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**.

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), 93 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 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].

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.

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

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, 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

$$\boldsymbol{\alpha_1}\!:\!\mathsf{acq} \qquad \boldsymbol{\alpha_2}\!:\!\mathsf{acq} \longrightarrow \boldsymbol{\alpha_2}\!:\!\mathsf{ok} \qquad \boldsymbol{\alpha_3}\!:\!\mathsf{acq} \qquad \boldsymbol{\alpha_2}\!:\!\mathsf{acq} \qquad \boldsymbol{\alpha_3}\!:\!\mathsf{ok} \qquad \boldsymbol{\alpha_2}\!:\!\mathsf{ok}$$

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

$$\boldsymbol{\alpha_2}\!:\!\mathsf{acq} \longrightarrow \boldsymbol{\alpha_2}\!:\!\mathsf{ok} \longrightarrow \boldsymbol{\alpha_2}\!:\!\mathsf{acq} \longrightarrow \boldsymbol{\alpha_2}\!:\!\mathsf{ok} \qquad \boldsymbol{\alpha_3}\!:\!\mathsf{acq} \longrightarrow \boldsymbol{\alpha_3}\!:\!\mathsf{ok}$$

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 $v'_{\mathsf{lock}} \subseteq P_{\dagger\mathsf{Lock}}$. To be correct this specification v'_{lock} should linearize to the atomic specification $v_{\mathsf{lock}} \subseteq P_{\dagger\mathsf{Lock}}$ given by the set of traces $s \in P_{\dagger\mathsf{Lock}}$ such that:

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

- 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:

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

Its concurrent specification v'_{queue} can be specified as the largest set of traces $s \in P_{\dagger\mathsf{Queue}}$ such that if $s = p \cdot \boldsymbol{\alpha}\!:\!\mathsf{deq} \cdot \boldsymbol{\alpha}\!:\!k \cdot s'$ and p is atomic then either $\mathsf{qstate}(p) = k :: q'$ or $\mathsf{qstate}(p) = [\,]$ and $k = \varnothing$, where qstate is an inductively defined function taking an atomic trace p and returning the state $\mathsf{qstate}(p)$ 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 $M_{\mathsf{squeue}} : \mathsf{Lock} \otimes \mathsf{Queue} \multimap \mathsf{Queue}$ 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 $v'_{\mathsf{lock}} \otimes v'_{\mathsf{queue}}$ (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

```
M_squeue :
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
}                   }
```

```
M_lock :
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

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

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:

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

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}.

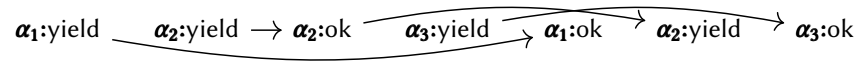
Occasionally, one implements the ticket lock so that it yields while spinning so to let other agents get access to the underlying computational resource (such as processor time). For some purposes, this is crucial to obtain better liveness properties. For this, we define a signature

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

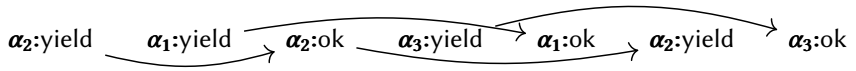
with concurrent specification v'_{yield} given by

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

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:



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:



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.

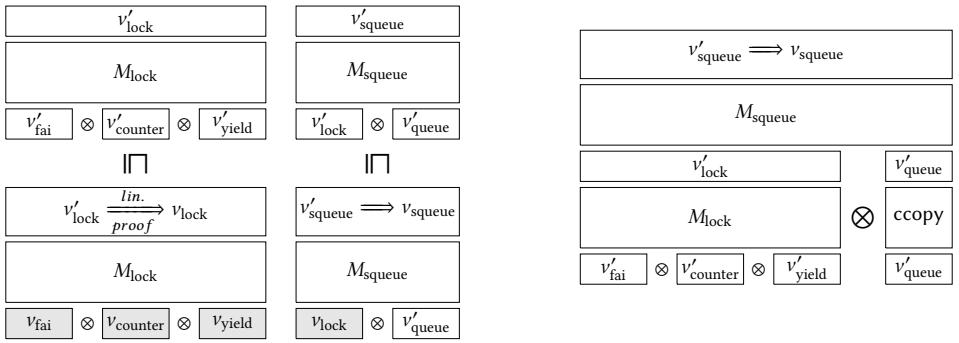


Fig. 2. In our compositional model, off-the-shelf components can be composed horizontally by using the linear logic tensor product $- \otimes -$. Each component's implementation is verified against its linearized specification individually (left). Refinement and generalized linearizability allow to use 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
}                 ||          ||  }
      ⋮           ||          ||        ⋮
```

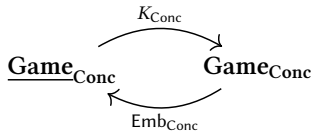


Fig. 3. Code corresponding to ccopy_ (left); Diagram depicting the operations K_{Conc} and $\mathrm{Emb}_{\mathrm{Conc}}$ (right)

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 $s\restriction_{\mathbf{A}}$ 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*

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

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). *$v'_A : \mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}$ is linearizable to $v_A : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ if and only if for all concurrent games $\mathbf{B}$ and $\sigma : \mathbf{A} \multimap \mathbf{B}$ it holds that*

$$v'_A; \sigma \subseteq v_A; \sigma$$

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 $\mathbf{Game}_{\mathbf{Seq}}$ 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 $s\restriction_A$ 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:

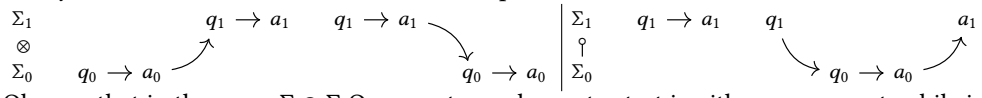
$$M^O_{A\otimes B} := M^O_A + M^O_B \quad M^P_{A\otimes B} := M^P_A + M^P_B \quad P_{A\otimes B} := \{s \in \mathsf{Alt}(M^O_{A\otimes B}, M^P_{A\otimes B}) \mid s\restriction_A \in P_A \wedge s\restriction_B \in P_B\}$$

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.



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 $\mathbf{Game}_{\mathbf{Seq}}$ of sequential games defined in the following.

Definition 3.8. The category $\mathbf{Game}_{\mathbf{Seq}}$ 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 $\mathrm{copy}_A : A \multimap A$.

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 $\mathbf{Game}_{\mathbf{Seq}}(E)$ 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 $\mathbf{Game}_{\mathbf{Seq}}(E)$. 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 $v \in \mathsf{ar}(e)$.

$$E : \qquad e \longrightarrow v \qquad\qquad \dagger E : \qquad e_1 \longrightarrow v_1 \longrightarrow e_2 \longrightarrow v_2 \longrightarrow \ldots \longrightarrow e_n \longrightarrow v_n$$

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.

	$\mathsf{inc} \longrightarrow \mathsf{ok}$	$\mathsf{get} \longrightarrow 3 \longrightarrow \mathsf{inc} \longrightarrow \mathsf{ok} \longrightarrow \mathsf{get} \longrightarrow 7 \longrightarrow \mathsf{get} \longrightarrow 2 \longrightarrow \mathsf{inc}$
$\forall n \in \mathbb{N}.$	$\mathsf{get} \longrightarrow n$	$\mathsf{inc} \longrightarrow \mathsf{ok} \longrightarrow \mathsf{get} \longrightarrow 1 \longrightarrow \mathsf{get} \longrightarrow 1 \longrightarrow \mathsf{inc} \longrightarrow \mathsf{ok}$

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}:

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

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 $P_{\boldsymbol{A}}$, where O-receptivity is defined by:

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

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 $-;-$.

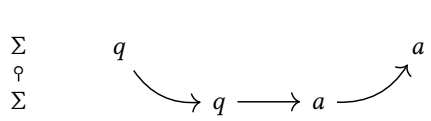
We define the game $\dagger\mathbf{E}$ of concurrent traces over the signature E by first defining $\mathbf{E} := (M_E, P_E)$ and then $\dagger\mathbf{E} := (M_{\dagger E}, P_{\dagger E})$. So the game $\dagger\mathbf{E}$ has each agent playing the corresponding sequential game $\dagger E$ concurrently. 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 Appendix E.

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 $\mathbf{Game}_{\mathbf{Conc}}$ 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):

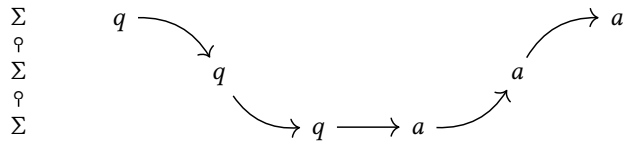



```
Import Σ

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:



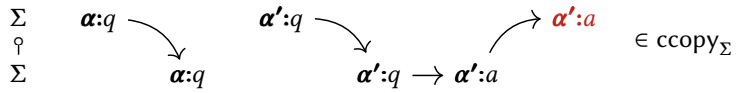

```
Import Σ₀

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

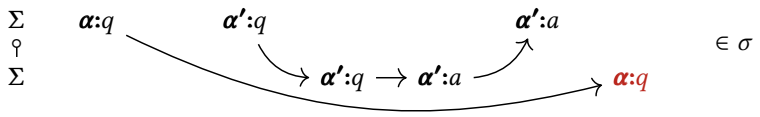
Fig. 5. Maximal play of $\mathrm{int}(\mathrm{copy}_\Sigma, \mathrm{copy}_\Sigma)$ (left) and corresponding pseudocode (right).

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.

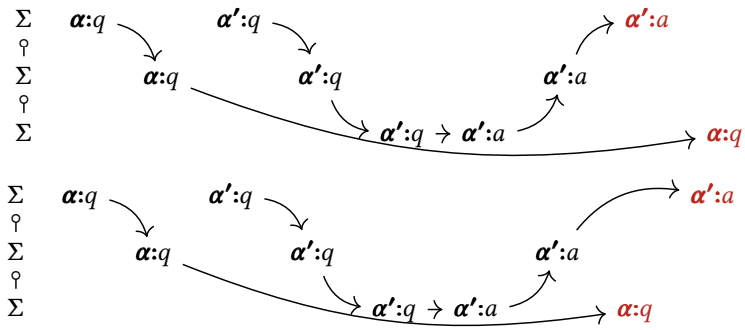
In the concurrent version $\Sigma \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ of Σ each agent of Υ locally plays Σ. The obvious neutral element in this situation would be to have each agent $\alpha, \alpha' \in \Upsilon$ locally run copy_Σ, a strategy we call $\mathrm{ccopy}_\Sigma : \Sigma \multimap \Sigma$, 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:



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



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



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:

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

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 $\mathrm{ccopy}\_$ 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 $\mathrm{ccopy}\_$. 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 $\mathrm{ccopy}\_$ 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 semifunctor 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

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

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 $(P_{\mathbf{A}}, \rightsquigarrow_{\boldsymbol{A}})$ with local rewrite rules:

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

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*

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

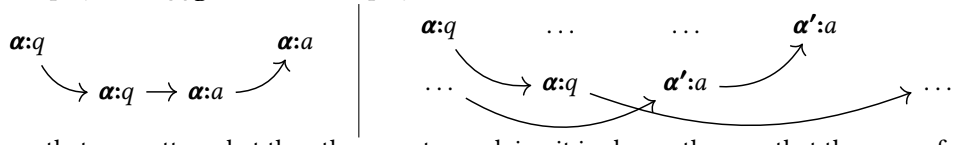
The key lemma to show this alternative characterization is the synchronization lemma, as coined by Ghica [2019], which plays a similar role to concurrent games as does the switching condition in sequential games. It essentially establishes that there is still synchronization happening under this liberal 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 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* $\sigma = \text{strat}\,(p \cdot \boldsymbol{\alpha}{:}m \cdot \boldsymbol{\alpha'}{:}m' \cdot p')$. *Then,*

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

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 $\text{ccopy}_{\mathbf{B}}$ behaves as displayed below on the left.



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 $\text{ccopy}_{\mathbf{B}}$, such as in strat (s) ; $\text{ccopy}_{\mathbf{B}}$, 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 $\mathbf{Game}_{\mathbf{Conc}}$ 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 $\tau : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ 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 Appendix D. When the linearized strategy is specialized to atomic strategies only, we obtain Herlihy-Wing linearizability. In Appendix B 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* $\tau : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$

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

Proof. Suppose $s \in K_{\mathsf{Conc}}\ \tau$. 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, $s \cdot s_P \in K_{\mathsf{Conc}}\ \tau$. By prefix-closure, $s \in K_{\mathsf{Conc}}\ \tau$, 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*

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

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:

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

Together with Corollary 5.3 it follows that $\sigma = K_{\mathsf{Conc}}\ \tau$ 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:

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

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 $\sigma = K_{\mathsf{Conc}}\ \tau$, as we showed above. Meanwhile, when σ is linearizable to τ but not strongly linearizable, we have a strict containment $\sigma \subset K_{\mathsf{Conc}}\ \tau$.

5.3 Computational Interpretation of Linearizability

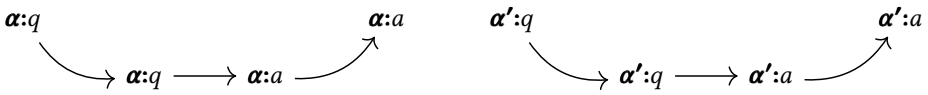
We just saw that linearizability can be characterized by the transformation K_{Conc}. We now offer yet another perspective on linearizability by providing a computational interpretation of linearizability proofs. Recall that in our discussion in §4.1 we observed that ccopy_ is the denotation of a concrete 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 $s \in \mathsf{ccopy}_{\mathbf{A}}$ 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 $s \in \mathsf{ccopy}_{\mathbf{A}}$ satisfies $s\restriction_{\mathbf{A}_1} \rightsquigarrow_{\mathbf{A}} s\restriction_{\mathbf{A}_0}$. Then, prefix-closure and receptivity of $\mathsf{ccopy}_{\mathbf{A}}$ allow for linearizability to be used instead of $- \rightsquigarrow_{\_} -$, similarly to the the proof of Prop. 5.2. See Appendix H 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 $\mathsf{ccopy}_{\mathbf{A}}$ 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}:



5.4 Interaction Refinement

One is often interested in implementing an interface of type **B** making use of some other interface of type **A** by using an implementation specified as a saturated strategy of type $\sigma : \mathbf{A} \multimap \mathbf{B}$. Now, the game **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). *$v'_A : \mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}$ is linearizable to $v_A : \mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}$ if and only if for all concurrent games $\mathbf{B}$ and $\sigma : \mathbf{A} \multimap \mathbf{B} \in \mathbf{Game}_{\mathbf{Conc}}$ it holds that*

$$v'_A; \sigma \subseteq v_A; \sigma$$

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

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

For the reverse direction, simply observe that:

$$v'_A \subseteq v'_A; \mathrm{ccopy}_{\mathbf{A}} \subseteq v_A; \mathrm{ccopy}_{\mathbf{A}} = \mathrm{ccopy}_{\mathbf{1}}; v_A; \mathrm{ccopy}_{\mathbf{A}} = K_{\mathrm{Conc}}\ v_A$$

□

This immediately implies a stronger result under strong linearizability

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

$$v'_A; \sigma = v_A; \sigma$$

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 $\mathbf{Game}_{\mathbf{Conc}}$.

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

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

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

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

This rather simple result is rather auspicious given the computational interpretation of $\mathrm{ccopy}_{\_}$ 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.

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

□

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*

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

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 $\mathsf{Emb}_{\mathrm{Conc}}$ in the main development. An extended version of this section is available in Appendix A.

Given a semicategory $\mathbf{C}$ the Karoubi envelope is the category $\mathsf{Kar}\ \mathbf{C}$ 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 $\mathsf{Kar}\ \mathbf{C}$ 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 $\mathsf{Kar}\ \mathbf{C}$ 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

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

which given a category $\mathbf{C}$ assigns a semicategory $\mathsf{Semi}\ \mathbf{C}$ 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

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

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

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:

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

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 $\mathbf{Game}_{\mathbf{Conc}} = (\underline{\mathbf{Game}}_{\mathbf{Conc}})_{\mathsf{ccopy}}$ 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

$$h \circ f \Rrightarrow h \circ g$$

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 Appendices E and F. 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.
- $\mathsf{assert}(\phi)$ takes a boolean function over the local environment and terminates computation if ϕ evaluates to False. $\mathsf{assert}(-)$ 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

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

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

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

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

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

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

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

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

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

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

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

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 Appendix E, 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

$$[\![M]\!] = \{s \mid \exists c \in \mathsf{Cont}. \exists \Delta \in (\Upsilon \to \mathsf{Env}). \langle c_0, \Delta_{\mathbf{0}}, \epsilon\rangle \twoheadrightarrow^{M} \langle c, \Delta, s\rangle\}$$

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 $s\restriction_{\mathbf{F}}$ 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):

$$\mathcal{G} \vdash_\alpha \{P\}\ B\ \{Q\} \iff$$
$$\forall (\Delta, s, \rho).\, s\restriction_{\mathbf{F}} \in v'_F \wedge (\Delta, s, \rho) \in P \wedge s\restriction_{\mathbf{E}} \in v_E \wedge (\Delta', s') \in [\![B]\!]_\alpha(\Delta, s) \Rightarrow$$
$$s'\restriction_{\mathbf{F}} \in v'_F \wedge \exists \rho'.\, (\Delta, s, \rho)\ Q\ (\Delta', s', \rho') \wedge (\Delta, s, \rho)\ \mathcal{G}\ (\Delta', s', \rho') \wedge \rho \dashrightarrow \rho'$$
$$\rho \dashrightarrow \rho' \iff \exists t_P \in (M^P_{\mathbf{F}})^*.\, \rho \cdot t_P \rightsquigarrow_{\dagger\mathbf{F}} \rho'$$

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

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 $C \in \mathsf{Com}$ 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.

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

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* $(v'_E : \dagger\mathbf{E}, v_E : \dagger\mathbf{E})$ *is a linearizable concurrent object then*

$$v'_E; [\![M[A]]\!] \cap v'_F \subseteq K_{\mathsf{Conc}}\ v_F$$

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 $K_{\mathsf{Conc}}\ v_F$, 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 F. 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 $\nu'_{\text{fai}} : \dagger\text{FAI}$, $\nu'_{\text{counter}} : \dagger\text{Counter}$ and $\nu'_{\text{yield}} : \dagger\text{Yield}$ assembling into linearizable objects

$$(\nu'_{\text{fai}} : \dagger\text{FAI}, \nu_{\text{fai}} : \dagger\text{FAI}) \quad (\nu'_{\text{counter}} : \dagger\text{Counter}, \nu_{\text{counter}} : \dagger\text{Counter}) \quad (\nu'_{\text{yield}} : \dagger\text{Yield}, \nu_{\text{yield}} : \dagger\text{Yield})$$

where ν_{fai} is the atomic FAI object specification, ν_{counter} is the semi-racy counter specification, and ν_{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 (ν'_E, ν_E):

$$(\nu'_E, \nu_E) := (\nu'_{\text{fai}} \otimes \nu'_{\text{counter}} \otimes \nu'_{\text{yield}}, \nu_{\text{fai}} \otimes \nu_{\text{counter}} \otimes \nu_{\text{yield}})$$

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 $(\nu'_{\text{lock}} : \dagger\mathbf{F}, \nu_{\text{lock}} : \dagger\mathbf{F})$ as described in §2.2 except for one extra assumption: that locally in ν'_{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 ν_E, for the source component. Because of that, it does not really matter what the actual concurrent object ν'_E is! It only matters that it linearizes to ν_E. For example, ν'_{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 $p \in \nu_{\text{lock}}$ 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:

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

$$\rho = p \cdot \rho_1 \cdot \boldsymbol{\alpha}\text{:rel} \cdot \rho_2 \xrightarrow{\texttt{inc()}} p \cdot \boldsymbol{\alpha}\text{:rel} \cdot \boldsymbol{\alpha}\text{:ok} \cdot \rho_1 \cdot \rho_2 = \rho'$$

Please check Appendix G for details. We denote the fact that M_{lock} is *correct* as:

$$[\![M_{\text{lock}}]\!] : (\nu'_E, \nu_E) \longrightarrow (\nu'_{\text{lock}}, \nu_{\text{lock}})$$

Along the same lines, we can verify that

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

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 $\text{ccopy}_{\dagger\text{Queue}} : \dagger\text{Queue} \to \dagger\text{Queue}$ to "pass-through" the queue object to M_{lock}, obtaining therefore an implementation $M_{\text{lock}} \otimes \text{ccopy}$ by using the code for ccopy_ shown in §4.1. This implementation satisfies that $[\![M_{\text{lock}} \otimes \text{ccopy}]\!] = [\![M_{\text{lock}}]\!] \otimes \text{ccopy}_{\dagger\text{Queue}}$ and therefore that

$$[\![M_{\text{lock}} \otimes \text{ccopy}]\!] : (\nu'_E \otimes \nu'_{\text{queue}}, \nu_E \otimes \nu'_{\text{queue}}) \longrightarrow (\nu'_{\text{lock}} \otimes \nu'_{\text{queue}}, \nu_{\text{lock}} \otimes \nu'_{\text{queue}})$$

By composing the two implementations together, we obtain that

$$[\![M_{\text{lock}} \otimes \text{ccopy}]\!]; [\![M_{\text{squeue}}]\!] : (\nu'_E \otimes \nu'_{\text{queue}}, \nu_E \otimes \nu'_{\text{queue}}) \longrightarrow (\nu'_{\text{squeue}}, \nu_{\text{squeue}})$$

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 Appendix E 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 and Appendix F.

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 Appendix B. 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 Appendix E. 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 Appendix D 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 Appendix F for a counterexample).

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* (Tokyo, Japan) *(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* (Mumbai, India) *(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* (Savannah, GA, USA) *(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 (Philadelphia, PA, USA) *(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* (Tokyo, Japan) *(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* (Uppsala, Sweden). 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* (Saarbrücken, Germany) *(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* (St. Petersburg, FL, USA) *(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* (Saarbrücken, Germany) *(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* (Los Angeles, California, USA) *(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

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* (Boston, Massachusetts, USA) *(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* (New York, New York, USA) *(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.

SUMMARY OF THE APPENDICES

A contains an extended development on the Karoubi envelope appering in §6 and other associated constructions that we use.
B develops the restriction of the theory of linearizability we developed in the core paper to the case where the linearized strategy is atomic, and then specialize the results appearing in §5 to Herlihy-Wing linearizability.
C gives a detailed account of the symmetric monoidal closed structure on concurrent games, and provides the proof of the key results (Prop. 5.10 and bi-semifunctoriality) required to show the generalized locality property.
D briefly compares our definition of linearizability with interval-sequential linearizability.
E defines an object-based semantics on concurrent games and gives a formal definition of linearizable concurrent objects.
F defines a program logic for showing that layered concurrent object implementations are linearizable to their specifications, which is sound for generalized linearizability, and potentially complete.
G gives detailed proofs of the examples on §2 using the program logic presented in §7.
H collects proofs omitted elsewhere in the text.

A KAROUBI ENVELOPE

In this section, we establish the main abstract tools we use to construct models of concurrent computation, and sometimes compare them with each other. Most of it requires only basic category theory, as well as knowing the definition of a semicategory, although basic knowledge about enriched category theory and weak notions of functoriality and adjointness is necessary for later parts. 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 $\mathsf{Emb}_{\mathsf{Conc}}$ in the main development.

A.1 The Karoubi Envelope

Typically, given a semicategory $\mathbf{C}$ we can construct its Karoubi envelope as the category $\mathsf{Kar}\,\mathbf{C}$ 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 $\mathsf{Kar}\,\mathbf{C}$ is a morphism $f : C \to C'$ of the underlying semicategory $\mathbf{C}$ that is invariant upon the idempotents involved in the sense that:

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

or equivalently:

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

which we call a *saturated* morphism of $\mathbf{C}$. Observe that by construction the Karoubi envelope $\mathsf{Kar}\,\mathbf{C}$ 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

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

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 similarly forgetting the fact that it maps neutral elements to neutral elements. Interestingly Semi admits a right adjoint

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

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

$$F : \mathbf{C} \to \mathbf{D} \overset{\mathsf{Kar}}{\longmapsto} \mathsf{Kar}\ F : \mathsf{Kar}\ \mathbf{C} \to \mathsf{Kar}\ \mathbf{D}$$

is defined by

$$(C, e) \overset{\mathsf{Kar}\ F}{\longmapsto} (F\,C, F\,e) \qquad f : (C, e_C) \to (C', e') \overset{\mathsf{Kar}\ F}{\longmapsto} F\,f : (F\,C, F\,e) \to (F\,C', F\,e')$$

Typically in the literature, one studies the Karoubi envelope from the perspective of categories by considering the monad associated to the adjunction. Instead, we put special focus on the comonad associated to the adjunction, so that we may study the Karoubi envelope from the perspective of semicategories:

$$\mathsf{SemiKar} : \mathbf{SemiCat} \to \mathbf{SemiCat}$$

Note that this comonad assigns to a semicategory $\mathbf{C}$ the semicategory Semi Kar $\mathbf{C}$, and acts as the identity on semifunctors.

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:

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

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. Worse yet, this mapping assembles into a functor if and only if for any $f : C \to D$ and $g : D \to E$ we have:

$$e_E \circ g \circ e_D \circ f \circ e_C = e_E \circ g \circ f \circ e_C$$

While this condition is trivial when $\mathbf{C}$ is a category and we take $e_C = \mathbf{id}_C$, in the semicategory case, given a family of idempotents $(e_C : C \to C)_{C \in \mathbf{C}}$ there is no canonical such semifunctor. Despite that there is always a forgetful semifunctor:

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

given by the mapping

$$(C, e) \overset{\mathsf{Emb}}{\longmapsto} C \qquad f : (C, e) \to (D, e') \overset{\mathsf{Emb}}{\longmapsto} f : C \to D$$

A.2 Embeddable Subcategories of the Karoubi Envelope

We just saw that the canonical functor embedding a category inside its Karoubi envelope amounts to a choice of an idempotent for each object of the category and that with semicategories there is no canonical family we can choose. 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 this 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 $\mathbf{C}$, obtained 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 also a candidate for a semifunctor in the reverse direction:

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

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.

We are now ready to define abstract linearizability. For this, we will assume that $\mathbf{C}$ is enriched semicategory.

Definition A.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 A.2. *In the following let* $\mathbf{C}$ *and* $\mathbf{C}_e$ *satisfy the conditions of 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 $f : C \longrightarrow C'$ and $g : D \longrightarrow D'$*

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

and if moreover, for all $C, C', D, D' \in \mathbf{C}$

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

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

Proof. Essentially the same proofs as the corresponding proofs we presented in §5.4 and §5.5.

Interaction Refinement

$$h \circ f \Rightarrow h \circ K_e\ g = h \circ (e_C \circ g \circ e_1) = (h \circ e_C) \circ (g \circ e_1) = h \circ g$$

For the reverse direction simply observe that:

$$f = e_C \circ f \Rightarrow e_C \circ g = e_C \circ g \circ e_1 = K_e\ g$$

Locality For the first claim:

$$\begin{aligned}
K_e(f \otimes g) &= e_{C' \otimes D'} \circ (f \otimes g) \circ e_{C \otimes D} && \text{(Def.)} \\
&= (e_{C'} \otimes e_{D'}) \circ (f \otimes g) \circ (e_C \otimes e_D) && (- \otimes - \text{ is a bifunctor in } \mathbf{C}_e) \\
&= (e_{C'} \circ f \circ e_C) \otimes (e_{D'} \circ g \circ e_D) && (\text{bi-semifunctoriality of } - \otimes -) \\
&= K_e\ f \otimes K_e\ g && \text{(Def.)}
\end{aligned}$$

For the second claim observe first that

$$f'_C \otimes f'_D \Rightarrow K_e\ f_C \otimes K_e\ f_D = K_e(f_C \otimes f_D)$$

for the reverse direction, observe that

$$f'_C \otimes f'_D \Rightarrow K_e\ (f_C \otimes f_D) = K_e\ f_C \otimes K_e\ f_D$$

by assumption we have that

$$\mathbf{C}_e(\mathbf{1}, C) \otimes \mathbf{C}_e(\mathbf{1}, D) \cong \mathbf{C}_e(\mathbf{1}, C) \times \mathbf{C}_e(\mathbf{1}, D)$$

and hence we obtain that

$$f'_C \Rightarrow K_e\ f_C \qquad\qquad f'_D \Rightarrow K_e\ f_D$$

□

Note that Prop. 6.2 does not require that K_e functorial in any way. In practice, K_e is often an (op)lax semifunctor in that there is either a 2-morphism:

$$K_e\ g \circ K_e\ f \Rightarrow K_e\ (g \circ f)$$

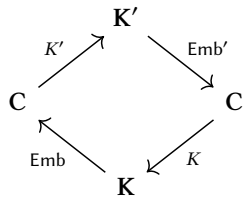
in which case K_e is called lax, or a 2-morphism:

$$K_e\ (g \circ f) \Rightarrow K_e\ g \circ K_e\ f$$

in which case K_e is oplax.

