interpretation function to a local small-step operational semantics $\langle C, \Delta, s\rangle \longrightarrow_\alpha \langle C', \Delta', s'\rangle$ encoding how α steps on commands in a mostly standard way following the Kleene algebra structure of commands. The key difference is that as we do not

$$\rightarrowtail\ \subseteq \mathsf{Com} \times \mathsf{Prim} \times \{O, P\} \times \mathsf{Com}$$

$$\frac{}{B \rightarrowtail^{O}_{B} B} \qquad \frac{}{B \rightarrowtail^{P}_{B} \mathsf{skip}}$$

$$\frac{C_1 \rightarrowtail^{X}_{B} C_1'}{C_1; C_2 \rightarrowtail^{X}_{B} C_1'; C_2} \qquad \frac{}{\mathsf{skip}; C \rightarrowtail^{X}_{\mathbf{id}} C}$$

$$\frac{}{C^* \rightarrowtail^{X}_{\mathbf{id}} C; C^*} \qquad \frac{}{C^* \rightarrowtail^{X}_{\mathbf{id}} \mathsf{skip}}$$

$$\frac{}{C_1 + C_2 \rightarrowtail^{X}_{\mathbf{id}} C_1} \qquad \frac{}{C_1 + C_2 \rightarrowtail^{X}_{\mathbf{id}} C_2}$$

$$\longrightarrow\ \subseteq (\mathsf{Com} \times \mathsf{UndState}) \times \Upsilon \times (\mathsf{Com} \times \mathsf{UndState})$$

$$\frac{(\Delta', s') \in [\![B]\!]^{X}_{\alpha}(\Delta, s) \qquad C \rightarrowtail^{X}_{B} C'}{\langle C, \Delta, s \rangle \longrightarrow_{\alpha} \langle C', \Delta', s' \rangle}$$

$$\twoheadrightarrow\ \subseteq (\mathsf{Cont} \times \mathsf{ModState}) \times \mathsf{CMod} \times (\mathsf{Cont} \times \mathsf{ModState})$$

$$\frac{f \in F \qquad a \in \mathsf{par}(f) \qquad \Delta' = \Delta[\alpha : [\mathsf{arg} : a]]}{\langle c[\alpha : \mathsf{idle}], \Delta, s \rangle \twoheadrightarrow^{M} \langle c[\alpha : M[\alpha]^{f}], \Delta', s \cdot \boldsymbol{\alpha}{:}f \rangle}$$

$$\frac{\langle C, \Delta, s \rangle \longrightarrow_{\alpha} \langle C', \Delta', s' \rangle}{\langle c[\alpha : C], \Delta, s \rangle \twoheadrightarrow^{M} \langle c[\alpha : C'], \Delta', s' \rangle}$$

$$\frac{\pi_{\alpha}(s \upharpoonright_{\mathbf{F}}) = p \cdot f \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(f) \qquad \Delta' = \Delta[\alpha : \varnothing]}{\langle c[\alpha : \mathsf{skip}], \Delta, s \rangle \twoheadrightarrow^{M} \langle c[\alpha : \mathsf{idle}], \Delta', s \cdot \boldsymbol{\alpha}{:}v \rangle}$$

Fig. 6. Command Reduction Rules ($\rightarrowtail$), Local Operational Semantics ($\longrightarrow$), and Concurrent Module Operational Semantics ($\twoheadrightarrow$)

assume the underlying object of type E is atomic, primitive commands execute in two separate steps, one for the invocation and the other for the return. Because of that, the interpretation function $[\![B]\!]_{\alpha}$ is decomposed into $[\![B]\!]^{O}_{\alpha}$, which is defined only on states where α's next move is an invocation, and $[\![B]\!]^{P}_{\alpha}$, which is defined only on states where α has a pending invocation (the remaining states). See Fig. 6 for the operational semantics rules. There, **id** stands for a primitive command that behaves just like skip but is used exclusively to define the operational semantics.

This small step operational semantics can be lifted to a concurrent module operational semantics

$$- \twoheadrightarrow^{-} - \subseteq (\mathsf{Cont} \times \mathsf{ModState}) \times \mathsf{CMod} \times (\mathsf{Cont} \times \mathsf{ModState})$$

which takes a continuation $\mathsf{Cont} := \Upsilon \to \{\mathsf{idle}\} + \{\mathsf{skip}\} + \mathsf{Com}$ and a module state $\mathsf{ModState} := (\Upsilon \to \mathsf{Env}) \times P_{\dagger\mathbf{E} \multimap \dagger\mathbf{F}}$ containing the local environments for all the agents, as well as the global trace of the system (see Fig. 6). The concurrent operational semantics $- \twoheadrightarrow^{M} -$ therefore describes the possible executions of the concurrent module M. The three rules correspond, from top to bottom, to a target component invocation, a step in the source component, and a return in the target component.

It is important to note that in our operational semantics, following the object-based semantics approach, which we develop in detail in Oliveira Vale et al. [2022], all shared state is encapsulated in the underlying object of type E. One of the many consequences of this is that the local environments can only be modified by their corresponding agents, and are initialized on a call on F and emptied on a return. This limits the lifetime of variables to a single execution of the body of a method.

