

Abstract

A Compositional Account of Concurrent Objects

Arthur Oliveira Vale

2026

Contemporary computer systems consist of interacting heterogeneous components spanning multiple levels of abstraction. Many of these systems are concurrent, and increasingly realized as distributed services. Concurrent systems are notoriously difficult to implement correctly, and the most pernicious errors arise not within individual components but in their interactions. Making matters worse, the theoretical foundations needed to support effective verification technology for these systems—be it program analysis, testing, runtime verification, or deductive verification—remain underdeveloped. For instance, most existing correctness conditions and verification techniques are confined to atomic components, leaving large classes of concurrent objects beyond their reach.

This dissertation develops a compositional theory of concurrent objects. It introduces a uniform semantic framework for specifying, verifying, and composing concurrent objects, and establishes how correctness properties of individual objects propagate to their compositions. Particular care is given to expressivity: the framework accommodates the gamut of concurrent objects, so as to address the heterogeneity challenge. Perhaps more importantly, it presents generic techniques that apply across computing models. This is demonstrated through a compositional generalization of linearizability encompassing sequential and concurrent components, atomic and non-atomic components, safety and liveness properties, and crash-aware and crash-oblivious components. The framework is validated by the mechanized verification of several objects and small systems of objects.

The technical development draws on concurrency theory, linear logic, game semantics, coalgebra, refinement calculus, program logics, and category theory, synthesizing ideas from these traditions into a single coherent account.

A Compositional Account of Concurrent Objects

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Arthur Oliveira Vale

Dissertation Director: Zhong Shao

May 2026

Copyright © 2026 by Arthur Oliveira Vale. This work is licensed under Creative Commons Attribution-ShareAlike 4.0 International. To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>.



Contents

Acknowledgements	xv
1 Introduction	1
1.1 Objects	1
1.2 Object-Oriented Ontology	2
1.3 Object-Oriented Programming	3
1.4 Abstraction, Refinement and Encapsulation	5
1.5 Encapsulation	6
1.6 Concurrency Model	7
1.7 Contributions and Dissertation Structure	8
1.8 Collaboration Disclosures	10
2 Related Work	12
2.1 Object-Based Semantics	12
2.2 Game Semantics	13
2.2.1 Foundations	13
2.2.2 Concurrent Game Models	14
2.2.3 Semicategories and the Karoubi Envelope	15
2.2.4 Crash-Aware Game Semantics	16
2.2.5 Winning Strategies and Liveness	16
2.3 Certified Abstraction Layers	17
2.4 Linearizability	19
2.4.1 Classical Linearizability	19

2.4.2	Variations of Linearizability	19
2.4.3	Linearizability and Observational Refinement	20
2.4.4	Beyond Linearizability	20
2.5	Linearizability with Crashes	21
2.6	Linearizability and Liveness	22
2.7	Verification with Crashes	22
2.8	Program Logics and Verification Methods	24
2.8.1	Possibility-Based Reasoning and Simulation	24
2.8.2	Concurrent Program Logics	26
2.8.3	Liveness Verification	26
2.8.4	Logical Atomicity	28
2.8.5	Contextual Refinement	29
2.8.6	Relational Hoare Logic	30
2.9	Mechanization	31
3	State Transition Systems	33
3.1	Labelled Transition Systems	33
3.2	Synchronizing Product	34
3.3	Behavioral Equivalence	35
3.4	Abstraction	37
3.5	Encapsulation	38
3.6	Revisiting Labelled Transition Systems and Composition	39
3.7	Tensor	41
3.8	Well-Formedness	42
3.9	Trace Semantics	43
3.10	Conclusion	45
I	Sequential Objects	46
4	Safety and Liveness	47

4.1	The Alpern-Schneider Theorem	48
4.2	Safety and Liveness in the Scott Topology	50
5	Plays and Games	53
5.1	Moves and Plays	54
5.2	Sequential Games	60
5.3	The Implication “Game”	63
6	Safety Specifications	67
6.1	Strategies	67
6.2	Strategy Composition	71
6.3	The Copycat and Neutrality	74
6.4	Independent Composition	75
6.5	Background: Domains and Continuity	77
6.6	Refinement	79
7	Background: Enriched (Semi)Category Theory	82
7.1	Categories, Functors, Natural Transformations	82
7.2	Monoidal Categories	86
7.3	Enriched (Semi)Categories	90
8	An Interactive Linear Logic Model of State	95
8.1	Background: Object-Based Semantics	96
8.1.1	The Basic Principles	96
8.1.2	Coherence Spaces and Linear Maps	97
8.1.3	Constructors on Coherence Spaces	98
8.1.4	The Before Operator	99
8.1.5	The Replay Modality	101
8.1.6	Regular Maps between Object Spaces	102
8.1.7	Background: Comonads and the Co-Kleisli Category	104
8.1.8	Background: Products	107
8.1.9	Background: Comonoids	108

8.1.10	Linear Logic Model of State	110
8.2	An Interactive Linear Logic Model of State	111
8.2.1	Finite Products	112
8.2.2	Before Operator	112
8.2.3	The $\dagger$ -Replay Modality	113
9	Modules and Their Interpretation	116
9.1	Syntax	116
9.2	Interpreting Modules Denotationally	118
9.3	Interpreting Modules Operationally	119
10	A Refinement Calculus of Certified Components	124
10.1	The Refinement Calculus Across Models	124
10.2	The Refinement Calculus for Safety of Sequential Objects	130
II	Safety	135
11	Introduction	136
11.1	The State of the Theory of Linearizability	138
11.2	Compositional Verification in Practice	141
11.2.1	Coarse-Grained Locking	141
11.2.2	Implementing a Lock	144
11.2.3	Overview	146
12	Concurrent Games	150
12.1	Concurrent Strategies	153
12.2	Concurrent Games and Synchronization	154
12.2.1	The Copycat Strategy	154
12.2.2	Saturated Strategies	157
12.2.3	The Semifunctors K_{copy} and Emb	158
12.3	Fine-Grained Synchronization in Concurrent Games	159

13 Safe Linearizability	162
13.1 Linearizability	162
13.2 Unary Linearizability	164
13.3 Computational Interpretation of Linearizability	168
13.4 Observational Refinement	169
13.5 Locality	170
14 Compositional Linearizability	174
14.1 Karoubi Envelope	174
14.1.1 The Karoubi Envelope	174
14.1.2 Sections of the Karoubi Envelope	177
14.2 Sketches of Compositional Linearizability	181
14.3 The Linearizability Galois Connection	186
14.4 The Compositional Linearizability Refinement Calculus	189
15 Equivalence with Linearizability	197
15.1 Atomicity	197
15.1.1 Sequential Atomic Games	197
15.1.2 Concurrent Atomic Games	199
15.1.3 Atomic Linearizability	202
15.1.4 Observational Refinement and Locality	205
15.2 Interval-Sequential Linearizability	206
16 An Analysis of Herlihy-Wing Possibilities	209
16.1 Copycat Positions	210
16.2 Possibilities	211
16.3 Linearization Points	213
16.3.1 Punctual Extensions under Atomicity	215
17 A Complete Program Logic for Compositional Linearizability	217
17.1 Concurrent Operational Semantics	218
17.2 Program Logic	220

III	Crash-Safety	226
18	Introduction	227
18.1	The FLiT Library	228
19	A Concurrent Game Semantics with Crashes	235
19.1	Games with Full-System Crashes	235
19.2	The Crashing Copycat and Saturation	239
20	Crash-Aware Linearizability	241
20.1	Crash-Aware Linearizability	242
20.2	Specifying a Buffered Memory Cell	244
20.3	Compositional Verification of a File System Fragment	245
21	Durable and Strict Linearizability	248
21.1	Background: Linearizability Criteria under Crashes	249
21.2	Contrasting Crash-Aware Linearizability	251
21.3	Crash Abstraction	254
21.4	Strict Linearizability	255
21.5	Durable Linearizability	257
21.5.1	Durable Linearizability	257
21.5.2	Observational Refinement and Locality	259
21.5.3	FLiT Correctness Theorem	260
IV	Liveness	262
22	Introduction	263
22.1	Linearizability and Progress	264
22.2	The Elimination-Backoff Stack	266
23	The Liveness Game Model	270
23.1	Win Conditions	272

23.2	Refinement and Composition for Win Conditions	276
23.3	Sidenote: A Unified Model	278
24	Live Linearizability	281
24.1	Progress Reflection	282
24.2	Honest Linearizability	285
25	Relating Safety to Liveness	287
25.1	Object Specifications	288
25.2	Totality, Completeness and Tightness	291
25.3	Completeness	295
26	Parallel Implementations	297
26.1	Sequential Implementations	297
26.2	Parallel Implementations	302
26.3	Symmetric Monoidal Structure	304
27	Composing Completeness and Tightness	306
27.1	Preliminary Results	306
27.2	Composing Completeness	307
27.3	Putting it Together	310
27.4	Relaxing Tightness	312
28	Live Linearizability Rely-Guarantee Logic	314
28.1	Operational Semantics	314
28.2	Program Logic	316
V	Conclusion	320
29	Conclusion	321

VI	Proof Compendium	323
30	Sequential Games	324
30.1	Safety and Liveness	324
30.2	The Replay Modality	326
31	Safety	330
31.1	Strategy Composition	330
31.2	Order Enrichment	331
31.3	Order-Enriched Semicategory	333
31.4	Symmetric Monoidal Structure	336
31.5	The K_{copy} Semifunctor	341
31.6	Fine-Grained Synchronization	342
31.7	Computational Interpretation	349
31.8	Atomicity	360
31.9	Atomic Linearizability	366
31.10	Linearization Points	373
32	Crash-Safety	380
32.1	Basic Semicategorical Structure	380
32.2	Volatile Lift and Idempotence	384
32.3	Concrete Saturation for Crash-Aware Games	387
32.4	Symmetric Monoidal Structure of Crash	394
32.5	Crash-Aware Linearizability	399
32.6	Crash Abstraction	403
32.7	Strict Linearizability	408
32.8	Durability	411
32.9	Symmetric Monoidal Structure of Dur	416
32.10	Durable Linearizability	418
33	Liveness	422
33.1	Composition	422

33.2 Sidenote: Associativity of divergent composition	425
33.3 P-maximal extensions	432
33.4 Live Linearizability	433
33.5 Honest Linearizability	437
33.6 Relating Safety and Liveness	438
33.7 Symmetric Monoidal Structure for Objects	439
33.8 Parallel Implementations	444
33.9 Composing Completeness and Tightness	457
34 Program Logics Soundness	467
34.1 Safety	467
34.2 Liveness	476

List of Figures

5.1	Position graph for well-formed plays.	58
10.1	A certified abstraction layer $L_{rb} \vdash_R M_{bq} : L_{bq}$ implementing a bounded queue of size N using a ring buffer. The transition systems $\mathcal{V}_{bq}$ and $\mathcal{V}_{rb}$ define the overlay and underlay interfaces. The implementation M_{bq} and the simulation relation R are used in the correctness proof.	132
11.1	Shared Queue implementation (left), and Lock implementation (right) . . .	143
11.2	Correctness of individual components implies correctness of the composed system. The lock implementation M_{lock} (bottom-left) is verified against the linearized specifications ν_{fai} , ν_{srcntr} , and ν_{yield} , while the shared queue implementation M_{squeue} (top-left) is verified using ν_{lock} and the racy queue ν_{rqueue} . By compositionality, these two individual results ($\wedge$) imply ($\implies$) that the vertically composed system (right) correctly implements ν_{squeue} . Horizontal composition uses the linear logic tensor $- \otimes -$, and copy denotes the neutral element for composition.	145
11.3	Code corresponding to copy₋ (left); Diagram depicting the operations K_{copy} and Emb (right)	147
12.1	Maximal play of seqcopy_{Unit} (left) and corresponding pseudocode (right) . .	155
12.2	Maximal play of int(seqcopy_{Unit}, seqcopy_{Unit}) (left) and corresponding pseudocode (right).	155
16.1	Possibility update rules.	212

17.1	Module Small-step semantics ($\longrightarrow$)	218
17.2	Other Proof Rules of HLHL	223
18.1	FLiT Memory Cell Implementation M_{FLiT}	230
18.2	(a) Using our program logic for durable linearizability, we verify the FLiT implementation; (b) Using the program logic for compositional linearizability we verify the crash-less snapshot object; (c) Using the FLiT correctness theorem, we lift the crash-less snapshot object into a durably linearizable snapshot object running on top of a FLiT array; (d) Using vertical and horizontal composition, we obtain (e) a durably linearizable snapshot object running on top of an array of buffered memory cells and volatile counters. .	232
19.1	Example of a play of Unit with crashes.	237
20.1	The structure of our File example.	246
22.1	Four counter implementations: (1) A typical shared counter implementation using a lock and racy counter; (2) A modification of (1) where the first inc call always diverges <i>after</i> incrementing the racy counter; (3) A modification of (1) where the first inc call always diverges <i>before</i> incrementing the racy counter; (4) A modification of (1) where inc calls can diverge depending on the scheduling.	264
22.2	The elimination-backoff stack code (right) and the hierarchy of underlying objects (left).	266
23.1	The definitions of: maximal progress (maxPr), minimal progress (minPr); fair scheduling (fair) and isolating scheduling (iso); lock-freedom (lf), wait-freedom (wf), deadlock-freedom (df), starvation-freedom (sf), and obstruction-freedom (of). All predicates are defined over P_E^{ws} . Our definitions are inspired by Herlihy and Shavit [2011] and feature a correction to obstruction-freedom from Liang et al. [2013].	275
28.1	Rules for managing scheduler counters	315

List of Tables

8.1	Coherences table for a positive product operator $A \odot B$	99
8.2	Possible positive product operators	100

Acknowledgements

I am deeply grateful to my advisor, Zhong Shao, who was seemingly always available, always attuned to his students' research and the surrounding literature, and always committed to rigorous scientific inquiry at the highest standard. I have valued the freedom he gave me to pursue my own ideas and his willingness to place his trust in me. Yale has also afforded me a comfortable life as a graduate student, and this is certainly in part due to Prof. Shao's ability to keep his Ph.D. students funded without our having to even think about it.