A.3 Comparing Embeddable Categories

We are interested in comparing two embeddable subcategories $\mathbf{C}_e$ and $\mathbf{C}_{e'}$, such as when $\mathbf{C}_e$ corresponds to concurrent games and $\mathbf{C}_{e'}$ corresponds to atomic games. For this we assume that the enrichment of $\mathbf{C}$ is then over a category, so that we may see it as a semibicategory. For the sake of brevity, we will call these two categories $\mathbf{K} = \mathbf{C}_e$ and $\mathbf{K}' = \mathbf{C}_{e'}$ and the corresponding mappings Emb, K and Emb', K'. Note that we can readily consider the square:



This suggests that we may define a pair of an oplax semifunctor L and a lax semifunctor R defined as:

$$L : \mathbf{K} \to \mathbf{K}' := K' \circ \mathsf{Emb} \qquad R : \mathbf{K}' \to \mathbf{K} := K \circ \mathsf{Emb}'$$

As the families e_- and e'_- define the identities on K and K' respectively, we wish to be as close as possible from producing functors between them, so we wish there to be 2-morphisms:

$$L\ e_A \Rightarrow e'_A \qquad\qquad e_A \Rightarrow R\ e'_A$$

which make L and R into oplax and lax functors respectively. Interestingly, this implies that L and R satisfy

$$f \Rightarrow R\ L\ f \qquad\qquad L\ R\ f' \Rightarrow f'$$

for any $f : A \to B \in \mathbf{K}$ and $f' : A \to B \in \mathbf{K}'$. This will be the key property to establish the following simple but important result:

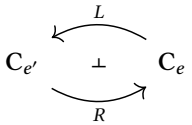
Proposition A.3. *Let*

$$e_- = \{e_C\}_{C \in \mathbf{C}} \qquad\qquad e'_- = \{e'_C\}_{C \in \mathbf{C}}$$

be families of idempotents such that the mappings L and R defined by

$$L : \mathbf{C}_e \to \mathbf{C}_{e'} := K' \circ \mathsf{Emb} \qquad R : \mathbf{C}_{e'} \to \mathbf{C}_e := K \circ \mathsf{Emb}'$$

define an oplax functor and a lax functor respectively. Then L and R assemble into an oplax-lax biadjunction of oplax and lax functors:



Proof. See §A.4 □

Note that in the above proposition we do not require K and K' to be even (op)lax semifunctors. Indeed, although in all of our models they will be, this is not required to show Proposition A.3. Moreover, the following simple condition is sufficient to establish the assumptions of Proposition A.3.

Proposition A.4. *If*

$$e_- = \{e_A\}_{A \in \mathbf{C}} \qquad\qquad e'_- = \{e'_A\}_{A \in \mathbf{C}}$$

are families of idempotents such that there are 2-morphisms:

$$e_{\mathbf{A}} \Rightarrow e'_{\mathbf{A}}$$

for every $A \in S$, then the mappings L and R defined by

$$L : \mathbf{C}_e \to \mathbf{C}_{e'} := K' \circ \mathsf{Emb} \qquad R : \mathbf{C}_{e'} \to \mathbf{C}_e := K \circ \mathsf{Emb}'$$

define an oplax functor and a lax functor respectively.

Proof. See §A.4. □

A.4 Proof of Prop. A.3 and Prop. A.4

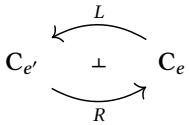
Proposition A.5. *Let*

$$e_- = \{e_A\}_{A \in \mathbf{C}} \qquad e'_- = \{e'_A\}_{A \in \mathbf{C}}$$

be families of idempotents such that the mappings L and R defined by

$$L : \mathbf{C}_e \to \mathbf{C}_{e'} := K' \circ \mathsf{Emb} \qquad R : \mathbf{C}_{e'} \to \mathbf{C}_e := K \circ \mathsf{Emb}'$$

define an oplax functor and a lax functor respectively. Then L and R assemble into an oplax-lax biadjunction of oplax and lax functors:



Proof. For convenience we let

$$\mathbf{K} = \mathbf{C}_e \qquad \mathbf{K}' = \mathbf{C}_{e'}$$

and

$$K = K_e \qquad \mathsf{Emb} = \mathsf{Emb}_e \qquad K' = K_{e'} \qquad \mathsf{Emb}' = \mathsf{Emb}_{e'}$$

We also find that it causes no confusion to refer to the objects

$$e_A \in \mathbf{K} \qquad e'_A \in \mathbf{K}'$$

as simply A, since there is a single object of $\mathbf{K}$ and $\mathbf{K}'$ that corresponds to an idempotent of $A \in \mathbf{C}$.

The key idea is to use as unit of the biadjunction

$$\eta_- : \mathbf{1} \Rightarrow R \circ L$$

the family e_- itself:

$$\eta_A : A \Rightarrow R\ L\ A := e_A$$

which is well-typed as

$$R\ L\ A = K\ \mathsf{Emb}'\ K'\ \mathsf{Emb}\ A = A$$

Similarly, we let

$$\epsilon_- : L \circ R \Rightarrow \mathbf{1}$$

be defined as the family e'_-:

$$\epsilon_A : L\ R\ A \Rightarrow A := e'_A$$

We first show that η_- and ϵ_- are lax natural transformations. We start by showing that for any $f : A \to B \in K$ and any $f' : A \to B \in K'$ we have:

$$f \Rightarrow R\ L\ f \qquad L\ R\ f' \Rightarrow f'$$

Indeed,

$$\begin{aligned}
f &= e_B \circ f \circ e_A && (\mathbf{K}\text{ def.})\\
&\Rightarrow R\, e'_B \circ f \circ R\, e'_A && (\text{lax functoriality})\\
&= e_B \circ e'_B \circ e_B \circ f \circ e_A \circ e'_A \circ e_A && (R\text{ def.})\\
&= e_B \circ e'_B \circ f \circ e'_A \circ e_A && (\mathbf{K}\text{ def.})\\
&= R\, L\, f && (\text{def.})
\end{aligned}$$

and

$$\begin{aligned}
L\, R\, f' &= e'_B \circ e_B \circ f' \circ e_A \circ e'_A && (\text{def.})\\
&= e'_B \circ e_B \circ e'_B \circ f' \circ e'_A \circ e_A \circ e'_A && (\mathbf{K}'\text{ def.})\\
&= L\, e_B \circ f' \circ L\, e_A && (L\text{ def.})\\
&\Rightarrow e'_B \circ f' \circ e'_A && (\text{Oplax functoriality})\\
&= f' && (\mathbf{K}'\text{ def.})
\end{aligned}$$

For η_- this means that the following square holds:

$$\begin{array}{ccc}
A & \xrightarrow{\eta_A} & R\, L\, A \\
{\scriptstyle f}\downarrow & \Rightarrow & \downarrow{\scriptstyle R\, L\, f} \\
B & \xrightarrow[\eta_B]{} & R\, L\, B
\end{array}$$

To see that it does hold, notice first that:

$$(R\, L\, f) \circ \eta_A = R\, L\, f$$

as $\eta_A = e_A$ is the identity in $\mathbf{K}$. By the same reasoning:

$$\eta_B \circ f = f$$

and then the 2-morphism:

$$f \Rightarrow R\, L\, f$$

is proved above.

Similarly, for the ϵ_- we must have the following square:

$$\begin{array}{ccc}
L\, R\, A & \xrightarrow{\epsilon_A} & A \\
{\scriptstyle L\, R\, g}\downarrow & \Rightarrow & \downarrow{\scriptstyle g} \\
L\, R\, B & \xrightarrow[\epsilon_B]{} & B
\end{array}$$

as in the unit case we have

$$\epsilon_B \circ L\, R\, g = L\, R\, g$$

because $\epsilon_B = e_B$ is the identity in $\mathbf{K}'$. For the same reason:

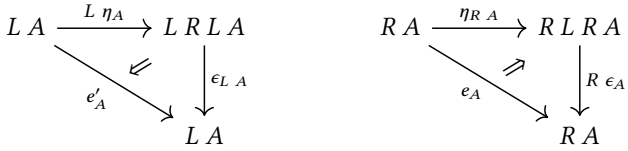
$$g \circ \epsilon_A = g$$

and then the 2-morphism:

$$L\, R\, g \Rightarrow g$$

is proved above.

In addition, η_- and ϵ_- satisfy the following lax triangle identities:



First note that

$$\epsilon'_{L\ A} \circ L\ \eta_A \Rightarrow \epsilon'_{L\ A} \circ e'_A = e'_A$$

$$e_A = e_A \circ \eta_{R\ A} \Rightarrow R\ \epsilon_A \circ \eta_{R\ A}$$

by oplax and lax functoriality as well as idempotency.

The remainder of the proof is just careful verification that the usual conversion of unit and counits do assemble into adjunct (lax) natural transformations. Indeed, this allows us to define two notions of adjuncts of a morphism, given by the usual formula. Explicitly, given a morphism

$$f' : L\ A \to B \in \mathbf{K}'$$

we define its right adjunct in the usual way as

$$(f')^{\vee} : A \to R\ B \coloneqq A \xrightarrow{\eta_A} R\ L\ A \xrightarrow{R\ f'} R\ \ B$$

while the left adjunct of

$$f : A \to R\ B \in \mathbf{K}$$

is given by

$$f^{\wedge} : L\ A \to B \coloneqq L\ A \xrightarrow{L\ f} L\ R\ B \xrightarrow{\epsilon_B} B$$

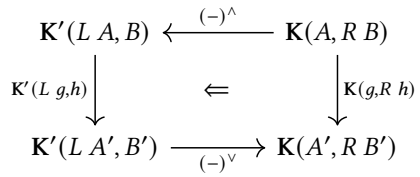
which are lax natural transformations in that:

$$\begin{array}{ccc} \mathbf{K}'(L\ A, B) & \xrightarrow{(-)^{\vee}} & \mathbf{K}(A, R\ B) \\ {\scriptstyle \mathbf{K}'(L\ g, h)}\downarrow & \Leftarrow & \downarrow{\scriptstyle \mathbf{K}(g, R\ h)} \\ \mathbf{K}'(L\ A', B') & \xrightarrow[(-)^{\vee}]{} & \mathbf{K}(A', R\ B') \end{array}$$

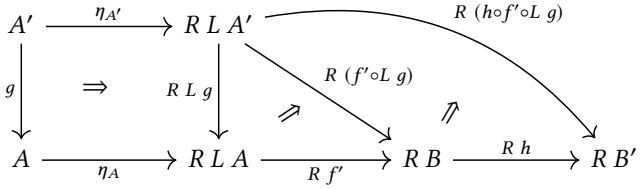
$$\begin{array}{ccc} \mathbf{K}'(L\ A, B) & \xleftarrow{(-)^{\wedge}} & \mathbf{K}(A, R\ B) \\ {\scriptstyle \mathbf{K}'(L\ g, h)}\downarrow & \Leftarrow & \downarrow{\scriptstyle \mathbf{K}(g, R\ h)} \\ \mathbf{K}'(L\ A', B') & \xleftarrow[(-)^{\wedge}]{} & \mathbf{K}(A', R\ B') \end{array}$$

and laxly satisfy the adjunction equations:

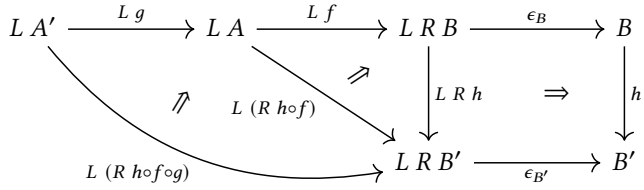
$$\begin{array}{ccc} \mathbf{K}'(L\ A, B) & \xrightarrow{(-)^{\vee}} & \mathbf{K}(A, R\ B) \\ {\scriptstyle \mathbf{K}'(L\ g, h)}\downarrow & \Rightarrow & \downarrow{\scriptstyle \mathbf{K}(g, R\ h)} \\ \mathbf{K}'(L\ A', B') & \xleftarrow{(-)^{\wedge}} & \mathbf{K}(A', R\ B') \end{array}$$



The naturality of the right adjunct mapping $-^\vee$ follows from:



where the leftmost 2-morphism comes from lax naturality, and the other two from lax functoriality. Similarly, for the left adjunct mapping $-^\wedge$:



where the rightmost 2-morphism comees from lax naturality, and the other two from oplax functoriality.

Now, for the lax adjunction equations we note that:

$$
\begin{aligned}
(R\ h \circ (f')^\vee \circ g)^\wedge &= \epsilon_{B'} \circ L\ (R\ h \circ (f')^\vee \circ g) && \text{(def.)} \\
&\Rightarrow \epsilon_{B'} \circ L\ R\ h \circ L\ (f')^\vee \circ L\ g && \text{(oplax functoriality)} \\
&= \epsilon_{B'} \circ L\ R\ h \circ L\ (R\ (f') \circ \eta_A) \circ L\ g && \text{(def.)} \\
&\Rightarrow \epsilon_{B'} \circ L\ R\ h \circ L\ R\ (f') \circ L\ \eta_A \circ L\ g && \text{(oplax functoriality)} \\
&\Rightarrow h \circ \epsilon_B \circ L\ R\ (f') \circ L\ \eta_A \circ L\ g && \text{(lax naturality)} \\
&\Rightarrow h \circ (f') \circ \epsilon_A \circ L\ \eta_A \circ L\ g && \text{(lax naturality)} \\
&\Rightarrow h \circ (f') \circ e'_A \circ L\ g && \text{(lax triangle identity)} \\
&= h \circ (f') \circ L\ g && \text{(identity in } \mathbf{C}')
\end{aligned}
$$

Similarly,

$$\begin{aligned}
R\,h \circ f \circ g &= R\,h \circ e_B \circ f \circ g && \text{(identity in } \mathbf{K}\text{)}\\
&\Rightarrow R\,h \circ R\,\epsilon_B \circ \eta_{R\,B} \circ f \circ g && \text{(lax triangle identity)}\\
&\Rightarrow R\,h \circ R\,\epsilon_B \circ R\,L\,f \circ \eta_A \circ g && \text{(lax naturality)}\\
&\Rightarrow R\,h \circ R\,\epsilon_B \circ R\,L\,f \circ R\,L\,g \circ \eta_{A'} && \text{(lax naturality)}\\
&\Rightarrow R\,h \circ R\,(\epsilon_B \circ L\,f) \circ R\,L\,g \circ \eta_{A'} && \text{(lax functoriality)}\\
&= R\,h \circ R\,f^{\wedge} \circ R\,L\,g \circ \eta_{A'} && \text{(def.)}\\
&\Rightarrow R\,(h \circ f^{\wedge} \circ L\,g) \circ \eta_{A'} && \text{(lax functoriality)}\\
&\Rightarrow (h \circ f^{\wedge} \circ L\,g)^{\vee} && \text{(def.)}
\end{aligned}$$

□

Proposition A.6. *If*

$$e_{-} = \{e_A\}_{A \in S} \qquad e'_{-} = \{e'_A\}_{A \in S}$$

are families of idempotents such that there are 2-morphisms:

$$e_{\mathbf{A}} \Rightarrow e'_{\mathbf{A}}$$

for every $A \in S$, then the mappings L and R defined by

$$L : \mathbf{C}_e \to \mathbf{C}_{e'} := K' \circ \mathsf{Emb} \qquad R : \mathbf{C}_{e'} \to \mathbf{C}_e := K \circ \mathsf{Emb}'$$

define an oplax functor and a lax functor respectively.

Proof. We start with L. Note first that

$$L\,e = e' \circ e \circ e' \Rightarrow e' \circ e' \circ e' = e'$$

moreover

$$L\,(g \circ f) = e' \circ g \circ f \circ e' = e' \circ g \circ e \circ f \circ e' \Rightarrow e' \circ g \circ e' \circ f \circ e' = e' \circ g \circ e' \circ e' \circ f \circ e' = L\,g \circ L\,f$$

For R, we have

$$e = e \circ e \circ e \Rightarrow e \circ e' \circ e = R\,e'$$

moreover

$$R\,g \circ R\,f = e \circ g \circ e \circ e \circ f \circ e = e \circ g \circ e \circ f \circ e \Rightarrow e \circ g \circ e' \circ f \circ e = e \circ g \circ f \circ e = R\,(g \circ f)$$

The enrichment of R and L follows from the fact that they are defined as formulas involving only composition. □

B ATOMICITY

In this section, we address Herlihy-Wing linearizability, which always assumes the linearized specification is atomic. We do this by specializing the theory of linearizability developed in §5 to Herlihy-Wing linearizability. Proofs for this section can be found in H.5.

B.1 Sequential Atomic Games

To set the stage for atomicity we start by defining a notion of atomic game.

Definition B.1. Let $A = (M_A, P_A) \in \mathbf{Game}_{\mathbf{Seq}}$ be a sequential game. We define its associated atomic game $!A = (M_{!A}, P_{!A})$ as follows:

$$M^O_{!A} := \sum_{\alpha \in \Upsilon} M^O_A \qquad M^P_{!A} := \sum_{\alpha \in \Upsilon} M^P_A \qquad P_{!A} := \{s \in \mathsf{Alt}(M^O_{!A}, M^P_{!A}) \mid \forall \alpha \in \Upsilon . \pi_\alpha(s) \in P_A\}$$

These games are atomic in that an O move by α is always followed by a P move by the same agent α, so that a typical play looks like:

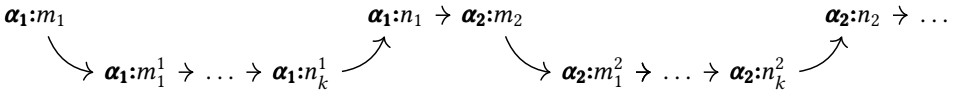
$$\boldsymbol{\alpha_1}{:}m_1 \longrightarrow \boldsymbol{\alpha_1}{:}n_1 \longrightarrow \boldsymbol{\alpha_2}{:}m_2 \longrightarrow \boldsymbol{\alpha_2}{:}n_2 \longrightarrow \boldsymbol{\alpha_3}{:}m_3 \longrightarrow \boldsymbol{\alpha_3}{:}n_3 \longrightarrow \ldots \longrightarrow \boldsymbol{\alpha_k}{:}m_k \longrightarrow \boldsymbol{\alpha_k}{:}n_k$$

where the m_i are O moves and the n_i are P moves. We may take $!A$ as an alternating version of $\mathbf{A}$, as any play of $!A$ may be seen as an alternating play of $\mathbf{A}$, a fact we frequently make use of. Note that a strategy $\sigma : !A \multimap !B$ does not need to respect the names of the agents. For instance, the following play is a valid play of $!\Sigma \multimap !\Sigma$

even when $\alpha \neq \alpha'$. This disagrees with our agent naming discipline on the concurrent games setting, as there the names of the agents must be preserved across components. Because of this, we must restrict the strategies $\sigma : !A \multimap !B$ so that they only allow agents to play moves that are labeled by their names in both components. We call such a strategy an *atomic strategy* and write the condition succinctly as:

$$\sigma \cap P_{!(A \multimap B)} = \sigma$$

by identifying plays of $!(A \multimap B)$ with plays of $!A \multimap !B$ in the obvious way. In a play of an atomic strategy $\sigma : !A \multimap !B$, if an α calls an O move in B then it cannot be preempted by another agent until it responds to that O move. That is to say, the typical play of an atomic strategy looks like:



It is important to note that the copycat strategy

$$\mathsf{copy}_{!A} : !A \multimap !A$$

is atomic. Furthermore, it is easy to see that composition of atomic strategies is well-defined.

Definition B.2. The category $\mathbf{Game}_{\mathbf{Atomic}}$ has atomic games $!A$, $!B$ as objects and atomic strategies as morphisms. Composition is given by usual sequential strategy composition and the identity is the sequential copycat $\mathsf{copy}_{!A}$.

By identifying atomic strategies $\sigma : !A \multimap !B$ with concurrent strategies in $\underline{\mathbf{Game}}_{\mathbf{Conc}}$ as discussed above we may define an oplax semifunctor

$$\mathsf{Lin}_{\mathsf{Atom}} : \mathsf{Semi}\, \mathbf{Game}_{\mathbf{Atomic}} \to \underline{\mathbf{Game}}_{\mathbf{Conc}}$$

by the usual formula

$$\mathsf{Lin}_{\mathsf{Atom}}\, \tau := \mathsf{ccopy}_{\mathbf{A}}; \tau; \mathsf{ccopy}_{\mathbf{B}}$$

B.2 Atomic Linearizability

We now endeavor in showing that our definition of linearizability is equivalent to Herlihy-Wing linearizability. Parts of our proof of this equivalence are adapted from Ghica and Murawski [2004]; Goubault et al. [2018]. In order to define Herlihy-Wing linearizability we must exhibit the happens-before ordering in our setting. We follow the approach of Goubault et al. [2018], which readily generalizes to our stronger setting. The key idea is that local sequentiality allows us to pair every Opponent move with a corresponding Proponent move by the same agent.

Definition B.3. Indeed, we define an *operation* of a play $s = m_1 \cdot \ldots \cdot m_k \in P_{\mathbf{A}}$ as a pair $(p, q?)$ such that m_p is an O move, and, moreover, either $q? = q$, $\alpha(m_q) = \alpha(m_p)$ and

$$\pi_{\alpha(m_p)}(s) = s_1 \cdot m_p \cdot m_q \cdot s_2$$

or $q? = \infty$ and

$$\pi_{\alpha(m_p)}(s) = s_1 \cdot m_p$$

In particular, $q?$ is an element of the total order $(\mathbb{N} + \infty, \leq)$ ordered in the obvious way. We say an operation $(p, q?)$ is by $\alpha \in \Upsilon$ when $\alpha(m_p) = \alpha$. We denote the set of operations of a play s by $\mathrm{op}(s)$.

With a notion of operation defined, we note that we may define a partial order, the happens-before order, associated to a play.

Definition B.4. We define the happens-before order assoaciated to a play s as the pair $(\mathrm{op}(s), \prec_s)$ where

$$(p, q) \prec_s (p', q') \iff q < p'$$

Definition B.5. We say two plays $s, s' \in P_{\mathbf{A}}$ are compatible when

$$\forall \alpha \in \Upsilon. \pi_\alpha(s) = \pi_\alpha(s')$$

Any two compatible plays have an associated bijection associating the i-th operation by α in s with the i-th operation by α in s', so we may implicitly apply it whenever needed and therefore assume that $\mathrm{op}(s) = \mathrm{op}(s')$ when convenient. We are now able to define Herlihy-Wing linearizability.

Definition B.6. For a play $s \in P_{\mathbf{A}}$ we call $\mathrm{complete}(s)$ the largest subsequence of s such that

$$\forall \alpha \in \Upsilon. \pi_\alpha(\mathrm{complete}(s)) = p \cdot m \Rightarrow \lambda_{\mathbf{A}}(m) = P$$

that is, the largest subsequence of s with no pending Opponent moves.

We say a play $s \in P_{\mathbf{A}}$ is Herlihy-Wing linearizable to a play $t \in P_{!A}$ if there exists a sequence of Proponent moves s_P such that $s' = \mathrm{complete}(s \cdot s_P)$ is compatible with t and moreover

$$\prec_{s'} \subseteq \prec_t$$

Now, we define an equivalence relation on plays based on $- \rightsquigarrow -$.

Definition B.7. The relation $- \equiv_{\mathbf{A}} -$ on plays $P_{\mathbf{A}}$ is the smallest relation satisfying:

$$s \equiv_{\mathbf{A}} t \iff s \rightsquigarrow_{\mathbf{A}} t \text{ using only } OO \text{ and } PP \text{ swaps}$$

This relation will appear again in our development on linearizability. Observe that by Prop. 4.7 if $\sigma : \mathbf{A}$ is saturated and $s \in \sigma$ then $[s]_{\equiv_{\mathbf{A}}} \subseteq \sigma$, where $[s]_{\equiv_{\mathbf{A}}}$ is the equivalence class of s under $\equiv_{\mathbf{A}}$.

The equivalence of our definition with their definition is predicated on the following two useful facts.

Proposition B.8. *If $s, t \in P_{\mathbf{A}}$ then $s \equiv_{\mathbf{A}} t$ if and only if s and t are compatible and $\prec_s = \prec_t$.*

Proposition B.9. *For plays $s, t \in P_{\mathbf{A}}$, there is a derivation*

$$s \rightsquigarrow_{\mathbf{A}} t$$

if and only if s is compatible with t and

$$\prec_{s'} \subseteq \prec_t$$

These give the following important corollary.

Corollary B.10. *A play $s \in P_{\mathbf{A}}$ is linearizable to an atomic play $t \in P_{!A}$ if and only if s is Herlihy-Wing linearizable to t.*

Moreover, our characterization of K_{Conc} in terms of general linearizability yields a characterization of the functor $\mathsf{Lin}_{\mathsf{Atom}}$.

Corollary B.11. *For any atomic strategy $\tau : !A$*

$$\mathsf{Lin}_{\mathsf{Atom}}\ \tau = \{s \in P_{\mathbf{A}} \mid s \text{ is Herlihy-Wing linearizable with respect to } \tau\}$$

Note that we arrived at the functor $\mathsf{Lin}_{\mathsf{Atom}}$ through the abstract construction of the Karoubi envelope, which can be understood as closing a computational model, represented by the semicategory $\underline{\mathbf{Game}}_{\mathbf{Conc}}$, by a synchronization pattern, represented by the choice of ccopy_ or atocopy_ as the unit. In this way, formally, Herlihy-Wing presents a solution to the problem of finding a concurrent strategy in $\mathbf{Game}_{\mathbf{Conc}}$ matching a certain atomic strategy in $\mathbf{Game}_{\mathbf{Atomic}}$.

Proposition B.11 also gives an alternative definition for Herlihy-Wing Linearizability in terms of the image of the functor $\mathsf{Lin}_{\mathsf{Atom}}$.

Corollary B.12. *A strategy $\sigma : \mathbf{A}$ is linearizable to an atomic strategy $\tau : !A$ if and only if*

$$\sigma \subseteq \mathsf{Lin}_{\mathsf{Atom}}\ \tau$$

B.3 Interaction Refinement and Locality

Herlihy-Wing linearizability admits its own computational intepretation of linearizability proofs, as a corollary of §5.3. Prop. 5.5 suggests defining a strategy

$$\mathsf{intcopy}_{\mathbf{A}} : \mathbf{A} \multimap \mathbf{A} := \{s \in \mathsf{ccopy}_{\mathbf{A}} \mid s\restriction_{\mathbf{A}_0} \in \Downarrow P_{\mathbf{A}}\}$$

That is, $\mathsf{intcopy}_{\mathbf{A}}$ is the substrategy of $\mathsf{ccopy}_{\mathbf{A}}$ that plays atomically in the source component of $\mathbf{A} \multimap \mathbf{A}$. By Prop. 5.5 and Prop. B.11 the plays of $\mathsf{intcopy}_{\mathbf{A}}$ correspond to proofs of Herlihy-Wing linearizability. Interestingly, intcopy_ is idempotent, so that it admits its own theory along the lines of A.

Corollary B.13 (Computational Interpretation of Herlihy-Wing Linearizability). *$s_1 \in P_{\mathbf{A}}$ is Herlihy-Wing linearizable to $s_0 \in P_{\mathbf{A}}$ if and only if there exists a play $s \in \mathsf{intcopy}_{\mathbf{A}}$ such that*

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

It is easy to see that the conditions of Prop. 6.2 are met by $\mathbf{Game}_{\mathbf{Atomic}}$. In particular, we have that

Proposition B.14. *$(\mathbf{Game}_{\mathbf{Atomic}}, K_{\mathsf{Atom}} \circ (- \otimes -), \mathbf{1})$ assembles into a symmetric monoidal category.*

which we discuss in §C. We take the freedom of overloading $- \otimes -$ for the atomic tensor product as well, (in particular omitting the use of K_{Atom}). It should be obvious which tensor we mean from context, as it will always be clear that the strategies involved are atomic. This readily gives that

Proposition B.15 (Interaction Refinement). *$v'_A : \mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}$ is Herlihy-Wing linearizable to $v_A : \mathbf{A} \in \mathbf{Game}_{\mathbf{Atomic}}$ if and only if for all concurrent games $\mathbf{B}$ and $\sigma : \mathbf{A} \multimap \mathbf{B} \in \mathbf{Game}_{\mathbf{Conc}}$ it holds that*

$$v'_A; \sigma \subseteq v_A; \sigma$$

Locality is also obtained by the same method as in §5.5, except that the source category is now $\mathbf{Game}_{\mathbf{Atomic}}$ and the oplax functor $\mathsf{Lin}_{\mathsf{Atom}}$ plays the role of K_{Conc}.

Proposition B.16 (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 $\mathbf{Game}_{\mathbf{Atomic}}$. Then*

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

C TENSOR PRODUCTS

In §5.5 we briefly discussed a notion of tensor product on $\underline{\mathbf{Game}}_{\mathbf{Conc}}$. We noted there that this notion of tensor product lifts to a symmetric monoidal closed structure in $\mathbf{Game}_{\mathbf{Conc}}$, what we develop in detail here. Moreover, we gave most, but omitted the proof of Prop. 5.10.

Proposition C.1.

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

Proof. Observe first that

$$\mathsf{ccopy}_{\mathbf{A}\otimes\mathbf{B}} = \mathsf{copy}^{\Phi}_{A\otimes B} = (\mathsf{copy}_A \otimes \mathsf{copy}_B)^{\Phi}$$

Now, assuming that s is sequentially consistent, observe that

$$s \in \mathsf{ccopy}_{\mathbf{A}\otimes\mathbf{B}} = (\mathsf{copy}_A \otimes \mathsf{copy}_B)^{\Phi}$$

if and only if for every $\alpha \in \Upsilon$:

$$\pi_\alpha(s\restriction_{\mathbf{A}}) \in \mathsf{copy}_{\mathbf{A}} \qquad \pi_\alpha(s\restriction_{\mathbf{B}}) \in \mathsf{copy}_{\mathbf{B}}$$

which is the case if and only if

$$s\restriction_{\mathbf{A}} \in \mathsf{ccopy}_{\mathbf{A}} \qquad s\restriction_{\mathbf{B}} \in \mathsf{ccopy}_{\mathbf{B}}$$

if and only if (as we have assumed sequential consistency):

$$s \in \mathsf{ccopy}_{\mathbf{A}} \otimes \mathsf{ccopy}_{\mathbf{B}}$$

Now, if $s \in \mathsf{ccopy}_{\mathbf{A}\otimes\mathbf{B}}$ then s is sequentially consistent, and if $s \in \mathsf{ccopy}_{\mathbf{A}} \otimes \mathsf{ccopy}_{\mathbf{B}}$ the same holds. □

Proposition C.2.

$$- \otimes - : \underline{\mathbf{Game}}_{\mathbf{Conc}} \otimes \underline{\mathbf{Game}}_{\mathbf{Conc}} \to \underline{\mathbf{Game}}_{\mathbf{Conc}}$$

is a bi-semifunctor.