7.1.3 Semantics. We give a concurrent module a denotation by the formula

$$[\![M]\!] = \{s \mid \exists c \in \mathsf{Cont}. \exists \Delta \in (\Upsilon \to \mathsf{Env}). \langle c_0, \boldsymbol{\Delta_0}, \epsilon \rangle \twoheadrightarrow^{M} \langle c, \boldsymbol{\Delta}, s \rangle\}$$

where c_0 is the initial continuation, and Δ_0 has every agent with an empty local environment. We specialize the operational semantics to the situation where a concurrent object specification $v_E : \dagger\mathbf{E}$ of type $\mathbf{E}$ is provided by defining an operational semantics $- \twoheadrightarrow^{M}_{v_E} -$ which runs M on top of v_E, what we denote as $\mathsf{Link}\ v_E; M$. We obtain the traces $[\![\mathsf{Link}\ v_E; M]\!]$ analogously to $[\![M]\!]$ by only considering steps in the source component that satisfy the specification v_E. The following result allows us to connect the programming language back with the theory we have developed so far.

Proposition 7.1. *For any $M \in \mathsf{CMod}$, $[\![M]\!] : \dagger\mathbf{E} \multimap \dagger\mathbf{F}$ is a strategy (in fact, a concurrent object implementation) and given $v_E : \dagger\mathbf{E}$,*

$$[\![\mathsf{Link}\ v_E; M]\!] = v_E; [\![M]\!]$$

7.2 Program Logic

Here, we present a simple, bare bones, program logic for proving implementations correctly implement linearizable objects. Despite its simplicity, it is expressive enough to reason about our notion of linearizability, and we believe it to be extensible.

We encapsulate the information necessary to define a linearizable concurrent object in a pair

$$(v' : \dagger\mathbf{A}, v : \dagger\mathbf{A}) \qquad \text{s.t.} \qquad v' \subseteq K_{\mathsf{Conc}}\ v$$

Throughout, we assume the following situation. We have a linearizable concurrent object $(v'_E : \dagger\mathbf{E}, v_E : \dagger\mathbf{E})$ and would like to show that an implementation $M : E \longrightarrow F$ is correct in that when it runs on top of v'_E it linearizes to a specification $v_F : \dagger\mathbf{F}$. When reasoning about $\mathsf{Link}\ v'_E; M$ it will be useful to restrict it with some invariants about its client. For example, usually when using a lock, one assumes that every lock user strictly alternates between calling acq and rel. So if all clients to the lock politely follow the lock policy, it is enough to verify only those traces. This policy of strict alternation is encoded in this strategy $v'_F : \dagger\mathbf{F}$ in our approach.

All in all, the program logic establishes that $(v'_E; [\![M]\!] \cap v'_F, v_F)$ is a linearizable concurrent object. For this purpose our program logic uses as proof configurations triples $(\Delta, s, \rho) \in \mathsf{Config} := \mathsf{ModState} \times \mathsf{Poss}$ where Poss is a set of *possibilities.* While Herlihy and Wing [1990] use sets of, so-called, linearized values, as possibilities, and Khyzha et al. [2017] uses an interval partial order, we use a play of $\mathsf{Poss} := K_{\mathsf{Conc}}\ v_F$. This means that throughout, if (Δ, s, ρ) is a configuration, we will always maintain as an invariant that $s\upharpoonright_{\mathbf{F}}$ is linearizable to ρ and that ρ is linearizable to v_F. Pre-conditions P are given by sets of configurations, while post-conditions Q, rely conditions $\mathcal{R}$, guarantee conditions $\mathcal{G}$ are specified as relations over the configurations.

There are three ways through which a configuration can be modified: through a relational predicate $\mathsf{invoke}_\alpha(-)$ which makes an invocation in $\mathbf{F}$, and simultaneously adds it to the state and the possibility; a *commit* rule $\mathcal{G} \vdash_\alpha \{P\}\ B\ \{Q\}$, where $B \in \mathsf{Prim}$, which allows one to modify the state by executing primitive commands over $\mathbf{E}$, but also to add early returns to ρ and to rewrite it according to $- \rightsquigarrow_{\mathbf{F}} -$; and a pair of post-conditions $\mathsf{returned}_\alpha(-)$ and $\mathsf{return}_\alpha(-)$ that check if at the end of execution there is a valid return in the possibility, and then adds it to the state.

Formally, the *commit* rule, which is the crux of the verification task, is defined below (P^O is the set of plays in P such that O is to move for α):

$$\begin{array}{l}\mathcal{G} \vdash_\alpha \{P\}\ B\ \{Q\} \iff \\ \quad \forall(\Delta, s, \rho). s\upharpoonright_{\mathbf{F}} \in v'_F \land (\Delta, s, \rho) \in P \land s\upharpoonright_{\mathbf{E}} \in v_E \land (\Delta', s') \in [\![B]\!]_\alpha(\Delta, s) \Rightarrow \\ \quad s'\upharpoonright_{\mathbf{F}} \in v'_F \land \exists \rho'.(\Delta, s, \rho)\ Q\ (\Delta', s', \rho') \land (\Delta, s, \rho)\ \mathcal{G}\ (\Delta', s', \rho') \land \rho \dashrightarrow \rho' \\ \qquad\qquad \rho \dashrightarrow \rho' \iff \exists t_P \in (M_{\mathbf{F}}^P)^*.\rho \cdot t_P \rightsquigarrow_{\dagger\mathbf{F}} \rho' \end{array}$$

$$\frac{\mathsf{stable}(\mathcal{R}, P) \qquad \mathsf{stable}(\mathcal{R}, Q) \qquad Q \circ P^O \subseteq P \qquad \mathcal{G} \vdash_\alpha \{P\}\ B\ \{Q\}}{\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ B\ \{Q\}}\ \textsc{Prim}$$