I must also thank Paul-André Melliès for serving on my dissertation committee. Beyond that, his mentorship early in my Ph.D. years made it much easier for me to learn the mathematical tools that proved imperative to the research in this dissertation. I am equally grateful to him for teaching me how to write a high-quality academic paper through our collaboration on my first publication.

I want to thank James Aspnes not only for serving on my dissertation committee, but also for his excellent course on the theory of distributed systems, which gave me the confidence to explore concurrency as part of my doctoral work—something I never imagined I would become an expert in before joining Yale. It was impressive how he managed to make such a challenging subject approachable even for the motivated undergraduates who take the class.

I extend my gratitude as well to Alex Lew for agreeing to serve on my committee on such short notice, and for his friendliness ever since joining Yale. I must also thank Ruzica Piskac, who served on my dissertation committee for all but the final stretch of my candidacy, and who helped me feel confident in coming to Yale in the first place. I am very grateful for our interactions over the years.

I must also recognize several current and former members of the FLINT group who either directly contributed to this work or whose discussions and participation in the group shaped it in one way or another. Foremost among them, my frequent collaborator Zhongye Wang, whose expertise has made our joint papers truly shine. Jérémie Koenig, who taught me much of what I know about the theory of system specifications, and whose work deeply influenced my own. Eashan Hatti and Yueyang Feng, who were a pleasure to mentor and learn from. Yixuan Chen, who always had a good example up his sleeve. Peixin You, who is always eager to talk about semantics. Beyond my direct collaborators, I also thank Richard Habeeb, Yuyang Sang, Yuting Wang, Longfei Qiu, Ben Chaimberg, Daniel Luick, Hao Chen, Yu Zhang, Jieung Kim, Wolf Honoré, Christian Altamirano Modesto, and Justin Restivo.

Beyond my group, I found many examples of excellent pedagogues among the professors and lecturers who supervised me as a Teaching Fellow. I thank in particular Dylan McKay, whose enthusiasm for teaching and the foundations of computer science is infectious, as well as Quanquan C. Liu, James Glenn, Andrew Bridy, and Gerry Brady. I found an equally impressive role model in Amy Zhao, whose presence any teaching team would be lucky to have.

To me, the Ph.D. journey started very early, and I must recognize the mentorship of those who helped me on this path. First, the extremely dedicated faculty of CMR, in particular Goretti Cabral and Emanuel Silva—I could not have asked for a better education. I thank the professors Henrique de Barros Correia Vitória and Eudes Naziazeno, from UFPE, who dedicated much of their time to developing my mathematical thinking from a very young age, a skill that has proven invaluable throughout my career. I am grateful to the organizers of the PIC Jr. program and the scientific olympiads in Brazil, who do more for science in Brazil than they receive credit for. I also thank Andrew Drucker, whose classes ultimately led me to change fields, and Stuart Kurtz, who advised me for many years and taught me much of the foundations of computer science and introduced me to the beauty of programming language theory, as well as Ravi Chugh and John H. Reppy, whose teaching further deepened that interest.

I also thank my colleagues in the Yale CS department, particularly Anargyros Oikonomou

and Jordan Schmerge; my friends Frankie Fung, Jan Ertl, Paolo Degiorgi, the members of math frat, and those members of the class of 2020 at the University of Chicago who helped make quarantine less lonely; as well as my friends Bruno Barbarioli, Jacob Stein, Daniel Mitsutani, Radhika Kaicker, Michael Berson, Matheus Oliveira, Natália Prado, Lorena Lima Tavares de Melo, and Vinícius Silva de Oliveira, for their friendship throughout many years.

Finally, I thank my mother for her unconditional love, for teaching me how to be a citizen of the world, and for her unbounded support of my education and aspirations. And to my wife, Has—thank you for your love, care, and support throughout my doctorate.

This work was supported in part by the National Science Foundation (NSF) grants 2019285, 1763399, 1521523, 2118851, and 2313433, and by the Defense Advanced Research Projects Agency (DARPA) Agreement No. HR00112590130 and DARPA and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018.

Chapter 1

Introduction

1.1 Objects

In this dissertation, there shall be much ado about *objects*. The question then is: what do we mean by an *object*? The unfortunate truth, to the computer scientist at least, is that the question is essentially philosophical. A naive account would have something to do with the notions of *state* and of *operations* to manipulate it. I take this as a non-definition for the following reason: it has no attitude.

For instance, Turing machines notably involve a very well-defined notion of state (some tape-head configuration) as well as a precise description of what operations may be performed on this piece of state. Nonetheless, one would hardly think of Turing machines as a model of objects, given that their essential computational content is that of computable functions. In other words, Turing machines are state machines with an attitude: describing computable functions.

Similarly, programming languages are often given their core semantics through operational semantics. This involves defining a notion of state for program evaluation and specifying what operations, i.e., reductions, are available at any given state. For example, in the case of the pure λ -calculus, we might think of the state as a λ -term and steps as β -reductions.¹ Alternatively, for the IMP language over a basic memory model, one can

1. More accurately, the notion of state depends on the evaluation strategy.

give an algebraic account where the state is a partial map from locations to values, and the reductions are given by the usual Kleene algebra (those implied by the partial order derived from the additive structure) enhanced with the semantics of assignment to and reading of locations. Again, there is a mismatch between the intent of operational semantics—describing the execution of programs—and the intent of objects.

It is essential then to pin down what we mean by an object.

1.2 Object-Oriented Ontology

Before moving to a more grounded account of objects, let’s give a chance to the philosopher to speak. Object-oriented ontology (OOO)² is a school of metaphysics that takes as central the notion of object. Because of this, OOO rejects anthropocentrism, and gives equal ontological footing to a rock and to a human. This also makes it a flat ontology, since it treats *everything* as an object. An object’s key characteristic in OOO is a notion of irreducibility: “‘object’ simply means anything that cannot be reduced either *downward* or *upward*, which means anything that has a surplus beyond its constituent pieces and beneath its sum total of effects on the world” [Harman, 2018]. To understand what is meant by this, a typical example in the OOO literature is to consider a hammer. A hammer is neither “just” its constituent atoms (downward reduction), nor “just” a tool for hammering (upward reduction). In computer science terms, a stack is neither just a linked list, nor just a LIFO container.

While taking OOO to its full extent is likely counter-productive for our purposes, we can take a few lessons from it. First, the reader will find that the idea that an object cannot be reduced downward corresponds well with the idea of abstraction. A stack, semantically, exceeds its implementation in terms of a linked list, in that we understand a stack as a stack as long as **pop** and **push** behave as we expect of a stack. Meanwhile, upward reduction is the idea that a concrete object is not reducible to just something that provides correct **push** and **pop** operations. That is, we still consider two different implementations of a stack as

2. “The term ‘object-oriented philosophy’ was initially borrowed in jest from computer science, but took on a life of its own” [Harman, 2013].

different objects (on the nose) even if they behave the same from the point of view of a certain context (an observer). Neither point of view is sufficient to pin down what is meant by a stack.

Another point we do take from OOO is the idea of a flat ontology. We too would like to consider any computational entity as an object. Although we could have maintained this commitment throughout, it would come at the cost of a more complex model than necessary for our purposes. Therefore, the reader will find that on occasion we do create distinctions between the notions of *object* and *object implementation*.

Now that we have humored the philosopher, we move to a first approximation of what we shall consider an object to be throughout the remainder of this dissertation.

1.3 Object-Oriented Programming

The notion of object in computer science was disseminated, perhaps even invented, as part of the history of the development of object-oriented programming (OOP). While we do not offer a thorough account of the history, we find it useful to bring our notion of object into the discussion by emphasizing the discourse around objects in a roughly historical fashion.

The first use of the word “object” in the sense understood by OOP appears in `Simula 67`, which extended `ALGOL 60` with the notions of *object* and *class*, among other features. A class is a way of describing a container for data, and what operations may be performed on that data. Meanwhile, an object is a particular instance of a class, obtained by instantiating the data with particular values.

The term *object-oriented programming* is attributed to Alan Kay, who coined it during the development of `Smalltalk`. He explains it as fulfilling several design goals, some of which I discuss here. First:

“I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful).” [Kay, 2003]

In fact, Alan Kay has even said he regrets calling what **Smalltalk** was doing “object-oriented” because to him “the big idea is ‘messaging’ ” [Kay, 1998]. We see here the beginnings of the tension that developed over the following decades between the shared-memory and message-passing paradigms. This tension is particularly prominent in the theory of distributed systems. In this dissertation, I make no distinction between the two. This is both a technical and paradigmatic choice. As will soon be introduced, even when I treat an object as a piece of data subject to some operations, its semantics is given by the possible sequences of messages that may be exchanged between the environment (the context) and the system (the object).

In practice, the line between the two paradigms is blurred. At all layers of the computing hierarchy, we observe that computation is achieved by having individual components (which are closer to a shared-memory object) interact with each other. At the hardware level, we have many storage devices, many types of computing units, many external devices, etc. At the software level, we have the OS, which must manage these interactions so as to provide the *illusion* of access to a shared memory object (as specified by the system calls). Nonetheless, IPC, file system interactions, and networking muddle this picture. Then, we build server clusters, which make the message-passing paradigm more apparent. But then, we build services like key-value stores on top of networked infrastructure to once again provide the *illusion* of a shared-memory object. From this point of view, every system really has a hybrid nature, reflecting the reality that neither paradigm is dominant.

Next, Alan Kay remarks that:

“I wanted to get rid of data. The B5000 almost did this via its almost unbelievable HW architecture. I realized that the cell/whole-computer metaphor would get rid of data, and that “<-” would be just another message token (it took me quite a while to think this out because I really thought of all these symbols as names for functions and procedures.” [Kay, 2003]

We see here what we now think of as *encapsulation*. That is, the data of an object is internal to it. To a context (an observer), it suffices to exchange messages with the object (even if those messages are operations like **push** and **pop**) without knowledge of

the underlying data. Encapsulation is deeply related to the concept of *abstraction*. The reason for this is that if the data inside a stack object is not exposed to a client, then the client should be insensitive to the particular data representation of the stack. It could be something very concrete, such as physical memory cells, or something more abstract, such as a linked list, or even a mathematical description of a stack as an inductively defined list of values.

1.4 Abstraction, Refinement and Encapsulation

The idea of “abstraction”, or rather the relation “ x is more concrete than y ” or “ x is less abstract than y ”, is often called *refinement*, which we will write as $x \sqsupseteq y$.

This was perhaps first formalized as *extensional equivalence* by Morris [1969] (today called *contextual equivalence*), or as *data refinement* by Hoare [1989] (i.e., refinement along state). The idea is that a λ -term M is a refinement of N when for any context $C[-]$, if $C[N]$ has a normal form then $C[M]$ has the same normal form. This is best understood as a substitution principle: in any context where N could be used, we can use M instead. In the context of OOP, this is echoed by the Liskov substitution principle:

“Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects of type S where S is a subtype of T ” [Liskov and Wing, 1994].