Proof. Let

$$\sigma : \mathbf{A}_1 \multimap \mathbf{A}_2 \qquad \sigma' : \mathbf{A}_2 \multimap \mathbf{A}_3$$
$$\tau : \mathbf{B}_1 \multimap \mathbf{B}_2 \qquad \tau' : \mathbf{B}_2 \multimap \mathbf{B}_3$$

Suppose first that

$$s \in ((\sigma;\sigma') \parallel (\tau;\tau')) \cap P_{(\mathbf{A}_1\multimap\mathbf{A}_3)\otimes(\mathbf{B}_1\multimap\mathbf{B}_3)}$$

then

$$s_A = s\restriction_{\mathbf{A}_1\multimap\mathbf{A}_3} \in \sigma;\sigma' \qquad s_B = s\restriction_{\mathbf{B}_1\multimap\mathbf{B}_3} \in \tau;\tau'$$

Hence, there are $t_A \in \mathsf{int}(\sigma, \sigma')$ and $t_B \in \mathsf{int}(\tau, \tau')$ such that

$$t_A \restriction_{\mathbf{A}_1,\mathbf{A}_3} = s_A \qquad t_B \restriction_{\mathbf{B}_1,\mathbf{B}_3} = s_B$$

But then, notice that

$$s \in s_A \parallel s_B$$

it is straight-forward to check that we can construct an interleaving

$$t \in t_A \parallel t_B$$

such that

$$t \restriction_{\mathbf{A}_1 \otimes \mathbf{B}_1, \mathbf{A}_3 \otimes \mathbf{B}_3} = s$$

and moreover

$$t \restriction_{\mathbf{A}_1,\mathbf{A}_2,\mathbf{A}_3} = t_A \qquad t \restriction_{\mathbf{B}_1,\mathbf{B}_2,\mathbf{B}_3} = t_B$$

so that

$$s \in (\sigma \parallel \tau); (\sigma' \parallel \tau')$$

Now, suppose

$$s = t \restriction_{\mathbf{A}_1 \otimes \mathbf{B}_1, \mathbf{A}_3 \otimes \mathbf{B}_3} \in (\sigma \parallel \tau); (\sigma' \parallel \tau')$$

Then,

$$t \restriction_{\mathbf{A}_1 \otimes \mathbf{B}_1, \mathbf{A}_2 \otimes \mathbf{B}_2} \in \sigma \parallel \tau \qquad t \restriction_{\mathbf{A}_2 \otimes \mathbf{B}_2, \mathbf{A}_3 \otimes \mathbf{B}_3} \in \sigma' \parallel \tau'$$

hence,

$$t \restriction_{\mathbf{A}_1,\mathbf{A}_2} \in \sigma \qquad t \restriction_{\mathbf{A}_2,\mathbf{A}_3} \in \sigma'$$

and

$$t \restriction_{\mathbf{B}_1,\mathbf{B}_2} \in \tau \qquad t \restriction_{\mathbf{B}_2,\mathbf{B}_3} \in \tau'$$

Hence,

$$t \restriction_{\mathbf{A}_1,\mathbf{A}_2,\mathbf{A}_3} \in \sigma; \sigma' \qquad t \restriction_{\mathbf{B}_1,\mathbf{B}_2,\mathbf{B}_3} \in \tau; \tau'$$

therefore

$$t \in (\sigma; \sigma') \parallel (\tau; \tau')$$

The enrichment is obvious. First, if $\sigma \subseteq \sigma'$ and $\tau \subseteq \tau'$ it follows immediately from the definition that

$$\sigma \parallel \tau \subseteq \sigma' \parallel \tau'$$

Unions are handled in the same way. □

Proposition C.3. $(\mathbf{Game}_{\mathbf{Conc}}, \otimes, \mathbf{1})$ *defines a symmetric monoidal closed category.*

Proof. We've already proven bi-semifunctoriality in Prop. C.2, and that neutral elements are preserved in C.1.

We now move to the monoidal structure.

Lemma C.4. $(\mathbf{Game}_{\mathbf{Conc}}, K_{\mathrm{Conc}} \circ - \otimes -, \mathbf{1})$ *defines a symmetric monoidal category.*

Proof. We start by showing that the structural morphisms assemble into natural isomorphisms:

$$\begin{array}{ccc}
\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) & \xrightarrow{\alpha_{\mathbf{A},\mathbf{B},\mathbf{C}}} & (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} \\
{\scriptstyle \sigma_A \otimes (\sigma_B \otimes \sigma_C)}\downarrow & \cong & \downarrow{\scriptstyle (\sigma_A \otimes \sigma_B) \otimes \sigma_C} \\
\mathbf{A}' \otimes (\mathbf{B}' \otimes \mathbf{C}') & \xrightarrow[\alpha_{\mathbf{A}',\mathbf{B}',\mathbf{C}'}]{} & (\mathbf{A}' \otimes \mathbf{B}') \otimes \mathbf{C}'
\end{array}
\qquad
\begin{array}{ccc}
\mathbf{1} \otimes \mathbf{A} & \xrightarrow{\lambda_{\mathbf{A}}} & \mathbf{A} \\
{\scriptstyle 1 \otimes \sigma}\downarrow & \cong & \downarrow{\scriptstyle \sigma} \\
\mathbf{1} \otimes \mathbf{B} & \xrightarrow[\lambda_{\mathbf{B}}]{} & \mathbf{B}
\end{array}
\qquad
\begin{array}{ccc}
\mathbf{A} \otimes \mathbf{1} & \xrightarrow{\rho_{\mathbf{A}}} & \mathbf{A} \\
{\scriptstyle \sigma \otimes 1}\downarrow & \cong & \downarrow{\scriptstyle \sigma} \\
\mathbf{B} \otimes \mathbf{1} & \xrightarrow[\rho_{\mathbf{B}}]{} & \mathbf{B}
\end{array}$$

The left and right unital are straight-forward. Indeed, they are simply the identity on the corresponding sequential games so that:

$$\lambda_{\mathbf{A}} = \lambda_A^{\Phi} = \mathsf{copy}_A^{\Phi} = \mathsf{ccopy}_{\mathbf{A}}$$

$$\rho_{\mathbf{A}} = \rho_A^{\Phi} = \mathsf{copy}_A^{\Phi} = \mathsf{ccopy}_{\mathbf{A}}$$

Meanwhile,

$$\mathbf{1} \otimes \sigma = \{\epsilon\} \parallel \sigma = \sigma$$

Therefore, we easily check that:

$$(\mathbf{1} \otimes \sigma); \lambda_{\mathbf{B}} = \sigma; \mathsf{ccopy}_{\mathbf{B}} = \sigma = \mathsf{ccopy}_{\mathbf{A}}; \sigma = \lambda_{\mathbf{A}}; \sigma$$

$$(\sigma \otimes \mathbf{1}); \rho_{\mathbf{B}} = \sigma; \mathsf{ccopy}_{\mathbf{B}} = \sigma = \mathsf{ccopy}_{\mathbf{A}}; \sigma = \rho_{\mathbf{A}}; \sigma$$

Now, for the associator, the equation essentially follows from the fact that:

$$\begin{aligned} \pi_\alpha(\sigma_A \otimes (\sigma_B \otimes \sigma_C)); \alpha_{A',B',C'} &= (\pi_\alpha(\sigma_A) \otimes (\pi_\alpha(\sigma_B) \otimes \pi_\alpha(\sigma_C))); \alpha_{A',B',C'} \\ &= \alpha_{A,B,C}; ((\pi_\alpha(\sigma_A) \otimes \pi_\alpha(\sigma_B)) \otimes \pi_\alpha(\sigma_C)) \\ &= \alpha_{A,B,C}; \pi_\alpha((\sigma_A \otimes \sigma_B) \otimes \sigma_C) \end{aligned}$$

this is the key step to establish that the naturality square commutes. The reverse direction follows similarly.

The coherence diagrams follow from functoriality of Conc, the fact that the structural morphisms are defined by lifting the sequential ones through Conc. Moreover,

$$\mathsf{Conc}\ \sigma \otimes \mathsf{Conc}\ \tau = \mathsf{Conc}\ (\sigma \otimes \tau)$$

as is easily checked.

The same argument shows that the braiding morphism is a natural transformation, that it is invertible and the functoriality of Conc implies that the hexagonal diagram commutes. □

Finally, we establish that $\mathbf{Game}_{\mathbf{Conc}}$ is closed.

Lemma C.5. *The symmetric monoidal category* $(\mathbf{Game}_{\mathbf{Conc}}, \otimes, \mathbf{1})$ *is closed.*

Proof. We start by noting that there is an isomorphism:

$$\mathbf{A} \otimes \mathbf{B} \multimap \mathbf{C} \cong \mathbf{A} \multimap (\mathbf{B} \multimap \mathbf{C})$$

Indeed, it immediately follows from the fact that the underlying sequential arenas are

$$A \otimes B \multimap C \cong A \multimap (B \multimap C)$$

which induces the necessary natural isomorphism of hom-sets. □

□

The argument for $(\mathbf{Game}_{\mathbf{Atomic}}, K_{\mathsf{Atom}} \circ - \otimes -, \mathbf{1})$ is analogous except that we construct an atomic interleaving functor

$$\mathsf{Atom} : \mathbf{Game}_{\mathbf{Seq}} \longrightarrow \mathbf{Game}_{\mathbf{Atomic}}$$

that plays the same role as Conc plays in the proof of C.3.

D INTERVAL-SEQUENTIAL LINEARIZABILITY

Goubault et al. [2018] noted, without proof, that their definition of linearizability is equivalent to interval-sequential linearizability (although it is actually the restriction of interval-sequential linearizability to total objects). Here, we show that our definition of linearizability in the context of our model of sequentially consistent concurrent computation corresponds to a generalization of interval-sequential linearization to handle blocking objects, which the original definition cannot [Castañeda et al. 2015].

In Castañeda et al. [2015] a trace is called interval-sequential if it is of the form

$$\langle I_1, R_1, \ldots, I_n, R_n \rangle$$

where the I_i are non-empty sets of invocations and the R_i are non-empty sets of responses such that

- Any two invocations in I_i are by different agents;
- Any two responses in R_i are by different agents;
- If $r \in R_j$ is a response by agent α, then there is $c \in I_i$ by the same agent for some $i \leq j$ such that for all k such that $i < k < j$, I_k has no invocations by α and R_k has no responses by α.

Interpreting O moves as invocations and P moves as responses we immediately see that the equivalence classes of plays $s \in P_{\mathbf{A}}$ under $- \equiv_{\mathbf{A}} -$ correspond precisely to plays of the form

$$\langle O_1, P_1, \ldots, O_n, P_n \rangle$$

where similarly to before the O_i are sets of Opponent moves and the P_i are sets of proponent moves. Otherwise, the same kind of happens-before order preservation is used to define linearizability to an interval-sequential trace.

Definition D.1. A play $s \in P_{\mathbf{A}}$ is interval-sequential linearizable to an equivalence class $[t]_{\equiv}$ of $- \equiv_{\mathbf{A}} -$ if for every $t' \in [t]_{\equiv}$, s is linearizable to t'.

We find it convenient to instead define a general notion of linearizability between concurrent strategies.

Definition D.2. We say a strategy $\sigma : \mathbf{A}$ is linearizable to a strategy $\tau : \mathbf{A}$ if every play of σ is linearizable to a play of τ.

The discussion above promptly let's us prove that.

Proposition D.3. *$s \in P_{\mathbf{A}}$ is linearizable to $t \in P_{\mathbf{A}}$ if and only if s is interval-sequential linearizable to $[t]_{\equiv}$, the equivalence class of t under $\equiv_{\mathbf{A}}$.*

Proof. Suppose first that s is linearizable with respect to t. Then, there is s_P a sequence of Proponent moves and s_O a sequence of Opponent moves such that.

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

So let $t' \in [t]_{\equiv}$. Then, note that in particular

$$t \rightsquigarrow_{\mathbf{A}} t'$$

and therefore

$$s \cdot s_P \rightsquigarrow_{\mathbf{A}} t \cdot s_O \rightsquigarrow_{\mathbf{A}} t' \cdot s_O$$

Now, suppose s is interval-sequential linearizable to $[t]_{\equiv}$. Then, s is linearizable to every $t' \in [t]_{\equiv}$ and in particular to t. □

Observe that we already showed the equivalence with happens-before order formulations of linearizability in Appendix B.2. The key difference between our formulation of interval-sequential linearizability and the original one is that we do not require that the linearization remove all uncompleted pending invocations. This essentially means that our definition of linearizability can handle blocking objects, while typical linearizability only handles non-blocking objects. This is vital. Consider our yield example. The trace

$$\boldsymbol{\alpha}\textbf{:}\mathsf{yield} \cdot \boldsymbol{\alpha'}\textbf{:}\mathsf{yield} \cdot \boldsymbol{\alpha}\textbf{:}\mathsf{ok}$$

linearizes to itself in our example. Now, suppose we were forced to either complete the pending invocation $\boldsymbol{\alpha'}$**:**yield or remove it to linearize the trace. Then we have to use one of the following traces as the linearization:

(1) $\boldsymbol{\alpha}$**:**yield $\cdot$ $\boldsymbol{\alpha'}$**:**yield $\cdot$ $\boldsymbol{\alpha}$**:**ok $\cdot$ $\boldsymbol{\alpha'}$**:**ok or any equivalent trace under $\equiv_{\dagger\mathsf{Yield}}$;
(2) $\boldsymbol{\alpha}$**:**yield $\cdot$ $\boldsymbol{\alpha}$**:**ok;
(3) $\boldsymbol{\alpha}$**:**yield $\cdot$ $\boldsymbol{\alpha}$**:**ok $\cdot$ $\boldsymbol{\alpha'}$**:**yield $\cdot$ $\boldsymbol{\alpha'}$**:**ok or $\boldsymbol{\alpha'}$**:**yield $\cdot$ $\boldsymbol{\alpha'}$**:**ok $\cdot$ $\boldsymbol{\alpha}$**:**yield $\cdot$ $\boldsymbol{\alpha}$**:**ok

Trace (1) does not make sense. Assume, without loss of generality, that α is the one that yielded first. Then, α is able to return because α' yielded after. But now there is no call to yield that justifies the return by α'. Traces in (2) and (3) do not make sense because no one yielded to α (or α' in (3)).

To state it more broadly, when all pending invocations are required to be removed the only way to signal that an invocation has already taken effect is by adding a return. Meanwhile, with our formulation an invocation may be effectful by itself, even when it is impossible to choose a return value for it.

E CONCURRENT OBJECT-BASED SEMANTICS AND LINEARIZABLE CONCURRENT OBJECTS

E.1 The Replay Modality and Concurrent Object Specifications

We start by recalling the definition of the replay modality on sequential games, which originally appears in Oliveira Vale et al. [2022].

Definition E.1. Let A be a game. We define the replay of the game A, the game $\dagger A$ as $\dagger A = (M_{\dagger A}, P_{\dagger A})$ where

$$M^O_{\dagger A} := \sum_{i \in \mathbb{N}} M^O_A \qquad M^P_{\dagger A} := \sum_{i \in \mathbb{N}} M^P_A$$

$$P_{\dagger A} := \{s_1 \cdot \ldots \cdot s_n \in \mathsf{Alt}(M^O_{\dagger A}, M^P_{\dagger A}) \mid \forall i. s_i \in \iota_i \; P_A\}$$

where, for a play s, $\iota_i \; s$ labels all the moves $m \in s$ as $\iota_i \; m$, and for a set of plays S the set of plays $\iota_i \; S$ is obtained by applying $\iota_i \; s$ to every play $s \in S$.

A key property of the sequential $\dagger -$ modality is that it is a comonad over $\mathbf{Game}_{\mathbf{Seq}}$.

Proposition E.2.

$$\dagger - : \mathbf{Game}_{\mathbf{Seq}} \to \mathbf{Game}_{\mathbf{Seq}}$$

defines a comonad.

Similarly to our approach to other operators on concurrent games, we define the concurrent $\dagger -$ as the lifting of the sequential one.

Definition E.3. For a concurrent game $\mathbf{A} = (M_A, P_A)$ we define the concurrent game $\dagger\mathbf{A}$ as $\dagger\mathbf{A} := (M_{\dagger A}, P_{\dagger A})$.

Proposition E.4.

$$\dagger - : \mathbf{Game}_{\mathbf{Conc}} \longrightarrow \mathbf{Game}_{\mathbf{Conc}}$$

defines a comonad.

Now, following the framework started by Reddy [1993, 1996], we define a concurrent object as follows:

Definition E.5. An object of type $\mathbf{A}$ is a strategy $v_A : \mathbf{1} \multimap \dagger\mathbf{A}$.

We are particularly interested in concurrent objects with types described by effect signatures, per the approach to modeling layered components in Oliveira Vale et al. [2022].

Definition E.6. 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:

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

Every effect signature defines a very simple sequential game $\mathbf{Game}_{\mathbf{Seq}}(E)$ with moves given by

$$M_E^O := E \qquad\qquad M_E^P := \cup_{e \in E} \mathsf{ar}(e)$$

and plays

$$P_E := \downarrow\{e \cdot v \mid e \in E \wedge v \in \mathsf{ar}(e)\}$$

We will often denote $\mathbf{Game}_{\mathbf{Seq}}(E)$ simply as E.

We can then define the associated concurrent game $\mathbf{Game}_{\mathbf{Conc}}(E) = (M_E, P_E)$ which we often will denote simply by $\mathbf{E}$.

Let's mull over what a layer game entails. In the sequential case, the game E has plays of the form:

$$e \longrightarrow v$$

consisting of an invocation of an effect $e \in E$ followed by a response $v \in \mathsf{ar}(e)$. Its replay $\dagger\mathbf{E}$ merely allows for several such interactions to be performed in sequence, like so

$$e_1 \longrightarrow v_1 \longrightarrow e_2 \longrightarrow v_2 \longrightarrow \ldots \longrightarrow e_n \longrightarrow v_n$$

where each $e_i \in E$ and each $v_i \in \mathsf{ar}(e_I)$. Its concurrent version simply allows each thread to play $\dagger\mathbf{E}$ locally. Most objects appearing in concrete systems can be modeled by an effect signature, and we have already provided many such examples in §2.2.

Then, we define a linearizable object simply as

Definition E.7. A *linearizable concurrent object* of type $\mathbf{A}$ consists of a pair of objects

$$(v'_A : \dagger\mathbf{A} \in \mathbf{Game}_{\mathbf{Conc}}, v_A : \dagger\mathbf{A} \in \underline{\mathbf{Game}}_{\mathbf{Conc}}) \qquad \text{such that} \qquad v'_A \subseteq K_{\mathrm{Conc}}\ v_A$$

It is called a *strongly linearizable concurrent object* if moreover

$$v_A \subseteq v'_A$$

E.2 Concurrent Object Implementations

Typically, in the sequential case, we would use as object implementations strategies

$$\widehat{M} : \dagger A \multimap \dagger B$$

which are moreover regular in the sense that they are †-coalgebra morphisms between the free †-coalgebras associated to A and B:

$$\widehat{M} : (\dagger A, \delta_A) \to (\dagger B, \delta_B)$$

We emphasize the $\widehat{-}$ on M because as $\widehat{M}$ lives in the co-Kleisli category of $\dagger -$ it may instead be described as a strategy

$$M : \dagger A \multimap B$$

and composition is as in the co-Kleisli category. This gives a minimal description of the associated coalgebra morphism $\widehat{M}$ and simplifies the process of specifying implementations. See any of Oliveira Vale et al. [2022]; Reddy [1993, 1996] for more details on the framework.

We extend that formulation to model concurrent object implementations as morphisms of the form

$$\|_{\alpha \in \Upsilon}\ \iota_\alpha(\widehat{M[\alpha]}) : \dagger \mathbf{A} \multimap \dagger \mathbf{B}$$

or, equivalently,

$$\bigotimes_{\alpha \in \Upsilon} \mathsf{strat}\left(\iota_\alpha(\widehat{M[\alpha]})\right) : \dagger \mathbf{A} \multimap \dagger \mathbf{B}$$

where each $M[\alpha]$ is a sequential strategy of type $\dagger A \multimap B$. Alternatively, we may characterize concurrent implementations as collections

$$(M[\alpha] : \dagger A \multimap B)_{\alpha \in \Upsilon}$$

Which define a concurrent implementation by the formula above. The intuition here is that each agent $\alpha \in \Upsilon$ locally runs a sequential object implementation. In practice, it is often the case that all agents run the same sequential implementation M in which case we can simply use

$$\mathsf{Conc}\ \widehat{M} : \dagger \mathbf{A} \multimap \dagger \mathbf{B}$$

We note that

Proposition E.8. *Concurrent object implementations are co-algebra morphisms.*

We now observe that for effect signatures E and F, any sequential object implementation:

$$M : \dagger E \multimap F$$

decomposes as a collection of implementations $(M^f : \dagger E \multimap \{f : \mathsf{ar}(f)\})_{f \in F}$ where

$$M^f := \epsilon \cup \{f \cdot s \in M \mid f \in F\}$$

that is, M^f is the set of plays of M starting with the operation f. Then

$$M = \bigcup_{f \in F} M^f$$

Moreover, any collection of strategies $(M^f : \dagger E \multimap \{f : \mathsf{ar}(f)\})_{f \in F}$ defines an implementation $M : \dagger E \multimap F$ by the formula above.

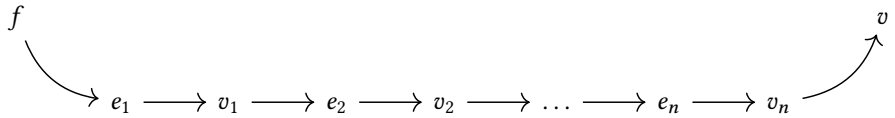
The computational interpretation is quite simple here. Along the lines of Oliveira Vale et al. [2022], every implementation $M[\alpha]^f$ corresponds to some code implementing the effect f using the effects in E. A full sequential implementation $M[\alpha]$ corresponds to all of the implementations for each $f \in F$ bundled together, such as in a file containing the code for all of those methods. The

concurrent object implementation then is analogous to the usual syntactic linking appearing in the syntactic approaches to concurrent computation. It is worth noting that:

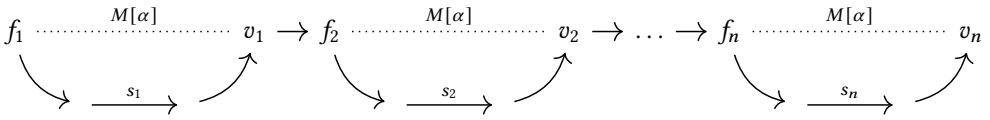
Definition E.9. Concurrent games and concurrent object implementations assemble into a category ConcObj, with composition inherited from the underlying category and the identity given by ccopy_.

The full subcategory obtained by restricting the objects of ConcObj to concurrent games coming from effect signatures, **E**, **F**, is the category of layered concurrent objects ConcLayer.

Again, let's consider what an implementation for a layer object consists of. Locally, the implementation $M[\alpha]^f : E \multimap \{f : \mathrm{ar}(f)\}$ of an effect $f \in F$ using events in E by $\alpha \in \Upsilon$ is a strategy consisting of plays of the following shape:



the implementation $M[\alpha] : \dagger E \multimap F$ of F using E by α is simply the collection of the implementations $M[\alpha]^f$ for each $f \in F$ by α, so that it is able to issue the right implementation on an environment request for any effect from F. Its regular extension $\widehat{M[\alpha]}$ *replays* the implementation $M[\alpha]$ in order to be able to handle several requests for effects in F by the environment. In this way, its plays are of the following shape:



where each sequence $f_i \cdot s_i \cdot v_i$ is a play of $M[\alpha]$, and in particular a play of $M[\alpha]^{f_i}$. The concurrent implementation $M : \mathbf{E} \multimap \mathbf{F}$ is simply the result of having each $\alpha \in \Upsilon$ playing their corresponding implementations $M[\alpha]$ in parallel. All the implementations discussed in §2.2 can be encoded as layer implementations.

It remains to give an account of when an implementation *correctly* implements an object. This is captured by the following definition.

Definition E.10. A *certified linearizable object implementation* $M : (v'_A, v_A) \longrightarrow (v'_B, v_B)$ is an implementation $M : \dagger\mathbf{A} \multimap \dagger\mathbf{B}$ which moreover satisfies:

$$v'_B \subseteq v'_A; M \qquad\qquad v'_B \subseteq K_{\mathrm{Conc}}\ v_B$$

Its immediate to see that linearizable concurrent objects together with certified linearizable object implementations assemble into a category, which may again be layered by restricting to games corresponding to effect signatures. This definition is readily adapted to strong linearizability.

F A PROGRAM LOGIC FOR LINEARIZABLE CONCURRENT OBJECTS

In this section we define a simple programming language for specifying concurrent object implementations, as well as a program logic for proving that they implement linearizable concurrent objects. We start by defining its syntax and semantics, and then move to the program logic. We then give an example to showcase one of the key differences between our program logic and that of Khyzha et al. [2017]. We finish by providing an outline of the proofs of the main results in this section. In Appendix G we treat the example of §2.2 using our program logic.

F.1 Programming Language

F.1.1 Syntax. For our language we find it useful to slightly generalize effect signatures to be of the form:

$$E = \{e : \mathsf{par}(e) \to \mathsf{ar}(e) \mid e \in E\}$$

here, $\mathsf{par}(e) \in \mathbf{Set}^*$ stands as the parameter set of e, and $\mathsf{ar}(e) \in \mathbf{Set}$ is its arity, as usual. These are interpreted as standard effect signatures of the form:

$$E = \{e(p) : \mathsf{ar}(e) \mid e \in E \wedge p \in \mathsf{par}(e)\}$$

namely, $e : \mathsf{par}(e) \to \mathsf{ar}(e)$ stands for a $\mathsf{par}(e)$-indexed family of effects all with arity $\mathsf{ar}(e)$.

The key idea then is to define a language of commands

$$\mathsf{Com} := \mathsf{Prim} \mid \mathsf{Com}; \mathsf{Com} \mid \mathsf{Com} + \mathsf{Com} \mid \mathsf{Com}^* \mid \mathsf{skip}$$

Where Prim is a collection of programming language primitives (to be defined soon) which serve as the base commands.

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.

F.1.2 Semantics. We start with the operational semantics of sequential commands. We define our state space as

$$\mathsf{UndState} = \mathsf{Env} \times P_{\dagger \mathbf{E}}$$

so that a state $(\Delta, s) \in \mathsf{UndState}$ contains a local environment $\Delta \in \mathsf{Env}$ (a partial map from a set of variable names Var to the set of possible values) 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.

Every language primitive $B \in \mathsf{Prim}$ receives an interpretation as a function

$$[\![B]\!]_\alpha : \mathsf{UndState} \to \mathcal{P}(\mathsf{UndState})$$

satisfying moreover that for all $(\Delta, s) \in \mathsf{UndState}$ if $(\Delta', s') \in [\![B]\!]_\alpha(\Delta, s)$ then:

- $\Delta \subseteq \Delta'$
- $\exists t. s' = s \cdot t \wedge \pi_{\Upsilon \setminus \alpha}(t) = \epsilon$

That is to say, the local environment cannot decrease in size by a language primitive; and a language primitive may only advance the state further, and only by adding events for the corresponding agent. Notice that we can split $[\![B]\!]_\alpha$ into two maps $[\![B]\!]^O_\alpha$ and $[\![B]\!]^P_\alpha$, the first only defined on O-positions for α and the later only defined on P-positions for α.

The local semantics is then given by

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

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

$$\frac{}{C^* \rightarrowtail^{X}_{\mathbf{id}} \mathsf{skip}} \qquad \frac{}{C_1 + C_2 \rightarrowtail^{X}_{\mathbf{id}} C_1} \qquad \frac{}{C_1 + C_2 \rightarrowtail^{X}_{\mathbf{id}} C_2}$$

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

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

There, **id** stands for a primitive command that behaves just like skip but is used exclusively to define the operational semantics. A key difference from traditional operational semantics is that language primitives take two steps to reduce. First, they execute their O-position effect, corresponding to an invocation, and then, separately, their P-position effect, corresponding to a return.

To move to the concurrent semantics, states are lifted as

$$\mathsf{ModState} = (\Upsilon \to \mathsf{Env}) \times P_{\dagger\mathbf{E} \multimap \dagger\mathbf{F}}$$

There is an obvious lifting of $[\![-]\!]_\alpha$ to ModState modifying only the local environment for α according to $[\![-]\!]_\alpha$ and advancing the play state further only by moves by α.

$$\longrightarrow\!\!\!\!\!\rightarrow \subseteq (\mathsf{Cont} \times \mathsf{ModState}) \times \mathsf{CMod} \times (\mathsf{Cont} \times \mathsf{ModState})$$

$$\frac{f \in F \qquad a \in \mathsf{par}(f) \qquad \Delta' = \Delta[\alpha : [\mathsf{arg} : a]]}{\langle c[\alpha : \mathsf{idle}], \Delta, s\rangle \longrightarrow\!\!\!\!\!\rightarrow^{M} \langle c[\alpha : M[\alpha]^{f}], \Delta', s \cdot \boldsymbol{\alpha}\mathbf{:}f\rangle} \qquad \frac{\langle C, \Delta, s\restriction_{\mathbf{E}}\rangle \longrightarrow_{\alpha} \langle C', \Delta', s'\restriction_{\mathbf{E}}\rangle}{\langle c[\alpha : C], \Delta, s\rangle \longrightarrow\!\!\!\!\!\rightarrow^{M} \langle c[\alpha : C'], \Delta', s'\rangle}$$

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

Here a continuation $c \in \mathsf{Cont}$ consists of a mapping $c : \Upsilon \to \{\mathsf{idle}\} + \{\mathsf{skip}\} + \mathsf{Com}$. The initial continuation c_0 is the mapping such that $c_0(\alpha) = \mathsf{idle}$ for all $\alpha \in \Upsilon$. The initial environment Δ_0 is defined as the empty mapping $\Delta_\alpha = \varnothing$ for every agent $\alpha \in \Upsilon$. The concurrent semantics simply models all the agents running their local implementations concurrently in an interleaved fashion. The three rules correspond, in order to, to a target component invocation, a step in the source component, and a return in the target component.