The rule considers every state (Δ', s') reachable by executing the primitive command B on behalf of α from a proof state (Δ, s, ρ) satisfying: the pre-condition P, the source component's linearized specification v_E and the target component's abstract invariant v'_F. The proof obligation is then to choose a new possibility ρ' and show that the reached state still satisfies v'_F, and that the step into the new proof configuration (Δ', s', ρ') satisfies the post-condition Q and the guarantee $\mathcal{G}$. This new possibility ρ' must be shown to satisfy $\rho \dashrightarrow \rho'$, which enforces that ρ' only differs from ρ by adding some returns t_P to ρ, and potentially linearizing the trace more by performing some rewrites ($\rho \cdot t_p \rightsquigarrow_{\dagger\mathbf{F}} \rho'$). Prim merely adds typical stability requirements on the operation. Lifting this rule to a Hoare-style judgement $\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C\ \{Q\}$ over any command $C \in \mathrm{Com}$ is straight-forward, which will be the program logic judgement for function bodies such as $M[\alpha]^f$.

Meanwhile, $\mathsf{invoke}_\alpha(-)$, $\mathsf{returned}_\alpha(-)$ and $\mathsf{return}_\alpha(-)$ are formally defined below, where idle_α is a predicate that checks if α is idle in a given state.

$$\begin{aligned}
&(\Delta, s, \rho)\ \mathsf{invoke}_\alpha(f(a))\ (\Delta', s', \rho') \iff \\
&\qquad (\Delta, s, \rho) \in \mathsf{idle}_\alpha \wedge s' \restriction_{\mathrm{F}} \in \nu'_F \wedge (\Delta'(\alpha) = [\mathsf{arg} : a] \wedge \forall \alpha' \neq \alpha . \Delta'(\alpha') = \Delta(\alpha') \wedge s' = s \cdot \boldsymbol{\alpha}{:}f \wedge \rho' = \rho \cdot \boldsymbol{\alpha}{:}f \\
&(\Delta, s, \rho)\ \mathsf{returned}_\alpha(f)\ (\Delta', s', \rho') \iff \\
&\qquad s' \restriction_{\mathrm{F}} \in \nu'_F \wedge (\Delta', s', \rho') = (\Delta, s, \rho) \wedge (\exists v \in \mathsf{ar}(f). \Delta(\alpha)(\mathsf{ret}) = v \wedge (\exists p . \pi_\alpha(\rho') = p \cdot v)) \\
&(\Delta, s, \rho)\ \mathsf{return}_\alpha(f)\ (\Delta', s', \rho') \iff \\
&\qquad \Delta' = \varnothing \wedge \rho' = \rho \wedge \exists v \in \mathsf{ar}(f). \exists p . \pi_\alpha(\rho) = p \cdot v \wedge s' = s \cdot \boldsymbol{\alpha}{:}v
\end{aligned}$$

Now, given a concurrent module $M = (M[\alpha])_{\alpha \in \Upsilon}$ where the local implementations are given by $M[\alpha] = (M[\alpha]^f)_{f \in F}$ verification is finalized by the following two rules:

$$\frac{\begin{array}{c}\forall f \in F . (\Delta_0, \epsilon, \epsilon) \in P[\alpha]^f \qquad \forall f \in F . P[\alpha]^f \subseteq \mathsf{idle}_\alpha \qquad \mathsf{stable}(\mathcal{R}[\alpha], P[\alpha]^f) \\ \mathsf{stable}(\mathcal{R}[\alpha], Q[\alpha]^f) \qquad \mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{\mathsf{invoke}_\alpha(f) \circ P[\alpha]^f\}\ M[\alpha]^f\ \{\mathsf{returned}_\alpha(f) \circ Q[\alpha]^f\} \\ \forall f, f' \in F . \mathsf{return}_\alpha(f') \circ \mathsf{returned}_\alpha(f') \circ Q[\alpha]^{f'} \circ \mathsf{invoke}_\alpha(f') \circ P[\alpha]^{f'} \subseteq P[\alpha]^f\end{array}}{\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{\cap_{f \in F} P[\alpha]^f\}\ M[\alpha]\ \{\cup_{f \in F} Q[\alpha]^f\}}\ \textsc{Local Impl}$$

$$\frac{\begin{array}{c}\forall \alpha \in A . \mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P[\alpha]\}\ M[\alpha]\ \{Q[\alpha]\} \\ \forall \alpha, \alpha' \in A . \alpha \neq \alpha' \Rightarrow \mathcal{G}[\alpha] \cup \mathsf{invoke}_\alpha(-) \cup \mathsf{return}_\alpha(-) \subseteq \mathcal{R}[\alpha']\end{array}}{\mathcal{R}[A], \mathcal{G}[A] \models_A \{\cap_{\alpha \in A} P[\alpha]\}\ M[A]\ \{\cup_{\alpha \in A} Q[\alpha]\}}\ \textsc{Conc Impl}$$