In fact, the idea of property preservation became the defining aspect of refinement, as per work on the refinement calculus [Back, 1981]. That is, a generic definition of *refinement* is that an object x is a refinement of y when for any property P that y satisfies, x also satisfies P . In other words, the implementation inherits all properties of the specification. Of course, there is a lot of flexibility with this definition: what is meant by property? What about objects?

Now, a key thing to notice is that there is a clear distinction between *contextual refinement* and *refinement* as just defined. Contextual refinement involves a notion of composing an object with a context, while refinement seems like a logical property at first glance.

Of course, if our domain of properties allows for stating the property $P(x) := “C[x] \text{ has normal form } v,”$ then contextual refinement is a special case of refinement. This shows how important the domain of properties is for what we even mean by refinement.

This dissertation concerns itself with refinement between objects, or components, as we will often call them. That is, throughout this dissertation, we will build semantic models of objects (answering what we mean by an object). These models will comprise the composition operations we consider between objects, as well as the properties these composition operations satisfy. We will define a notion of property, in some cases conflated with the notion of object for uniformity of treatment, and then define refinement for objects. Then, we will show that our notion of refinement implies contextual refinement, i.e., for any composition operation $\otimes$ and components $x' \sqsupseteq x$ and $y' \sqsupseteq y$, $x' \otimes y' \sqsupseteq x \otimes y$. We will see how doing this results in a compositional calculus for verification of components, in the refinement calculus tradition.

1.5 Encapsulation

Part of the point of OOP was the idea of *encapsulation*. Kay [2003] explicitly said he wanted to get rid of data. What he meant by that is that an object should fully encapsulate its data, by bundling together the underlying state representation and the operations on the state, so that the internal representation of the data is hidden from clients. However, and this is a point of disagreement between my work and many OOP approaches, usually in OOP-based developments it is still possible to interfere with that internal representation. For instance, when implementing a counter from a memory cell, one could save an alias for that memory cell, which then gets arbitrarily modified externally to the counter. This could turn the internal data into garbage (say, by making it a negative integer), breaking an otherwise correct counter. Clearly, this precludes compositional verification.

So I take a stronger perspective on encapsulation. What I mean by the data internal to an object is actually just another object, as I will model everything as a kind of object. When an object, such as the memory cell, is used to implement a new object, such as a counter, that memory cell is entirely encapsulated inside the counter, with no possibility of

leaking an alias. Note that this does not necessarily mean the memory cell is of exclusive “ownership” to the counter. This just means that any possible interference from the outside must already be taken into account in the semantics of the internalized memory cell. This will also make my models *affine* in the sense that an implementation “consumes” its source object to implement the target object.

1.6 Concurrency Model

These ideas are fairly well understood for sequential objects: objects that can be used “one operation at a time”. The next operation may only be invoked once the last one has provided a response. They are significantly less developed for concurrent and distributed objects. This is no accident, as the latter systems are considerably more complex. For instance, n threads executing some code generate super-exponentially more behaviors than the same program when run by a single thread.

But concurrency is also an opportunity for more interesting refinement notions. In the sequential case, refinement is essentially about data refinement, that is, abstraction “along space”. Concurrency adds a new axis of abstraction, where we could try to abstract components “along time” in order to abstract a concrete concurrent object, with all its complex interleavings, into a much simpler object with a lower degree of concurrency.

The core contribution of my doctoral work is obtaining a notion of refinement satisfying all the constraints described above. This is done at a high level of abstraction, so that the same general technique for dealing with concurrent refinement easily translates across models. We call the resulting notion of refinement *compositional linearizability* due to its relationship to atomic linearizability [Herlihy and Wing, 1990] when instantiated in the context of standard concurrency models.

We use a simple concurrency model which is consistent with previous work in concurrency theory, particularly linearizability [Herlihy and Wing, 1990]. We assume a set of *agents* Υ . An $\alpha \in \Upsilon$ could be a processor core, a thread, a node in a distributed system, a device that is part of a larger system, or a mix of these. We assume each agent is sequential, i.e., it strictly alternates between environment actions and system actions; this is sometimes

conflated with the *sequential consistency* condition [Lamport, 1979], but we prefer to call it *locally sequential* or *parallel* to avoid confusion with that correctness criterion.

Generally, agents are assumed to be scheduled fairly, unless they have been starved by the scheduler. In other words, while the scheduler has not decided to starve an agent, it guarantees it will schedule the agent fairly alongside other agents. The scheduler might, however, non-deterministically starve any agent, at any point. Otherwise, the scheduling is entirely non-deterministic. When we want to state a property such as starvation-freedom, we simply state it conditionally as “if the scheduling is fair, then every α always eventually makes progress”.

1.7 Contributions and Dissertation Structure

The dissertation starts with a short chapter (Ch. 3) addressing the absence of *state* for most of the dissertation, showing how coalgebra theory and insights from denotational and game semantics allow for much of the theoretical treatment to be done using only traces. The remainder is organized into five parts.

Part I: Sequential Objects. I develop the foundations of the compositional framework in a sequential setting. This includes the core game-semantic notions (moves, plays, strategies, effect signatures), the Alpern-Schneider decomposition of trace properties into safety and liveness, enriched semicategory theory, an interactive linear logic model of stateful objects extending Reddy’s object-based semantics, and a compositional refinement calculus for certified components.

The part features the first detailed treatment of Reddy’s $\dagger$ -modality [Reddy, 1996] in a game model, ultimately resulting in my approach to object implementations in both the sequential and concurrent setting. Then, I describe a construction of a compositional refinement calculus by marrying the setting of enriched categories with ideas from the refinement calculus [Back, 1981], which leads to a general recipe to obtain a compositional verification framework given a particular model of computation and a notion of refinement.

Part II: Safety. This part investigates how the compositional refinement calculus of Part I applies to concurrent computation. The key observation is that there is an opportunity to improve what is meant by refinement in a concurrent setting, as we can attempt to abstract computation not only along space (data refinement) but also along time (linearizability). This leads to an entirely new approach to the theory of linearizability, which gives a compositional account of it resulting in the notion of compositional linearizability. I re-develop the theory of linearizability from this perspective and show that compositional linearizability can also be given a categorical description compatible with the compositional refinement calculus developed in Part I. I finish by presenting a complete program logic for compositional linearizability, itself a key contribution as it is the first program logic which is sound for non-atomic notions of linearizability, as well as complete for them.

Part III: Crash-Safety. I showcase the compositional linearizability construction in the context of systems with full-system crashes, extending the game model and linearizability theory to account for crash and recovery events. This results in a game model with an interesting technical approach to handling crashes. But, moreover, the compositional linearizability construction results in a new linearizability criterion I call *crash-aware linearizability*. I show crash-aware linearizability is expressive enough to handle a variety of types of crash behaviors, but also that it helps generalize and further develop the theory of durable linearizability and strict linearizability, pre-existing criteria for persistent objects.

Part IV: Liveness. I address the liveness dimension of concurrent objects. This results in a game-semantic model that unifies two ideas from the literature, *winning strategies* and *divergence tracking*, into a kind of strategy we call a *win condition*. This leads to a novel model featuring a formal treatment of scheduler starvation and divergence, allowing for the compositional verification of liveness properties. I then show how safety and liveness relate to each other, assembling the full notion of a concurrent object as one with both a safety and a liveness specification. Then, by applying the compositional linearizability construction, I derive a criterion I call *live linearizabil-*

ity, which I then show reflects all the usual progress properties [Herlihy and Shavit, 2011]. Then, I extend the program logic from Part II to handle live linearizability. This provides the first verification approach which is both compositional and handles liveness verification beyond a single progress property.

Part V: Conclusion. I summarize the main contributions and discuss directions for future work.

Part VI: Proof Compendium. Proofs that are deferred from the main text are collected in the final part.

Most proofs are omitted, and instead a reference is given for where the proof may be found. Where a proof would help make an argument, we do present it.

1.8 Collaboration Disclosures

Each of the parts of the dissertation benefited from the contribution of other researchers. Here, I briefly clarify what parts are my contribution and what parts were contributed by others. It should be highlighted that all of the work in this dissertation was developed under the supervision of Zhong Shao.

Part I: Sequential Objects. The original version of the material here comes from Oliveira Vale et al. [2022]. There are fundamental changes made to better match the concurrent semantics we develop using the sequential semantics, so to some extent the variation here is entirely my contribution. Ideas and expertise from all the authors of Oliveira Vale et al. [2022] contributed to the material that appears there. All the results presented here were originally proven by me, however. The only exception is the ring buffer example (which appears in Ch. 10.2), which was mostly written by Jérémie Koenig and Zhong Shao, and modified by me to better fit this dissertation.

Part II: Safety. All of the results in this section were my contribution, including their development and proofs. The handling of possibilities and the rules of the program logic are presented using versions developed after the original inception of the program

logic, in Oliveira Vale et al. [2024a] and Oliveira Vale et al. [2023], and were developed in collaboration with Zhongye Wang and Eashan Hatti as part of later work. They are nonetheless equivalent to my original formulation. The proof of completeness of the program logic is due mostly to Eashan Hatti in work under my supervision, although it benefitted from my ideas. The lock example in this part was originally proved using my program logic by Yixuan Chen, and later mechanized by Eashan Hatti and Yueyang Feng.

Part III: Crash-Safety. The overall theory is my contribution, as well as much of the design of the program logic that appears in the paper [Oliveira Vale et al., 2024b]. Some of the proofs of the theory were written by Peixin You under my supervision. The program logic was designed by me and Zhongye Wang, but its proof of soundness was written by the latter using my proof of soundness for the program logic for safety as a template. The examples were suggested by Yixuan Chen, who also proved the correctness of the file system in our program logic. The parts of the FLiT example that use the program logic were proven by Zhongye Wang, while the glueing of these parts into the final theorem is my work. The program logic does not appear in this dissertation, however.

Part IV: Liveness. All of the theory in this part is my contribution, as well as the suggestion of the examples. The program logic was developed in collaboration with Zhongye Wang, and is mostly his work including the proof of soundness. He also proved all the examples used in this part using the program logic.

Chapter 2

Related Work

This chapter surveys the literature surrounding the contributions of this dissertation. We organize it thematically, starting from the foundations of object-based semantics and game semantics (Part I), through linearizability theory and its extensions to crashes and liveness (Parts II–IV), to verification methodology and mechanization.

2.1 Object-Based Semantics

Object-based semantics was introduced by Reddy [Reddy, 1996], building on Girard’s coherence spaces [Girard, 1987b] and Retoré [1997]’s before operator. The key idea is to model imperative programs not as functions on global states, but instead by their observable interactive behavior. Reddy formalized this using a novel comonad $\dagger$ on coherence spaces, and axiomatized the required categorical structure as a Linear Logic Model of State (LLMS) [Reddy, 1993]. His subsequent work [O’Hearn and Reddy, 1999; Reddy, 2002; Reddy and Dunphy, 2012; Reddy, 2013] combined the event-based and state-based approaches to give a full semantics of Algol-like languages. Reddy’s work was also seminal in the later development of game semantics. In Part I of this dissertation, we re-introduce and extend Reddy’s ideas to develop an interactive, game-semantic account of object-based semantics.

2.2 Game Semantics

2.2.1 Foundations

Game semantics has been around for more than 30 years. It has been extremely successful in describing the fine-grained semantics of a large class of programming languages including PCF [Abramsky et al., 2000; Hyland and Ong, 2000], imperative languages [Abramsky and McCusker, 1997, 1999; Ghica and Murawski, 2004], and object-oriented languages [Murawski and Tzevelekos, 2014]. Despite its importance and promising support for compositional reasoning, it has not been used in large formal verification projects based on proof assistants. Instead, the formal verification community has preferred to use simple small-step or mixed-step operational semantics to verify programs because game semantics is often seen as too complex to be smoothly mechanized in any proof assistant. Despite that, it has recently been mechanized by Borthelle et al. [2025], and a particular point-of-view on game semantics has been mechanized as part of [Zhang et al., 2025].

Our work as well as Koenig and Shao [2020]; Zhang et al. [2025] can be seen as significant steps toward applying game semantics to the mechanized verification of large systems. We have had a pleasant experience in mechanizing coherence spaces due to their simplicity. Furthermore, while we give a traditional game semantics presentation to our model in Ch. 8, we believe Koenig [2021] provides an equivalent model (in the sequential setting) amenable to mechanization.

The game semantics literature has focused on giving the semantics of a specific programming language and then using it to prove soundness and full abstraction properties. These works are complex because they use game semantics to model command- or expression-level interaction in the core programs. These languages and their game semantics are not primarily designed for program verification, and there are no equivalent notions of layer interfaces or certified layers. By focusing on certified layers, we take the best idea from game semantics to support certified composition. To make things simple, our key idea is that these certified layers must fully encapsulate their states; otherwise, their interfaces would be too complex and then make composition difficult.