We obtain a concurrent object implementation $[\![M]\!] : \dagger\mathbf{E} \multimap \dagger\mathbf{F}$ from a concurrent module $M \in \mathsf{CMod}$ by:

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

A program M can be linked with a specification ν_E for its source component given by a strategy $\nu_E : \dagger\mathbf{E}$, what we denote by $\mathsf{Link}\ \nu_E; M$. The operational semantics of $\mathsf{Link}\ \nu_E; M$ is given by:

$$\longrightarrow\!\!\!\!\!\rightarrow \subseteq (\mathsf{Cont} \times \mathsf{UndState}_{\mathsf{Conc}}) \times (\underline{\mathbf{Game}}_{\mathbf{Conc}}(\mathbf{1}, \mathbf{E}) \times \mathsf{CMod}) \times (\mathsf{Cont} \times \mathsf{ModState})$$

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

$$\frac{\langle C, \Delta, s\rangle \longrightarrow_\alpha \langle C', \Delta', s'\rangle \qquad s' \in \nu_E}{\langle c[\alpha : C], \Delta, s\rangle \longrightarrow\!\!\!\!\!\rightarrow^{M}_{\nu_E} \langle c[\alpha : C'], \Delta', s'\rangle}$$

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

Observe that

$$- \longrightarrow\!\!\!\!\!\rightarrow^{M} - \qquad = \qquad - \longrightarrow\!\!\!\!\!\rightarrow^{M}_{P_{\dagger\mathbf{E}}} -$$

From a linked program $\mathsf{Link}\ \nu_E; M$ we can obtain a corresponding strategy $[\![\mathsf{Link}\ \nu_E; M]\!] : \dagger\mathbf{F}$ similarly to before:

$$[\![\mathsf{Link}\ \nu_E; M]\!] = \{s \mid \exists c \in \mathsf{Cont}.\exists \Delta \in (\Upsilon \to \mathsf{Env}).\langle c_0, \Delta_0, \epsilon\rangle \longrightarrow\!\!\!\!\!\rightarrow^{M}_{\nu_E} \langle c, \Delta, s\rangle\}$$

The following result allows us to connect the program language back with the theory we have developed so far.

Proposition F.1. *For any $M \in \mathsf{CMod}$, $[\![M]\!]$ is a concurrent object implementation of type $\dagger\mathbf{E} \multimap \dagger\mathbf{F}$ and given $\nu_E : \dagger\mathbf{E}$,*

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

F.1.3 Language Primitives. We now introduce the language primitives we will use for our purposes. First, we have a command **id** with interpretation given by:

$$[\![\mathbf{id}]\!]_\alpha(\Delta, s) = \{(\Delta, s)\}$$

which behaves exactly like skip.

A command ret $-$ with interpretation:

$$[\![\mathsf{ret}\ v]\!]_\alpha(\Delta, s) = \begin{cases} (\Delta[\mathsf{ret} : v], s), \ \Delta(\mathsf{ret}) = \bot \\ \varnothing, \ \Delta(\mathsf{ret}) \neq \bot \end{cases}$$

ret $-$ reserves a location for returns to be written to. Observe that a return may only be called once in any execution.

A more interesting primitive is a primitive of the form $x \leftarrow e(a)$ where $e \in E$ and $a \in \mathsf{Var}+\mathsf{par}(e)$ and $x \in \mathsf{Var}$ with interpretation $[\![x \leftarrow e(a)]\!]_\alpha(\Delta, s)$ given by:

- If $\mathsf{even}(\pi_\alpha(s))$ then $[\![x \leftarrow e(a)]\!]_\alpha(\Delta, s) := \begin{cases} \{(\Delta, s \cdot \boldsymbol{\alpha}{:}e(a))\}, \ a \in \mathsf{par}(e) \\ \{(\Delta, s \cdot \boldsymbol{\alpha}{:}e(\Delta(a)))\}, \ a \in \mathsf{Var} \wedge \Delta(a) \in \mathsf{par}(e) \\ \varnothing, \ \text{otherwise} \end{cases}$
- If $\pi_\alpha(s) = p \cdot e(a')$ where either $a' = a$ or $a' = \Delta(a)$ then

$$[\![x \leftarrow e(a)]\!]_\alpha(\Delta, s) := \{(\Delta[x : v], s \cdot \boldsymbol{\alpha}{:}v) \mid v \in \mathsf{ar}(e)\}$$

- Otherwise, $[\![x \leftarrow e(a)]\!]_\alpha(\Delta, s) := \varnothing$.

This models the fact that the implementation may call effects from its source component E. $x \leftarrow e(a)$ executes the effect $e \in E$ with argument a, which might contain variables defined in a local environment $\Delta \in \mathsf{Env}$.

Finally, to implement branching we have a command $\mathsf{assert}(\phi)$ where $\phi : \mathsf{Env} \to \mathbf{Bool}$ is interpreted by

$$[\![\mathsf{assert}(\phi)]\!]_\alpha(\Delta, s) := \begin{cases} \{(\Delta, s)\}, \phi(\Delta) = \mathsf{True} \\ \varnothing, \ \mathsf{Otherwise} \end{cases}$$

$\mathsf{assert}(-)$ can be used to implement a while loop and if conditionals in the usual way.

F.2 Program Logic

Throughout, we assume the following situation. We have a linearizable concurrent object $(\nu'_E : \dagger\mathbf{E}, \nu_E : \dagger\mathbf{E})$ and would like to show that an implementation $M : E \to F$ is correct in that when it runs on top of ν'_E it linearizes to a specification $\nu_F : \dagger\mathbf{F}$. When reasoning about $\mathsf{Link}\ \nu'_E; M$ 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 $\nu'_F : \dagger\mathbf{F}$ in our approach.

We define a configuration as a triple

$$\mathsf{Config} := \mathsf{ModState} \times \mathsf{Poss}$$

where

$$\mathsf{Poss} := K_{\mathsf{Conc}}\ \nu_F$$

A possibility is a play of $\dagger\mathbf{F}$ that moreover is linearizable to ν_F, by definition. As we will see, in the process of verification, ν_F may be rewritten using $- \rightsquigarrow_{\dagger\mathbf{F}} -$. This means that throughout, if (Δ, s, ρ) is a configuration, we will always maintain as an invariant that $s\restriction_{\mathbf{F}}$ is linearizable to ρ and that ρ is linearizable to ν_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.

Pre-Conditions, Post-Conditions, Rely and Guarantees are given by relational predicates:

$$P \subseteq \mathsf{Config} \qquad Q \subseteq \mathsf{Config} \times \mathsf{Config} \qquad \mathcal{R}, \mathcal{G} \subseteq \mathsf{Config} \times \mathsf{Config}$$

As usual, we define stability requirements on pre-conditions P and post-conditions Q:

$$\mathsf{stable}(\mathcal{R}, P) = \mathcal{R} \circ P \subseteq P \qquad \mathsf{stable}(\mathcal{R}, Q) = \mathcal{R} \circ Q \subseteq Q \wedge Q \circ \mathcal{R} \subseteq Q$$

The key logical judgment is:

$$\mathcal{G} \vdash_\alpha \{P\}\ B\ \{Q\} \iff \forall(\Delta, s, \rho).s\restriction_{\mathbf{F}} \in \nu'_F \wedge (\Delta, s, \rho) \in P \wedge s\restriction_{\mathbf{E}} \in \nu_E \wedge (\Delta', s') \in [\![B]\!]_\alpha(\Delta, s) \Rightarrow$$

$$s'\restriction_{\mathbf{F}} \in \nu'_F \wedge \exists\rho'.(\Delta, s, \rho)\ Q\ (\Delta', s', \rho') \wedge (\Delta, s, \rho)\ \mathcal{G}\ (\Delta', s', \rho') \wedge \rho \dashrightarrow \rho'$$

$$\rho \dashrightarrow \rho' \iff \exists t_P \in (M^P_{\mathbf{F}})^*.\rho \cdot t_P \rightsquigarrow_{\dagger\mathbf{F}} \rho'$$

Comparing this judgment with the possible edges of a punctual extension, we see that this rule closely models a $\mathsf{commit}_\alpha(-)$ edge. The remaining rules for commands are the usual Hoare-style rules.

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

$$\frac{\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C\ \{Q\} \qquad \mathcal{R}, \mathcal{G} \models_\alpha \{Q \circ P\}\ C'\ \{Q'\}}{\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C; C'\ \{Q'\}}\ \textsc{Seq} \qquad \frac{}{\top, \mathsf{ID} \models_\alpha \{\top\}\ \epsilon\ \{\mathsf{ID}\}}\ \textsc{Skip}$$

$$\frac{\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C\ \{Q\} \qquad Q \circ P \subseteq P}{\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C^*\ \{Q\}}\ \textsc{Iter} \qquad \frac{\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C_1\ \{Q\} \qquad \mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C_2\ \{Q\}}{\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C_1 + C_2\ \{Q\}}\ \textsc{Choice}$$

$$\frac{\begin{array}{c}\mathsf{stable}(\mathcal{R}', P') \\ P' \subseteq P \qquad \mathcal{R}' \subseteq \mathcal{R} \qquad \mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ C\ \{Q\} \qquad \mathcal{G} \subseteq \mathcal{G}' \qquad Q \subseteq Q' \qquad \mathsf{stable}(\mathcal{R}', Q')\end{array}}{\mathcal{R}', \mathcal{G}' \models_\alpha \{P'\}\ C\ \{Q'\}}\ \textsc{Weaken}$$

Now, for a local implementation $M[\alpha] = (M[\alpha]^f)_{f \in F}$ we start by defining:

$$(\Delta, s, \rho) \in \mathsf{idle}_\alpha \iff \Delta_\alpha = \varnothing \wedge \mathsf{even}(\pi_\alpha(s\restriction_{\mathrm{F}})) \wedge \mathsf{even}(\pi_\alpha(\rho))$$

$$(\Delta, s, \rho)\ \mathsf{invoke}_\alpha(f(a))\ (\Delta', s', \rho') \iff$$
$$(\Delta, s, \rho) \in \mathsf{idle}_\alpha \wedge s'\restriction_{\mathrm{F}} \in \nu'_F \wedge (\Delta'(\alpha) = [\mathrm{arg} : a] \wedge \forall \alpha' \neq \alpha. \Delta'(\alpha') = \Delta(\alpha') \wedge s' = s \cdot \boldsymbol{\alpha}{:}f \wedge \rho' = \rho \cdot \boldsymbol{\alpha}{:}f$$

$$(\Delta, s, \rho)\ \mathsf{returned}_\alpha(f)\ (\Delta', s', \rho') \iff$$
$$s'\restriction_{\mathrm{F}} \in \nu'_F \wedge (\Delta', s', \rho') = (\Delta, s, \rho) \wedge (\exists v \in \mathsf{ar}(f). \Delta(\alpha)(\mathrm{ret}) = v \wedge (\exists p. \pi_\alpha(\rho') = p \cdot v))$$

$$(\Delta, s, \rho)\ \mathsf{return}_\alpha(f)\ (\Delta', s', \rho') \iff$$
$$\Delta' = \varnothing \wedge \rho' = \rho \wedge \exists v \in \mathsf{ar}(f). \exists p. \pi_\alpha(\rho) = p \cdot v \wedge s' = s \cdot \boldsymbol{\alpha}{:}v$$

invoke and return receive their names due to their relationship with the corresponding edges of a punctual extension. $\mathsf{invoke}_\alpha(f)$ allows α to invoke f as long as that is possible, $\mathsf{returned}_\alpha(f)$ checks if f has received an appropriate return in the possibility, while $\mathsf{return}_\alpha(f)$ triggers the return and empties the local environment. Then the rule for a local implementation $M[\alpha] = (M[\alpha]^f)_{f \in F}$ is:

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

and for a concurrent implementation $M = (M[\alpha])_{\alpha \in A}$:

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

where

$$\mathsf{invoke}_\alpha(-) := \bigcup_{f \in F} \mathsf{invoke}_\alpha(f) \qquad \mathsf{return}_\alpha(-) := \bigcup_{f \in F} \mathsf{return}_\alpha(f)$$

and given relies $\mathcal{R}[\alpha]$ and guarantees $\mathcal{G}[\alpha]$ for every $\alpha \in A$, we define:

$$\mathcal{R}[A] \coloneqq \bigcap_{\alpha \in A} \mathcal{R}[\alpha] \qquad \mathcal{G}[A] \coloneqq \bigcup_{\alpha \in A} \mathcal{G}[\alpha]$$

we also write

$$P[A] \coloneqq \bigcap_{\alpha \in A} P[\alpha] \qquad Q[A] \coloneqq \bigcup_{\alpha \in A} Q[\alpha]$$

for per-conditions $P[\alpha]$ and postconditions $Q[\alpha]$ for every $\alpha \in A$.

Let's briefly explain each of the premises of these two rules.

We start with Local Impl. That the initial state is in the pre-condition $P[\alpha]^f$ of every agent ensures that every method may be called to start with. Now, for a method to be called, the α must be idle, which justifies the second condition. Stability requirements are standard in rely-guarantee reasoning. The fourth premise checks that the body of each method's $f \in F$ implementation satisfies the corresponding pre ($P[\alpha]^f$) and post-conditions ($Q[\alpha]^f$), after invoking the method, by means of $\mathsf{invoke}_\alpha(f)$, from the pre-condition; moreover, at the end of the body, we check that the value stored in ret makes sense and that it appears in the possibility by means of $\mathsf{returned}_\alpha(f)$. The final condition ensures that if method f' is invoked ($\mathsf{invoke}_\alpha(f')$) from a state satisfying the pre-condition $P[\alpha]^{f'}$, then it takes its effect ($Q[\alpha]^{f'}$) and then we both check it returned accordingly ($\mathsf{returned}_\alpha(f')$) and then we actually return in the target component ($\mathsf{return}_\alpha(f')$) then we obtain a state in which any other operation of F may be called. Now, Conc Impl merely checks that every α's local implementation satisfies their specifications, and that their relies and guarantees agree with each other, and moreover that their relies take into account the possibility of other Υ invoking and returning.

This supports the parallel composition rule that follows:

$$\frac{\begin{array}{c} A \cap B = \varnothing \qquad \mathcal{R}[A], \mathcal{G}[A] \models_A \{P[A]\}\ M[A]\ \{Q[A]\} \qquad \mathcal{G}[A] \cup \mathsf{invoke}_A(-) \cup \mathsf{return}_A(-) \subseteq \mathcal{R}[B] \\ \mathcal{R}[B], \mathcal{G}[B] \models_B \{P[B]\}\ M[B]\ \{Q[B]\} \qquad \mathcal{G}[B] \cup \mathsf{invoke}_B(-) \cup \mathsf{return}_B(-) \subseteq \mathcal{R}[A] \end{array}}{\mathcal{R}[A] \cap \mathcal{R}[B], \mathcal{G}[A] \cup \mathcal{G}[B] \models_{A \uplus B} \{P[A] \cap P[B]\}\ M[A \uplus B]\ \{Q[A] \cup Q[B]\}}\ \textsc{PComp}$$

The rules are justified by the following soundness theorem:

Proposition F.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*

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

Although we present the simplest version of our program logic, it can be extended without much trouble. A straight-forward, but useful extension is to keep track of a set of possibilities instead. While we *believe* that this program logic is complete as is, keeping track of a set of possibilities affords the user extra flexibility in verification. For instance, they may choose to simply keep track of every linearized trace that is valid at each point, instead of keeping track of a single concurrent trace representing them.

F.3 The Middle Queue Concurrent Object

Consider a concurrent object with signature

$$\mathsf{MidQueue} \coloneqq \{\mathsf{middeq} : \mathbb{N}, \mathsf{enq} : \mathbb{N} \to \mathbf{1}\}$$

Its semantics is similar to a regular queue. An $\mathsf{enq}(n)$ adds n to the end of the queue, like the usual enq in a queue object. $\mathsf{middeq}()$, on the other hand, instead of dequeuing the front element of the queue, dequeues the center element of the queue (if the queue is even-length, it returns the nearest of the two elements to front of the queue).

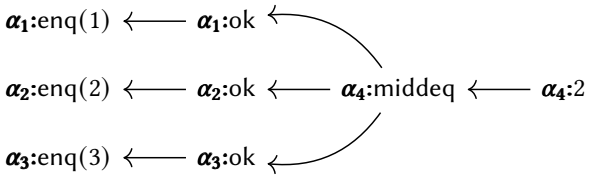
We argue now that Khyzha et al. [2017]'s methodology cannot prove the middle queue object is linearizable. The issue is in that they keep as invariant that every linearization of their possibility, represented as an interval partial order, is valid (in the sense that it satisfies the linearized specification). Consider the trace:

$$s = \boldsymbol{\alpha_1}\text{:enq}(1) \cdot \boldsymbol{\alpha_2}\text{:enq}(2) \cdot \boldsymbol{\alpha_3}\text{:enq}(3) \cdot \boldsymbol{\alpha_1}\text{:ok} \cdot \boldsymbol{\alpha_2}\text{:ok} \cdot \boldsymbol{\alpha_3}\text{:ok} \cdot \boldsymbol{\alpha_4}\text{:middeq} \cdot \boldsymbol{\alpha_4}\text{:2}$$

We will write:

$$t_{x,y,z} = \boldsymbol{\alpha_x}\text{:enq}(x) \cdot \boldsymbol{\alpha_x}\text{:ok} \cdot \boldsymbol{\alpha_y}\text{:enq}(y) \cdot \boldsymbol{\alpha_y}\text{:ok} \cdot \boldsymbol{\alpha_z}\text{:enq}(z) \cdot \boldsymbol{\alpha_z}\text{:ok} \cdot \boldsymbol{\alpha_4}\text{:middeq} \cdot \boldsymbol{\alpha_4}\text{:2}$$

The only two valid linearizations of s are $t_{1,2,3}$ and $t_{3,2,1}$. Because of happens before ordering, the least ordered interval partial order that can be kept at this point is



This partial order does not satisfy their invariant as $t_{2,1,3}$, $t_{2,3,1}$, $t_{1,3,2}$, $t_{3,1,2}$ are not valid linearization but are a linearization of this partial order. We must therefore use a more ordered partial order that orders the enq(2) between the enq(1) and the enq(3) to rule out these linearizations. So we must choose between

$$\boldsymbol{\alpha_1}\text{:enq}(1) \leftarrow \boldsymbol{\alpha_1}\text{:ok} \leftarrow \boldsymbol{\alpha_2}\text{:enq}(2) \leftarrow \boldsymbol{\alpha_2}\text{:ok} \leftarrow \boldsymbol{\alpha_3}\text{:enq}(3) \leftarrow \boldsymbol{\alpha_3}\text{:ok} \leftarrow \boldsymbol{\alpha_4}\text{:middeq} \leftarrow \boldsymbol{\alpha_4}\text{:2}$$

and

$$\boldsymbol{\alpha_3}\text{:enq}(3) \leftarrow \boldsymbol{\alpha_3}\text{:ok} \leftarrow \boldsymbol{\alpha_2}\text{:enq}(2) \leftarrow \boldsymbol{\alpha_2}\text{:ok} \leftarrow \boldsymbol{\alpha_1}\text{:enq}(1) \leftarrow \boldsymbol{\alpha_1}\text{:ok} \leftarrow \boldsymbol{\alpha_4}\text{:middeq} \leftarrow \boldsymbol{\alpha_4}\text{:2}$$

But no choice is sound at this point, as we can invalidate each choice by extending the trace with $\boldsymbol{\alpha_4}$:middeq $\cdot$ $\boldsymbol{\alpha_4}$:2 or $\boldsymbol{\alpha_4}$:middeq $\cdot$ $\boldsymbol{\alpha_4}$:1, respectively. As our invariant merely requires us to guarantee that there *exists* a valid linearization for our possibility, we are able to keep the least ordered interval partial order we showed without harm.

F.4 Soundness

We briefly outline the key reasons why the operational semantics agrees with the denotation.

Proposition F.3. *For any $M \in \mathsf{CMod}$, $[\![M]\!]$ is a strategy of type $\dagger\mathbf{E} \multimap \dagger\mathbf{F}$ and given $v_E : \dagger\mathbf{E}$,*

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

Proof. The proof for this is straight-forward, but tedious, and therefore we merely give the outline. $[\![M]\!]$ is well-defined, as by definition of ModState, the play in the state is a play of $P_{\dagger\mathbf{E}\multimap\dagger\mathbf{F}}$. Moreover, it is prefix-closed and receptive by definition. Now, that

$$[\![M]\!] = \|_{\alpha\in\Upsilon}\ \iota_\alpha(\pi_\alpha[\![M]\!])$$

follows from the fact that in the concurrent semantics, in any state, any agent can take a step. Moreover, a step either does not modify the underlying state s (in the case of **id**, ret $-$ or assert$(-)$), or, in the case of $x \leftarrow e(a)$ it either adds the move $\boldsymbol{\alpha}$:e (O-position case) or some response $\boldsymbol{\alpha}$:v (P-position case). Hence, any (sequentially consistent) interleaving of the projections can be produced. So it remains to prove that $\pi_\alpha([\![M]\!])$ is always a sequential implementation. Was it not for the local environment, this would be immediate, as between an O-move $f \in \mathbf{F}$ and its response the executed code is generated from the same command $M[\alpha]^f$. Now, the local environment is emptied on every

response in **F**, hence on every O move in **F** it is empty prior to the invocation. Hence, under the same arguments, the same traces are produced by $M[\alpha]^f$ every time, which implies regularity. That

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

can be observed from the fact that the operational semantics merely restricts steps to those that play as v_E in the source component, which is the same as composing with v_E. □

Our proof of soundness is adapted from that from Khyzha et al. [2017]. Define $\mathsf{rely}(\mathcal{R}, P)$ of a pre-condition P by a rely $\mathcal{R}$:

$$\mathsf{rely}(\mathcal{R}, P) = P \cup \mathcal{R} \circ P$$

Then, we define the judgment

$$\mathsf{safe}_\alpha(\mathcal{R}, \mathcal{G}, P, s, Q)$$

inductively as follows:

$$\frac{\mathsf{rely}(\mathcal{R}, P) \subseteq Q \circ P_0}{\mathsf{safe}_\alpha(\mathcal{R}, \mathcal{G}, P_0, P, \mathsf{skip}, Q)}\ \textsc{Done}$$

$$\frac{\forall C'.C \rightarrowtail^X_B C' \Rightarrow \exists P'.\mathcal{R}, \mathcal{G} \models_\alpha \{\mathsf{rely}(\mathcal{R}, P)\}\ B\ \{P'\}\ \textsc{Prim} \quad \mathsf{safe}_\alpha(\mathcal{R}, \mathcal{G}, P_0, P' \circ \mathsf{rely}(\mathcal{R}, P), C', Q)}{\mathsf{safe}_\alpha(\mathcal{R}, \mathcal{G}, P_0, P, C, Q)}\ \textsc{Step}$$

A straight-forward proof by induction shows that.

Lemma F.4. *If*

$$\mathcal{R}, \mathcal{G} \models_\alpha \{P\}\ s\ \{Q\}$$

then

$$\mathsf{safe}_\alpha(\mathcal{R}, \mathcal{G}, P, P, s, Q)$$

Proposition F.5 (Soundness). *If* $\mathcal{R}[A], \mathcal{G}[A] \models_A \{P[A]\}\ M[A]\ \{Q[A]\}$ *and* $(v'_E : \dagger\mathbf{E}, v_E : \dagger\mathbf{E})$ *is a linearizable concurrent object then*

$$v'_E; [\![M[A]]\!] \cap v'_F \subseteq K_{\mathsf{Conc}}\ v_F$$

Proof. Start by noting that by assumption and Prop. 5.6 it follows that if

$$v_E; [\![M[A]]\!] \cap v'_F \subseteq K_{\mathsf{Conc}}\ v_F$$

then

$$v'_E; [\![M[A]]\!] \cap v'_F \subseteq v_E; [\![M[A]]\!] \cap v'_F \subseteq K_{\mathsf{Conc}}\ v_F$$

so it is enough to show

$$v_E; [\![M[A]]\!] \cap v'_F \subseteq K_{\mathsf{Conc}}\ v_F$$

By definition

$$P[A] = \bigcap_{\alpha \in A} P[\alpha] \qquad Q = \bigcup_{\alpha \in A} Q[\alpha]$$

where for each $\alpha \in A$

$$P[\alpha] = \bigcap_{f \in F} P[\alpha]^f \qquad Q[\alpha] = \bigcup_{f \in F} Q[\alpha]^f$$

moreover, for every $\alpha \in A$ such that

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P[\alpha]\}\ M[\alpha]\ \{Q[\alpha]\}$$

and hence for every $f \in F$:

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P[\alpha]^f\}\ M[\alpha]^f\ \{Q[\alpha]^f\}$$

We prove the result by induction on the length of

$$\langle c_0, \Delta_0, \epsilon\rangle \longrightarrow\!\!\!\!\!\rightarrow^{M}_{\nu_E} \langle c, \Delta, s\rangle$$

for which we maintain the invariant that

$$s\restriction_{\mathbf{F}} \in \nu'_F$$

and that there is a position ρ_F such that

$$s\restriction_{\mathbf{F}} \dashrightarrow \rho_F$$

and that there are pre-conditions P_α for every $\alpha \in A$ such that

$$(\Delta, s, \rho_F) \in P_\alpha \qquad \mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)$$

and moreover:

$$c(\alpha) = \mathsf{idle} \Rightarrow P_\alpha \subseteq \mathsf{idle}_\alpha \wedge P_\alpha \subseteq P[\alpha]$$

$$c(\alpha) \neq \mathsf{idle} \Rightarrow \exists f \in F.\mathsf{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, P_\alpha, c(\alpha), \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f)$$

We note at this point that if this invariant holds about $p = s$ then in particular

$$s\restriction_{\mathbf{F}} \dashrightarrow \rho_F$$

and by the definition of possibility and Prop. 5.3 it follows that

$$s\restriction_{\mathbf{F}} \in K_{\mathsf{Conc}}\ \nu_F$$

We now start the proof proper. We will not bother with the invariant $s\restriction_{\mathbf{F}} \in \nu'_F$ from the definitions of invoke, return and Prim. In the case where

$$\langle c, \Delta, s\rangle = \langle c_0, \Delta_0, \epsilon\rangle$$

we set $P_\alpha = P[\alpha]$ and $\rho_F = \epsilon$. Most of the invariants are easily established. We stress the invariants around P_α. Note that in this case $c(\alpha) = \mathsf{idle}$. Now note that $(\Delta, p, \rho_F) \in \mathsf{idle}_\alpha$ for every $\alpha \in A$ by definition. Moreover

$$P_\alpha = P[\alpha] = \cap_{f\in F} P[\alpha]^f \subseteq \mathsf{idle}_\alpha$$

by assumption that ConcImpl holds. By definition:

$$P_\alpha = P[\alpha] \subseteq P[\alpha]$$

Furthermore,

$$(\Delta, p, \rho_F) \in P_\alpha \qquad \mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)$$

by ConcImpl, and $P[\alpha]$ is stable by ConcImpl.

For the inductive step we have that

$$\langle c_0, \Delta_0, \epsilon\rangle \longrightarrow\!\!\!\!\!\rightarrow^{M}_{\nu_E} \langle c, \Delta, s\rangle \longrightarrow\!\!\!\!\!\rightarrow^{M}_{\nu_E} \langle c', \Delta', s'\rangle$$

Moreover, we have

$$s\restriction_{\mathbf{F}} \dashrightarrow \rho_F$$

and a pre-condition P_α for each agent $\alpha \in A$ such that

$$(\Delta, s, \rho_F) \in P_\alpha \qquad \mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)$$

and moreover:

$$c(\alpha) = \mathsf{idle} \Rightarrow P_\alpha \subseteq \mathsf{idle}_\alpha \wedge P_\alpha \subseteq P[\alpha]$$

$$c(\alpha) \neq \mathsf{idle} \Rightarrow \exists f \in F.\mathsf{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, P_\alpha, c(\alpha), \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f)$$

We split the proof into cases depending on the continuation for the agent α that modifies the state in the last step.