Several of the premises of Local Impl and Conc Impl are typical of rely-guarantee reasoning, and the remaining ones are very similar to those found in Khyzha et al. [2017, 2016]. Of note, is the highlighted premise in LocalImpl, which makes sure that the pre and post-conditions are defined in such a way that after executing a method $f' \in F$ the system satisfies all the requirements to safely execute any other method $f \in F$. Meanwhile, the highlighted premise in ConcImpl makes sure that the rely condition is stable not only under the guarantee but also under invocations and returns by other agents. These two program logic rules are justified by the following soundness theorem.

Proposition 7.2 (Soundness). *If* $\mathcal{R}[A], \mathcal{G}[A] \models_A \{P[A]\}\ M[A]\ \{Q[A]\}$ *and* $(\nu'_E : \dagger\mathbf{E}, \nu_E : \dagger\mathbf{E})$ *is a linearizable concurrent object then*

$$\nu'_E ; [\![M[A]]\!] \cap \nu'_F \subseteq K_{\mathrm{Conc}}\ \nu_F$$

The program logic can be extended with quality-of-life features like ghost state, and fancier notions of possibilities such as using a set of plays of $K_{\mathrm{Conc}}\ \nu_F$, instead of a single play, for added flexibility. Another point is that, other than paradigmatic modifications, our programming language and program logic are close to those of Khyzha et al. [2017]. There are two major differences. First, our program logic is built to reason about *our* notion of linearizability (Def. 5.1), while theirs focuses on Herlihy-Wing linearizability. In particular, their operational semantics can assume that operations in the source component are atomic, while we cannot. The second is that we maintain that *there exists* a valid linearization of the possibility, while they maintain that *every* linearization is valid. There are linearizable concurrent objects for which the stronger invariant on possibilities cannot be maintained, see our technical report [Oliveira Vale et al. 2022]. This means that our program logic is more expressive, and therefore any proof achievable with theirs should admit a straight-forward adaption to ours.

7.3 Example Revisited

We now revisit the example of §2.2. We start by assuming we have concurrent objects $v'_{\text{fai}} : \dagger\mathsf{FAI}$, $v'_{\text{counter}} : \dagger\mathsf{Counter}$ and $v'_{\text{yield}} : \dagger\mathsf{Yield}$ assembling into linearizable objects

$$(v'_{\text{fai}} : \dagger\mathsf{FAI}, v_{\text{fai}} : \dagger\mathsf{FAI}) \quad (v'_{\text{counter}} : \dagger\mathsf{Counter}, v_{\text{counter}} : \dagger\mathsf{Counter}) \quad (v'_{\text{yield}} : \dagger\mathsf{Yield}, v_{\text{yield}} : \dagger\mathsf{Yield})$$

where v_{fai} is the atomic FAI object specification, v_{counter} is the semi-racy counter specification, and v_{yield} is the less concurrent Yield specification, all as described in §2.2. Using the *locality* property, we can combine these linearizable objects into a composed linearizable object, written as (v'_E, v_E):

$$(v'_E, v_E) := (v'_{\text{fai}} \otimes v'_{\text{counter}} \otimes v'_{\text{yield}}, v_{\text{fai}} \otimes v_{\text{counter}} \otimes v_{\text{yield}})$$

Observe that the code for M_{lock} appearing in Fig. 1 can be encoded in the programming language of §7.1. We wish therefore to show that M_{lock} correctly implements a linearizable object $(v'_{\text{lock}} : \dagger\mathbf{F}, v_{\text{lock}} : \dagger\mathbf{F})$ as described in §2.2 except for one extra assumption: that locally in v'_{lock}, each agent alternates between invoking acq and rel. This extra assumption becomes available in our program logic. Because of the *interaction refinement* property, we need only consider linearized traces, those in v_E, for the source component. Because of that, it does not really matter what the actual concurrent object v'_E is! It only matters that it linearizes to v_E. For example, v'_{counter} could very well be an atomic Counter provided by hardware somehow, or a Counter implementation that misbehaves when two increments occur at the same time. Even then, it still linearizes to the semi-racy counter specification, so the proof of correctness of M_{lock} will remain valid.

Verification with the program logic is straight-forward. The main invariant maintains that the possibility ρ satisfies $\rho = p \cdot \rho_O$ where $p \in v_{\text{lock}}$ is an atomic trace representing the already linearized operations, while ρ_O is a sequence of pending invocations yet to be linearized. When an agent leaves the while loop in the code of acq, or executes the inc command in the body of rel we add the corresponding return ok and linearize the operation to the end of p, like so:

$$\rho = p \cdot \rho_1 \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot \rho_2 \xrightarrow{\texttt{assert(cur\_tick = my\_tick)}} p \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot \rho_1 \cdot \rho_2 = \rho'$$
$$\rho = p \cdot \rho_1 \cdot \boldsymbol{\alpha}{:}\mathsf{rel} \cdot \rho_2 \xrightarrow{\texttt{inc()}} p \cdot \boldsymbol{\alpha}{:}\mathsf{rel} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot \rho_1 \cdot \rho_2 = \rho'$$