This is why Reddy’s approach to handling global state is particularly attractive. Starting

from the seminal work by Abramsky and McCusker on Idealized Algol [Abramsky and McCusker, 1997], the game semantics community took inspiration from Reddy’s idea to give fully abstract models to imperative languages, for programming in the small. Here, we show that the very idea could have a big impact for certified layer programming in the large. This happens to match the best practice on how abstraction layers are used by real-world engineers. Also, despite its early influence on game semantics, the $\dagger$ modality has been largely forgotten in the game semantics community in favor of linear logic’s $!$ modality. Our work seeks to bring back into focus the relevance of the $\dagger$ modality by showcasing its simplicity and expressiveness.

Our approach to game semantics traces back to [Abramsky and McCusker, 1997], and our game model is an extension of the one developed in Part II, which is itself inspired by [Hyland, 1997] for the sequential game model and [Ghica and Murawski, 2004; Ghica, 2023] for its concurrent variation.

2.2.2 Concurrent Game Models

Our concurrent game model is heavily inspired by the model appearing in Ghica and Murawski [2004] and Ghica [2023], and the genesis of our key result lies in the observation that the rewrite system they use for saturation is deeply related to happens-before order preservation. 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 removes many of the complications involved in managing pointers, making it simpler to manipulate and to certify in a proof assistant.

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 as to obtain a Cartesian category. We instead use an affine composition operation that leaves the left strategy as is, and fits our purposes better. An-

other 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 [2023], but was not explored in detail. In particular, none of the material in §14.1 appears in their work. Neither of these works deals with linearizability in any way, nor observes the relationship between their rewrite relation and happens-before preservation.

The authors likely noticed that the rewrite relation in Ghica and Murawski [2004]; Ghica [2023] is related to linearizability, as a variation of it appears in Murawski and Tzevelekos [2019]. There, 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. None of these works observe the relationship between `copy_` and the Karoubi envelope with linearizability.

The notion of rewrite saturation in games comes from [Laird, 2001], and is a common feature of concurrent game models. Other concurrent game semantics that differ substantially from ours in their formulation have also been proposed [Abramsky and Melliès, 1999; Melliès and Mimram, 2007; Rideau and Winskel, 2011; Castellan et al., 2017]. More broadly, our motivations fit into a program started by Koenig and Shao [2020]. Game semantics has been used to analyze concurrent program logics in Melliès and Stefanescu [2020] to a much larger extent than what we endeavor in Ch. 17.

2.2.3 Semicategories and the Karoubi Envelope

Semicategories have been studied extensively in the context of the 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 Rideau and Winskel [2011]; Castellan et al. [2017] (though not explicitly mentioned).

2.2.4 Crash-Aware Game Semantics

Our game semantics model for crash-aware reasoning (Part III) is directly based on that of Part II, and our use of object-based game semantics traces back to Part I and [Reddy, 1993, 1996]. To develop our crash-aware model, we indirectly made use of insights from Melliès [2019]. In our goal of describing systems written in imperative languages, our game semantics is related to some of the work by Ghica and Tzevelekos [2012]; Koenig and Shao [2020]. It is important to note that crashes are not accurately modeled as a separate computational agent responsible for issuing crashes: crashes are instantaneous and pervasive, synchronous across components, are not invoked, and are unimplemented. Because of this, our crash-aware model is rather unorthodox in that it breaks the tradition of having only two players (Opponent and Proponent) by adding an extra player for crash events. It models crash events differently from usual moves in traditional game models by having crash events happen instantaneously and synchronously across all components, while typically, a move belongs to a single component and happens mostly asynchronously. As far as we are aware, this is the first game semantics of its kind. Because of this, while we build on the model from Part II and benefit significantly from the theory there, our model needs to address the intrusive effects of properly modeling crashes.

2.2.5 Winning Strategies and Liveness

Winning strategies originally appear in order to realize fragments of linear logic [Joyal, 1977; Abramsky and Jagadeesan, 1992; Blass, 1992], and in a concurrent setting in [Abramsky and Melliès, 1999; Melliès, 2005a]. These works do not use winning strategies as progress specifications, and usually make totality a separate requirement for winning strategies,

which obfuscates their usefulness as more general progress specifications. An exception to this is [Clairambault et al., 2012], which does not assume totality, but still does not observe the potential use of win conditions as progress specifications.

Different from previous works, which typically attach win conditions to games, we attach them to strategies as our approach essentially treats a strategy $\nu : A$ as a refinement type for A seen as the strategy $P_A : A$. Moreover, as far as we are aware, the concepts of well-scheduling, win conditions, and $- \wp -$ are novel to Part IV of this dissertation. Our handling of divergences is inspired by [Gowers and Laird, 2018; Harmer and McCusker, 1999], which only discuss the sequential case. Meanwhile, we extend tracking of divergences to the concurrent case, and we realize divergence tracking in a unified fashion with win conditions (typically used to keep track of convergence instead), which we believe is more foundational than either technique in isolation.

Although the interplay between refinement and composition has always been present in game models, our particular approach and application come from [Koenig and Shao, 2020].

2.3 Certified Abstraction Layers

Koenig and Shao [2020] model certified abstraction layers using categories whose objects are effect signatures and whose morphisms are *strategy specifications*, enriched with a complete refinement lattice structure. Layer interfaces and implementations are both modeled as strategy specifications. Layer correctness can be stated as $L_F \sqsubseteq M \circ L_E$, where $L_E : 1 \rightarrow E$ is the underlay interface, $M : E \rightarrow F$ is the layer implementation, and $L_F : 1 \rightarrow F$ is the overlay interface. However, this elegant picture is complicated by their treatment of state. The set of states used by a layer interface must be encoded as part of its signature, and interactions must follow a “state-passing” discipline. Likewise, the simulation relation used to establish a layer’s correctness must be internalized as a morphism, then composed with the implementation to translate between underlay and overlay states. While Koenig and Shao also explore a model featuring stateful and reentrant strategies, which could in principle realize the encapsulation of state, this comes at the cost of the simplicity and elegance of their main development, and they do not extend their treatment of certified

layers to this setting. They also do not consider layers with concurrency.

By contrast, our approach avoids complex combinations of features by maintaining a strong distinction between layer interfaces and implementations. Layer implementations are two-sided (they both *use* underlay operations and *provide* overlay operations) but they can remain stateless and deterministic. Layer interfaces are stateful, but because they are one-sided the structure of their plays can remain simple. In turn, the statefulness of layer interfaces and our direct approach to formalizing layer correctness mean we do not need an explicit internalization of simulation relations. This allows us to limit our treatment of nondeterminism to *demonic* nondeterminism, which is sufficient to express implementation freedom.

The idea of certified abstraction layers [Gu et al., 2015] was inspired by the systems community’s best practice in using abstraction layers to build large-scale software and hardware systems [Saltzer and Kaashoek, 2009]. Certified abstraction layers rely on using a pair of underlay and overlay interfaces to encapsulate the implementation effects and eliminate undesirable dependencies from other components. Gu et al. [2016] has shown the effectiveness of this approach in building a certified concurrent OS kernel.

Gu et al. [2018] developed *Certified Concurrent Abstraction Layers (CCAL)* and applied them to build a certified concurrent OS kernel [Gu et al., 2016, 2019]. They used game-semantic strategies to model the interaction behavior of each thread (or CPU core) against its environment context, and developed a program logic for reasoning about both the safety and progress of concurrent objects. The marriage of Reddy’s work [Reddy, 1996, 1994] with our layered game semantics offers a promising direction for developing compositional models for CCAL-style shared-memory concurrency. An appealing challenge for future work in that direction will be to articulate the results of Part I with the asynchronous and interactive accounts based on action and footstep trace semantics [Brookes, 2007, 2006] and template game semantics [Melliès and Stefanescu, 2018, 2020] of Concurrent Separation Logic [O’Hearn, 2004].

Similarly, there is a significant body of work on correctness conditions for concurrent programs [Herlihy and Wing, 1990; Filipovic et al., 2010; Cerone et al., 2014; Murawski and Tzevelekos, 2019]. Most notably, Cerone et al. [2014] and Murawski and Tzevelekos

[2019] provide generalizations of linearizability to layers encompassing both an underlay and an overlay, including potentially higher-order computation. As far as we are aware, this is the only work in this line that discusses a notion of layer with underlay and overlay, and we believe that there is an opportunity to connect the ideas from there with our model. This is a promising avenue for future work and will likely involve a connection with concurrent models such as Ghica and Murawski [2004], Cerone et al. [2014], and Murawski and Tzevelekos [2019].

2.4 Linearizability

2.4.1 Classical Linearizability

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

Linearizability was originally proposed by Herlihy and Wing [1990], who proved locality. Many generalizations of it have been proposed [Neiger, 1994; Castañeda et al., 2015; Murawski and Tzevelekos, 2019; Cerone et al., 2014]. Our particular generalization of linearizability is closely related to interval linearizability [Castañeda et al., 2015]. Part IV conservatively generalizes the safety criterion from Part II to take liveness into account, making it strictly more expressive. We use the abstract framework of compositional linearizability as a roadmap to derive our criterion and its metatheory.

2.4.2 Variations of Linearizability

There are numerous works that discuss variations of linearizability [Haas et al., 2016; Neiger, 1994; Hemed et al., 2015; Castañeda et al., 2015]. Notably, in defining a criterion for linearizability in the context of crashes and abortions, Aguilera and Frølund [2003] make use of a rewrite system not unlike the one used by us and [Goubault et al., 2018] (see also §2.5 on linearizability with crashes). 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 developed 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 the original Herlihy-Wing linearizability, which we address in depth in §15.1. 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 §8.1. 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 §15.2 for a detailed comparison.

2.4.3 Linearizability and Observational Refinement

The equivalence between linearizability and observational refinement, which now is a crucial part of linearizability theory, was originally proved in Filipovic et al. [2010]. It has been extended to a generalization of interval linearizability [Castañeda et al., 2015] to handle non-blocking objects in Part II. Interval linearizability has been shown to be a complete criterion for refined distributed tasks [Castañeda et al., 2015; Goubault et al., 2018], and (with a minor modification) for concurrent objects (Part II).

2.4.4 Beyond Linearizability

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). 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. In particular, they establish a Galois connection related to linearizability, which we reproduce using our abstract formulation, as opposed to their proofs, which used a concrete formulation of linearizability. They also do not discuss horizontal composition and locality.

2.5 Linearizability with Crashes

We discussed some of the history of linearizability criteria with crashes throughout Part III [Aguilera and Frølund, 2003; Guerraoui and Levy, 2004; Berryhill et al., 2016; Izraelevitz et al., 2016]. In Part III, we address strict linearizability (in the context of full-system crashes) and durable linearizability. We generalize both of them by not requiring the linearized specifications to be atomic and by allowing for blocking objects. This makes our variations of these linearizability criteria closer to interval-sequential linearizability [Castañeda et al., 2015]. We formulate these criteria in the style of compositional linearizability (Part II), which is novel. This allows us to give simple proofs of locality, develop a compositional verification framework around these criteria, give the first proof of observational refinement properties for these two criteria, and provide a counterpart to the analogous result proved for Herlihy-Wing linearizability [Filipovic et al., 2010] and for compositional linearizability (Ch. 14).

We also discover that the notion of linearizability inherent to crash-aware objects is the linearizability criterion we call crash-aware linearizability, satisfying locality and observa-

tional refinement. Although related to strict linearizability, it does not appear elsewhere. We note that while crash-aware linearizability is the compositional linearizability one gets from our model **Crash**, our formulations of strict and durable linearizability impose new challenges and new structures, in particular, because they relate two distinct models of computation (concurrency with and without crashes). We conjecture that this different structure can be reconciled with that from compositional linearizability through a weakening of the notion of a Grothendieck fibration, following ideas from functorial refinement [Melliès and Zeilberger, 2015].

2.6 Linearizability and Liveness

As far as we are aware, the only other linearizability criteria that preserve progress properties are the linearizability of Gotsman and Yang [2011], and, at least in theory, strong linearizability [Golab et al., 2011]. The former differs from our notion of linearizability in that it does not allow for completions or removals of pending invocations to all traces, instead of disallowing it only for the traces relevant to liveness reasoning. This significantly hinders its abstraction power (it never admits sequential specs). They do not consider matters of compositionality, which are central to our results. Meanwhile, the reason that strong linearizability might reflect progress properties is that it reflects any hyperproperty [Attia and Enea, 2019]. Because of this, it is significantly less expressive than even Herlihy-Wing linearizability, and cannot verify, for instance, the Herlihy-Wing queue [Herlihy and Wing, 1990]. Our more specialized criterion (Part IV), instead, achieves liveness reflection without loss of expressiveness.