$c(\alpha) = \mathsf{idle}$ Note that in this case, $c' = c[\alpha : M[\alpha]^f]$ for some $f \in F$, $s' = s \cdot \boldsymbol{\alpha}{:}f$. By the invariant, $P_\alpha \subseteq \mathsf{idle}_\alpha$, and in particular $(\Delta, s, \rho_F) \in \mathsf{idle}_\alpha$. Let (Δ', s', ρ_F') be such that ρ_F' is any ρ_F' such that

$$(s, \rho_F)\ \mathsf{invoke}_\alpha(f)\ (p', \rho_F')$$

Note that as $(\Delta, s, \rho_F) \in \mathsf{idle}_\alpha$ it immediately follows that there is exactly one such ρ_F' (given by just appending $\boldsymbol{\alpha}{:}f$ to ρ_F). We argue that:

$$\{s' \restriction_{\mathbf{F}}\} \dashrightarrow \rho_F'$$

By definition,

$$\rho_F' = \rho_F \cdot \boldsymbol{\alpha}{:}f$$

Now, by induction there is t_P such that

$$s \restriction_{\mathbf{F}} \cdot t_P \rightsquigarrow_{\dagger \mathbf{F}} \rho_F$$

but then

$$s \restriction_{\mathbf{F}} \cdot \boldsymbol{\alpha}{:}f \cdot t_P \rightsquigarrow_{\dagger \mathbf{F}} s \restriction_{\mathbf{F}} \cdot t_P \cdot \boldsymbol{\alpha}{:}f \rightsquigarrow_{\dagger \mathbf{F}} \rho_F \cdot \boldsymbol{\alpha}{:}f = \rho_F$$

it follows that

$$\{s' \restriction_{\mathbf{F}}\} \dashrightarrow \rho_F'$$

Note moreover that as $(\Delta, s, \rho_F) \in P_\alpha$, by induction

$$(\Delta, p, \rho_F) \in P_\alpha \subseteq P[\alpha] \subseteq P[\alpha]^f$$

and by construction

$$(\Delta, p, \rho_F)\ \mathsf{invoke}_\alpha(f)\ (\Delta', p', \rho_F')$$

so that

$$(\Delta', p', \rho_F') \in \mathsf{invoke}_{\alpha'}(f) \circ P[\alpha']^f$$

In addition, by assumption

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{\mathsf{invoke}_\alpha(f) \circ P[\alpha]^f\}\ M[\alpha]^f\ \{\mathsf{returned}_\alpha(f) \circ Q[\alpha]^f\}$$

By Lemma F.4 it follows that

$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, M[\alpha]^f, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f)$$

and

$$\mathsf{stable}(\mathcal{R}[\alpha], \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f)$$

so if we let

$$P_\alpha' = \mathsf{rely}(\mathcal{R}, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f)$$

Then it is almost immediate from the definition of safe that

$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, P_\alpha', M[\alpha]^f, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f)$$

Moreover, by definition P_α' is stable. Hence, P_α' satisfies all the necessary invariants.
Now, for $\alpha' \in A$ such that $\alpha \neq \alpha'$ we set $P_{\alpha'}' = P_{\alpha'}$. We must show that $(\Delta', s', \rho_F') \in P_{\alpha'}$. For that, note that by induction $P_{\alpha'}$ is stable and by assumption $\mathcal{R}[\alpha']$ contains $\mathsf{invoke}_\alpha(f) \subseteq \mathsf{invoke}_\alpha(-)$ so that $P_{\alpha'}$ is stable under $\mathsf{invoke}_\alpha(f)$. Now, $(\Delta', s', \rho_F') \in \mathsf{idle}_\alpha \iff (\Delta, s, \rho_F) \in \mathsf{idle}_\alpha$ by definition. It is obvious that if $(\Delta', s', \rho_F') \in \mathsf{idle}_\alpha$ then all the conditions are still satisfied by induction. Finally, if $(\Delta', s', \rho_F') \notin \mathsf{idle}_\alpha$ then there is an operation f' for which it holds that

$$\mathsf{safe}_{\alpha'}(\mathcal{R}[\alpha'], \mathcal{G}[\alpha'], \mathsf{invoke}_{\alpha'}(f') \circ P[\alpha']^{f'}, P_{\alpha'}, c(\alpha'), \mathsf{returned}_{\alpha'}(f') \circ Q[\alpha']^{f'})$$

But then, it is immediate that $c'(\alpha') = c(\alpha')$ so that

$$\mathsf{safe}_{\alpha'}(\mathcal{R}[\alpha'], \mathcal{G}[\alpha'], \mathsf{invoke}_{\alpha'}(f') \circ P[\alpha']^{f'}, P_{\alpha'}, c'(\alpha'), \mathsf{returned}_{\alpha'}(f') \circ Q[\alpha']^{f'})$$

$c(\alpha) = \mathsf{skip}$ In this case it must be that $c' = c[\alpha : \mathsf{idle}]$, $s' = s \cdot \boldsymbol{\alpha}{:}v$ for some $v \in \mathsf{ar}(f)$ and $\Delta' = \Delta[\alpha : \varnothing]$. By induction there exists $f \in F$ such that (note that the f may be taken to be the same because of the definition of safe and the fact that $s' = s \cdot \boldsymbol{\alpha}{:}v$ is a valid play)

$$\mathsf{safe}_{\alpha}(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_{\alpha}(f) \circ P[\alpha]^{f}, P_{\alpha}, c(\alpha), \mathsf{returned}_{\alpha}(f) \circ Q[\alpha]^{f})$$

In this case safe_α consists of a Done rule, and therefore:

$$\mathsf{rely}(\mathcal{R}, P_\alpha) \subseteq \mathsf{return}_\alpha(f) \circ Q[\alpha]^f$$

In particular

$$(\Delta, s, \rho_F) \in P_\alpha \subseteq \mathsf{rely}(\mathcal{R}, P_\alpha) \subseteq \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f$$

Therefore, ρ_F already has the return v to f for α. Then, we have that if we let $\rho'_F = \rho_F$ then:

$$(\Delta, s, \rho_F)\ \mathsf{return}_\alpha(f)\ (\Delta', s', \rho'_F)$$

Moreover, by induction

$$s \upharpoonright_{\mathbf{F}} \dashrightarrow \rho_F$$

and therefore there is t_P proving the above derivation. Now,

$$s' \upharpoonright_{\mathbf{F}} = s \upharpoonright_{\mathbf{F}} \cdot \boldsymbol{\alpha}{:}v$$

Hence

$$s' \upharpoonright_{\mathbf{F}} \dashrightarrow_{\nu_F} \rho'_F = \rho_F$$

by choosing $t'_P = t_P \setminus \boldsymbol{\alpha}{:}v$. So, we set

$$P'_\alpha = P[\alpha]$$

Then, by construction and by LocalImpl:

$$(\Delta', p', \rho'_F) \in \mathsf{return}_\alpha(f) \circ \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f \circ \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f \subseteq P'_\alpha$$

$$\mathsf{stable}(\mathcal{R}[\alpha], P'_\alpha)$$

Moreover,

$$P'_\alpha = P[\alpha] \subseteq \mathsf{idle}_\alpha$$

and

$$P'_\alpha = P[\alpha] \subseteq P[\alpha]$$

For the other agents $\alpha' \in A$ the invariants all hold by induction by setting $P'_{\alpha'} = P_{\alpha'}$. Indeed, the point of pressure is showing that $(\Delta', s', \rho'_F) \in P'_{\alpha'}$ but $P'_{\alpha'}$ is stable under $\mathsf{return}_{\alpha'}(-)$ by assumption $(\Delta, p, \rho_F) \in P_{\alpha'}$ so that $(\Delta', p', \rho'_F) \in P'_{\alpha'}$.

$c(\alpha) = C$ **and** $C \neq \mathsf{skip}$ In this case, we have that $C \rightarrowtail^X_B C'$ and $(\Delta', s') \in [\![B]\!]_\alpha(\Delta, s)$. The interesting case is when B is an command issuing an effect from E, so we assume $s' = s \cdot \boldsymbol{\alpha}{:}m$ where m is the move resulting from B. Moreover, there is some $f \in F$

$$\mathsf{safe}_{\alpha}(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_{\alpha}(f) \circ P[\alpha]^{f}, P_{\alpha}, C, \mathsf{returned}_{\alpha}(f) \circ Q[\alpha]^{f})$$

Now, notice that it follows by safe_α that

$$\exists P'.\mathcal{R}, \mathcal{G} \models_\alpha \{\mathsf{rely}(\mathcal{R}[\alpha], P_\alpha)\}\ B\ \{P'\}$$

and

$$\mathsf{safe}_{\alpha}(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_{\alpha}(f) \circ P[\alpha]^{f}, P' \circ \mathsf{rely}(\mathcal{R}, P_\alpha), C', \mathsf{returned}_{\alpha}(f) \circ Q[\alpha]^{f})$$

Now, by assumption $(\Delta, p, \rho_F) \in P_\alpha$ and $s\restriction_{\mathbf{E}} \cdot \boldsymbol{\alpha}{:}m \in \nu_E$. Therefore, by

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{\mathsf{rely}(\mathcal{R}[\alpha], P_\alpha)\}\ B\ \{P'\}$$

it follows that there is some ρ'_F such that

$$(\Delta, p, \rho_F)\ P'\ (\Delta', s \cdot \boldsymbol{\alpha}{:}m, \rho'_F) \qquad (\Delta, s, \rho_F)\ \mathcal{G}\ (\Delta', s \cdot \boldsymbol{\alpha}{:}m, \rho'_F) \qquad \rho_F \dashrightarrow \rho'_F$$

by assumption

$$s\restriction_{\mathbf{F}} \dashrightarrow \rho_F$$

so that

$$s'\restriction_{\mathbf{F}} = s\restriction_{\mathbf{F}} \dashrightarrow \rho_F \dashrightarrow \rho'_F$$

Moreover, if we set $P'_\alpha = P' \circ \mathsf{rely}(\mathcal{R}[\alpha], P_\alpha)$ then

$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, P'_\alpha, C', \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f)$$

and moreover

$$(\Delta', s', \rho'_F) \in P'_\alpha$$

which meets all of the necessary invariants.
For agents $\alpha' \in A$ such that $\alpha \neq \alpha'$, the invariants all still hold by induction, except for perhaps $(\Delta', p', \rho'_F) \in P_{\alpha'}$. But as

$$(\Delta, p, \rho_F)\ \mathcal{G}[\alpha']\ (\Delta', p', \rho'_F)$$

and

$$(\Delta, p, \rho_F) \in P_{\alpha'}$$

it follows from assumption that

$$\mathcal{G}[\alpha] \subseteq \mathcal{R}[\alpha']$$

and by stability that

$$(\Delta', p', \rho'_F) \in P_{\alpha'}$$

as desired.

□

G VERIFICATION OF A TICKET LOCK AND SHARED QUEUE IMPLEMENTATIONS

In this section we give detailed proofs, using the program logic from §F, that the components in §2 assemble into certified linearizable object implementations. In G.1 we show the proof for the ticket lock implementation M_{lock}, and in G.2 for M_{squeue}.

For practical purposes it is often useful to assume ν'_F is not receptive. This does not affect the result as if $\nu'_F \subseteq K_{\mathsf{Conc}}\ \nu_F$ then $\mathsf{strat}\left(\nu'_F\right) \subseteq K_{\mathsf{Conc}}\ \nu_F$, and similarly for $\nu_E; [\![M]\!]$.

G.1 Ticket Lock

Here, we assume that

$$(\nu'_{\mathrm{fai}} : \dagger\mathsf{FAI}, \nu_{\mathrm{fai}} : \dagger\mathsf{FAI}) \quad (\nu'_{\mathrm{counter}} : \dagger\mathsf{Counter}, \nu_{\mathrm{counter}} : \dagger\mathsf{Counter}) \quad (\nu'_{\mathrm{yield}} : \dagger\mathsf{Yield}, \nu_{\mathrm{yield}} : \dagger\mathsf{Yield})$$

are linearizable objects. Therefore, by *locality*

$$(\nu'_E, \nu_E) := (\nu'_{\mathrm{fai}} \otimes \nu'_{\mathrm{counter}} \otimes \nu'_{\mathrm{yield}}, \nu_{\mathrm{fai}} \otimes \nu_{\mathrm{counter}} \otimes \nu_{\mathrm{yield}})$$

is a linearizable object. We therefore seek to show that

$$[\![M_{\mathrm{lock}}]\!] : (\nu'_E, \nu_E) \longrightarrow \left(\mathsf{strat}\left(\nu'_{\mathrm{lock}}\right), \nu_{\mathrm{lock}}\right)$$

by using our program logic. By the remarks at the beginning of this section, here v'_{lock} is the set of plays $s \in P_{\dagger\text{Lock}}$ such that

$$\forall \alpha \in \Upsilon. \exists t \in (\mathsf{acq} \cdot \mathsf{ok} \cdot \mathsf{rel} \cdot \mathsf{ok})^*. \pi_\alpha(s) \sqsubseteq t$$

we are allowed to take this v'_{lock}, which is not receptive, because of the remarks in the beginning of this section. With the proof setup explained, we proceed to the proof proper.

We apply the program logic developed in §F on the ticket lock implementation discussed in §2.2.2. In particular, we concern ourselves to the adapted implementation in Figure 7, written in the language introduced in §F.1, and already de-sugared.

```
1  acq() {
2    my_tick <- fai();
3    ( assert (cur_tick ≠ my_tick);
4      yield();
5      cur_tick <- get() )* ;
6    assert (cur_tick = my_tick);
7    ret ok
8  }
```

```
1  rel() {
2    inc();
3    ret ok
4  }
```

Fig. 7. Ticket Lock Implementation in language developed in §F.1

Before go into details, we briefly describe the intuition behind ticket locks. Each agent tries to acquire a lock first by atomically fetching a ticket number and incrementing its value, making sure the next agent will get a greater ticket number. Afterwards, each agent waits for the "now serving" counter to become its ticket number, at which point they are granted access to the shared resource protected by the lock. When the lock holder tries to release the lock, it simply (non-atomically) increments the counter value. Part of the correctness proof is to establish that write-write will never happen on the counter, otherwise it would lead to undefined behavior.

Formally, we need to prove the following judgment,

$$\mathcal{R}[\Upsilon], \mathcal{G}[\Upsilon] \models_\Upsilon \{P[A]\}\ M_{\text{Lock}}[\Upsilon]\ \{Q[A]\}$$

according to the Conc Impl and Local Impl rule and symmetry, in addition to other obligations, we need to find a definition of $P[\alpha]^f$ and $Q[\alpha]^f$ for $f \in \{\mathsf{acq}, \mathsf{rel}\}$, $\mathcal{R}[\alpha]$, and $\mathcal{G}[\alpha]$ (same for every $\alpha \in \Upsilon$) such that the following three judgments holds,

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P[\alpha]^{\mathsf{acq}}\}\ M_{\text{Lock}}[\alpha]^{\mathsf{acq}}\ \{Q[\alpha]^{\mathsf{acq}}\}$$

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P[\alpha]^{\mathsf{rel}}\}\ M_{\text{Lock}}[\alpha]^{\mathsf{rel}}\ \{Q[\alpha]^{\mathsf{rel}}\}$$

$$\forall \alpha, \alpha' \in \Upsilon. \alpha \neq \alpha' \implies \mathcal{G}[\alpha] \cup \mathsf{invoke}_\alpha(-) \cup \mathsf{return}_\alpha(-,-) \subseteq \mathcal{R}[\alpha']$$

To define preconditions and postconditions of acquire and release and rely/guarantee conditions, it would be helpful to have access to the current counter value, ticket value, lock owner, etc., in addition to the history. To this end, we define a set of functions that take different types of plays to calculate these state values.

We first define three functions over lock events,

$$\mathsf{linowner} : P_{!\text{Lock}} \to \Upsilon + \{\varnothing\} + \{\bot\} \quad \mathsf{lin} : P_{\dagger\text{Lock}} \to P_{!\text{Lock}} \quad \mathsf{owner} : P_{\dagger\text{Lock}} \to \Upsilon + \{\varnothing\} + \{\bot\}$$

$$\mathsf{linowner}(p) := \begin{cases} \varnothing & p = \epsilon \\ \alpha & p = p' \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \mathsf{linowner}(p') = \varnothing \\ \varnothing & p = p' \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot \boldsymbol{\alpha}{:}\mathsf{rel} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \mathsf{linowner}(p') = \varnothing \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{lin}(p) := p' \text{ s.t. } p' \in P_{!\mathsf{Lock}} \wedge p' \sqsubseteq p \wedge \mathsf{linowner}(p') \neq \bot \wedge (\forall p''.p'' \sqsubseteq p \wedge \mathsf{linowner}(p'') \neq \bot \implies p'' \sqsubseteq p')$$

$$\mathsf{owner}(p) := \mathsf{linowner}(\mathsf{lin}(p))$$

linowner takes an atomic play of Lock as input. It checks for the lock invariant (acquire is always followed by release of the same thread) and returns the current owner agent. The function lin takes a concurrent play of Lock and returns the longest prefix of it that is atomic and satisfies the lock invariant. Finally, the function owner takes any concurrent play of Lock and returns the owner calculated by linowner ∘ lin.

We then define three functions over underlay events $\mathsf{ctrval} : P_{\dagger\mathsf{Counter}\otimes\dagger\mathsf{FAI}\otimes\dagger\mathsf{Yield}} \to \mathbb{N} + \{\bot\}$, $\mathsf{mytkt} : P_{\dagger\mathsf{Counter}\otimes\dagger\mathsf{FAI}\otimes\dagger\mathsf{Yield}} \to \mathbb{N} + \{\varnothing\}$, and $\mathsf{newtkt} : P_{\dagger\mathsf{Counter}\otimes\dagger\mathsf{FAI}\otimes\dagger\mathsf{Yield}} \to \mathbb{N}$.

$$\mathsf{ctrval}(p) := \begin{cases} \left\lceil \frac{\left|(p \restriction_{\mathsf{Counter}}) \restriction_{\{\mathsf{inc}:\mathbf{1}\}}\right|}{2} \right\rceil & (p \restriction_{\mathsf{Counter}}) \restriction_{\{\mathsf{inc}:\mathbf{1}\}} \in P_{!\{\mathsf{inc}:\mathbf{1}\}} \\ \bot & \text{otherwise} \end{cases}$$

ctrval accepts any trace that contains only atomic $\{\mathsf{inc} : \mathbf{1}\}$ sequences for the Counter object. It returns the number of inc calls in the trace, which is also the return value of get if invoked at the time, according to ν_{Counter}.

$$\mathsf{mytkt}_\alpha(p) := \begin{cases} n & \exists p', p''.\pi_\alpha(p) = p' \cdot \mathsf{fai} \cdot n \cdot p'' \wedge p'' \sqsubseteq (\mathsf{yield} \cdot \mathsf{ok} \cdot \mathsf{get} \cdot n')^* \cdot \mathsf{inc} \\ \varnothing & \text{otherwise} \end{cases}$$

$$\mathsf{newtkt}(p) := \left\lceil \frac{\left|(p \restriction_{\mathsf{FAI}}) \restriction_{\{\mathsf{fai}:\mathbb{N}\}}\right|}{2} \right\rceil$$

mytkt_α returns the current ticket for a particular agent. It will only return if the ticket is still active, *i.e.*, the agent has already acquired a ticket in acq but haven't reached the linearization point in the matching rel. On the other hand, newtkt always returns the next ticket to be issued.

With the helper functions defined, we can now state the preconditions and postconditions as follows,

$$\mathsf{acqed}[\alpha](\Delta, s, \rho) \iff I(\Delta, s, \rho) \wedge \mathsf{owner}(\rho) \neq \alpha$$
$$\mathsf{reled}[\alpha](\Delta, s, \rho) \iff I(\Delta, s, \rho) \wedge \mathsf{owner}(\rho) = \alpha$$
$$P[\alpha]^{\mathsf{acq}} := \mathsf{reled}[\alpha] \cap \mathsf{idle}_\alpha$$
$$(\Delta, s, \rho)\ Q[\alpha]^{\mathsf{acq}}\ (\Delta', s', \rho') \iff \mathsf{acqed}[\alpha](\Delta', s', \rho')$$
$$P[\alpha]^{\mathsf{rel}} := \mathsf{acqed}[\alpha] \cap \mathsf{idle}_\alpha$$
$$(\Delta, s, \rho)\ Q[\alpha]^{\mathsf{rel}}\ (\Delta', s', \rho') \iff \mathsf{reled}[\alpha](\Delta', s', \rho')$$

One may notice that postconditions, while being an relation, is only predicated over the post-state. This is true for most of the reasoning except for the linearization point as we shall see later. All predicates (relations) are composed of a shared invariant I and an ownership assertion. The

definition of I is given below,

$$\begin{aligned}
\mathsf{mytick}[\alpha](\Delta, s) &\iff \Delta(\alpha)(\mathtt{my_tick}) \neq \bot \implies \Delta(\alpha)(\mathtt{my_tick}) = \mathsf{mytkt}_\alpha(s) \\
\mathsf{curtick}[\alpha](\Delta, s) &\iff \Delta(\alpha)(\mathtt{cur_tick}) \neq \bot \implies \exists s'.s' \sqsubseteq s \wedge \mathsf{ctrval}(s') = \Delta(\alpha)(\mathtt{cur_tick})
\end{aligned}$$

$$I[\alpha](\Delta, s, \rho) \iff \begin{pmatrix}
\mathsf{owner}(\rho) \neq \bot \wedge \mathsf{ctrval}(s) \neq \bot \quad \wedge \\
\mathsf{mytkt}_\alpha(s) \neq \varnothing \implies \mathsf{ctrval}(s) \leq \mathsf{mytkt}_\alpha(s) \quad \wedge \\
\mathsf{ctrval}(s) \leq \mathsf{newtkt}(s) \quad \wedge \\
\mathsf{mytkt}_\alpha(s) = \mathsf{ctrval}(s) \implies \mathsf{owner}(\rho) \in \{\varnothing, \alpha\} \quad \wedge \\
\mathsf{newtkt}(s) = \mathsf{ctrval}(s) \implies \mathsf{owner}(\rho) = \varnothing \quad \wedge \\
\mathsf{owner}(\rho) = \alpha \implies \mathsf{mytkt}_\alpha(s) = \mathsf{ctrval}(s) \quad \wedge \\
\mathsf{mytick}[\alpha](\Delta, s) \wedge \mathsf{curtick}[\alpha](\Delta, s)
\end{pmatrix}$$

The invariant I not only relates the local environment to the shared objects, it also specify the expected behavior of shared objects, such as the current value of the counter object is never greater than the next ticket to be dispensed. As we shall describe later, we also maintain I during execution inside the functions.

To prove $\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P[\alpha]^f\}\ M_{\mathsf{Lock}}[\alpha]^f\ \{Q[\alpha]^f\}$, we need such a $\mathcal{R}[\alpha]$ that both $\mathsf{stable}(\mathcal{R}, P[\alpha]^f)$ and $\mathsf{stable}(\mathcal{R}, Q[\alpha]^f)$ holds. We define $\mathcal{R}[\alpha]$ in such a way that the stability is trivial to prove,

$$(\Delta, s, \rho)\ \mathcal{R}[\alpha]\ (\Delta, s', \rho') \iff \begin{pmatrix}
(\Delta, s, \rho)\ \mathsf{invoke}_{A\setminus\alpha}(-)\ (\Delta, s', \rho') \vee \\
(\Delta, s, \rho)\ \mathsf{return}_{A\setminus\alpha}(-)\ (\Delta, s', \rho') \vee \\
\begin{pmatrix}
\mathsf{owner}(\rho') \neq \bot \wedge \mathsf{ctrval}(s') \neq \bot \quad \wedge \\
\begin{pmatrix}
\left(\mathsf{mytkt}_\alpha(s) \neq \varnothing \implies \mathsf{ctrval}(s) \leq \mathsf{mytkt}_\alpha(s)\right) \implies \\
\left(\mathsf{mytkt}_\alpha(s') \neq \varnothing \implies \mathsf{ctrval}(s') \leq \mathsf{mytkt}_\alpha(s')\right)
\end{pmatrix} \wedge \\
\mathsf{ctrval}(s') \leq \mathsf{newtkt}(s') \quad \wedge \\
\begin{pmatrix}
\left(\mathsf{mytkt}_\alpha(s) = \mathsf{ctrval}(s) \implies \mathsf{owner}(\rho) \in \{\varnothing, \alpha\}\right) \implies \\
\left(\mathsf{mytkt}_\alpha(s') = \mathsf{ctrval}(s') \implies \mathsf{owner}(\rho') \in \{\varnothing, \alpha\}\right)
\end{pmatrix} \wedge \\
\mathsf{newtkt}(s') = \mathsf{ctrval}(s') \implies \mathsf{owner}(\rho') = \varnothing \quad \wedge \\
\mathsf{owner}(\rho) = \alpha \implies (\mathsf{lin}(\rho) = \mathsf{lin}(\rho') \wedge \mathsf{ctrval}(s) = \mathsf{ctrval}(s')) \quad \wedge \\
\mathsf{owner}(\rho) \neq \alpha \implies \mathsf{owner}(\rho') \neq \alpha
\end{pmatrix}
\end{pmatrix}$$

while the stability of the invariant I is mostly self-evident, we will present the stability argument for the ownership assertions below,

- If $\mathsf{owner}(\rho) = \alpha$, we can rely on that $\mathsf{lin}(\rho) = \mathsf{lin}(\rho')$, through the definition of owner we can deduce that $\mathsf{owner}(\rho') = \mathsf{owner}(\rho) = \alpha$. With a similar argument we can also deduce that $\mathsf{mytkt}_\alpha(s') = \mathsf{ctrval}(s')$ assuming ownership of the lock,
- If $\mathsf{owner}(\rho) \neq \alpha$, the last conjunct of $\mathcal{R}[\alpha]$ enforces that we won't become the owner by any other agent's action.

In addition to $\mathcal{R}[\alpha]$, we also need to define $\mathcal{G}[\alpha]$ such that $\mathcal{G}[\alpha] \cup \mathsf{invoke}_\alpha(-) \cup \mathsf{return}_\alpha(-,-) \subseteq \mathcal{R}[\alpha]$ holds. Similar to the design of $\mathcal{R}[\alpha]$, we define $\mathcal{G}[\alpha]$ in such a way that the subset relation is

trivial,

$$(\Delta, s, \rho)\ \mathcal{G}[\alpha]\ (\Delta, s', \rho') \iff \left(\begin{array}{c} \mathsf{owner}(\rho') \neq \bot \wedge \mathsf{ctrval}(s') \neq \bot \quad \wedge \\ \forall \alpha'. \left(\begin{array}{c} \left(\mathsf{mytkt}_{\alpha'}(s) \neq \varnothing \implies \mathsf{ctrval}(s) \leq \mathsf{mytkt}_{\alpha'}(s)\right) \implies \\ \left(\mathsf{mytkt}_{\alpha'}(s') \neq \varnothing \implies \mathsf{ctrval}(s') \leq \mathsf{mytkt}_{\alpha'}(s')\right) \end{array}\right) \wedge \\ \mathsf{ctrval}(s') \leq \mathsf{newtkt}(s') \quad \wedge \\ \forall \alpha'. \left(\begin{array}{c} \left(\mathsf{mytkt}_{\alpha'}(s) = \mathsf{ctrval}(s) \implies \mathsf{owner}(\rho) \in \{\varnothing, \alpha'\}\right) \implies \\ \left(\mathsf{mytkt}_{\alpha'}(s') = \mathsf{ctrval}(s') \implies \mathsf{owner}(\rho') \in \{\varnothing, \alpha'\}\right) \end{array}\right) \wedge \\ \mathsf{newtkt}(s') = \mathsf{ctrval}(s') \implies \mathsf{owner}(\rho') = \varnothing \quad \wedge \\ \mathsf{owner}(\rho) \notin \{\varnothing, \alpha\} \implies \mathsf{lin}(\rho) = \mathsf{lin}(\rho') \quad \wedge \\ \mathsf{owner}(\rho) \neq \alpha \implies \mathsf{ctrval}(s) = \mathsf{ctrval}(s') \quad \wedge \\ \mathsf{owner}(\rho') \in \{\varnothing, \alpha, \mathsf{owner}(\rho)\}\} \end{array}\right)$$

Most conjuncts in $\mathcal{R}$ have direct correspondence in $\mathcal{G}$, and we will present a short argument for those doesn't. Assuming α is the rely agent and α' is the actor (guarantee) agent,

- if $\mathsf{owner}(\rho) = \alpha$ and therefore $\mathsf{owner}(\rho) \neq \alpha'$, by the second and third last conjuncts in $\mathcal{G}[\alpha']$, we know that $\mathsf{lin}(\rho) = \mathsf{lin}(\rho')$ and $\mathsf{ctrval}(s) = \mathsf{ctrval}(s')$,
- if $\mathsf{owner}(\rho) \neq \alpha$, we know from the last conjunct that $\mathsf{owner}(\rho)$ can only be $\varnothing$, α', or $\mathsf{owner}(\rho)$, none of which is α.

Even though the $\mathcal{R}[\alpha]$ and $\mathcal{G}[\alpha]$ are defined in such a way that the stability and subset relation are easy to verify, it remains to be proven that $\mathcal{G}[\alpha]$ is correct with respect to the implementation, though $\mathcal{G}[\alpha]$ is held trivially at steps that don't update s or ρ.

Now that we have all the proof obligations defined, we will prove that

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P[\alpha]^f\}\ M_{\mathsf{Lock}}[\alpha]^f\ \{Q[\alpha]^f\}$$

using the primitive rule and structure rules. The general idea is to prove that, in the case for acquire and symmetric for release, $\mathsf{reled}[\alpha]^f$ is maintained at every step, in the form of

$$\{\mathsf{reled}[\alpha](\Delta, s, \rho)\}\ B\ \{\mathsf{reled}[\alpha](\Delta', s', \rho')\}$$

before linearization. While $\mathsf{acqed}[\alpha]$ is maintained at every step after linearization in the form of

$$\{\mathsf{acqed}[\alpha](\Delta, s, \rho)\}\ B\ \{\mathsf{acqed}[\alpha](\Delta', s', \rho')\}$$

At linearization points (line 6 for acq and line 2 for rel), the precondition is transformed into corresponding postcondition while updating the possibility ρ according to the commit functions defined below,

$$\mathsf{commit}[\alpha](\rho)^{\mathsf{acq}} := \begin{cases} \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot p_1 \cdot p_2 & \rho = \mathsf{lin}(\rho) \cdot p_1 \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot p_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{G} \vdash_\alpha \{\mathsf{reled}[\alpha](\Delta, s, \rho)\}\ \mathsf{assert}(\mathsf{cur\_tick} = \mathsf{my\_tick})\ \{\mathsf{acqed}[\alpha](\Delta', s', \rho') \wedge \mathsf{commit}[\alpha]^{\mathsf{acq}}(\rho) \sqsubseteq \rho'\}$$

$$\mathsf{commit}[\alpha]^{\mathsf{rel}}(\rho) := \begin{cases} \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{rel} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot p_1 \cdot p_2 & \rho = \mathsf{lin}(\rho) \cdot p_1 \cdot \boldsymbol{\alpha}{:}\mathsf{rel} \cdot p_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{G} \vdash_\alpha \{\mathsf{acqed}[\alpha](\Delta, s, \rho)\}\ \mathsf{inc}()\ \{\mathsf{reled}(\Delta', s', \rho') \wedge \mathsf{commit}[\alpha]^{\mathsf{rel}}(\rho) \sqsubseteq \rho'\}$$

notice we can only obtain a prefix relation in the postcondition, this is due to stability requirement as other agents might changes ρ' after linearization, but at least the linearized prefix is kept the same. while the commit functions may return $\bot$, the invariant I and concurrent specification v'_{Lock} makes sure that this won't happen during execution.

Figure 8 provides a more detailed proof sketch of acq. The green component in each assertions are already complete for the reasoning, we highlight the crucial conjuncts inside the invariant in the blue component to better illustrate the reasoning. We also discuss in details the crucial steps below,

(1) on line 2, the local variable `my_tick` is updated to be the value of $\mathsf{newtkt}(s)$ while simultaneously increasing the value of $\mathsf{newtkt}(s')$ by 1. In case of $\mathsf{newtkt}(s) = \mathsf{ctrval}(s)$, the invariant in the precondition implies empty ownership of the lock maintaining itself. This operation also increment newtkt, but all guarantee conditions and invariants are justified after the update,
(2) on line 5, the local variable `cur_tick` is updated to be the value of $\mathsf{ctrval}(s)$. While the trace s will grow in the future, we have the knowledge that there exists a prefix s' of s such that $\mathsf{ctrval}(s') = \Delta(\texttt{cur\_tick})$. On the other hand, since $\mathsf{ctrval}(s)$ is non-decreasing w.r.t. s, we know a lower bound of the value for the future $\mathsf{ctrval}(s)$,
(3) on line 6, we compare the value of `my_tick` and `cur_tick`, which is equal to the current value of $\mathsf{mytkt}_\alpha(s)$ and a lower bound of the current value $\mathsf{ctrval}(s)$ respectively. If the values coincides, we can deduce that $\mathsf{mytkt}_\alpha(s) = \mathsf{ctrval}(s)$. According to the invariant in the precondition, it implies the lock is either owned by α or nobody. On the other hand, we know that α is not the owner at the beginning of the function, and it is maintained by $\mathcal{R}[\alpha]$. Therefore we know the lock is free. We then linearize the acq event at this point by updating ρ with $\mathsf{commit}[\alpha]^{\mathsf{acq}}$. $\mathcal{G}[\alpha]$ is justified at this step since the only change is $\mathsf{owner}(\rho')$ becoming α, which doesn't fit in any premises of $\mathcal{G}[\alpha]$.