Please check our technical report for details. We denote the fact that M_{lock} is *correct* as:

$$[\![M_{\text{lock}}]\!] : (v'_E, v_E) \longrightarrow (v'_{\text{lock}}, v_{\text{lock}})$$

Along the same lines, we can verify that

$$[\![M_{\text{squeue}}]\!] : (v'_{\text{lock}} \otimes v'_{\text{queue}}, v_{\text{lock}} \otimes v'_{\text{queue}}) \longrightarrow (v'_{\text{squeue}}, v_{\text{squeue}})$$

At this point, the two implementations can be composed together by using the *tensor product* of concurrent games, the *locality property* and *strategy composition*. First, we use $\mathsf{ccopy}_{\dagger\mathsf{Queue}} : \dagger\mathsf{Queue} \to \dagger\mathsf{Queue}$ to "pass-through" the queue object to M_{lock}, obtaining therefore an implementation $M_{\text{lock}} \otimes \mathsf{ccopy}$ by using the code for ccopy_ shown in §4.1. This implementation satisfies that $[\![M_{\text{lock}} \otimes \mathsf{ccopy}]\!] = [\![M_{\text{lock}}]\!] \otimes \mathsf{ccopy}_{\dagger\mathsf{Queue}}$ and therefore that

$$[\![M_{\text{lock}} \otimes \mathsf{ccopy}]\!] : (v'_E \otimes v'_{\text{queue}}, v_E \otimes v'_{\text{queue}}) \longrightarrow (v'_{\text{lock}} \otimes v'_{\text{queue}}, v_{\text{lock}} \otimes v'_{\text{queue}})$$

By composing the two implementations together, we obtain that

$$[\![M_{\text{lock}} \otimes \mathsf{ccopy}]\!]; [\![M_{\text{squeue}}]\!] : (v'_E \otimes v'_{\text{queue}}, v_E \otimes v'_{\text{queue}}) \longrightarrow (v'_{\text{squeue}}, v_{\text{squeue}})$$

immediately from the fact that each of the two implementations is known to be correct.

8 RELATED WORK AND CONCLUSION

Herlihy and Wing [1990]. We revisit many, if not all, of the major points of their now classical paper. In particular we generalize their definition and provide a new proof of locality. Overall, we present new foundations to their original definition of linearizability.

Ghica [2019]; Ghica and Murawski [2004]; Murawski and Tzevelekos [2019]. Our concurrent game model is heavily inspired by the model appearing in Ghica and Murawski [2004] and Ghica [2019], and the genesis of our key result lies in the observation we outlined in §2.1.2. Despite that, our game model both *simplifies* and *modifies* the one appearing there. It simplifies it in that they use arena-based games, relying on justification pointers. They also have more structure on their plays around a second classification of moves into *questions* or *answers*, in order to model Idealized Concurrent Algol precisely. We believe that our formulation of linearizability readily extends to other, more sophisticated formulations of concurrent games, including theirs. Our choice of this simple game semantics is justified in §1.2. We also make a significant modification to their game model in that we change the strategy composition operation. Theirs always applies a non-linear self-interleaving operation on the left strategy so to obtain a Cartesian category. We instead use a linear composition operation that leaves the left strategy as is, and fits our purposes better. Another difference is that theirs is single-threaded (a single opening O move) while ours is multi-threaded. They do use a multi-threaded model to explain the categorical structure of their model, but they do not use the multi-threaded model as extensively as we do.

The fact that the category defined in Ghica and Murawski [2004] is a Karoubi envelope was observed in a manuscript by Ghica [2019], but was not explored in detail. In particular, none of the material in §6 appears in their work. Neither of these works deal with linearizability in any way, nor observe the relationship between their rewrite relation and happens-before preservation.

The authors likely did notice that the rewrite relation in Ghica [2019]; Ghica and Murawski [2004] is related to linearizability, as a variation of it appears in Murawski and Tzevelekos [2019]. In this paper, they revisit a higher-order variation of linearizability originally introduced in Cerone et al. [2014] and strengthen the results from there. Meanwhile, we only address the more traditional first-order linearizability, though we believe it could be generalized to a higher-order setting. Despite that, they use a trace semantics, which, though inspired by game semantics, still relies on syntactic linking operations and lacks a notion of composition beyond syntactic linking at the single layer level. The approach fits into the typical approach we outline in §1. None of these works observe the relationship between ccopy_ and the Karoubi envelope with linearizability.

Goubault et al. [2018]. As we described in §2.1.2, Goubault et al. [2018] is another major reference for our work. Many of our results are significant generalizations of theirs. They focus just on concurrent object specifications, and use untyped specifications. We go beyond that by considering a compositional model, featuring linear logic types, and strategy composition. Given the definition of concurrent specification they use, and the background of the authors, they were likely inspired by game semantics, and leave for future work a compositional variant of their results, which our work addresses. Moreover, they only model non-blocking total objects, while we assume neither restriction on our objects. Some of our results are generalizations of their results along several lines, as our model is compositional, typed and does not assume totality (this last one is explicitly used to simplify several of their proofs). In particular, while they prove a Galois connection, we prove a weak biadjunction. Several of these generalizations are established using our novel techniques, such as the algebraic characterization in terms of the Karoubi envelope, as opposed to proofs involving the rewrite system. They also do not discuss horizontal composition and locality.