2.7 Verification with Crashes

There are approaches for verifying systems with crashes that do not involve linearizability. Much of the work along these lines has been done in the context of file system verification. A notable starting point is the development of Crash Hoare Logic [Chen et al., 2015], later refined into recovery refinement [Chajed et al., 2019a], and generalized to handle concurrent systems [Chajed et al., 2021, 2019b]. Of these, only Chajed et al. [2019a], which

only handles sequential systems, formally proves a refinement theorem that enables building large systems. The later variants that handle concurrency lack such a contextual refinement theorem. These works, different from ours, have been mechanized.

Another important work is Khyzha and Lahav [2022], which proves a contextual refinement theorem for programs with crashes. Quite interesting is the fact that their approach is reminiscent of that used in Part II and by us in Part III, in that they define a notion of refinement by composition with a “Most General Client”. This most general client seems to be a special case of the copycat strategies that appear in our game models. Since they do this using operational semantics, we believe their work is further evidence of the practicality of our approach. Moreover, their programming language features a buffered memory interface with global flushes, which our example does not. Despite the similarities, they only address linearizability by providing a few examples where linearizability specifications can be encoded in their framework, but they do not describe a generic framework to do so, nor prove a formal connection with linearizability. Modeling a buffered memory with global flushes in our framework is straightforward: its specification is almost the same as our buffered memory cell arrays, but with a requirement of proving a memory separation property, like they had to do. We do not do this here as it was not required for our examples.

A recent line of work proves linearizability specifications, but only for a single component [D’Ousualdo et al., 2023], and focuses on data structures implemented on top of NVM only. It is quite impressive in that it assumes a weak memory model, which requires handling weak consistency models, which we do not. Despite that, they do not provide a program logic and are closer to axiomatic approaches, which could hinder scalability. It is unlikely that their framework could be generalized to a compositional verification methodology without significant effort.

Concurrently with our development of Part III, Bodenmüller et al. [2024] verified the FLiT library and have a mechanized proof of correctness. Part of their simulation-based technique is reminiscent of our use of refinement and $\text{dur}(-)$, which they define as a specific transformation of a state-transition system into another and do not note its relationship to the structure of some compositional model (which they do not develop). Their technique is restricted to durable linearizability w.r.t. atomic specifications and is specialized in verifying

persistence libraries over NVM. Our work is, therefore, significantly more general in scope. Our FLiT correctness theorem shows that linearizable objects in the sense of Part II are transformed into durable linearizable libraries in our sense, and therefore applies even to non-atomic and blocking objects, proving a stronger correctness theorem for FLiT (in fact, stronger than the FLiT authors’ informal claim of correctness, for the same reasons).

Our program logic in Part III is the first to verify a linearizability criterion with crashes. It is based on the program logic from Part II and Khyzha et al. [2017], and takes inspiration from Crash Hoare Logic and Argosy [Chajed et al., 2019a]. It differs from the aforementioned works in that it proves durable and crash-aware linearizability specifications. The compositional framework, which we directly connect with our program logic, is the only one that simultaneously provides refinement, linearizability specifications, and vertical and horizontal composition. Our theory allows us to state the correctness of systems like FLiT [Wei et al., 2022]. We also show we can verify a simplified variant of a file system interface. Note that previous file system interfaces are not verified against linearizability specifications, which are deemed more intuitive than the kind of specifications one gets from HOCAP-style specifications [Svendsen and Birkedal, 2014; Dinsdale-Young et al., 2010].

2.8 Program Logics and Verification Methods

2.8.1 Possibility-Based Reasoning and Simulation

Herlihy and Wing [1990] not only proposed the notion of linearizability but also introduced possibility reasoning as an axiomatic method for proving atomic linearizability, and showed it is complete and sound for proofs of linearizability for individual traces. Their original technique has recently been reformulated as a forward simulation and mechanized by [Jayanti et al., 2024]. Their technique focuses only on atomicity, however, and only on verifying a single component, and does not provide the ability to either verify large object systems modularly or compositionally. An approach based on a combination of forward and backward simulations has been shown to be complete [Schellhorn et al., 2014]. Backward simulations are often deemed hard to reason about, however, because they go against the natural forward flow of executions, introduce backward non-determinism, require knowl-

edge of the future, and often require introducing history and prophecy variables to provide information that would be available under a forward technique.

Khyzha et al. [2017] packaged possibility reasoning in a program logic for the first time, and used interval partial orders to represent a set of possibilities of an execution. They also apply the possibilities technique in their rely-guarantee program logic. All of these approaches are restricted to atomic linearized specifications. A modular technique for verifying set linearizability has been proposed by Hemed et al. [2015].

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. Oliveira Vale et al. [2024a] has shown that Khyzha et al. [2017]’s program logic is not complete for atomically linearizable objects, and we speculate that this last modification should make our program logic complete (see Ch. 17 for a counterexample). Since our program logic strictly generalizes theirs, we can translate to our program logic any proof using Khyzha et al. [2017]. The mechanization in §2.9 builds upon this theory, and extends the possibilities with state, making possibility reasoning simpler and more usable than using either partial orders or traces, and embeds this method into a mechanized program logic. Although we use the particular program logic in Ch. 17, 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.

Overall, Part II extends possibility reasoning [Herlihy and Wing, 1990] to a complete axiomatic method for proving compositional linearizability, and provides a rely-guarantee program logic based on the axiomatic method. Part IV extends this with improved possibility design and metrics ensuring liveness. It is sound with respect to live linearizability, subsuming the safety-only version, and allows for simultaneous reasoning about lineariz-

ability and progress.

2.8.2 Concurrent Program Logics

There has been much work on building program logics for reasoning about concurrent programs [Vafeiadis and Parkinson, 2007; Vafeiadis et al., 2006; da Rocha Pinto et al., 2014; Dinsdale-Young et al., 2010; Feng et al., 2007; Fu et al., 2010; Turon et al., 2013; Svendsen and Birkedal, 2014; Nanevski et al., 2014; Jung et al., 2018], including those capable of reasoning about termination-preservation of refinements under fair schedulers [Liang et al., 2014a; Gähler et al., 2022; Lee et al., 2023, 2025]. 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, which 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 their notion of refinement, differs from that in Filipovic et al. [2010]. Notable exceptions in this matter are Khyzha et al. [2017]; Liang and Feng [2016]; Birkedal et al. [2021].

Moreover, as far as we are aware, the contextual refinement approaches cited above all assume a program’s library is atomic, making our observational refinement significantly more general.

2.8.3 Liveness Verification

Among the concurrent program verification works discussed above, some also support liveness verification. Liang et al. [2014a] propose using termination-preserving refinement to specify and verify lock-free objects. It is then extended to the program logic LiLi [Liang and Feng, 2016, 2017] to verify progress properties of blocking objects. They unify safety and liveness verification with contextual refinement as the correctness criterion, which implies linearizability when only considering atomic objects. However, atomicity makes specifying liveness difficult, as divergence is not an atomic behavior and blocking objects that make minimal progress cannot be captured by atomic abstract objects. They have to wrap the abstract operation in loops to make it non-atomic and then show correctness against this

wrapped abstract object. This makes their approach less compositional as these complicated wrapped objects are not intuitive, and composing above them makes the client harder to verify. We build our theory above compositional linearizability (Part II) without the dependency on atomicity, making it capable of encoding divergence and our liveness specification more intuitive and compositional.

D’Oswaldo et al. [2021] extends logical atomicity [Birkedal et al., 2021] in [da Rocha Pinto et al., 2014] to support termination verification. Their logical atomicity triple (LAT) ensures atomicity of an overlay operation and also termination. By guarding LAT with obligation lists, they specify the requirements on other threads and clients for the current operation to terminate and solve the termination dependency among threads. Their approach is compositional in that the prover can reuse a LAT in its overlay’s proof. However, their work suffers from several major drawbacks. Firstly, their LAT is only designed for specifying termination, and lacks the ability to formulate more general liveness properties that depend on non-termination and cannot handle different scheduler assumptions. Secondly, their obligation lists and LAT specifications depend not only on the syntax of a programming language, but also the semantics of the language and the program logic, making their approach less scalable. Lastly, it suffers from the same drawbacks discussed for other approaches reliant on atomicity.

Another line of work on liveness verification is based on fair operational semantics [Lee et al., 2023] and Lilo [Lee et al., 2025]. Fair operational semantics [Lee et al., 2023] introduces fairness events and a fairness map to the programming language and its semantics, and defines a simulation that establishes progress-preservation between the target and source programs. By inserting fairness events into a program, their semantics can interpret the liveness properties indicated by those events and thus specify more general liveness properties. This makes their specifications tightly bound to the semantics of the fairness events, as without them, the specification does not carry any information about the liveness property. In comparison, our trace-based specification only depends on the object signature, and our compositional theory assumes nothing beneath the specification when composing objects and proofs. Lilo [Lee et al., 2025] is a higher-order relational concurrent separation logic for liveness that builds upon the fair operational semantics. It is equipped with tools like

progress credits, obligation lists, and promises that make their proofs more thread-local and modular. They also verified the safety and liveness of the elimination-backoff stack, but with exchanger code inlined in the overlay stack, as they only handle atomic objects. This indicates a lower degree of compositionality as they need to inline non-atomic objects when verifying other objects using them, making the verification more monolithic, while we can directly build proofs above non-atomic objects.

Some of these works [Liang et al., 2014b; Liang and Feng, 2016; Lee et al., 2025] also utilize the idea that failed operations are caused by successful ones to show progress. However, they have to introduce an auxiliary variable as a progress metric that can be delayed by the successful operations. Maintaining these auxiliary variables along with other progress metrics is a burden for the prover and introduces unnecessary code. Our approach directly formalizes this idea using set-linearizable specifications, and the compositionality of our framework ensures the safety and liveness of the overlay object intrinsically.

Nevertheless, all these works present more mature program logics than ours. In particular, concurrent-separation-logic approaches are effective at verifying higher-order programs. It is noteworthy that Lilo [Lee et al., 2025] has been mechanized, and since this was done on top of Iris [Jung et al., 2015, 2018], Lilo can enjoy the robust tooling in Iris. In contrast, Part IV presents LLHL mainly to show that our new correctness criterion can be verified in practice. Extensive future work is needed to develop and mechanize more robust verification technology for safe and live linearizability.

2.8.4 Logical Atomicity

Logical atomicity [da Rocha Pinto et al., 2014; Jung et al., 2018, 2019] uses logically atomic Hoare triples $\langle P \rangle \text{op}() \langle Q \rangle$ to specify the ability of the given operation to perform the update abstracted as a transition from assertion P to assertion Q at a single physically atomic statement. It was invented to make it easier to prove linearizability in Hoare logics. Since its focus is on atomicity, it cannot generally specify set and interval linearizable objects satisfactorily. It is compositional, as one may directly use the logical atomic Hoare triple in proofs of other functions that use this operation, although this compositionality is limited to within the confines of a particular program logic framework such as Iris [Jung et al., 2015].

It has recently been shown [Birkedal et al., 2021] that logical atomicity implies Herlihy-Wing linearizability. There is no reason to believe the reverse implication is provable, so its completeness is unknown. Moreover, logical atomic triples rely on a concrete assertion language, a concrete programming language, and their underlying semantic models. Strictly speaking, logical atomicity only characterizes objects realizable in a particular operational semantics and expressible in a particular program logic. 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. Our approach can specify any object using an abstract LTS, without assuming the actual semantic model or the program logic verifying the object against the specification. Our treatment in Part II characterizes sequentially consistent concurrent objects whose operations behave as if they had linearization intervals.

2.8.5 Contextual Refinement

Contextual refinement provides an alternative to linearizability. It uses an abstract piece of code A_E to specify the concrete implementation M_E by requiring that the behavior of M_E running in any context C is bounded by the behavior of A_E running in the same context, usually denoted as $M_E \sqsubseteq A_E$. Contextual refinement is equivalent [Filipovic et al., 2010] to Herlihy-Wing linearizability and supports vertical composition in a sequential setting [Song et al., 2023]. However, contextual refinement relies on a concrete programming language for the abstract code, and an embedding of this language into the original language for the concrete code is necessary for composing a concrete context with an “abstract” specification. In contrast, our framework only uses the signature of an object and applies to any programming language whose behaviors are characterized by call and return events, without any modification to the language. This is a major benefit of verifying objects directly against linearizability specifications. Moreover, there has been no work to support modular verification of non-atomic objects like the exchanger and the interval-linearizable write-snapshot within the contextual refinement framework.