Similarly, Figure 9 provides a proof sketch of rel and we highlight the crucial steps below,

(1) on line 2, we know that we currently holds the lock, and that currently $\mathsf{ctrval}(s) = \mathsf{mytkt}_\alpha(s) < \mathsf{newtkt}(s)$ from the invariant, which also implies the invariant between $\mathsf{newtkt}(s')$ and $\mathsf{ctrval}(s')$ will be maintained after incrementing the counter. Furthermore, we can linearize the rel event by updating ρ with $\mathsf{commit}[\alpha]^{\mathsf{rel}}$. $\mathcal{G}[\alpha]$ may be easily verified except for the second conjunct, whose proof would benefit from the following lemma,

$$\forall \alpha, \alpha' \in \Upsilon.\mathsf{mytkt}_\alpha(s) \neq \varnothing \wedge \mathsf{mytkt}_{\alpha'}(s) \neq \varnothing \implies \mathsf{mytkt}_\alpha(s) \neq \mathsf{mytkt}_{\alpha'}(s)$$

in other words, no two agents share the same ticket. This is provable by the underlay spec. v_{FAI}. Combined with the fact that $\mathsf{ctrval}(s) = \mathsf{mytkt}_\alpha(s)$, we know $\mathsf{ctrval}(s) \neq \mathsf{mytkt}_{\alpha'}(s)$ for all other agent α' in the system. Assuming the premise of the second conjunct, we can derive that for any other agent α' such that $\mathsf{mytkt}_{\alpha'}(s) \neq \varnothing$, it must be that $\mathsf{ctrval}(s) < \mathsf{mytkt}_{\alpha'}(s) = \mathsf{mytkt}_{\alpha'}(s')$. After the increment, we would still have $\mathsf{ctrval}(s') \leq \mathsf{mytkt}_{\alpha'}(s')$, therefore maintaining the same assertion.

$\{\mathsf{invoke}_\alpha(\mathsf{acq}) \circ P[\alpha]^{\mathsf{acq}}\}$
$\mathsf{acq}()\{$
  $\mathsf{my\_tick} \leftarrow \mathsf{fai}();$
  $\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge \Delta(\mathsf{my\_tick}) = \mathsf{mytkt}_\alpha(s)\}$
  $\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge \mathsf{mytick}[\alpha](\Delta, s)\}$
  $($
    $\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge \mathsf{curtick}(\Delta, s)\}$
    $\mathsf{assert}(\mathsf{cur\_tick} \neq \mathsf{my\_tick});$
    $\{\mathsf{reled}[\alpha](\Delta, s, \rho)\}$
    $\mathsf{yield}();$
    $\{\mathsf{reled}[\alpha](\Delta, s, \rho)\}$
    $\mathsf{cur\_tick} \leftarrow \mathsf{get}();$
    $\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge \Delta(\mathsf{cur\_tick}) = \mathsf{ctrval}(s)\}$
    $\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge \mathsf{curtick}(\Delta, s)\}$
  $)^*;$
  $\left\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge \begin{pmatrix} \mathsf{ctrval}(s) \leq \mathsf{mytkt}_\alpha(s) \wedge \\ \mathsf{mytkt}_\alpha(s) = \mathsf{ctrval}(s) \implies \mathsf{owner}(\rho) \in \{\varnothing, \alpha\} \wedge \\ \Delta(\mathsf{cur\_tick}) \leq \mathsf{ctrval}(s) \wedge \Delta(\mathsf{my\_tick}) = \mathsf{mytkt}_\alpha(s) \end{pmatrix}\right\}$
  $\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge (\Delta(\mathsf{cur\_tick}) = \Delta(\mathsf{my\_tick}) \implies \mathsf{owner}(\rho) = \varnothing \wedge \mathsf{ctrval}(s) = \mathsf{mytkt}_\alpha(s))\}$
  $\mathsf{assert}(\mathsf{cur\_tick} = \mathsf{my\_tick});$
  $\left\{\mathsf{reled}[\alpha](\Delta, s, \rho) \wedge \mathsf{commit}[\alpha]^{\mathsf{acq}}(\rho) \sqsubseteq \rho' \wedge \begin{pmatrix} \mathsf{owner}(\rho) = \varnothing \wedge \mathsf{owner}(\rho') = \alpha \wedge \\ \mathsf{ctrval}(s') = \mathsf{mytkt}_\alpha(s') \wedge (\Delta, s) = (\Delta, s') \end{pmatrix}\right\}$
  $\{\mathsf{acqed}[\alpha](\Delta', s', \rho') \wedge \mathsf{commit}[\alpha]^{\mathsf{acq}}(\rho) \sqsubseteq \rho'\}$
  $\mathsf{ret\ ok}$
$\}$
$\{\mathsf{returned}_\alpha(\mathsf{acq}) \circ Q[\alpha]^{\mathsf{acq}}\}$

Fig. 8. Proof for acq.

$\left\{\mathsf{invoke}_\alpha(\mathsf{rel}) \circ P[\alpha]^{\mathsf{rel}}\right\}$
$\mathsf{rel}()\{$
  $\{\mathsf{acqed}(\Delta, s, \rho) \wedge \mathsf{owner}(\rho) = \alpha \wedge \mathsf{mytkt}_\alpha(s) = \mathsf{ctrval}(s) < \mathsf{newtkt}(s)\}$
  $\mathsf{inc}();$
  $\left\{\mathsf{acqed}(\Delta, s, \rho) \wedge \mathsf{commit}_{\mathsf{rel}}(\rho) \sqsubseteq \rho' \wedge \begin{pmatrix} \mathsf{mytkt}_\alpha(s') < \mathsf{ctrval}(s') \leq \mathsf{newtkt}(s') \wedge \\ \mathsf{owner}(\rho) = \alpha \wedge \\ \Delta = \Delta' \wedge s\restriction_{\mathsf{FAI,Yield}} = s'\restriction_{\mathsf{FAI,Yield}} \end{pmatrix}\right\}$
  $\{\mathsf{reled}(\Delta', s', \rho') \wedge \mathsf{commit}_{\mathsf{rel}}(\rho) \sqsubseteq \rho'\}$
  $\mathsf{ret\ ok}$
$\}$
$\left\{\mathsf{returned}_\alpha(\mathsf{rel}) \circ Q[\alpha]^{\mathsf{rel}}\right\}$

Fig. 9. Proof for rel.

Gathering together all the resources we have collected so far, we have proven,

- $(\Delta_0, \epsilon, \epsilon) \in P[\alpha]^f$ for $f \in \{\mathsf{acq}, \mathsf{rel}\}$, since $\mathsf{ctrval}(\epsilon) = \mathsf{newtkt}(\epsilon) = 0$, $\mathsf{mytkt}_\alpha(\epsilon) = \varnothing$, and $\mathsf{owner}(\rho) = \varnothing$,
- $\mathsf{stable}(\mathcal{R}[\alpha], P[\alpha]^f) \wedge \mathsf{stable}(\mathcal{R}[\alpha], Q[\alpha]^f)$ for $f \in \{\mathsf{acq}, \mathsf{rel}\}$ by construction,
- $\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{\mathsf{invoke}_\alpha(f) \circ P[\alpha]^f\}\ M_{\mathsf{Lock}}[\alpha]^f\ \{\mathsf{return}_\alpha(f) \circ Q[\alpha]^f\}$ verified using the logic,

- $\forall f, f' \in F.\mathsf{return}_\alpha(f') \circ \mathsf{returned}_\alpha(f') \circ Q[\alpha]^{f'} \circ \mathsf{invoke}_\alpha(f') \circ P[\alpha]^{f'} \subseteq P[\alpha]^f$ for $f, f' \in \{\mathsf{acq}, \mathsf{rel}\}$, since the only traces that doesn't satisfy the subset relation will be rejected by the ν'_{Lock}.

With the Local Impl rule, we can derive the following judgement

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

We furthermore have $\mathcal{G}[\alpha] \cup \mathsf{invoke}_\alpha(-) \cup \mathsf{return}_\alpha(-) \subseteq \mathcal{R}[\alpha']$ for $\alpha, \alpha' \in \Upsilon$ and $\alpha \neq \alpha'$ by construction. We then can obtain the top level theorem by invoking Conc Impl rule,

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

In other words, we have proven that M_{Lock} is a linearizable lock object w.r.t. ν_{Lock} for the entire system.

G.2 Concurrent Queue

In this subsection, we present a short proof that the concurrent queue implementation is correct using the same program logic. The intuition behind the correctness is that the sequential queue is protected by the lock. Formally, we will relate the history of the sequential queue to the ownership of the lock. The set up is as follows. We have linearizable concurrent objects

$$(\nu'_{\mathsf{lock}} : \dagger\mathsf{Lock}, \nu_{\mathsf{lock}} : \dagger\mathsf{Lock}) \qquad (\nu'_{\mathsf{queue}} : \dagger\mathsf{Queue}, \nu'_{\mathsf{queue}} : \dagger\mathsf{Queue})$$

by locality we can construct the linearizable object

$$(\nu'_{\mathsf{lock}} \otimes \nu'_{\mathsf{queue}}, \nu_{\mathsf{lock}} \otimes \nu'_{\mathsf{queue}})$$

We therefore seek to show that

$$[\![M_{\mathsf{squeue}}]\!] : (\nu'_{\mathsf{lock}} \otimes \nu'_{\mathsf{queue}}, \nu_{\mathsf{lock}} \otimes \nu'_{\mathsf{queue}}) \to (\nu'_{\mathsf{squeue}}, \nu_{\mathsf{squeue}})$$

is a linearizable object implementation. Similar to verification of the lock, we will define several helper functions.

We first define a function

$$\mathsf{owner} : P_{\dagger\mathsf{Lock} \otimes \dagger\mathsf{Queue}} \to \Upsilon + \{\bot\} + \{\varnothing\}$$

to denote the ownership of the lock object, defined as follows

$$\mathsf{owner}(p) := \begin{cases} \varnothing & p \upharpoonright_{\mathsf{Lock}} = \epsilon \cdot \boldsymbol{\alpha}{:}\mathsf{acq}? \\ \alpha & p \upharpoonright_{\mathsf{Lock}} = p' \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot \boldsymbol{\alpha}{:}\mathsf{rel}? \wedge \mathsf{owner}(p') = \varnothing \\ \varnothing & p \upharpoonright_{\mathsf{Lock}} = p' \cdot \boldsymbol{\alpha}{:}\mathsf{acq} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot \boldsymbol{\alpha}{:}\mathsf{rel} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \cdot \boldsymbol{\alpha'}{:}\mathsf{acq}? \wedge \mathsf{owner}(p') = \varnothing \\ \bot & \text{otherwise} \end{cases}$$

while this owner function looks similar to the other owner function defined in the proof for the lock, there is a major differences between them: we can now assume Lock is linearized to an atomic specification, we no longer need to reason about interleaving between acquires and releases.

We also define a function $\mathsf{lin} : P_{\dagger \mathrm{Queue}} \to P_{!\mathrm{Queue}}$ to denote the longest linearized prefix of ρ,

$$\mathsf{lin}(p) = p_0 \iff p_0 \in P_{!\mathrm{Queue}} \wedge p_0 \sqsubseteq p \wedge \forall p' \in P_{!\mathrm{Queue}}.p' \sqsubseteq p \implies p' \sqsubseteq p_0$$

Finally, we define a function $\mathsf{queue} : P_{!\mathrm{Queue}} \to \mathsf{list}\ \mathbb{N} + \{\bot\} + \{\varnothing\}$, which is the functional specification for both the sequential queue and shared queue.

$$\mathsf{queue}(p) := \begin{cases} [\,] & p = \epsilon \cdot e? \\ q +\!\!+ [n] & p = p' \cdot \mathsf{enq}(n) \cdot \mathsf{ok} \cdot e? \wedge \mathsf{queue}(p') = q \\ q & p = p' \cdot \mathsf{deq} \cdot n \cdot e? \wedge \mathsf{queue}(p') = n :: q \\ [\,] & p = p' \cdot \mathsf{deq} \cdot \varnothing \cdot e? \wedge \mathsf{queue}(p') = [] \\ \bot & \text{otherwise} \end{cases}$$

We can now prove the correctness using the program logic. We start by defining the shared invariant I, rely condition $\mathcal{R}[\alpha]$, and guarantee condition $\mathcal{G}[\alpha]$,

$$I(\Delta, s, \rho) \iff \begin{pmatrix} \mathsf{owner}(s) \neq \bot \wedge \mathsf{queue}(\mathsf{lin}(\rho)) \neq \bot \quad \wedge \\ \big(\mathsf{owner}(s) = \varnothing \implies (\mathsf{lin}(\rho) = s\restriction_{\mathrm{Queue}_0})\big) \end{pmatrix}$$

$$(\Delta, s, \rho)\ \mathcal{R}[\alpha]\ (\Delta', s', \rho') \iff \left(\begin{pmatrix} \mathsf{invoke}_{A\setminus\alpha}(-) \vee \mathsf{return}_{A\setminus\alpha}(-) \vee \\ \mathsf{owner}(s') \neq \bot \wedge \mathsf{queue}(\mathsf{lin}(\rho')) \neq \bot \quad \wedge \\ \mathsf{owner}(s) = \alpha \implies \big(s\restriction_{\mathrm{Queue}_0} = s'\restriction_{\mathrm{Queue}_0} \wedge \mathsf{lin}(\rho) = \mathsf{lin}(\rho')\big) \end{pmatrix} \right)$$

$$(\Delta, s, \rho)\ \mathcal{G}[\alpha]\ (\Delta', s', \rho') \iff \begin{pmatrix} \mathsf{owner}(s') \neq \bot \wedge \mathsf{queue}(\mathsf{lin}(\rho')) \neq \bot \quad \wedge \\ \mathsf{owner}(s) \neq \alpha \implies \big(s\restriction_{\mathrm{Queue}_0} = s'\restriction_{\mathrm{Queue}_0} \wedge \mathsf{lin}(\rho) = \mathsf{lin}(\rho')\big) \end{pmatrix}$$

We can then give the same precondition and postcondition to enq and deq,

$$P[\alpha]^f(\Delta, s, \rho) \iff \mathsf{idle}_\alpha \wedge I(\Delta, s, \rho) \wedge \mathsf{owner}(s) \neq \alpha$$

$$(\Delta, s, \rho)\ Q[\alpha]^f\ (\Delta', s', \rho') \iff I(\Delta', s', \rho') \wedge \mathsf{owner}(s') \neq \alpha$$

Finally, we can define the commit functions to linearize the enq and deq events,

$$\mathsf{commit}[\alpha]^{\mathsf{enq}}(\rho) := \begin{cases} \mathsf{lin}(\rho) \cdot \mathsf{enq}(n) \cdot \mathsf{ok} \cdot p_1 \cdot p_2 & \exists p_1, p_2.\rho = \mathsf{lin}(\rho) \cdot p_1 \cdot \boldsymbol{\alpha}\mathbf{:}\mathsf{enq}(n) \cdot p_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{commit}[\alpha]^{\mathsf{deq}}(\rho, n) := \begin{cases} \mathsf{lin}(\rho) \cdot \mathsf{deq} \cdot n \cdot p_1 \cdot p_2 & \exists p_1, p_2.\rho = \mathsf{lin}(\rho) \cdot p_1 \cdot \boldsymbol{\alpha}\mathbf{:}\mathsf{deq} \cdot p_2 \\ \bot & \text{otherwise} \end{cases}$$

The proof for deq is sketched in Figure 10. The proof for enq is ommited as it's symmetric to deq. We highlight the curcial steps below,

(1) when the agent successfully acquires the lock, we know from ν_{Lock} that the pre-state of L.acq() must satisfy that $\mathsf{owner}(s) = 0$, which allows us to open the invariant and infer that $\mathsf{lin}(\rho) = s\restriction_{\mathrm{Queue}_0}$. We also know that $\mathsf{owner}(s') = \alpha$, which allows us to temporarily break the lock invariant while holding the lock,
(2) while holding the lock, we can safely access the sequential queue. This is justified by $\mathcal{R}[\alpha]$, specifically $\mathsf{owner}(s) = \alpha \implies \big(s\restriction_{\mathrm{Queue}_0} = s'\restriction_{\mathrm{Queue}_0} \wedge \mathsf{lin}(\rho) = \mathsf{lin}(\rho')\big)$,
(3) when it's time to release the lock, we linearize the lock with $\mathsf{commit}[\alpha]^{\mathsf{deq}}$. This is justified because $\mathsf{commit}[\alpha]^{\mathsf{deq}}(\rho, \Delta(\alpha)(\mathsf{r})) = \mathsf{s}\restriction_{\mathrm{Queue}_0}$, and we also know that $\mathsf{queue}(s\restriction_{\mathrm{Queue}_0}) \neq \bot$ by ν'_{queue} and the fact that $s\restriction_{\mathrm{Queue}_0} = \rho \cdot \mathsf{deq} \cdot \Delta(\alpha)(\mathsf{r}) \in P_{!\mathrm{Queue}}$,

(4) $\mathcal{G}[\alpha]$ is easily justified since the agent only modifies the sequential queue or the linearized shared queue while holding the lock.

$\left\{\mathsf{invoke}_\alpha(\mathsf{rel}) \circ P[\alpha]^{\mathsf{rel}}\right\}$
deq(){
$\{I(\Delta, s, \rho) \wedge \mathsf{owner}(s) \neq \alpha\}$
acq();
$\left\{I(\Delta, s, \rho) \wedge \mathsf{owner}(s) = \varnothing \wedge \mathsf{owner}(s') = \alpha \wedge (\Delta(\alpha), s\upharpoonright_{\mathsf{Queue}_0}, \rho) = (\Delta'(\alpha), s'\upharpoonright_{\mathsf{Queue}_0}, \rho')\right\}$
$\left\{I(\Delta, s, \rho) \wedge \mathsf{owner}(s) = \alpha \wedge \mathsf{lin}(\rho) = s\upharpoonright_{\mathsf{Queue}_0}\right\}$
r ← deq();
$\left\{I(\Delta, s, \rho) \wedge \mathsf{owner}(s) = \alpha \wedge \mathsf{lin}(\rho) \cdot \mathsf{deq} \cdot \Delta(\alpha)(\mathsf{r}) = s\upharpoonright_{\mathsf{Queue}_0}\right\}$
rel();
$\left\{I(\Delta, s, \rho) \wedge \mathsf{owner}(s') \neq \alpha \wedge \mathsf{commit}[\alpha]^{\mathsf{deq}}(\rho, \Delta(\alpha)(\mathsf{r})) \sqsubseteq \rho' \wedge \Delta(\alpha) = \Delta'(\alpha)\right\}$
ret r
}
$\left\{\mathsf{returned}_\alpha(\mathsf{rel}) \circ Q[\alpha]^{\mathsf{rel}}\right\}$

Fig. 10. Proof for deq.

H PROOF COMPENDIUM

H.1 Proof of 3.5

Proposition H.1. *Strategy composition is well-defined and associative.*

Proof. **Well-Defined** Indeed, suppose $\sigma : \mathbf{A} \multimap \mathbf{B}$ and $\tau : \mathbf{B} \multimap \mathbf{C}$. Since $\epsilon \in \sigma$ and $\epsilon \in \tau$ it follows that taking

$$\epsilon \in \mathsf{int}(\mathbf{A}, \mathbf{B}, \mathbf{C})$$

we have that

$$\epsilon\upharpoonright_{\mathbf{A},\mathbf{B}} = \epsilon \qquad \epsilon\upharpoonright_{\mathbf{B},\mathbf{C}} = \epsilon$$

And therefore

$$\epsilon\upharpoonright_{\mathbf{A},\mathbf{C}} = \epsilon \in \sigma;\tau$$

from which it follows that $\sigma;\tau$ is non-empty.
Now, suppose $s \in \sigma;\tau$ and that $p \sqsubseteq s$. Then, there exists $s' \in \mathsf{int}(\sigma, \tau)$ such that $s'\upharpoonright_{\mathbf{A},\mathbf{C}} = s$. In particular $p \sqsubseteq s'\upharpoonright_{\mathbf{A},\mathbf{C}}$. Hence, there is prefix $p' \sqsubseteq s'$ such that $p'\upharpoonright_{\mathbf{A},\mathbf{C}} = p$. Now, consider p'. Since $s'\upharpoonright_{\mathbf{A},\mathbf{B}} \in \sigma$ and σ is prefix-closed it follows that because $p'\upharpoonright_{\mathbf{A},\mathbf{B}} \sqsubseteq s'\upharpoonright_{\mathbf{A},\mathbf{B}} \in \sigma$ we have that $p'\upharpoonright_{\mathbf{A},\mathbf{B}} \in \sigma$. Similarly, $p'\upharpoonright_{\mathbf{B},\mathbf{C}} \in \tau$. Hence, it follows that $p' \in \mathsf{int}(\sigma, \tau)$ and therefore $p \in \sigma;\tau$.
Suppose $s \in \sigma;\tau$ and that $s \cdot o \in P_{\mathbf{A}\multimap\mathbf{C}}$ and o is an Opponent move. Then, there is $s' \in \mathsf{int}(\sigma, \tau)$ such that $s'\upharpoonright_{\mathbf{A},\mathbf{C}} = s$. If o is a move in $\mathbf{A}$ then note that since $s' \in \mathsf{int}(\mathbf{A}, \mathbf{B}, \mathbf{C})$ it is such that $s'\upharpoonright_{\mathbf{A},\mathbf{B}} \in \sigma \subseteq P_{\mathbf{A}\multimap\mathbf{B}}$. Now, suppose o is a move by agent $\alpha \in \Upsilon$ and consider $s_\alpha = \pi_\alpha(s'\upharpoonright_{\mathbf{A},\mathbf{B}})$. By the switching condition of the sequential game $A = (M_A, P_A)$ and the fact that $s_\alpha\upharpoonright_A \cdot o \in \iota_\alpha(P_A)$ it follows that s_α had its last move in A. But then, $s'\upharpoonright_{\mathbf{A},\mathbf{B}} \cdot o \in P_{\mathbf{A}\multimap\mathbf{B}}$ and hence, since σ is receptive and $s'\upharpoonright_{\mathbf{A},\mathbf{B}} \in \sigma$ it follows that $s'\upharpoonright_{\mathbf{A},\mathbf{B}} \cdot o \in \sigma$. Again, by switching, $\pi_\alpha(s')$ must have had its last move in A. Hence, $s' \cdot o \in \mathsf{int}(\mathbf{A}, \mathbf{B}, \mathbf{C})$ from which it follows that $s' \cdot o \in \mathsf{int}(\sigma, \tau)$ and therefore $s \cdot o \in \sigma;\tau$ as desired. The argument for o a move in $\mathbf{C}$ is dual appealing to the receptivity of τ.

Associative Indeed, suppose $\sigma : \mathbf{A} \multimap \mathbf{B}$, $\tau : \mathbf{B} \multimap \mathbf{C}$, and $\nu : \mathbf{C} \multimap \mathbf{D}$. Let $s\restriction_{\mathbf{A},\mathbf{D}} \in (\sigma;\tau);\nu$ where $s \in \mathsf{int}((\sigma;\tau),\nu)$. Then, there is $t \in \mathsf{int}(\sigma,\tau)$ such that $s\restriction_{\mathbf{A},\mathbf{C}} = t\restriction_{\mathbf{A},\mathbf{C}} \in \sigma;\tau$.
Then, because $s\restriction_{\mathbf{A},\mathbf{C}} = t\restriction_{\mathbf{A},\mathbf{C}}$ we can define v a sequence of moves such that $v\restriction_{\mathbf{A},\mathbf{C},\mathbf{D}} = s$ and $v\restriction_{\mathbf{A},\mathbf{B},\mathbf{C}} = t$. Finally, since $v\restriction_{\mathbf{B},\mathbf{C}} = t\restriction_{\mathbf{B},\mathbf{C}}$ it follows that $v\restriction_{\mathbf{B},\mathbf{C}} \in \tau$. Similarly, $v\restriction_{\mathbf{C},\mathbf{D}} = s\restriction_{\mathbf{C},\mathbf{D}}$ implies $v\restriction_{\mathbf{C},\mathbf{D}} \in \nu$. Hence, $v\restriction_{\mathbf{B},\mathbf{C},\mathbf{D}} \in \mathsf{int}(\tau,\nu)$ and $v\restriction_{\mathbf{B},\mathbf{D}} \in \tau;\nu$. Now, $v\restriction_{\mathbf{A},\mathbf{B}} = t\restriction_{\mathbf{A},\mathbf{B}} \in \sigma$, it follows then that $v\restriction_{\mathbf{A},\mathbf{B},\mathbf{D}} \in \mathsf{int}(\sigma,(\tau;\nu))$ and hence

$$s\restriction_{\mathbf{A},\mathbf{D}} = v\restriction_{\mathbf{A},\mathbf{D}} \in \sigma;(\tau;\nu)$$

The other inclusion is symmetric.

□

H.2 Order Enrichment

Lemma H.2. *Let* $\mathbf{A} = (M_A, P_A)$, $\mathbf{B} = (M_B, P_B)$ *and* $\mathbf{C} = (M_C, P_C)$ *be concurrent games and* $\sigma : \mathbf{A} \multimap \mathbf{B}$ *and* $\tau : \mathbf{B} \multimap \mathbf{C}$ *be concurrent strategies. Then, for every* $\alpha \in \Upsilon$*:*

$$\pi_\alpha(\sigma;\tau) \subseteq \pi_\alpha(\sigma);\pi_\alpha(\tau)$$

Proof. Suppose $s\restriction_{\mathbf{A},\mathbf{C}} \in \sigma;\tau$ where $s \in \mathsf{int}(\sigma,\tau)$. Then:

$$\pi_\alpha(s)\restriction_{\mathbf{A},\mathbf{B}} = \pi_\alpha(s\restriction_{\mathbf{A},\mathbf{B}}) \in \pi_\alpha(\sigma)$$

and similarly:

$$\pi_\alpha(s)\restriction_{\mathbf{B},\mathbf{C}} = \pi_\alpha(s\restriction_{\mathbf{B},\mathbf{C}}) \in \pi_\alpha(\tau)$$

and hence:

$$\pi_\alpha(s) \in \mathsf{int}(\pi_\alpha(\sigma),\pi_\alpha(\tau))$$

so that

$$\pi_\alpha(s)\restriction_{\mathbf{A},\mathbf{C}} \in \pi_\alpha(\sigma);\pi_\alpha(\tau)$$

□

Proposition H.3.

$$\mathsf{Conc} : \mathsf{Semi}\ \mathbf{Game}_{\mathbf{Seq}} \to \underline{\mathbf{Game}}_{\mathbf{Conc}}$$

defines a semifunctor.

Proof. Let $\sigma : A \multimap B$. It is straight-forward to see that $\mathsf{Conc}\ \sigma$ is well-defined. Indeed, as $\epsilon \in \sigma$ it follows that $\epsilon \in \mathsf{Conc}\ \sigma$. Now, suppose $s \in \mathsf{Conc}\ \sigma$ and $p \sqsubseteq s$. Then, p is still an interleaving of plays of σ as σ is prefix-closed. For receptivity note that if $s \in \mathsf{Conc}\ \sigma$ and $s \cdot o \in P_{\mathbf{A}\multimap\mathbf{B}}$ where o is an Opponent move then $\pi_{\alpha(o)}(s) \cdot o \in P_{A\multimap B}$ by definition and by receptivity of σ it follows that $\pi_{\alpha(o)}(s) \cdot o \in \sigma$ and therefore $s \cdot o$ is still an interleaving of σ plays. Therefore, $\mathsf{Conc}\ \sigma$ is indeed a concurrent strategy of the appropriate type.