Other Works. There are other approaches to concurrent game semantics such as Abramsky and Mellies [1999] and Melliès and Mimram [2007] (this later one also involving a rewrite system), and to concurrent models of computation [Castellan et al. 2017; Rideau and Winskel 2011]. An important reference on the game semantics side, though we do not use the methods from there explicitly, is Mellies [2019]. Our treatment of concurrent objects, appearing in §2.2, in §7.3 and in our technical report traces back to Reddy [1993, 1996], which has been recently brought back to attention by Oliveira Vale et al. [2022]. More broadly, our motivations seem to fit into a program started by Koenig and Shao [2020]. Game semantics has been used to analyze concurrent program logics in Melliès and Stefanesco [2020] to a much larger extent than what we endeavor in §7 .

Semicategories have been studied extensively in the context of theory of computation in order to provide category theoretical formulations for models of the λ-calculus, notably in Hayashi [1985]; Hyland et al. [2006]. Our notions of semi-biadjunction and enriched semicategories trace back to Hayashi [1985] and Moens et al. [2002] respectively. Semifunctors have been thoroughly studied in Hoofman and Moerdijk [1995]. The Karoubi envelope often appears in the context of concurrent models of computation beyond the already mentioned Ghica and Murawski [2004]; for instance in Ghica [2013] to model delay insensitive circuits, in Gaucher [2020] on the flow model of concurrent computation, in Piedeleu [2019] to give a graphical language to distributed systems, or in Castellan et al. [2017]; Rideau and Winskel [2011] (though not explicitly mentioned).

As we noted in §2.2 there are many works that discuss variations of linearizability [Castañeda et al. 2015; Haas et al. 2016; Hemed et al. 2015; Neiger 1994]. Crucially, our methodology and formulation differ widely from previous works. In particular, we do not propose a notion of linearizability. Instead, we define a model of concurrent computation and derive the appropriate definition of linearizability intrinsic to the model. As far as we are aware, the only work that has noticed a relationship between the copycat and linearizability is Lesani et al. [2022], which likely happened concurrently with our own discovery. Despite that, they only discuss atomic linearizability, and do not explore the theory surrounding their definition of linearizability. In particular, they do not prove the equivalence of their definition to original Herlihy-Wing linearizability, which we address in depth in our technical report [Oliveira Vale et al. 2022]. In this way, our work generalizes their development around linearizability and, moreover, formally explains why their definition of linearizability is appropriate. In terms of methodology, our work still differs widely and subsumes their model of computation, especially when considering the object-based semantics model appearing in our technical report [Oliveira Vale et al. 2022]. The main contribution of their paper is in showing how linearizability can elegantly model transactional objects, a matter which is orthogonal to our development and readily adaptable to our setting. All the works cited *supra* are strictly less expressive than the notion of linearizability we derive. Our notion of linearizability corresponds to a generalization of interval-sequential linearizability [Castañeda et al. 2015] (the most expressive notion of linearizability prior to our work) to potentially blocking concurrent objects (while they only model non-blocking objects, as is typical in the linearizability literature). See our technical report [Oliveira Vale et al. 2022] for a detailed comparison.

For our results on proof methods for proving linearizability we must mention Herlihy and Wing [1990]; Khyzha et al. [2017]; Schellhorn et al. [2014]. In particular, our program logic and programming language are adapted from Khyzha et al. [2017, 2016], but with some substantial modifications: instead of interval partial orders, we use just a concurrent trace as our notion of possibility; we follow the object-based semantics paradigm and therefore encapsulate all state in objects instead of having programming language constructs that directly modify the shared state; while they maintain as an invariant that every linearization of their possibility is valid, we only maintain that there exists at least one valid linearization. We speculate that this last modification should make our program logic complete, while theirs is not (see our technical report [Oliveira Vale et al. 2022]). Since our program logic strictly generalizes theirs, we can translate to our program logic any proof using Khyzha et al. [2017]. Although we use the particular program logic in §7, we do not see our program logic as a major contribution of our work. Rather, it serves the purpose of illustrating the interaction of the theory with a concrete verification methodology and that objects linearizable under our notion of linearizability are verifiable. We believe that other program logics, and other proof methodologies can be connected with our framework.

There has been much work in building program logics for reasoning about concurrent programs [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Feng et al. 2007; Fu et al. 2010; Jung et al. 2018; Nanevski et al. 2014; Svendsen and Birkedal 2014; Turon et al. 2013; Vafeiadis et al. 2006; Vafeiadis and Parkinson 2007]. Most of these works only prove soundness with respect to the particular combination of Rely/Guarantee, Separation Logic and/or Concurrent Separation Logic involved, but not against linearizability. This sometimes happens even when a proof method for establishing linearizability is presented, what they justify by citing Filipovic et al. [2010] and by claiming that they can show observational refinement. This is despite the fact that their programming language, and hence, notion of refinement differs from that in Filipovic et al. [2010]. Notable exceptions in this matter are Birkedal et al. [2021]; Khyzha et al. [2017]; Liang and Feng [2016].