2.8.6 Relational Hoare Logic

Relational Hoare logic [Benton, 2004] is widely used to show contextual refinement. Liang and Feng [2013] used local rely-guarantee reasoning to implement a relational logic that verifies the contextual refinement of concurrent objects w.r.t. their atomic abstract code. Xu et al. [2016] later applied this technique to verify an operating system using their relational program logic. They achieved this using logically atomic triples [da Rocha Pinto et al., 2014] to establish a contextual refinement between the two. Their work allows modular verification, but with limited encapsulation: an object’s abstract code A_F must also include all its sub-components’ abstract code A_E . They can verify sub-components against corresponding abstract code A_E and use their Hoare triples to execute A_E embedded in A_F . Consequently, the top-level specification consists of very complicated abstract code that does not intuitively specify its behaviors. In contrast, our LTS specification encapsulates all details of the underlay object and provides an implementation-agnostic interface for users to verify overlay objects.

Turon et al. [2013] encodes the contextual refinement obligation into a ghost state to verify it with CSL. Iris [Jung et al., 2018] is a higher-order logic for verifying fine-grained concurrent programs and also features logical atomicity. It leads to the ReLoC framework [Frumin et al., 2018, 2021; Vindum et al., 2022], which shows contextual refinement of concurrent programs using a program logic, with prophecy variables [Jung et al., 2019] enabling future-dependent linearization. ReLoC achieves compositionality by first encapsulating reusable code into smaller functions, then verifying these functions against logically atomic Hoare triples encoded as a logical relation, and lastly verifying the refinement of the top-level object using these intermediate proofs. However, it is unknown how to reuse the top-level refinement specification when verifying objects that use this top-level object. As a result, every time they build new objects using the one verified against the refinement specification, they need to prove a different functional correctness specification before they can reuse its proof. By contrast, our framework imposes LTS specifications uniformly at any level of abstraction.

Both lines of work have verified the elimination-backoff stack but failed to achieve the

same level of compositionality as we do. Liang and Feng [2013] directly inlined all underlay code in the **EBStack** code. ReLoC [Frumin et al., 2018] broke down the exchanger into three atomic operations: `make_offer`, `accept_offer`, and `revoke_offer`, exposing implementation details, and implemented the **EBStack** using these operations. They built and verified the **EBStack** using these three methods and their specifications. This effectively inlines a specific protocol for implementing the exchanger object within the stack implementation. As a result, to verify other objects using the exchanger, nothing except the specifications for these low-level atomic operations can be salvaged and reused from their proof. They still need to inline all the implementation code of the exchanger in the overlay object. In contrast, our approach allows direct composition above the ready-to-use exchanger implementation and specification.

2.9 Mechanization

This dissertation provides the first mechanized proof of linearizability of the one-shot write-snapshot [Castañeda et al., 2015] object against its interval-sequential specification, and we believe it also provides the first truly modular proof of the elimination-backoff stack.

Regarding mechanization, the closest work to ours is [Lesani et al., 2022]. It focuses on using linearizability to verify serializability. Our encoding of modules as interaction trees is the same as theirs, and some aspects of their treatment of atomic linearizability are compatible with ours. Despite that, their framework is restricted to atomicity. This means that their LTSs pair call and return events and cannot handle non-atomic objects. As a result, they cannot compute overlay objects $(\nu ; M)$, as these can be non-atomic even if ν is atomic, so they define an object as a pair of an underlay (atomic) LTS with a module instead. Meanwhile, an object for us is just an LTS. While they prove several compositional properties of the modules, which boil down to properties of interaction trees, we prove several more properties related to computing overlay objects, monotonicity w.r.t. trace refinement, etc. These enable the compositional linearizability framework and make up a significant portion of our mechanization. Due to the compatibility of their approach with ours, we believe that their results around encoding serializability as atomic linearizability

could be carried over.

Chapter 3

State Transition Systems

Perhaps the most common way of modeling systems, across all corners of computer science, is with labelled state transition systems (LTS). This chapter shows how our approach to modeling systems relates to state transition systems. I will avoid introducing all the formalism necessary to fully develop this chapter, as it serves mostly as a stepping stone toward the actual technical development of the dissertation. I hope this will allow the reader familiar with traditional approaches to system modeling to ease into my approach inspired by game semantics.

3.1 Labelled Transition Systems

Traditionally, a labelled transition system $\mathcal{S}$ consists of a quadruple $((S, L, - \xrightarrow{\quad} -), s_0)$ where: S is a set of *states* $s \in S$, L is a set of labels $l \in L$, and $- \xrightarrow{\quad} -$ is a relation $- \xrightarrow{\quad} - \subseteq S \times L \times S$, together with an initial state $s_0 \in S$.

Example 3.1.1. For example, a (sequential) stack object may be specified by the transition system $\mathcal{S}_{\text{Stack}}$ with:

- states taken as $S_{\text{Stack}} := \text{Val}^* \times \text{Control}$, that is, a finite sequence $s \in \text{Val}^*$ of values together with some state used for control $\text{Control} := (\{\text{push}(v) \mid v \in \text{Val}\} \uplus \{\text{pop}()\}) + \perp$, as shall be explained shortly;
- labels taken from the set $L_{\text{Stack}} := \{\text{push}(v) \mid v \in \text{Val}\} \uplus \text{ok} \uplus \{\text{pop}()\} \uplus (\text{Val} + \perp)$. Here,

$\text{push}(v)$ and $\text{pop}()$ are invocation events, and ok is the response to a push invocation, and $v \in \text{Val}$ and $\perp$ are the possible responses to pop ($\perp$ is returned on an empty stack);

- the relation $-\xrightarrow{\text{Stack}}-$ is defined by the following transitions:

$$\begin{aligned} (s, \perp) &\xrightarrow{\alpha:\text{push}(v)}_{\text{Stack}} (s, \text{push}(v)) & (s, \text{push}(v)) &\xrightarrow{\text{ok}}_{\text{Stack}} (v \cdot s, \perp) \\ (s, \perp) &\xrightarrow{\alpha:\text{pop}()}_{\text{Stack}} (s, \text{pop}()) & (v \cdot s, \text{pop}()) &\xrightarrow{v}_{\text{Stack}} (s, \perp) & (\epsilon, \text{pop}()) &\xrightarrow{\perp}_{\text{Stack}} (\epsilon, \perp) \end{aligned}$$

- its initial state is $(\epsilon, \perp)$.

We make an important observation: here, we already make a significant departure from most of the literature in the way we model a sequential stack. Typically, $\text{push}(v)$ and its response ok would be a single event $\text{push}(v).\text{ok}$, and similarly for pop . While this is viable, for our purposes we strictly separate the events that are due to the environment and those due to the system. This has a deep impact on our modeling and is inspired by the literature on game semantics. This choice, however, does make it easier to give a uniform account of sequential and concurrent computation, as in the latter separating invocations and responses is essential.

The control state is used to ensure that the transitions alternate between invocations and responses. When the control state is $\perp$, it means the stack is in a system position, i.e., the system is satisfied and waiting for its environment to make an invocation. When the control state is $\text{push}(v)$ or $\text{pop}()$, then the stack is in an environment position, where the environment is satisfied and waiting for the system to provide a response. The system uses the choice of invocation from the environment to decide how to modify the state, which follows the usual semantics of a stack.

3.2 Synchronizing Product

It is common to assume the set L comes from some universe U of labels, i.e., any transition system under consideration by a theory would assume $L \subseteq U$. This essentially set-theoretic assumption has implications for how operations on labelled transition sys-

tems are defined. The traditional way of composing transition systems is by what might be called the *synchronizing product* of transition systems [Arnold and Nivat, 1982]. Let $\mathcal{S} = ((S_{\mathcal{S}}, L_{\mathcal{S}}, - \xrightarrow{\quad} -, s_0)$ and $\mathcal{T} = ((S_{\mathcal{T}}, L_{\mathcal{T}}, - \xrightarrow{\quad} -, t_0)$, then their synchronizing product is given by $\mathcal{S} * \mathcal{T} := ((S_{\mathcal{S}*\mathcal{T}}, L_{\mathcal{S}*\mathcal{T}}, - \xrightarrow{\quad} -, (s_0, t_0))$, where:

$$S_{\mathcal{S}*\mathcal{T}} := S_{\mathcal{S}} \times S_{\mathcal{T}} \qquad L_{\mathcal{S}*\mathcal{T}} := L_{\mathcal{S}} \cup L_{\mathcal{T}}$$

and the transition relation is defined by:

$$\frac{s \xrightarrow{l}_{\mathcal{S}} s' \quad t \xrightarrow{l}_{\mathcal{T}} t'}{(s, t) \xrightarrow{l}_{\mathcal{S}*\mathcal{T}} (s', t')} \qquad \frac{s \xrightarrow{l}_{\mathcal{S}} s' \quad l \notin L_{\mathcal{T}}}{(s, t) \xrightarrow{l}_{\mathcal{S}*\mathcal{T}} (s', t)} \qquad \frac{l \notin L_{\mathcal{S}} \quad t \xrightarrow{l}_{\mathcal{T}} t'}{(s, t) \xrightarrow{l}_{\mathcal{S}*\mathcal{T}} (s, t')}$$

the initial state of $\mathcal{S} * \mathcal{T}$ is the pair (s_0, t_0) .

The synchronizing product $\mathcal{S} * \mathcal{T}$ synchronizes transitions in the intersection $L_{\mathcal{S}} \cap L_{\mathcal{T}}$, taking a transition in both $\mathcal{S}$ and $\mathcal{T}$ at the same time. Otherwise, it allows for transitions to be taken only in $\mathcal{S}$ or $\mathcal{T}$.

Intuitively, $- * -$ should be associative, and the transition system $\mathcal{S}_1 := ((\{\bullet\}, \emptyset, \emptyset), \bullet)$ the unit for $- * -$. However, this is clearly not true on the nose. For example, the set of states for $(\mathcal{S} * \mathcal{T}) * \mathcal{U}$ is $(S_{\mathcal{S}} \times S_{\mathcal{T}}) \times S_{\mathcal{U}}$, while for $\mathcal{S} * (\mathcal{T} * \mathcal{U})$ it is $S_{\mathcal{S}} \times (S_{\mathcal{T}} \times S_{\mathcal{U}})$. The more accurate statement is that $- * -$ is associative and unital up-to-equivalence, but what should be taken as the notion of equivalence?

3.3 Behavioral Equivalence

Given a transition system $\mathcal{S} = ((S, L, - \xrightarrow{\quad} -, s_0)$, we define its set of traces at $s \in S$ by

$$\mathcal{S}^\#s := \{t \in L^* \mid \exists ss \in S^*.ss[0] = s \wedge \forall n \in \mathbb{N}.ss[n] \xrightarrow{t[n]} ss[n+1]\}$$

where, for a set A , $A^* := A^* \cup A^\infty$ (the set of countable sequences over A), and for a sequence $s \in A^*$, $s[i]$ is the i -th entry in the sequence.

We say two transition systems $\mathcal{S} = (S, L, - \xrightarrow{\quad} -)$ and $\mathcal{S}' = (S', L, - \xrightarrow{\quad} -)$ over

the same set of labels L are *behaviorally equivalent*, written $\mathcal{S} \sim \mathcal{S}'$, when $\mathcal{S} \# s_0 = \mathcal{S}' \# s'_0$. It is straightforward that this defines an equivalence relation on state-transition systems with the same set of labels L .

Example 3.3.1. For example, for a set of values Val , we may define the set $\text{LinkedList} \subseteq (\mathbb{N} \times ((\text{Val} \times \mathbb{N}) + \perp))^\infty$ as the set of pairs (l_0, m) of a location l_0 together with a sequence $m \in ((\text{Val} \times \mathbb{N}) + \perp)^\infty$ satisfying a few conditions. First, m must have finite support (i.e., there are finitely many locations such that $m[i] \neq \perp$). Moreover, define the sequence of locations $ll_n \in \mathbb{N} + \perp$ by $ll_0 := l_0$ and $ll_n := m[ll_{n-1}][1]$ if $m[ll_{n-1}]$ is defined, and $ll_n = \perp$ otherwise. Then, we require that there is a k such that $\forall i < k$, ll_i is defined and distinct from all other ll_j for $j < k$, and for $i \geq k$, $ll_i = \perp$. In other words, starting at $m[l_0]$ and then chasing the locations stored in the previous cell as if it were a pointer, we find a finite sequence of values.