It remains to show that $\mathsf{Conc}\ (\sigma;\tau) = \mathsf{Conc}\ \sigma;\mathsf{Conc}\ \tau$. By definition and Lemma H.2

$$\forall \alpha \in \Upsilon.\pi_\alpha(\mathsf{Conc}\ \sigma;\mathsf{Conc}\ \tau) \subseteq \pi_\alpha(\mathsf{Conc}\ \sigma);\pi_\alpha(\mathsf{Conc}\ \tau) = \sigma;\tau$$

and hence

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

Now suppose $s \in \mathsf{Conc}\ (\sigma;\tau)$. This means that:

$$\forall \alpha \in \Upsilon.\pi_\alpha(s) \in \sigma;\tau$$

In particular, for all $\alpha \in \Upsilon$ there is a a play $s_\alpha \in \mathsf{int}(\sigma,\tau)$ such that

$$s_\alpha\restriction_{A,C} = \pi_\alpha(s)$$

while

$$s_\alpha \restriction_{A,B} \in \sigma \qquad s_\alpha \restriction_{B,C} \in \tau$$

It is straight-forward to show that one can construct a sequence $s' \in \mathsf{int}(\mathsf{Conc}\ \sigma, \mathsf{Conc}\ \tau)$ by interleaving the s_α such that

$$s' \restriction_{\mathbf{A},\mathbf{C}} = s$$

□

Proposition H.4. *$\mathsf{ccopy}_{\mathbf{A}}$ is idempotent for every $\mathbf{A}$.*

Proof. By Prop. H.3

$$\mathsf{ccopy}_{\mathbf{A}}; \mathsf{ccopy}_{\mathbf{A}} = \mathsf{Conc}\ \mathsf{copy}_A; \mathsf{Conc}\ \mathsf{copy}_A = \mathsf{Conc}\ (\mathsf{copy}_A; \mathsf{copy}_A) = \mathsf{Conc}\ \mathsf{copy}_A = \mathsf{ccopy}_{\mathbf{A}}$$

□

H.3 Proposition 4.4

Proposition H.5. *For strategies*

$$\sigma : \mathbf{A} \multimap \mathbf{B} \qquad \tau : \mathbf{B} \multimap \mathbf{C}$$

the following all hold:

(1) *If*

$$\sigma \subseteq \sigma' : \mathbf{A} \multimap \mathbf{B} \qquad \tau \subseteq \tau' : \mathbf{B} \multimap \mathbf{C}$$

then

$$\sigma; \tau \subseteq \sigma'; \tau'$$

(2) *Given a family*

$$\sigma_i : \mathbf{A} \multimap \mathbf{B}$$

it holds that

$$\left(\bigcup_{i \in I} \sigma_i\right); \tau = \bigcup_{i \in I} (\sigma_i; \tau)$$

(3) *Given a family*

$$\tau_i : \mathbf{B} \multimap \mathbf{C}$$

it holds that

$$\sigma; \bigcup_{i \in I} \tau_i = \bigcup_{i \in I} (\sigma; \tau_i)$$

Proof. (1) Supose $s \restriction_{\mathbf{A},\mathbf{C}} \in \sigma; \tau$. Then:

$$s \restriction_{\mathbf{A},\mathbf{B}} \in \sigma \Rightarrow s \restriction_{\mathbf{A},\mathbf{B}} \in \sigma'$$
$$s \restriction_{\mathbf{B},\mathbf{C}} \in \tau \Rightarrow s \restriction_{\mathbf{B},\mathbf{C}} \in \tau'$$

hence

$$s \in \mathsf{int}(\sigma', \tau') \Rightarrow s \restriction_{\mathbf{A},\mathbf{C}} \in \sigma'; \tau'$$

(2) One direction is simple as we have that

$$\sigma_i \subseteq \bigcup_{i \in I} \sigma_i$$

so that

$$\sigma_i; \tau \subseteq \left(\bigcup_{i \in I} \sigma_i\right); \tau$$

by monotonicity and hence

$$\bigcup_{i\in I}(\sigma_i;\tau) \subseteq \left(\bigcup_{i\in I}\sigma\right);\tau$$

For the other direction, suppose

$$s\restriction_{\mathbf{A},\mathbf{C}} \in \left(\bigcup_{i\in I}\sigma\right);\tau$$

then

$$s\restriction_{\mathbf{A},\mathbf{B}} \in \bigcup_{i\in I}\sigma \qquad\qquad s\restriction_{\mathbf{B},\mathbf{C}} \in \tau$$

so there is $j \in I$ such that:

$$s\restriction_{\mathbf{A},\mathbf{B}} \in \sigma_j$$

and therefore

$$s\restriction_{\mathbf{A},\mathbf{C}} \in \sigma_j;\tau$$

so that

$$s\restriction_{\mathbf{A},\mathbf{B}} \in \bigcup_{i\in I}(\sigma_j;\tau)$$

(3) One direction is simple as we have that

$$\tau_i \subseteq \bigcup_{i\in I}\tau_i$$

so that

$$\sigma;\tau_i \subseteq \sigma;\bigcup_{i\in I}\tau_i$$

by monotonicity and hence

$$\bigcup_{i\in I}(\sigma;\tau_i) \subseteq \sigma;\bigcup_{i\in I}\tau_i$$

For the other direction, suppose

$$s\restriction_{\mathbf{A},\mathbf{C}} \in \sigma;\bigcup_{i\in I}\tau_i$$

then

$$s\restriction_{\mathbf{A},\mathbf{B}} \in \sigma \qquad\qquad s\restriction_{\mathbf{B},\mathbf{C}} \in \bigcup_{i\in I}\tau_i$$

so there is $j \in I$ such that:

$$s\restriction_{\mathbf{B},\mathbf{C}} \in \tau_j$$

and therefore

$$s\restriction_{\mathbf{A},\mathbf{C}} \in \sigma;\tau_j$$

so that

$$s\restriction_{\mathbf{A},\mathbf{C}} \in \bigcup_{i\in I}(\sigma;\tau_i)$$

□

H.4 Proofs for Appendix B

Proposition H.6. *Composition of atomic strategies is well-defined.*

Proof. Suppose $\sigma : !A \multimap !B$ and $\tau : !B \multimap !C$ are atomic. It follows from sequential composition that $\sigma;\tau : !A \multimap !C$. It remains to show that it is atomic. So let $s \restriction_{\mathbf{A},\mathbf{C}} \in \mathsf{int}(\sigma,\tau)$. First, let $s \restriction_{\mathbf{C}} = p_1 \cdot m \cdot m' \cdot p_2$ where $\alpha(m) = \alpha(m')$ and $\lambda_{\mathbf{C}}(m) = O$. Then, since τ is atomic, $s \restriction_{\mathbf{B},\mathbf{C}} = p_1' \cdot m \cdot s' \cdot m' \cdot p_2'$ is such that every move in s' is by $\alpha(m)$. By the same reasoning, every move between two moves in s' is by $\alpha(m)$. Hence, if $s \restriction_{\mathbf{A},\mathbf{C}} = p_1'' \cdot m \cdot s'' \cdot m' \cdot p_2''$ then every move in s'' is by $\alpha(m)$. The argument is analogous for $s \restriction_{\mathbf{C}} = p \cdot m$ with $\lambda_{\mathbf{C}}(m) = O$. □

Lemma H.7. *Let $s \in P_{\mathbf{A}\multimap\mathbf{B}}$. Then if $s \restriction_{\mathbf{B}} \in P_{!B}$ then $s \in P_{!A\multimap!B}$.*

Proof. We argue by induction by keeping track of a prefix $p \sqsubseteq_{\mathsf{even}} s$ such that $p \in P_{!A\multimap!B}$ and such that every agents last move was in $\mathbf{B}$. For the base case we note that if $s = \epsilon$ then we are done (we also consider this case as a case where every agents last move is $\mathbf{B}$). Otherwise, let $p \sqsubseteq_{\mathsf{even}} s$ be such that $p \in P_{!A\multimap!B}$. If $p = s$ we are also done. Otherwise, there is m a move in $\mathbf{B}$ (by the switching condition and the inductive hypothesis) such that $p \cdot m \sqsubseteq s$. Suppose first that

$$p \cdot m \cdot s' = s$$

is such that every move in s' happens in $\mathbf{A}$. As at p every agent had its last move in $\mathbf{B}$ at $p \cdot m$ only $\alpha(m)$ can move in $\mathbf{A}$, and since $\alpha(m)$ plays as in $A \multimap B$ it follows then that $s = p \cdot m \cdot s' \in P_{!A\multimap!B}$ as desired. Otherwise, there are s' a sequence of moves in $\mathbf{A}$ and a move m' in $\mathbf{B}$ such that

$$p \cdot m \cdot s' \cdot m' \sqsubseteq s$$

By alternation in $\mathbf{B}$ it follows that m' is a move by $\alpha(m)$. By the same reasoning as above, it follows that s' only involves moves by $\alpha(m)$ and since every agent behaves sequentially it follows that $p \cdot m \cdot s' \cdot m' \in P_{!A\multimap!B}$. It is easy to check that all the other invariants still hold. □

H.5 Proofs of B.2

Proposition H.8. *If $s, t \in P_{\mathbf{A}}$ then $s \equiv_{\mathbf{A}} t$ if and only if s and t are compatible and $\prec_s = \prec_t$.*

Proof. ($\Rightarrow$) Since all the swaps allowed by $\rightsquigarrow_{\mathbf{A}}$ are between agents, it immediately follows that s and t are compatible. Moreover, no swap $OO \rightsquigarrow OO$ or $PP \rightsquigarrow PP$ swap modifies the happens before order as the happens before order is defined by comparing the position of a P move with the position of an O move.

($\Leftarrow$) For the reverse direction, suppose s and t are compatible but $s \not\equiv_{\mathbf{A}} t$. Then, there must be moves m, and Opponent move, and n a Proponent move such that

$$s = s_1 \cdot m \cdot s_2 \cdot n \cdot s_3$$

but

$$t = t_1 \cdot n \cdot t_2 \cdot m \cdot t_3$$

or

$$t = t_1 \cdot m \cdot t_2 \cdot n \cdot t_3$$

and

$$s = s_1 \cdot n \cdot s_2 \cdot m \cdot s_3$$

Without loss of generality, we assume the first situation (otherwise, reverse the roles of s and t). Let o be the operation corresponding to m and o' the operation corresponding to n. Then,

$$o' \prec_t o$$

by definition. Meanwhile, in s either e and e' are not comparable, or $o \prec_s o'$, which contradicts that $\prec_t = \prec_s$. □

Proposition H.9. *For plays $s, t \in P_{\mathrm{A}}$, there is a derivation*

$$s \rightsquigarrow_{\mathrm{A}} t$$

if and only if s is compatible with t and

$$\prec_{s'} \subseteq \prec_t$$

Proof. ($\Rightarrow$) Note that if

$$s \rightsquigarrow^{1}_{\mathrm{A}} t$$

then either $\prec_s = \prec_t$ by Prop. B.8 or the derivation is a $OP \rightsquigarrow PO$ swap. We argue that

$$\prec_s \subseteq \prec_t$$

in that case. Indeed, suppose

$$s = s_1 \cdot m \cdot n \cdot s_2 \rightsquigarrow^{1}_{\mathrm{A}} s_1 \cdot n \cdot m \cdot s_2 = t$$

Let o be the operation associated to m and o' the operation associated to n. Note first that for any o_1, o_2 where at least one of o_1 and o_2 are distinct from o and o' it is the case that

$$o_1 \prec_s o_2 \iff \rho(o_1) \prec_t \rho(o_2)$$

where ρ is the associated bijection. Indeed, if they are both distinct from o and o' then in fact

$$\rho(o_1) = o_1 \qquad \rho(o_2) = o_2$$

and the equivalence holds. Otherwise, consider the four possible cases:

$o_1 = o$ Then, we have that

$$o_1 = (p, q) \qquad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p + 1, q) \qquad \rho(o_2) = (p', q')$$

hence

$$o_1 \prec_s o_2 \iff q < p' \iff \rho(o_1) \prec_t \rho(o_2)$$

$o_1 = o'$ Then, we have that

$$o_1 = (p, q) \qquad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p, q - 1) \qquad \rho(o_2) = (p', q')$$

hence

$$o_1 \prec_s o_2 \iff q < p' \iff q - 1 < q < p' \iff \rho(o_1) \prec_t \rho(o_2)$$

where the middle equivalence holds because o_2 is not o.

$o_2 = o$ Then, we have that

$$o_1 = (p, q) \qquad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p, q) \qquad \rho(o_2) = (p' + 1, q')$$

hence

$$o_1 \prec_s o_2 \iff q < p' \iff q < p' < p' + 1 \iff \rho(o_1) \prec_t \rho(o_2)$$

where the second equivalence holds because o_1 is not o'.

$o_2 = o'$ Then, we have that

$$o_1 = (p, q) \qquad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p, q) \qquad \rho(o_2) = (p', q' - 1)$$

hence

$$o_1 \prec_s o_2 \iff q < p' \iff \rho(o_1) \prec_t \rho(o_2)$$

Finally, note that o and o' are not comparable in $\prec_s$. Meanwhile, in $\prec_t$ we have $o' \prec_t o$.

($\Leftarrow$) By Prop. B.8, if $\prec_s = \prec_t$ we are done, so suppose $\prec_s \neq \prec_t$. We construct a play s' such that $\prec_s \subset \prec_{s'} \subseteq \prec_t$ and $s \rightsquigarrow_{\mathbf{A}} s'$. Because $\prec_s$ is strictly contained in $\prec_t$ there is a pair $o \prec_t o'$ but o and o' are incomparable in s. Hence, if

$$o = (p, q) \qquad o' = (p', q')$$

in s, we may choose the pair of o and o' incomparable in s such that $q - p'$ is minimal. Let m be the O move associated to o' and n the P move associated to o. Then:

$$s = s_1 \cdot m \cdot s_2 \cdot n \cdot s_3$$

Note that by minimality, s_2 decomposes as

$$s_2 = s_O \cdot s_P$$

where s_P is a sequence of P moves and s_O a sequence of O moves. Indeed, otherwise we have:

$$s_2 = s_1' \cdot n' \cdot s_2' \cdot m' \cdot s_3'$$

where n' is a P move and m' an O move. Let o_1 be the operation associated to m' and o_2 the operation associated to n'. Then note that

$$s = s_1 \cdot m \cdot s_1' \cdot n' \cdot s_2' \cdot m' \cdot s_3' \cdot n \cdot s_3$$

Note that if

$$o_1 = (p_1, q_1) \qquad o_2 = (p_2, q_2)$$

then,

$$p' < q_2 < p_1 < q$$

Note then that

$$q - p_1, q_2 - p' < q - p'$$

So as long as either the pair o, o_1 is incomparable or o_2, o' is incomparable then it breaks minimality. Hence,

$$q_1 < p \qquad \text{and} \qquad q' < p_2$$

But then

$$q' < p_2 < q_2 < p_1 < q_1 < p$$

and therefore

$$o' \prec_s o$$

a contradiction. Hence, it must be that

$$s = s_1 \cdot m \cdot s_O \cdot s_P \cdot n \cdot s_3$$

and therefore:

$$s = s_1 \cdot m \cdot s_O \cdot s_P \cdot n \cdot s_3 \equiv_{\mathbf{A}} s_1 \cdot s_O \cdot m \cdot s_P \cdot n \cdot s_3 \equiv_{\mathbf{A}} s_1 \cdot s_O \cdot m \cdot n \cdot s_P \cdot s_3 \rightsquigarrow_{\mathbf{A}} \cdot s_O \cdot n \cdot m \cdot s_P \cdot s_3$$

So we let

$$s' = s_O \cdot n \cdot m \cdot s_P \cdot s_3$$

By the argument from the forward direction we have that

$$\prec_s \subset \prec_{s'}$$

Moreover, by our choice of o and o'

$$\prec_{s'} \subseteq \prec_t$$

We may continue this procedure until $s' = t$, which must happen as there are finitely many partial orders over the finite set $\mathrm{op}(s)$. □

The following couple of lemmas are straight-forward.

Lemma H.10. *If*

$$s \cdot m \cdot t \rightsquigarrow_{\mathbf{A}} s' \cdot m$$

then

$$s \cdot t \rightsquigarrow_{\mathbf{A}} s'$$

Lemma H.11. *Let $s, s' \in P_{\mathbf{A}}$, s_P a sequence of Proponent moves and s_O a sequence of Opponent moves. If*

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

then let $(s \setminus s_O) \in P_{\mathbf{A}}$ be the subsequence of s obtained by removing the pending Opponent moves that appear in s_O, then

$$s \cdot s_P \rightsquigarrow_{\mathbf{A}} (s \setminus s_O) \cdot s_P \cdot s_O \rightsquigarrow_{\mathbf{A}} s' \cdot s_O$$

Proposition H.12. *A play $s \in P_{\mathbf{A}}$ is linearizable to an atomic play $t \in P_{!A}$ if and only if s is Herlihy-Wing linearizable to t.*

Proof. ($\Rightarrow$) By assumption there is a sequence of Opponent moves s_O and a sequence of Proponent moves s_P such that

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

If there are no pending O moves in t then, s_O contains all pending moves in $s \cdot s_P$ so that by Lemma H.11

$$s \cdot s_P \rightsquigarrow_{\mathbf{A}} \mathsf{complete}(s \cdot s_P) \cdot s_O \rightsquigarrow t \cdot s_O$$

and then by Lemma H.10 we have that

$$\mathsf{complete}(s \cdot s_P) \rightsquigarrow t$$

so that by Prop. B.9 the result follows. Now, suppose there is a pending Opponent move o in t. Then, o must be the last move of t. Indeed, suppose otherwise. Then, $t = u \cdot o \cdot v$ for non-empty v. Since o is pending, no move in v is by the same agent as that of o. But since t is sequential, the first move of v must be a Proponent move by the same agent as o, a contradiction. Hence, $t = t' \cdot o$ for some pending Opponent move o. We argue that $\mathsf{complete}(s \cdot s_P) \rightsquigarrow t'$. By Lemma H.11 we have that there is s' such that

$$s \cdot s_P \rightsquigarrow_{\mathbf{A}} s' \cdot s_P \cdot s_O \rightsquigarrow_{\mathbf{A}} t \cdot s_O$$

but then, by the reasoning above, there is at most one pending Opponent move in s' so that

$$s' \cdot s_O \cdot s_P \rightsquigarrow_{\mathbf{A}} s' \cdot s_P \cdot s_O \rightsquigarrow_{\mathbf{A}} t \cdot s_O = t' \cdot (o \cdot s_O)$$

implies by H.11 that there is $s'' \in P_{\mathbf{A}}$ such that

$$s' \cdot s_O \cdot s_P \rightsquigarrow_{\mathbf{A}} s'' \cdot s_P \cdot (o \cdot s_O) \rightsquigarrow_{\mathbf{A}} t' \cdot (o \cdot s_O)$$

But, s'' is s' with o removed, and s' is s with all moves in s_O removed. Moreover, s'' has no pending Opponent moves, as t' does not. Therefore, $s'' \cdot s_P = \mathsf{complete}(s \cdot s_P)$. By the previous reasoning, the result follows.

($\Leftarrow$) By Proposition B.9 it follows that there is a reduction $\mathsf{complete}(s \cdot s_P) \rightsquigarrow_{\mathbf{A}} t$. Now, let s_O be a sequence containing all the Opponent moves removed by $\mathsf{complete}(-)$. Note that there is at most one move per agent in s_O, and, moreover, that any agent that appears in s_O does not appear in s_P. Then:

$$s \cdot s_P \rightsquigarrow_{\mathbf{A}} \mathsf{complete}(s \cdot s_P) \cdot s_O \rightsquigarrow_{\mathbf{A}} t \cdot s_O$$

proving that s is linearizable to t. □

H.6 K_{Conc} is an oplax semifunctor

Proposition H.13. *For any $\sigma : \mathbf{A} \multimap \mathbf{B}$ and $\tau : \mathbf{B} \multimap \mathbf{C}$:*

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

Proof. The argument is quite simple, we just verify the following sequence of equalities and inclusions taking note of the use of Lemma H.19, Proposition 4.4, Proposition 4.2 and associativity of interaction:

$$\begin{aligned} K_{\mathsf{Conc}}(\sigma;\tau) &= \mathsf{ccopy}_{\mathbf{A}}; \sigma; \tau; \mathsf{ccopy}_{\mathbf{C}} \\ &\subseteq \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}; \tau; \mathsf{ccopy}_{\mathbf{C}} \\ &= \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}; \mathsf{ccopy}_{\mathbf{B}}; \tau; \mathsf{ccopy}_{\mathbf{C}} \\ &= K_{\mathsf{Conc}}(\sigma); K_{\mathsf{Conc}}(\tau) \end{aligned}$$

□

Proposition H.14. *K_{Conc} is monotonic and join-preserving.*

Proof. Suppose $\sigma \subseteq \sigma'$. Then:

$$K_{\mathsf{Conc}}\ \sigma = \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}} \subseteq \mathsf{ccopy}_{\mathbf{A}}; \sigma'; \mathsf{ccopy}_{\mathbf{B}} = K_{\mathsf{Conc}}\ \sigma'$$

by Proposition 4.4.

Similarly, if we have a collection

$$\{\sigma_i\}_{i \in I}$$

we have

$$K_{\mathsf{Conc}} \left(\bigcup_{i \in I} \sigma_i \right) = \mathsf{ccopy}_{\mathbf{A}}; \left(\bigcup_{i \in I} \sigma_i \right); \mathsf{ccopy}_{\mathbf{B}} = \bigcup_{i \in I} \mathsf{ccopy}_{\mathbf{A}}; \sigma_i; \mathsf{ccopy}_{\mathbf{B}} = \bigcup_{i \in I} K_{\mathsf{Conc}}\ \sigma_i$$

by Proposition 4.4. □

Corollary H.15.

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

defines an oplax semifunctor.

Lemma H.16. *If*

$$e_- = \{e_A\}_{A \in S} \qquad e'_- = \{e'_A\}_{A \in S}$$

are families of idempotents such that there are 2-morphisms:

$$e_{\mathbf{A}} \Rightarrow e'_{\mathbf{A}}$$

for every $A \in S$, then the mappings L and R defined by

$$L : \tilde{\mathbf{C}}_e \to \tilde{\mathbf{C}}_{e'} := K' \circ \mathsf{Emb} \qquad R : \tilde{\mathbf{C}}_{e'} \to \tilde{\mathbf{C}}_e := K \circ \mathsf{Emb}'$$

define an oplax functor and a lax functor respectively.

H.7 Proofs for §4.5

Proposition H.17 (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 $\sigma = \mathsf{strat}\,(p \cdot m \cdot m' \cdot p')$. Then,*

$$p \cdot m' \cdot m \cdot p' \in \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}} \iff m' \cdot m \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} m \cdot m'$$

Proof. We need to consider all possibilities for the polarity and components of the moves m and m', between O and P and A or B respectively. A lot of cases are very similar to each other, so we will reference previous cases when that happens.

Just so we take another component of variation out of the way, note that if $\alpha(m) = \alpha(m')$ then it is immediate that $p \cdot m' \cdot m \cdot p'$ cannot be in $\mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$ as $p \cdot m' \cdot m \cdot p'$ is not locally alternating and hence is not in $P_{\mathbf{A} \multimap \mathbf{B}}$. On the other hand, in that case no rewrite rule applies, as all of them assume the agents are distinct. Therefore, assume in the remaining cases that $\alpha(m) \neq \alpha(m')$.

The key idea is to consider how the copying is happening in $\mathsf{ccopy}_{\mathbf{A}} : \mathbf{A}_0 \multimap \mathbf{A}_1$ and $\mathsf{ccopy}_{\mathbf{B}} : \mathbf{B}_0 \multimap \mathbf{B}_1$. If m is a move in A then it appears in a play of $\mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$ as a result of the projection to A_0. It has a corresponding copy in $\mathbf{A}_1$ which is the move that actually appeared in some play of σ. The key point is that the fact that $\alpha(m)$ is locally alternating and running the sequential copycat strategy means that if $\lambda_{\mathbf{A}}(m) = O$ then its copy appeared *earlier* in $\mathbf{A}_1$, while if it was a P move then its copy will appear *later* in $\mathbf{A}_1$. Meanwhile, if m is a move in B then it appears as a result of the projection to $\mathbf{B}_1$. Hence, if $\lambda_{\mathbf{B}}(m) = O$ its copy will appear *later* in $\mathbf{B}_0$, while if it is a P move then its copy has already appeared *earlier* in $\mathbf{B}_0$.

$m, m' \in M_B$

$\lambda_{\mathbf{A} \multimap \mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A} \multimap \mathbf{B}}(m') = O$ Note that in this case we have that

$$\lambda_{\mathbf{B}_0 \multimap \mathbf{B}_1}(m) = \lambda_{\mathbf{B}_0 \multimap \mathbf{B}_1}(m') = P$$

so that by the reasoning above their copy appeared earlier in $\mathbf{B}_0$ as O moves. Since $\mathsf{ccopy}_{\mathbf{B}}$ allows for both orderings. In particular,

$$p \cdot m' \cdot m \cdot p' \in \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$$

$\lambda_{\mathbf{A} \multimap \mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A} \multimap \mathbf{B}}(m') = P$ The reasoning here is analogous to the previous case, except that in this case the corresponding moves appear later in $\mathsf{ccopy}_{\mathbf{B}}$ but both orderings are still allowed.

$\lambda_{\mathbf{A} \multimap \mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A} \multimap \mathbf{B}}(m') = P$ In this case $\lambda_{\mathbf{B}_0 \multimap \mathbf{B}_1}(m) = P$ and $\lambda_{\mathbf{B}_0 \multimap \mathbf{B}_1}(m') = O$. Hence, m is the copy of an earlier move in $\mathsf{ccopy}_{\mathbf{B}}$ and m' is copied later in $\mathsf{ccopy}_{\mathbf{B}}$. But m already occurs before m' so that so will their copies in $\mathbf{B}_1$. Hence, the only order possible is m before m', giving the only negative case. But notice that it does not hold that $m' \cdot m \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} m \cdot m'$ in this case either.

$\lambda_{\mathbf{A} \multimap \mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A} \multimap \mathbf{B}}(m') = O$ In this case, $\lambda_{\mathbf{B}_0 \multimap \mathbf{B}_1}(m) = O$ and $\lambda_{\mathbf{B}_0 \multimap \mathbf{B}_1}(m') = P$. So that the copy of m in $\mathbf{B}_1$ appears later while the copy of m' appears earlier. In particular, there is a play of $\mathsf{ccopy}_{\mathbf{B}}$ where the copy of m' appears earlier then the copy of m and therefore

$$p \cdot m' \cdot m \cdot p' \in \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$$

$m, m' \in M_A$

$\lambda_{\mathbf{A} \multimap \mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A} \multimap \mathbf{B}}(m') = O$ Similarly to before, the polarities are dualized once we consider the move within the game $\mathbf{A}_0 \multimap \mathbf{A}_1$ so that:

$$\lambda_{\mathbf{A}_0 \multimap \mathbf{A}_1}(m) = \lambda_{\mathbf{A}_0 \multimap \mathbf{A}_1}(m') = P$$

and their respective copies in $\mathbf{A}_1$ therefore appear earlier in the $\mathsf{ccopy}_{\mathbf{A}}$ play. Other than that, $\mathsf{ccopy}_{\mathbf{A}}$ does not prescribe any particular ordering between them, so both are allowed.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = P$ As before the polarities switch so that:

$$\lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m) = \lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m') = O$$

and hence their copies in $\mathbf{A}_0$ appear later with no particular order required.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = P$ In this case:

$$\lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m) = P \qquad \text{and} \qquad \lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m') = O$$

in this case the copy of m in $\mathbf{A}_0$ appears earlier in $\text{ccopy}_{\mathbf{A}}$ while the copy of m' appears later. Hence their order cannot be changed, and this is precisely the only case in this group where it does not hold that $m' \cdot m \rightsquigarrow_{\mathbf{A}\multimap\mathbf{B}} m \cdot m'$.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = O$ In this case we have:

$$\lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m) = O \qquad \text{and} \qquad \lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m') = P$$

So that the copy of m in $\mathbf{A}_0$ appears later than m in ccopy_A while the copy of m' appears earlier. In particular, both orders are allowed.

$m \in M_B$ **and** $m' \in M_A$

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = O$ In this case

$$\lambda_{\mathbf{B}_0\multimap\mathbf{B}_1}(m) = P \qquad \text{and} \qquad \lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m') = P$$

but as m occurs in $\mathbf{B}_0$ while m' occurs in $\mathbf{A}_1$ the copy of m in $\mathbf{B}_1$ so that both copies appear earlier in the respective plays of $\text{ccopy}_{\mathbf{B}}$ and $\text{ccopy}_{\mathbf{A}}$ so that both orderings are possible.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = P$ The situation in this case is analogous to the previous case except that the copies of m and m' appear later.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = P$ In this case we are in a situation where

$$\lambda_{\mathbf{B}_0\multimap\mathbf{B}_1}(m) = P \qquad \text{and} \qquad \lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m') = O$$

so that m's copy appears earlier while m''s copy appears later. Hence, the ordering must still be m preceded by m' in $\text{ccopy}_{\mathbf{A}}; \sigma; \text{ccopy}_{\mathbf{B}}$ so that

$$p \cdot m' \cdot m \cdot p' \notin \text{ccopy}_{\mathbf{A}}; \sigma; \text{ccopy}_{\mathbf{B}}$$

but this is the only case where it does not hold $m' \cdot m \rightsquigarrow_{\mathbf{A}\multimap\mathbf{B}} m \cdot m'$.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = O$ In this case: that means that the copy of m in $\mathbf{B}_0$ appears later in $\text{ccopy}_{\mathbf{B}}$ as well as the copy of m' in $\mathbf{A}_1$, and therefore no order is imposed on them.

$m \in M_A$ **and** $m' \in M_B$

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = O$ We have the polarities:

$$\lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m) = P \qquad \text{and} \qquad \lambda_{\mathbf{B}_0\multimap\mathbf{B}_1}(m') = P$$

so that the copy of m in $\mathbf{A}_0$ happens earlier as does the copy of m' in $\mathbf{B}_1$. No order is required between them and therefore both orderings is possible.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = P$ This case works as before, except that the corresponding copies into $\mathbf{A}_0$ and $\mathbf{B}_1$ happen later, but still no particular order is required.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = O$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = P$ We have the polarities:

$$\lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m) = P \qquad \text{and} \qquad \lambda_{\mathbf{B}_0\multimap\mathbf{B}_1}(m') = O$$

which means that the copy of m in $\mathbf{A}_0$ happens earlier while the copy of m' in $\mathbf{B}_1$ happens later. Hence, the only possible order allowed is for m to precede m'. But this is the only negative case, where $m' \cdot m \rightsquigarrow_{\mathbf{A}\multimap\mathbf{B}} m \cdot m'$ does not hold.

$\lambda_{\mathbf{A}\multimap\mathbf{B}}(m) = P$ **and** $\lambda_{\mathbf{A}\multimap\mathbf{B}}(m') = O$ In this case:

$$\lambda_{\mathbf{A}_0\multimap\mathbf{A}_1}(m) = O \qquad \text{and} \qquad \lambda_{\mathbf{B}_0\multimap\mathbf{B}_1}(m') = P$$

so that m's copy in $\mathbf{A}_0$ happens later while m''s copy in $\mathbf{B}_1$ happens earlier. No order is required between them.