A close relative to linearizability is *logical atomicity* [da Rocha Pinto et al. 2014; Jung et al. 2019, 2015]. Logical atomicity does address some of the biases delineated in §1, and Jung et al. [2015]'s framework, Iris, is compositional, although only within the confines of Iris. In fact, logical atomicity is intimately tied to a program logic. Strictly speaking, it only characterizes objects realizable in a particular operational semantics, and expressible in a particular program logic. It was invented to make it easier to prove linearizability in Hoare logics. Until recently, there was no formal account of the relationship between the two. It has been recently shown [Birkedal et al. 2021] that logical atomicity implies Herlihy-Wing linearizability. There is no reason to believe the reverse implication is provable. It is, moreover, tied to atomicity. Meanwhile, linearizability (both in our treatment and in the original Herlihy-Wing paper) is not tied to a particular logical framework, or to realizability under a programming language. In the original Herlihy-Wing paper, it characterizes any non-blocking sequentially consistent concurrent object that behaves as if their operations happened atomically. The concrete part of our paper characterizes sequentially consistent concurrent objects whose operations behave as if they had linearization intervals.

Conclusion. We believe that linearizability beyond atomicity is currently underdeveloped in the theory, and hope that our analysis contributes to divorcing linearizability from atomicity, as it presents a strong argument that preservation of happens-before order is the core insight of linearizability. Along these lines, there are both practical (relaxed memory models and architectures) and theoretical (strengthening some results appearing in the appendices) reasons to consider models that are not sequentially consistent. We believe the framework presented here readily generalizes to many contexts, what we intend to explore in the future. Finally, one of the main intended applications of this work is to provide a fertile ground for developing compositional verification methods for concurrent systems, and for proving theoretical properties of such systems.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 2019285, 1763399, and 2118851, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Inf. Comput.* 163, 2 (2000), 409–470. https://doi.org/10.1006/inco.2000.2930

Samson Abramsky and Guy McCusker. 1999. Game Semantics. In *Computational Logic*, Ulrich Berger and Helmut Schwichtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–55. https://doi.org/10.1007/978-3-642-58622-4_1

S. Abramsky and P.-A. Mellies. 1999. Concurrent games and full completeness. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. IEEE Computer Society, USA, 431–442. https://doi.org/10.1109/LICS.1999.782638

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *Proc. ACM Program. Lang.* 5, ICFP, Article 81 (aug 2021), 29 pages. https://doi.org/10.1145/3473586

Andreas Blass. 1992. A Game Semantics for Linear Logic. *Ann. Pure Appl. Log.* 56, 1–3 (1992), 183–220. https://doi.org/10.1016/0168-0072(92)90073-9

Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2015. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363 (DISC 2015)*. Springer-Verlag, Berlin, Heidelberg, 420–435. https://doi.org/10.1007/978-3-662-48653-5_28

Simon Castellan, Pierre Clairambault, Silvain Rideau, and Glynn Winskel. 2017. Games and Strategies as Event Structures. *Logical Methods in Computer Science* Volume 13, Issue 3 (Sept. 2017), 49. https://doi.org/10.23638/LMCS-13(3:35)2017

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2014. Parameterised Linearisability. In *Automata, Languages, and Programming*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–109. https://doi.org/10.1007/978-3-662-43951-7_9

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *Programming Languages and Systems*, Rocco De Nicola (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–188. https://doi.org/10.5555/1762174.1762193

Ivana Filipovic, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for Concurrent Objects. *Theor. Comput. Sci.* 411, 51–52 (dec 2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Philippe Gaucher. 2020. Flows revisited: the model category structure and its left determinedness. *Cahiers de topologie et géométrie différentielle catégoriques* LXI, 2 (2020), 208–226. https://hal.archives-ouvertes.fr/hal-01919037

Dan R. Ghica. 2013. Diagrammatic Reasoning for Delay-Insensitive Asynchronous Circuits. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky: Essays Dedicated to Samson Abramsky on the Occasion of His 60th Birthday*, Bob Coecke, Luke Ong, and Prakash Panangaden (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–68. https://doi.org/10.1007/978-3-642-38164-5_5

Dan R. Ghica. 2019. The far side of the cube. *CoRR* abs/1908.04291 (2019). arXiv:1908.04291 http://arxiv.org/abs/1908.04291

Dan R. Ghica and Andrzej S. Murawski. 2004. Angelic Semantics of Fine-Grained Concurrency. In *Foundations of Software Science and Computation Structures*, Igor Walukiewicz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–225. https://doi.org/10.1016/j.apal.2007.10.005

Éric Goubault, Jérémy Ledent, and Samuel Mimram. 2018. Concurrent Specifications Beyond Linearizability. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira (Eds.), Vol. 125. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 28:1–28:16. https://doi.org/10.4230/LIPIcs.OPODIS.2018.28

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 653–669.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM*

SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018). Association for Computing Machinery, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

Rachid Guerraoui and Eric Ruppert. 2014. Linearizability Is Not Always a Safety Property. In *Networked Systems*, Guevara Noubir and Michel Raynal (Eds.). Springer International Publishing, Cham, 57–69. https://doi.org/10.1007/978-3-319-09581-3_5

Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. 2016. Local Linearizability for Concurrent Container-Type Data Structures. In *27th International Conference on Concurrency Theory (CONCUR 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Josée Desharnais and Radha Jagadeesan (Eds.), Vol. 59. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.6

Susumu Hayashi. 1985. Adjunction of semifunctors: Categorical structures in nonextensional λ calculus. *Theoretical Computer Science* 41 (1985), 95–104. https://doi.org/10.1016/0304-3975(85)90062-3

Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363 (DISC 2015)*. Springer-Verlag, Berlin, Heidelberg, 371–387. https://doi.org/10.1007/978-3-662-48653-5_25

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

R. Hoofman and I. Moerdijk. 1995. A remark on the theory of semi-functors. *Mathematical Structures in Computer Science* 5, 1 (1995), 1–8. https://doi.org/10.1017/S096012950000061X

J. M. E. Hyland and C.-H. L. Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. https://doi.org/10.1006/inco.2000.2917

Martin Hyland. 1997. Game Semantics. In *Semantics and Logics of Computation*, Andrew M. Pitts and P.Editors Dybjer (Eds.). Cambridge University Press, Cambridge, UK, 131–184. https://doi.org/10.1017/CBO9780511526619.005

Martin Hyland, Misao Nagayama, John Power, and Giuseppe Rosolini. 2006. A Category Theoretic Formulation for Engeler-style Models of the Untyped λ-Calculus. *Electronic Notes in Theoretical Computer Science* 161 (2006), 43–57. https://doi.org/10.1016/j.entcs.2006.04.024 Proceedings of the Third Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT 2004).

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

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2019. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 45 (dec 2019), 32 pages. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *SIGPLAN Not.* 50, 1 (jan 2015), 637–650. https://doi.org/10.1145/2775051.2676980

Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. 2017. Proving Linearizability Using Partial Orders. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings.* Springer-Verlag, Berlin, Heidelberg, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

Artem Khyzha, Alexey Gotsman, and Matthew Parkinson. 2016. A Generic Logic for Proving Linearizability. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 426–443. https://doi.org/10.1007/978-3-319-48989-6_26

Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. Association for Computing Machinery, New York, NY, USA, 633–647. https://doi.org/10.1145/3373718.3394799

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: Verified Transactional Objects. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 80 (apr 2022), 31 pages. https://doi.org/10.1145/3527324

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects under Fair Scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 385–399. https://doi.org/10.1145/2837614.2837635

Paul-André Mellies. 2019. Categorical Combinatorics of Scheduling and Synchronization in Game Semantics. *Proc. ACM Program. Lang.* 3, POPL, Article 23 (jan 2019), 30 pages. https://doi.org/10.1145/3290336

Paul-André Melliès and Samuel Mimram. 2007. Asynchronous Games: Innocence Without Alternation. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 395–411. https://doi.org/10.1007/978-3-540-74407-8_27

Paul-André Melliès and Léo Stefanesco. 2020. Concurrent Separation Logic Meets Template Games. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3373718.3394762

M.-A. Moens, U. Berni-Canani, and Francis Borceux. 2002. On regular presheaves and regular semi-categories. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 43, 3 (2002), 163–190. http://www.numdam.org/item/CTGDC_2002__43_3_163_0/

Andrzej S. Murawski and Nikos Tzevelekos. 2019. Higher-order linearisability. *Journal of Logical and Algebraic Methods in Programming* 104 (2019), 86–116. https://doi.org/10.1016/j.jlamp.2019.01.002

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

Gil Neiger. 1994. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '94)*. Association for Computing Machinery, New York, NY, USA, 396. https://doi.org/10.1145/197917.198176

Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. 2022. Layered and Object-Based Game Semantics. *Proc. ACM Program. Lang.* 6, POPL, Article 42 (jan 2022), 32 pages. https://doi.org/10.1145/3498703

Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. 2022. *A Compositional Theory of Linearizability.* Technical Report YALEU/DCS/TR-1564. Yale Univ. https://flint.cs.yale.edu/publications/ctlinear.html

R Piedeleu. 2019. *Picturing resources in concurrency.* Ph.D. Dissertation. University of Oxford.

Uday S. Reddy. 1993. *A Linear Logic Model of State.* Technical Report. Dept. of Computer Science, UIUC, Urbana, IL.

Uday S. Reddy. 1996. Global State Considered Unnecessary: An Introduction to Object-Based Semantics. *LISP Symb. Comput.* 9, 1 (1996), 7–76. https://doi.org/10.1007/978-1-4757-3851-3_9

Silvain Rideau and Glynn Winskel. 2011. Concurrent Strategies. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, USA, 409–418. https://doi.org/10.1109/LICS.2011.13

Gerhard Schellhorn, John Derrick, and Heike Wehrheim. 2014. A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures. *ACM Trans. Comput. Logic* 15, 4, Article 31 (sep 2014), 37 pages. https://doi.org/10.1145/2629496

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 377–390. https://doi.org/10.1145/2500365.2500600

Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. 2006. Proving Correctness of Highly-Concurrent Linearisable Objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. Association for Computing Machinery, New York, NY, USA, 129–136. https://doi.org/10.1145/1122971.1122992

Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–271.

Received 2022-07-07; accepted 2022-11-07