Then, we may define a transition system $\mathcal{S}'_{\text{Stack}}$ with states $\text{LinkedList} \times \text{Control}$, labels L_{Stack} , and transitions described by (where $m[l \mapsto x]$ is the sequence that is equal to m in all locations except at l , where it is equal to x):

$$\begin{array}{c}
\frac{m[l'_0] = \perp}{((l_0, m), \text{push}(v)) \xrightarrow{\text{ok}}_{\text{Stack}} ((l'_0, m[l'_0 \mapsto (v, l_0)]), \perp)} \\
((l_0, m), \perp) \xrightarrow{\alpha: \text{push}(v)}_{\text{Stack}} ((l_0, m), \text{push}(v)) \\
((l_0, m), \perp) \xrightarrow{\alpha: \text{pop}()}_{\text{Stack}} ((l_0, m), \text{pop}()) \\
\frac{m[l_0] = (v, l_1)}{((l_0, m), \text{pop}()) \xrightarrow{v}_{\text{Stack}} ((l_1, m[l_0 \mapsto \perp]), \perp)} \quad \frac{m[l_0] = \perp}{((l_0, m), \text{pop}()) \xrightarrow{\perp}_{\text{Stack}} ((l_0, m), \perp)}
\end{array}$$

the initial state may be taken to be any pair (l_0, m_0) such that $m_0[l_0] = \perp$, say $m_0[i] = \perp$ for every i .

It is possible to show that $\mathcal{S}_{\text{Stack}} \sim \mathcal{S}'_{\text{Stack}}$ for such a choice of initial state (that is, any m_0 such that $m_0[l_0] = \perp$).

We see then that behavioral equivalence is insensitive to the state representation. This is enough to be able to state that, for any state transition systems $\mathcal{S}$, $\mathcal{T}$, $\mathcal{U}$, and states

$s \in S_S$, $t \in S_T$, and $u \in S_U$ ¹:

$$(\mathcal{S} * \mathcal{T}) * \mathcal{U} \sim \mathcal{S} * (\mathcal{T} * \mathcal{U}) \qquad \mathcal{S} * \mathcal{S}_1 \sim \mathcal{S}$$

In fact, using the theory of coalgebras [Rutten, 2000; Aczel, 1988], it could be shown that the mapping $-\sharp-$ is rather special in that it satisfies quite a strong universal property. With a careful setup, it would be the unique mapping into the final coalgebra for the type of coalgebras that capture the dynamics of the LTS we describe here (i.e., for the right choice of functor F). I am intentionally vague to avoid digressing into coalgebra theory. The takeaway, however, is that behavioral equivalence is the same as equality after mapping the LTS through this canonical mapping, providing some formal justification for behavioral equivalence as the notion of equivalence we might want to use.

3.4 Abstraction

We just noted how behavioral equivalence is insensitive to state representation. This reminds us of our goals in modeling objects: we seek to have a good notion of abstraction. If we define $\mathcal{S}' \lesssim \mathcal{S}$ to mean $\mathcal{S}'\sharp s'_0 \subseteq \mathcal{S}\sharp s_0$, then we obtain our base notion of refinement, which we call behavioral refinement (at times, simply refinement). First, it defines a pre-order on LTSs with the same set of labels, as expected of refinement. In fact, we could even show that:

$$\mathcal{S}' \lesssim \mathcal{S} \implies \forall \mathcal{T}. \mathcal{S}' * \mathcal{T} \lesssim \mathcal{S} * \mathcal{T}$$

It is also possible to show that

$$\forall \mathcal{T}. \mathcal{S}' * \mathcal{T} \lesssim \mathcal{S} * \mathcal{T} \implies \mathcal{S}' \lesssim \mathcal{S}$$

by instantiating $\mathcal{T}$ with $\mathcal{S}_1$. This shows that $-\lesssim-$ is equivalent to contextual refinement (when our algebra of components only includes the operation $-\ast-$). Mathematically

1. It would have been enough to state these equations up-to isomorphism (invertible labelled transition system homomorphism) of state transition systems. We prefer to state it under the weaker notion of behavioral equivalence as it better suits our goals.

speaking, the first direction above is monotonicity of $- * -$ w.r.t. $-\lesssim-$, while the second property shows it is in fact an order-embedding (an order-preserving and order-reflecting mapping). This is a first inkling of how we might formalize these ideas in a rather generic setting.

3.5 Encapsulation

So far, the traditional formalism has served us well, except perhaps for the annoyance of working up-to equivalence, as opposed to up-to equality. However, the synchronization product is not compatible with our take on encapsulation, as $\mathcal{S} * \mathcal{T}$ still leaves the internal data available for other components to synchronize on (that is, a third component can still synchronize on the labels in $L_{\mathcal{S}} \cap L_{\mathcal{T}}$).

For example, say we implement a “reverser” object by using a stack as follows: on an invocation of `reverse(s)` for some string of values $s \in \text{Val}^*$, the implementation pushes each character of s from head to tail, and then pops the stack until it is empty while storing locally the sequence of characters that was popped. This could be depicted by the following program:

```

M:Reverser
Import S:Stack

def reverse(s : Val*) {
  for i in len(s) {
    S.push(s[i]);
  };
  r <- ε;
  for i in len(s) {
    v <- S.pop();
    r <- r · v;
  }
  ret r
}

```

This implementation could be modeled by a transition system $\mathcal{M}$ by using, say, some operational semantics. We avoid giving the detailed definition. Instead, we remark that such a transition system would take as labels $L_{\text{Stack}} \uplus L_{\text{Reverser}}$ where $L_{\text{Reverser}} := (\{\text{reverse}(s) \mid$

$s \in \text{Val}^*\} + \text{Val}^*)$. That is, it transitions on events of either the object it implements (the reverser) or the object it uses (the stack). Note, though, that we would expect that $\mathcal{S}_{\text{Stack}} * \mathcal{M}$ gives us a reverser object. Indeed, we have linked the Stack object needed by $\mathcal{M}$ to implement the reverser. However, $\mathcal{S}_{\text{Stack}} * \mathcal{M}$ is an LTS with labels in $L_{\text{Stack}} \cup (L_{\text{Stack}} \cup L_{\text{Reverser}}) = L_{\text{Stack}} \cup L_{\text{Reverser}}$. But a reverser object should not have events such as $\text{push}(v)$ and $\text{pop}()$, it should only have $\text{reverse}(s)$ and its response. Moreover, say we have an incorrectly implemented stack object $\mathcal{S}'_{\text{Stack}}$. Then, even after we compose $\mathcal{S}_{\text{Stack}} * \mathcal{M}$, we can still compose $\mathcal{S}'_{\text{Stack}} * (\mathcal{S}_{\text{Stack}} * \mathcal{M})$, creating the odd situation where the reverser simultaneously uses a correct and an incorrect stack, and only outputs in the special cases where both agree on their executions. This is problematic for our goals of compositional verification.

What should happen is that when we compose an implementation such as $\mathcal{M}$ with a stack, we should obtain a reverser object where the underlying data (the stack) is fully encapsulated *within* the reverser object, and not available for others to inspect unless intentionally exposed by the interface of the Reverser (which is not the case). This justifies a different approach to composition, and in fact, to how we manage our labels.

3.6 Revisiting Labelled Transition Systems and Composition

The key motto for the kind of composition I need to model objects will be “synchronization + hiding”. This is a point learned from game semantics, which, as far as I am aware, has coined the motto to describe strategy composition². To achieve this, we find it convenient, if not required, to use a slightly refined formalism for LTSs.

First, we find it useful to consider LTSs with silent steps. A transition system with silent steps consists of the usual data for a transition system $\mathcal{S} = ((S, L, - \xrightarrow{\quad} -), s_0)$ except that $- \xrightarrow{\quad} - \subseteq S \times (L + \epsilon) \times S$, where ϵ denotes a silent transition. It is also common and useful to assume that for every state s , $s \xrightarrow{\epsilon} s$. This assumption will not play a role in our informal discussion, however. Silent transitions are so called because they are not considered visible.

2. Note that the separate ideas of *synchronizing product* and *hiding* pre-exist game semantics [Milner, 1980].

This is reflected in an updated definition of our behavior-assigning map. First, given a transition relation with silent steps $\xrightarrow{-}$, we define its ϵ -closure $\xRightarrow{-} \subseteq S \times L \times S$ as the relation inductively defined by:

$$\frac{l \in L \quad s \xrightarrow{l} s'}{s \xRightarrow{l} s'} \quad \frac{s \xRightarrow{l} s' \quad s' \xrightarrow{\epsilon} s''}{s \xRightarrow{l} s''} \quad \frac{s \xrightarrow{\epsilon} s' \quad s' \xRightarrow{l} s''}{s \xRightarrow{l} s''}$$

That is, $s \xRightarrow{l} s'$ if and only if there is a sequence $s \xrightarrow{(\epsilon)^*} t \xrightarrow{l} t' \xrightarrow{(\epsilon)^*} s'$. Then, the set of traces of an LTS $\mathcal{S}$ with silent steps at $s \in S$ is given by:

$$\mathcal{S}\#s := \{t \in L^* \mid \exists ss \in S^*. ss[0] = s \wedge \forall n \in \mathbb{N}. ss[n] \xRightarrow{t[n]} ss[n+1]\}$$

We also introduce a notion of typing for LTSs. We will say an LTS $\mathcal{S}$ is of type $A \rightarrow B$, written $\mathcal{S} : A \rightarrow B$, when its set of labels is $A + B$. Note that by taking $\mathbf{1} := \emptyset$, we have that $\mathcal{S} : \mathbf{1} \rightarrow B$ are in a bijective correspondence with LTSs labelled by B . Given LTS $\mathcal{S} : A \rightarrow B$ and $\mathcal{T} : B \rightarrow C$, we define their composition $\mathcal{S} \odot \mathcal{T} : A \rightarrow C$ by the following data:

- its set of states $S_{\mathcal{S} \odot \mathcal{T}}$ is exactly $S_{\mathcal{S}} \times S_{\mathcal{T}}$, and the initial state is (s_0, t_0)
- the set of labels is defined already by the typing: $A + C$,
- its transitions are given by:

$$\frac{s \xrightarrow{\text{inl } a}_{\mathcal{S}} s' \quad a \in A}{(s, t) \xrightarrow{\text{inl } a}_{\mathcal{S} \odot \mathcal{T}} (s', t)} \quad \frac{t \xrightarrow{\text{inr } c}_{\mathcal{T}} t' \quad c \in C}{(s, t) \xrightarrow{\text{inr } c}_{\mathcal{S} \odot \mathcal{T}} (s, t')}$$

$$\frac{b \in B \quad s \xrightarrow{\text{inr } b}_{\mathcal{S}} s' \quad t \xrightarrow{\text{inl } b}_{\mathcal{T}} t'}{(s, t) \xrightarrow{\epsilon}_{\mathcal{S} \odot \mathcal{T}} (s', t')}$$

Other than the change in the format of the labels of LTSs, the key difference between synchronizing product and composition of LTSs $(- \odot -)$ is that in the case where $\mathcal{S}$ and $\mathcal{T}$ take a synchronous B -action, the action becomes a silent step in the composition. This

could be formalized using the synchronizing product followed by a hiding operation. First, we would coerce both $\mathcal{S}$ and $\mathcal{T}$ into transition systems with labels in $A + B + C$ (a relabelling homomorphism). Then, we would take their synchronizing product, followed by hiding (a relabelling where all the B -actions are mapped to the silent transition label ϵ). While such a formulation makes the motto “synchronization + hiding” evident, it would be significantly lengthier, so instead we define the composition operation directly. Versions of the operation $- \odot -$ will appear throughout this dissertation, but all of them have this flavor of “synchronization + hiding”. In the context of systems verification, we will call it vertical composition, in other contexts, it will be called categorical composition, or just composition. Composition is associative up-to behavioral equivalence. With the setup I have described, it is not unital, a problem we will deal with later.

Example 3.6.1. Revisiting our informal example, with some relabelling, we can type $\mathcal{S}_{\text{Stack}} : \mathbf{1} \rightarrow L_{\text{Stack}}$ and $\mathcal{M} : L_{\text{Stack}} \rightarrow L_{\text{Reverser}}$. Then, their composition $\mathcal{S}_{\text{Stack}} \odot \mathcal{M} : \mathbf{1} \rightarrow L_{\text{Reverser}}$ makes it evident already from the typing that the stack has been encapsulated. In fact, the only visible events are **Reverser** events, and moreover $\mathcal{S}_{\text{Stack}} \odot \mathcal{M} : \mathbf{1} \rightarrow L_{\text{Reverser}}$ can now only be composed with components $\mathcal{N} : L_{\text{Reverser}} \rightarrow L$.