□

Corollary H.18. *Let $s \in P_{\mathbf{A}\multimap\mathbf{B}}$ and that t is a play such that*

$$\forall \alpha \in \Upsilon.\pi_\alpha(t) = \pi_\alpha(s)$$

and moreover

$$t \in \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}}$$

then,

$$t \rightsquigarrow_{\mathbf{A}\multimap\mathbf{B}} s$$

Proof. Note that as s is the only play of strat (s) satisfying the sequential consistency condition on t. By the Synchronization Lemma (Prop. 4.8) it follows that any play that can be obtained by a single move swap from s is in $\mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}}$ if and only if that swap is allowed by $\rightsquigarrow_{\mathbf{A}\multimap\mathbf{B}}$. So let

$$\sigma_0 = \mathsf{strat}\,(s)$$

and

$$\sigma_i = \{t' \in P_{\mathbf{A}\multimap\mathbf{B}} \mid \exists s' \in \sigma_i . t' \rightsquigarrow^1_{\mathbf{A}\multimap\mathbf{B}} s' \vee t' = s'\}$$

Then, note that by the Synchronization Lemma (Prop. 4.8) $t' \in \sigma_i$ if and only if there is a derivation of length at most i such that

$$t' \rightsquigarrow^i_{\mathbf{A}\multimap\mathbf{B}} s'$$

where $s' \in \sigma_0$. Note moreover that if t' is sequentially consistent with s then

$$t' \rightsquigarrow^i_{\mathbf{A}\multimap\mathbf{B}} s$$

Now, we argue that there exists k such that

$$\sigma_{k+1} = \sigma_k$$

Indeed, it is easy to observe that

$$\sigma_i \subseteq \sigma_{i+1}$$

As strategies form a complete partial order it follows that there is σ' such that

$$\sigma' = \cup_{i\in\mathbb{N}}\sigma_i$$

but note that there are finitely many plays t' such that

$$\exists s' \in \mathsf{strat}\,(s)\,. t' \rightsquigarrow_{\mathbf{A}\multimap\mathbf{B}} s'$$

as there are finitely many permutations for any play in strat (s). Therefore, there must be a k such that

$$\sigma_k = \sigma'$$

but note that, by the Synchronization Lemma (Prop. 4.8), $\mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}}$ is a fixed point of the chain and therefore,

$$\sigma' = \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}}$$

from which the result follows. □

Lemma H.19. *For every strategy $\sigma : \mathbf{A} \multimap \mathbf{B}$:*

$$\sigma \subseteq \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$$

Proof. Suppose $s \in \sigma$. We inductively construct a play of $t \in \mathsf{int}(\mathbf{A}_0, \mathbf{A}_1, \mathbf{B}_0, \mathbf{B}_1)$ such that

$$t \upharpoonright_{\mathbf{A}_0,\mathbf{A}_1} \in \mathsf{ccopy}_{\mathbf{A}} \qquad t \upharpoonright_{\mathbf{A},\mathbf{B}} \in \sigma \qquad t \upharpoonright_{\mathbf{B}_0,\mathbf{B}_1} \in \mathsf{ccopy}_{\mathbf{B}} \qquad t \upharpoonright_{\mathbf{A}_0,\mathbf{B}_1} = s$$

Indeed, if $s = \epsilon$ we simply take $t = \epsilon$. Otherwise, let t be the current play satisfying the invariants above with the last one modified to

$$t \upharpoonright_{\mathbf{A}_0,\mathbf{B}_1} \sqsubseteq s$$

If $t = s$ we are done. Otherwise there is a move m such that

$$t \upharpoonright_{\mathbf{A}_0,\mathbf{B}_1} \cdot m \sqsubseteq s$$

Suppose m is a move in $\mathbf{A}$ in s. If it is an O move we simply append to t a copy of m in $\mathbf{A}_0$ and m as a move in $\mathbf{A}_1$ as in that case the last move by $\alpha(m)$ in t was a P move in component $\mathbf{A}_0$. If it is a P move then the last move by $\alpha(m)$ was in $\mathbf{B}_0$. In that case we append the move m in $\mathbf{A}_1$ and its copy in $\mathbf{A}_0$.

Otherwise, m is a move in $\mathbf{B}$ in s. In that case if it is an O move we add a $\mathbf{B}_1$ copy to it and the move m in $\mathbf{B}_0$. If it is a P move then we add the move m in $\mathbf{B}_1$ and a copy in $\mathbf{B}_0$.

It is straight-forward to check that this builds a play with all the desired conditions. □

Lemma H.20. *For every strategy $\sigma : \mathbf{A}$ it holds that:*

$$\sigma = \bigcup_{s \in \sigma} \mathsf{strat}\,(s)$$

Proof. Since $\mathsf{strat}\,(s)$ contains $\{s\}$ by definition it follows that if $s \in \sigma$ then $s \in \mathsf{strat}\,(s)$ and hence

$$s \in \bigcup_{s' \in \sigma} \mathsf{strat}\,(s')$$

proving one containment.

For the other direction if

$$s \in \bigcup_{s' \in \sigma} \mathsf{strat}\,(s')$$

then either s is in $\mathsf{strat}\,(t)$ for some $t \in \sigma$. But then, either $s \sqsubseteq t$ or s is obtained from some prefix $p \sqsubseteq t$ by appending Opponent moves. In the first case $s \in \sigma$ because σ is prefix-closed, and in the later we simply apply prefix-closure and receptivity of σ to obtain that $s \in \sigma$. □

Proposition H.21. *A strategy $\sigma : \mathbf{A} \multimap \mathbf{B}$ is saturated if and only if*

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

Proof. Suppose σ is saturated. It follows that if $s \in \sigma = \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$ and $t \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$ then there is a sequence of single steps:

$$t = t_0 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} \ldots \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_n = s$$

then by applying the Sychronization Lemma (Prop. 4.8) starting with

$$t_{n-1} \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$$

to conclude that

$$t_{n-1} \in \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}} \subseteq \sigma$$

in a finite number of applications we obtain that

$$t = t_0 \in \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(t_1)\,; \mathsf{ccopy}_{\mathbf{A}} \subseteq \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}} \subseteq \sigma$$

as desired.

Note that for every strategy $\sigma : \mathbf{A} \multimap \mathbf{B}$ it holds that:

$$\sigma = \bigcup_{s \in \sigma} \mathsf{strat}\,(s)$$

by lemma H.20.

But

$$\mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}} = \bigcup_{s \in \sigma} \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}}$$

by the fact that composition is join-preserving. Hence,

$$t \in \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}} \iff \exists s \in \sigma. t \in \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}}$$

moreover, by the definition of $\mathsf{ccopy}\_$, s can be chosen so that

$$\forall \alpha \in \Upsilon. \pi_\alpha(t) = \pi_\alpha(s)$$

by corollary to the Synchronization Lemma (Prop. 4.8) it follows that

$$t \in \mathsf{ccopy}_{\mathbf{A}}; \mathsf{strat}\,(s)\,; \mathsf{ccopy}_{\mathbf{B}} \iff t \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$$

And hence

$$t \in \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}} \iff \exists s \in \sigma. t \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$$

So, suppose $t \in \mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}}$. Then, there is some $s \in \sigma$ such that $t \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$ and hence by assumption $t \in \sigma$. Hence,

$$\mathsf{ccopy}_{\mathbf{A}}; \sigma; \mathsf{ccopy}_{\mathbf{B}} \subseteq \sigma$$

the reverse containment is exactly lemma H.19 so that it follows that

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

and hence σ is saturated. □

H.8 Computational Interpretation Proof

Lemma H.22. *Let $s \in P_{\mathbf{A}}$. Then, there exists t an alternating play and s_O a sequence of Opponent moves such that*

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

Proof. We prove the result by induction. If $s = \epsilon$ we let $t = s_O = \epsilon$ and the result follows. Otherwise, let

$$s = p \cdot m$$

by induction there is an alternating play p' and sequence of Opponent moves p_O such that

$$p \rightsquigarrow_{\mathbf{A}} p' \cdot p_O$$

Hence,

$$s = p \cdot m \rightsquigarrow_{\mathbf{A}} p' \cdot p_O \cdot m$$

Note that without loss of generality we may assume that the last move in p' is a Proponent move, as otherwise we can add that last O move to p_O without harm. We now split into cases depending on whether m is an Opponent or Proponent move. If m is an Opponent move. then we let $s_O = p_O \cdot m$ and $t = p'$ and the result follows immediately. Otherwise, m is a Proponent move. By local sequentiality, ti follows that the last move by $\alpha(m)$ is an Opponent move, and moreover, as p' is alternating and its last move is a P move it follows that the last O move m' by $\alpha(m)$ is in p_O. So we let $t = p' \cdot m' \cdot m$, and if $p_O = p_1 \cdot m' \cdot p_2$ then we let $s_O = p_1 \cdot p_2$, that is, the subsequence of p_O obtained by removing

the move m'. Note that there is a single move by $\alpha(m)$ in p_O because of local sequentiality. This, together with the inductive hypothesis justifies the following derivation.

$$s = p \cdot m \rightsquigarrow_{\mathbf{A}} p' \cdot p_O \cdot m = p' \cdot p_1 \cdot m' \cdot p_2 \cdot m \rightsquigarrow_{\mathbf{A}} p' \cdot m' \cdot p_1 \cdot p_2 \cdot m \rightsquigarrow_{\mathbf{A}} p' \cdot m' \cdot m \cdot p_1 \cdot p_2 = t \cdot s_O$$

□

Lemma H.23. *Let $s \in P_{\mathbf{A} \multimap \mathbf{B}}$. Then, for any $s'_A \in P_{\mathbf{A}}$ and $s'_B \in P_{\mathbf{B}}$ such that*

$$s'_B \rightsquigarrow_{\mathbf{B}} s \upharpoonright_{\mathbf{B}} \qquad\qquad s \upharpoonright_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} s'_A$$

then there exists an

$$s' \in P_{\mathbf{A} \multimap \mathbf{B}}$$

such that

$$s' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s \qquad\qquad s' \upharpoonright_{\mathbf{A}} = s'_A \qquad\qquad s' \upharpoonright_{\mathbf{B}} = s'_B$$

Proof. We let $s_A = s \upharpoonright_{\mathbf{A}}$ and $s_B = s \upharpoonright_{\mathbf{B}}$.

Suppose first that $s_A \rightsquigarrow_{\mathbf{A}} s'_A$. We construct by induction on the length of the derivation $s_A \rightsquigarrow_{\mathbf{A}} s'_A$ an s' such that $s' \upharpoonright_{\mathbf{A}} = s'_A$ and $s' \upharpoonright_{\mathbf{B}} = s_B$. If the length of the derivation is 0 then $s_A = s'_A$ and the result is immediate by taking $s' = s$. Now, Suppose

$$s_A \rightsquigarrow_{\mathbf{A}} s_1 \cdot m \cdot n \cdot s_2 \rightsquigarrow_{\mathbf{A}} s_1 \cdot n \cdot m \cdot s_2 = s'_A$$

By induction we have $s' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$ such that $s' \upharpoonright_{\mathbf{A}} = s_1 \cdot m \cdot n \cdot s_2$ and $s' \upharpoonright_{\mathbf{B}} = s_B$. Then, we have that

$$s' = t_1 \cdot m \cdot t_2 \cdot n \cdot t_3$$

where t_2 only has moves in $\mathbf{B}$ and $t_1 \upharpoonright_{\mathbf{A}} = s_1$ and $t_3 \upharpoonright_{\mathbf{A}} = s_2$. We split into cases depending on the polarity of m, n in $\mathbf{A}$. Note that since we can swap m and n in $\mathbf{A}$ it follows that either $\lambda_{\mathbf{A}}(m) = \lambda_{\mathbf{A}}(n)$ or $\lambda_{\mathbf{A}}(m) = O$ and $\lambda_{\mathbf{A}}(n) = P$.

- m **is** O **and** n **is** P Then, in $\mathbf{A} \multimap \mathbf{B}$ m is P and n is O. Now, since m is P the next move by its agent is O and therefore must also happen in $\mathbf{A}$. Hence, there is no move by the same agent as m in t_2. Therefore:

  $$s'' = t_1 \cdot t_2 \cdot n \cdot m \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot t_2 \cdot m \cdot n \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot m \cdot t_2 \cdot n \cdot t_3 = s'$$

- m **is** O **and** n **is** O Then, in $\mathbf{A} \multimap \mathbf{B}$ m is P and n is P. Then, as before, there is no move by the same agent as m in t_2 justifying the sequence of derivations below

  $$s'' = t_1 \cdot t_2 \cdot n \cdot m \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot t_2 \cdot m \cdot n \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot m \cdot t_2 \cdot n \cdot t_3 = s'$$

- m **is** P **and** n **is** P Then, in $\mathbf{A} \multimap \mathbf{B}$ m is O and n is O. Now, the previous move by the same agent as n must have been a P move in the same component as n. But there is no $\mathbf{A}$ move between m and n so must be that there is no move in t_2 by the same agent as n. Then:

  $$s'' = t_1 \cdot n \cdot m \cdot t_2 \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot m \cdot n \cdot t_2 \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot m \cdot t_2 \cdot n \cdot t_3 = s'$$

in all cases, since $s'' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s' \rightsquigarrow s$. Furthermore, in all cases

$$s'' \upharpoonright_{\mathbf{A}} = t_1 \upharpoonright_{\mathbf{A}} \cdot n \cdot m \cdot t_3 \upharpoonright_{\mathbf{A}} = s_1 \cdot n \cdot m \cdot s_2 = s'_A \qquad s'' \upharpoonright_{\mathbf{B}} = s' \upharpoonright_{\mathbf{B}} = s_B$$

as desired.

Now, suppose $s'_B \rightsquigarrow_{\mathbf{B}} s_B$. We construct by induction on the length of the derivation $s'_B \rightsquigarrow_{\mathbf{B}} s_B$ an $s' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$ such that $s' \upharpoonright_{\mathbf{A}} = s_A$ and $s' \upharpoonright_{\mathbf{B}} = s'_B$. If the length of the derivation is o then $s_B = s'_B$ and the result is immediate by taking $s' = s$. Now, suppose

$$s'_B = s_1 \cdot m \cdot n \cdot s_2 \rightsquigarrow s_1 \cdot n \cdot m \cdot s_2 \rightsquigarrow s_B$$

By induction we have $s' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$ such that $s' \upharpoonright_{\mathbf{A}} = s_A$ and $s' \upharpoonright_{\mathbf{B}} = s_1 \cdot n \cdot m \cdot s_2$. Then, we have that

$$s' = t_1 \cdot n \cdot t_2 \cdot m \cdot t_3$$

where t_2 only has **A** moves and $t_1 \restriction_{\mathbf{B}} = s_1$ and $t_3 \restriction_{\mathbf{B}} = s_2$. We split into cases depending on the polarity of m, n in **B**. Note that since we can swap m and n in **B** we have that $\lambda_{\mathbf{B}}(m) = \lambda_{\mathbf{B}}(n)$ or $\lambda_{\mathbf{B}}(m) = O$ and $\lambda_{\mathbf{B}}(n) = P$. In all cases the polarity is preserved as **B** is positive in $\mathbf{A} \multimap \mathbf{B}$.

m **is** *O* **and** *n* **is** *P* Since m is an O move there can't be any moves by the same agent as m in t_2. Hence:

$$s'' = t_1 \cdot m \cdot n \cdot t_2 \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot n \cdot m \cdot t_2 \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot n \cdot t_2 \cdot m \cdot t_3 = s'$$

m **is** *O* **and** *n* **is** *O* This goes the same as the previous case. Since m is an O move there can't be any moves by the same agent as m in t_2. Hence:

$$s'' = t_1 \cdot m \cdot n \cdot t_2 \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot n \cdot m \cdot t_2 \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot n \cdot t_2 \cdot m \cdot t_3 = s'$$

m **is** *P* **and** *n* **is** *P* In this case, as n is a Proponent move there can't be any moves by the same agent as n in t_2. Hence, the following derivation is justified

$$s'' = t_1 \cdot t_2 \cdot m \cdot n \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot t_2 \cdot n \cdot m \cdot t_3 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t_1 \cdot n \cdot t_2 \cdot m \cdot t_3 = s'$$

In all cases, since $s' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$ it follows that $s'' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} s$. Furthermore, in all cases

$$s'' \restriction_{\mathbf{A}} = s' \restriction_{\mathbf{A}} = s_A \qquad s'' \restriction_{\mathbf{B}} = t_1 \restriction_{\mathbf{B}} \cdot m \cdot n \cdot t_3 \restriction_{\mathbf{B}} = s_1 \cdot m \cdot n \cdot s_2 = s'_B$$

as desired.

The claim follows from applying the two arguments above in sequence. □

Lemma H.24. *If*

$$s \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t$$

then

$$s \restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t \restriction_{\mathbf{B}} \qquad\qquad t \restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} s \restriction_{\mathbf{A}}$$

Proof. We prove the result by induction on the length of the derivation

$$s \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t$$

If the derivation has length 0 then $s = t$ and hence

$$s \restriction_{\mathbf{A}} = t \restriction_{\mathbf{A}} \qquad\qquad s \restriction_{\mathbf{B}} = t \restriction_{\mathbf{B}}$$

and in particular

$$s \restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t \restriction_{\mathbf{B}} \qquad\qquad t \restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} s \restriction_{\mathbf{A}}$$

Otherwise, suppose

$$s = s_1 \cdot m \cdot n \cdot s_2 \rightsquigarrow^{1}_{\mathbf{A} \multimap \mathbf{B}} s_1 \cdot n \cdot m \cdot s_2 \rightsquigarrow_{\mathbf{A} \multimap \mathbf{B}} t$$

By induction there are derivations

$$(s_1 \cdot n \cdot m \cdot s_2) \restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t \restriction_{\mathbf{B}} \qquad\qquad t \restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} (s_1 \cdot n \cdot m \cdot s_2) \restriction_{\mathbf{A}}$$

We split into cases depending on the components in which m and n are played:

m **is a move in B and** *n* **is a move in B** In this case,

$$s \restriction_{\mathbf{B}} = s_1 \restriction_{\mathbf{B}} \cdot m \cdot n \cdot s_2 \restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} s_1 \restriction_{\mathbf{B}} \cdot n \cdot m \cdot s_2 \restriction_{\mathbf{B}} = s_1 \cdot n \cdot m \cdot s_2 \restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t \restriction_{\mathbf{B}}$$

and

$$t \restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} (s_1 \cdot n \cdot m \cdot s_2) \restriction_{\mathbf{A}} = (s_1 \cdot s_2) \restriction_{\mathbf{A}} = s \restriction_{\mathbf{A}}$$

m **is a move in B and** *n* **is a move in A** Note that in this case:

$$s \restriction_{\mathbf{B}} = s_1 \restriction_{\mathbf{B}} \cdot m \cdot s_2 \restriction_{\mathbf{B}} = (s_1 \cdot n \cdot m \cdot s_2) \restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t \restriction_{\mathbf{B}}$$

$$t \restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} (s_1 \cdot n \cdot m \cdot s_2) \restriction_{\mathbf{A}} = s_1 \restriction_{\mathbf{A}} \cdot n \cdot s_2 \restriction_{\mathbf{A}} = s \restriction_{\mathbf{A}}$$

***m* is a move in A and *n* is a move in B**

$$s\restriction_{\mathbf{B}} = s_1\restriction_{\mathbf{B}} \cdot n \cdot s_2\restriction_{\mathbf{B}} = (s_1 \cdot n \cdot m \cdot s_2)\restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t\restriction_{\mathbf{B}}$$

$$t\restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} (s_1 \cdot n \cdot m \cdot s_2)\restriction_{\mathbf{A}} = s_1\restriction_{\mathbf{A}} \cdot m \cdot s_2\restriction_{\mathbf{A}} = s\restriction_{\mathbf{A}}$$

***m* is a move in A and *n* is a move in A** In this case we have that as m and n have the opposite polarity in $\mathbf{A} \multimap \mathbf{B}$ than they have in $\mathbf{A}$, so that

$$n \cdot m \rightsquigarrow_{\mathbf{A}} m \cdot n$$

This justifies that:

$$s\restriction_{\mathbf{B}} = s_1\restriction_{\mathbf{B}} \cdot s_2\restriction_{\mathbf{B}} = s_1 \cdot n \cdot m \cdot s_2\restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t\restriction_{\mathbf{B}}$$

and

$$t\restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} (s_1 \cdot n \cdot m \cdot s_2)\restriction_{\mathbf{A}} = s_1\restriction_{\mathbf{A}} \cdot n \cdot m \cdot s_2\restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} s_1\restriction_{\mathbf{A}} \cdot m \cdot n \cdot s_2\restriction_{\mathbf{A}} (s_1 \cdot s_2)\restriction_{\mathbf{A}} = (s_1 \cdot m \cdot n \cdot s_2)\restriction_{\mathbf{A}} = s\restriction_{\mathbf{A}}$$

In all cases we obtain derivations

$$s\restriction_{\mathbf{B}} \rightsquigarrow_{\mathbf{B}} t\restriction_{\mathbf{B}} \qquad\qquad t\restriction_{\mathbf{A}} \rightsquigarrow_{\mathbf{A}} s\restriction_{\mathbf{A}}$$

□

Lemma H.25. *For plays $s_0, s_1 \in P_{\mathbf{A}}$ such that*

$$\forall \alpha \in \Upsilon. \pi_\alpha(s_0) = \pi_\alpha(s_1)$$

there is a derivation

$$s_1 \rightsquigarrow_{\mathbf{A}} s_0$$

if and only if there is a play $s \in \mathrm{ccopy}_{\mathbf{A}}$ such that

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

Proof. For the forward direction, note that by the definition of $\mathrm{ccopy}_{\mathbf{A}}$ there is at least one play $s' \in \mathrm{ccopy}_{\mathbf{A}}$ such that

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

By Lemma H.23 it follows that there is a play s such that

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

and then, by Proposition 4.7 it follows that

$$s \in \mathrm{ccopy}_{\mathbf{A}}$$

For the reverse direction, we prove the result by induction. Let p be the largest even-length prefix of s such that p is alternating and

$$p\restriction_{\mathbf{A}_0} = p\restriction_{\mathbf{A}_1}$$

If $p = s$ then, In particular,

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

Otherwise,

For the reverse direction, first note that if a play $t \in \mathrm{ccopy}_{\mathbf{A}}$ is alternating then

$$t\restriction_{\mathbf{A}_0} = t\restriction_{\mathbf{A}_1}$$

Indeed, since the play is alternating it follows that if

$$t = t_1 \cdot m_1 \cdot m_2 \cdot t_2$$

and $\lambda_{\mathbf{A}\multimap\mathbf{A}}(m_1) = O$ then $\alpha(m_1) = \alpha(m_2)$. But as every agent plays according to $\mathrm{copy}_{\mathbf{A}}$ that it follows that m_2 is the counterpart for m_1 in the other component. A simple argument by induction on the even-length prefixes of t shows that then

$$t\restriction_{\mathbf{A}_0} = t\restriction_{\mathbf{A}_1}$$

Now, note that by Lemma H.22 there exists t an alternating play and s_O a sequence of Opponent moves such that

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

We start by arguing that we can take $s_O = \epsilon$. Indeed, note that as $\rightsquigarrow_{\_}$ never swaps moves by the same agent we have that

$$\pi_\alpha(t \cdot s_O) = \pi_\alpha(s) \in \mathrm{copy}_{\mathbf{A}}$$

Note that in particular, t can be taken to be an even-length play, as

$$\forall \alpha \in \Upsilon. \pi_\alpha(s_0) = \pi_\alpha(s_1)$$

But then, suppose $s_O = m \cdot s'_O$. As

$$\forall \alpha \in \Upsilon. \pi_\alpha(s_0) = \pi_\alpha(s_1)$$

it follows that m has a counterpart m' that appears after m in $t \cdot s_O$. Hence, m' must appear in s'_O. But s'_O only has Opponent moves, and m' is a Proponent move, a contradiction. Hence, $s_O = \epsilon$. But now, note that we have that

$$s \rightsquigarrow_{\mathbf{A}\multimap\mathbf{A}} t$$

In particular, by Lemma H.24,

$$s\restriction_{\mathbf{A}_1} \rightsquigarrow_{\mathbf{A}} t\restriction_{\mathbf{A}_1} \qquad\qquad t\restriction_{\mathbf{A}_0} \rightsquigarrow_{\mathbf{A}} s\restriction_{\mathbf{A}_0}$$

but then:

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

as desired. □

Proposition H.26. *s_1 linearizes to s_0 if and only if there exists a play $s \in \mathrm{ccopy}_{\mathbf{A}}$ such that*

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

Proof. If s_1 linearizes to s_0 then there are sequences of Opponent and Proponent moves, respectively, s_O and s_P, such that

$$s_1 \cdot s_P \rightsquigarrow_{\mathbf{A}} s_0 \cdot s_O$$

But then, note that

$$s_1/s_O \cdot s_P \rightsquigarrow_{\mathbf{A}} s_0$$

by Lemma H.11. Hence, there is a play s of $\mathrm{ccopy}_{\mathbf{A}}$ such that

$$s\restriction_{\mathbf{A}_0} = s_1/s_O \cdot s_P \qquad\qquad s\restriction_{\mathbf{A}_1} = s_0$$

Now, notice that as s_P only has Proponent moves, by the switching condition, if m is a move in s_P then there are no moves by $\alpha(m)$ after m in s. Hence,

$$s/s_P \cdot s_P \rightsquigarrow_{\mathbf{A}\multimap\mathbf{A}} s$$

so that s/s_P (the subsequence of s where the moves in s_P have been removed) is in $\mathrm{ccopy}_{\mathbf{A}}$ by Prop. 4.7 and prefix-closure. Note that

$$(s/s_P)\restriction_{\mathbf{A}_1} = s_1/s_O \qquad\qquad (s/s_P)\restriction_{\mathbf{A}_0} = s_0$$

Now, let m be a move in s_O. Because,

$$s_1 \cdot s_P \rightsquigarrow_{\mathbf{A}} s_0 \cdot s_O$$

it follows that m does not appear in s_0. Moreover, the last move by $\alpha(m)$ in s must be a P move in $\mathbf{A}_1$, and therefore by the switching condition the last move by $\alpha(m)$ is that P move. Therefore, there must be s' such that

$$s' \rightsquigarrow_{\mathbf{A} \multimap \mathbf{A}} s/s_P \cdot s_O$$

and moreover

$$s' \restriction_{\mathbf{A}_0} = s_1$$

constructed by reversing the swaps involving s_O up to the swaps with $\mathbf{A}_0$ moves and the removal of the swaps that involve s_P, which is possible by the remarks above. By Prop. 4.7 $s' \in \mathrm{ccopy}_{\mathbf{A}}$. Moreover, as the derivation above does not involve swaps between two moves of $\mathbf{A}_0$ it follows that

$$s' \restriction_{\mathbf{A}_0} = (s/s_P) \restriction_{\mathbf{A}_0} = s_0$$

And therefore s' is the desired play.

Conversely, suppose there exists such a play $s \in \mathrm{ccopy}_{\mathbf{A}}$. Then, note that for any $\alpha \in \Upsilon$,

$$\pi_\alpha(s) \in \mathrm{copy}_{\mathbf{A}}$$

so that in particular there is a sequence of at most one Opponent move s_α such that either

$$\pi_\alpha(s) \restriction_{\mathbf{A}_0} \cdot s_\alpha = \pi_\alpha(s) \restriction_{\mathbf{A}_1}$$

or

$$\pi_\alpha(s) \restriction_{\mathbf{A}_0} = \pi_\alpha(s) \restriction_{\mathbf{A}_1} \cdot s_\alpha$$

Let then

$$s'_O = \cdot_{\alpha \in \Upsilon} s_\alpha$$

that is, the concatenation of all the s_α. Notice that this is a finite sequence as there are at most finitely many $\alpha \in \Upsilon$ for which $s_\alpha \neq \epsilon$. Then, we note that the play s/s'_O satisfies:

$$\forall \alpha \in \Upsilon. \pi_\alpha((s/s_O) \restriction_{\mathbf{A}_0}) = \pi_\alpha((s/s'_O) \restriction_{\mathbf{A}_1})$$

so let $p = s/s'_O$ and note that by Lemma H.25 it follows that

$$p \restriction_{\mathbf{A}_1} \rightsquigarrow_{\mathbf{B}} p \restriction_{\mathbf{A}_0}$$

Now, note that $s'_O \restriction_{\mathbf{A}_0}$ is a sequence of P moves in $\mathbf{A}$ while $s'_O \restriction_{\mathbf{A}_1}$ is a sequence of O moves in $\mathbf{A}$. We claim that:

$$(p \cdot s'_O) \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_0} \rightsquigarrow_{\mathbf{A}} (p \cdot s'_O) \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_1}$$

Indeed, note that

$$(p \cdot s'_O) \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_0} = p \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_0} \rightsquigarrow_{\mathbf{A}} p \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_1} = p \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_1} = (p \cdot s'_O) \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_1}$$

is valid as long as

$$s'_O \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_0} \rightsquigarrow_{\mathbf{A}} s'_O \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_1}$$

As $s'_O \restriction_{\mathbf{A}_1}$ only contains O moves and $s'_O \restriction_{\mathbf{A}_0}$ only contains P moves the reduction is valid as long as no agent that appears in $s'_O \restriction_{\mathbf{A}_1}$, appears in $s'_O \restriction_{\mathbf{A}_0}$. But note that in s, all of the moves in s'_O are Opponent, and as agents are locally sequential no two moves can be by the same agent. So the derivation is indeed valid. Now, notice that

$$s_1 = s \restriction_{\mathbf{A}_1} \rightsquigarrow_{\mathbf{A}} (s/s'_O) \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_1} = (s/s'_O \cdot s'_O) \restriction_{\mathbf{A}_1} = (p \cdot s'_O) \restriction_{\mathbf{A}_1}$$

and that

$$(p \cdot s'_O) \restriction_{\mathbf{A}_0} = (s/s'_O \cdot s'_O) \restriction_{\mathbf{A}_0} = (s/s'_O) \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_0} \rightsquigarrow_{\mathbf{A}} s \restriction_{\mathbf{A}_0} = s_0$$

Hence, by finally taking

$$s_P = s'_O \restriction_{\mathbf{A}_0} \qquad s_O = s'_O \restriction_{\mathbf{A}_1}$$

$$s_1 = s \restriction_{\mathbf{A}_1} \cdot s_P \rightsquigarrow_{\mathbf{A}} (p \cdot s'_O) \restriction_{\mathbf{A}_1} \cdot s'_O \restriction_{\mathbf{A}_0} \rightsquigarrow_{\mathbf{A}} (p \cdot s'_O) \restriction_{\mathbf{A}_0} \cdot s'_O \restriction_{\mathbf{A}_1} = s \restriction_{\mathbf{A}_0} \cdot s_O \rightsquigarrow_{\mathbf{A}} s_0 \cdot s_O$$

so that s_1 linearizes to s_0. □

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