It is therefore by the combination of comparing LTSs only for behavioral refinement, and by the change in the algebraic structure of the LTS model, through using the idea of synchronization followed by hiding, that we obtain the kind of strong encapsulation and abstraction principles that we seek for a proper modeling of objects.

3.7 Tensor

There is another important operation for composing components. Say we wanted to implement a stack object which also supports a length operation. This could easily be done if we had access to both a stack and a counter. In order to treat situations where we need many objects simultaneously, we could instead assume we have a single object providing the interfaces of a stack and a counter in such a way that operations on the stack do not interfere with operations on the counter, and vice-versa. In other words, given objects $\mathcal{S}_{\text{Stack}}$

and $\mathcal{S}_{\text{Counter}}$, we want an object $\mathcal{S}_{\text{Stack}} \otimes \mathcal{S}_{\text{Counter}}$ which behaves like having a stack and a counter.

Indeed, given LTS $\mathcal{S} : A \rightarrow B$ and $\mathcal{T} : C \rightarrow D$, we define their tensor $\mathcal{S} \otimes \mathcal{T} : A + C \rightarrow B + D$ by the following data:

- its set of states $S_{\mathcal{S} \otimes \mathcal{T}}$ is given by the product $S_{\mathcal{S}} \times S_{\mathcal{T}}$, with initial state (s_0, t_0) ,
- labels are implied by the typing, i.e., $(A + C) + (B + D)$
- its transitions are given by:

$$\frac{s \xrightarrow{l}_{\mathcal{S}} s'}{(s, t) \xrightarrow{l}_{\mathcal{S} \otimes \mathcal{T}} (s', t)} \qquad \frac{t \xrightarrow{l}_{\mathcal{T}} t'}{(s, t) \xrightarrow{l}_{\mathcal{S} \otimes \mathcal{T}} (s, t')}$$

where we make a slight abuse of notation, as some reindexing would be necessary to take $l \in A + B$ or $l \in C + D$ into $(A + C) + (B + D)$.

The tensor may therefore be understood as an asynchronous product of transition systems. In the context of systems verification, we will call the tensor *horizontal composition*. Note that now, a thorough contextual refinement theorem would require showing that both $-\odot-$ and $-\otimes-$ are monotonic with respect to behavioral refinement $-\lesssim-$.

3.8 Well-Formedness

I have ignored so far an important issue. Recall that our LTS $\mathcal{S}_{\text{Stack}}$ makes use of some control state to ensure sequentiality, despite having separate events for invocations and responses. Suppose now we compute the tensor $\mathcal{S}_{\text{Stack}} \otimes \mathcal{S}_{\text{Stack}}$. If we assume we work under a model of sequential computation, then we wish $\mathcal{S}_{\text{Stack}} \otimes \mathcal{S}_{\text{Stack}}$ to still be a sequential object. That is, we expect that it strictly alternates between invocation and response transitions. This is, however, not the case. From the initial state, by definition, the following sequence of transitions is allowed:

$$((\epsilon, \perp), (\epsilon, \perp)) \xrightarrow{\text{inr inl pop()}} ((\epsilon, \text{pop()}), (\epsilon, \perp)) \xrightarrow{\text{inr inr pop()}} ((\epsilon, \text{pop()}), (\epsilon, \text{pop()}))$$

This is not a sequential behavior. The issue is that the composition operations we have defined do not take into account that the domain of LTSs we have in mind has extra well-formedness conditions.

There are several ways to approach this issue. An elegant and thorough treatment is provided by Mellès [2019], or the coalgebraic approach of [Zhang et al., 2025]. We consider here two simpler, although less elegant, solutions. One simple solution, when it works, is to modify the trace mapping. For example, we could define a mapping from transition systems to their sequential traces:

$$\text{seqtraces}_s(\mathcal{S}) = \{t \in \mathcal{S}^\#_s \mid t \text{ is sequential}\}$$

Implicitly, this is the approach I take in most of my models and it is common in the game semantics literature.

In our mechanization, however, we take a slightly different approach. Instead, we modify the composition operations on LTSs so that they also add some control state to the composed system. The extra control state enforces the necessary well-formedness condition (usually local sequentiality). In this way, working only with LTSs that are (locally) sequential, and composing them, preserves their (local) sequentiality. In particular, with such a setup $\mathcal{S}_{\text{Stack}} \otimes' \mathcal{S}_{\text{Stack}}^\#((\epsilon, \perp), (\epsilon, \perp)) = \text{seqtraces}_{((\epsilon, \perp), (\epsilon, \perp))}(\mathcal{S}_{\text{Stack}} \otimes' \mathcal{S}_{\text{Stack}})$ (where $- \otimes' -$ is the tensor operation modified in this way). This solution, although not so elegant, is workable because of our commitment to abstraction and encapsulation: we do not care much that we are polluting the state with extra, often redundant, control state whenever we compose components.

3.9 Trace Semantics

Throughout this chapter, I have established my goal of working with objects at least up-to behavioral equivalence, if not even more permissive notions of equivalence. As I remarked, behavioral equivalence is insensitive to data representation. A natural consequence of this, then, is that we do not need to work with state at all! Indeed, by the substitution principle

implied by contextual refinement, two equivalent transition systems $\mathcal{S} \sim \mathcal{S}'$ may be replaced freely by each other in any context where they appear. If we just picked a canonical representative of each equivalence class of LTSs, we could work instead up-to equality.

Fortunately, this can be done. First, for any $\mathcal{S}$, we can assign an LTS $\text{Traces}(\mathcal{S})$ whose states are non-empty prefix-closed sets of traces, and labels are the same as $\mathcal{S}$. Transitions are given by Brzowski derivatives (assuming the derivative is non-empty):

$$T \xrightarrow{l} \{t \in L_{\mathcal{S}}^* \mid l \cdot t \in T\}$$

and the initial state is $\mathcal{S}\sharp s_0$. Then, it holds that $\text{Traces}(\mathcal{S}) \sim \mathcal{S}$. But note that $\text{Traces}(\mathcal{S})$ makes no reference to the original state representation $S_{\mathcal{S}}$ used by $\mathcal{S}$.

This representation is still redundant in that we are using a *state* transition system, even though we should not need any particular data representation. We can refine this by instead working directly with the set $\mathcal{S}\sharp s_0$ as our canonical representation for transition systems that are observationally equivalent to $\mathcal{S}$. We can always retrieve $\text{Traces}(\mathcal{S})$ from this representation as follows. A trace $s \in \mathcal{S}\sharp s_0$ represents the set of all states reachable by taking the path s from s_0 in $\mathcal{S}$. Then, we may compute a corresponding set of residual traces of s , that is, $\{t \in L_{\mathcal{S}}^* \mid s \cdot t \in \mathcal{S}\sharp s_0\}$. Transitions are as before. This approach identifies an object exactly with the set of behaviors we are interested in. It has the advantage that to compare two objects, now represented as sets of traces, it is enough to compare the sets for equality $- = -$, with no need to work up-to behavioral equivalence $- \sim -$. Similarly, behavioral refinement $- \lesssim -$ becomes just subset inclusion $- \subseteq -$. This greatly simplifies the mathematics we need.

Now, we must still define how to do vertical and horizontal composition, but now directly on traces. While we do not define them here, we note that they are defined so as to satisfy equations such as (including well-formedness when appropriate):

$$\mathcal{S} \odot \mathcal{T}\sharp(s_0, t_0) = \mathcal{S}\sharp s_0; \mathcal{T}\sharp t_0 \qquad \mathcal{S} \otimes \mathcal{T}\sharp(s_0, t_0) = \mathcal{S}\sharp s_0 \otimes \mathcal{T}\sharp t_0$$

where $-; -$ is one of the symbols we use for vertical composition on this trace-set represen-

tation.

3.10 Conclusion

We hope the reader familiar with state-based modeling of systems will find this discussion helpful in translating between the notions they are accustomed to and our treatment of objects as sets of traces. I also note that in all our mechanized developments, and in many of the examples I will present in this dissertation, we make use of transition systems to describe objects and sometimes even to achieve core verification tasks, before encapsulating the state by computing its representation as sets of traces. This is because state transition systems are often more tractable for verifying individual components, while the trace-based representation is more tractable for both dealing with composition and developing much of the theory presented in this dissertation.

Part I

Sequential Objects

Chapter 4

Safety and Liveness

In this chapter, we start the technical development of my dissertation. We begin with sequential objects for two key reasons: first, doing so proves useful later in analyzing the structure of parallel programs, and second, it lets us explore some of the key ideas and definitions in this dissertation from the relatively tame domain of sequential computation. We start by saying a word about the two kinds of properties that organize our approach: safety and liveness.

We use the Alpern-Schneider theorem as an organizing force in our model. Denotations, be they of specifications (state-transition systems) or of programs, modules, libraries, and other computational entities, will be given by a pair of trace properties (S, L) where S is understood as a safety specification and L as a liveness specification. We note that this does not mean, necessarily, that the underlying topology that would justify the decomposition is the same as the one we discussed in §4.2, although it is a close approximation. The key distinction is that we will introduce into the traces logical tokens that have a special status in the models, and, moreover, conditions such as receptivity (environment moves are always enabled), will eventually appear in our model, and these would require using more sophisticated partial orders than just finite prefixing.

This decomposition has a few advantages. First, we can afford to build compositional models for safety and liveness *separately* from each other, at least at first. This provides a nice separation of concerns, and also clarifies how this work relates to previous works, since most previous works only focused on safety. But it also provides the appropriate

foundations to then carefully relate these two worlds, providing a needed bridge between them. This ultimately results in the ability to verify safety and liveness simultaneously by focusing only on pairs (S, L) that satisfy certain side-conditions.

4.1 The Alpern-Schneider Theorem

In their seminal paper, Alpern and Schneider [1985] showed that every trace property decomposes as the intersection of a safety and a liveness property, a result I will call the Alpern-Schneider Theorem, formally:

Proposition 4.1.1. *Every property is the intersection of a safety and a liveness property.*

A safety property, informally, is a property that says “something bad never happens,” while a liveness property is a property that says “something good eventually happens”.

At the heart of the Alpern-Schneider Theorem is a topological characterization of properties. The key idea is that the basic open sets should be sets of plays which all share the same common prefix. Our first goal is to dissect the original argument. Toward that end, we first define a few useful topological constructions.

Definition 4.1.1. Given a set Σ , we define the discrete topology $\top$ on Σ as the topology that makes all subsets of Σ open sets.

We define the indiscrete topology $\perp$ as the coarsest topology, i.e., $\perp := \{\emptyset, \Sigma\}$.

Definition 4.1.2. Let I be some indexing set and let Σ_i be sets with topologies $\mathcal{T}_i$ for every $i \in I$.

We define the topology $\prod_{i \in I} \mathcal{T}_i$ over the set $\prod_{i \in I} \Sigma_i$ as the coarsest topology making the projections $\pi_j : \prod_{i \in I} \Sigma_i \rightarrow \Sigma_j$ continuous.

It is well-known that:

Proposition 4.1.2. *The sets of the form $\prod_{i \in I} U_i$ where for every i , $U_i \in \mathcal{T}_i$, and for only finitely many i , $U_i \neq \Sigma_i$, define a basis for $\prod_{i \in I} \mathcal{T}_i$.*

Proposition 4.1.3. *The topology $\mathcal{T}$ used by Alpern-Schneider (the Cantor topology) on the set of traces $\text{Traces} := \Sigma^\omega$ is equivalent to the topology $\prod_{i \in \omega} \top$.*

Proof. See Proof 30.1.1. □

A property is then defined as any subset $P \subseteq \Sigma^\omega$, and a property P implies another property P' when $P \subseteq P'$. Note that if we specify objects by their sets of traces $O \subseteq \Sigma^\omega$, then objects themselves are just properties. We say $s \in \Sigma^\omega$ satisfies a property P when $\{s\} \subseteq P$, from now on written simply as $s \models P$.

Definition 4.1.3. A *safety property* is a property $P \subseteq \Sigma^\omega$ such that

$$s \not\models P \implies \exists p \in \Sigma^*. p \sqsubseteq s \wedge \forall t. p \cdot t \not\models P$$

A *liveness property* is a property $P \subseteq \Sigma^\omega$ such that

$$\forall s \in \Sigma^*. \exists t \in \Sigma^\omega. s \cdot t \models P$$

Their key observation is then that

Proposition 4.1.4. *Safety properties are exactly the closed sets of $\prod_{i \in \omega} \top$, and liveness properties are exactly the dense sets of $\prod_{i \in \omega} \top$.*

Proof. See Proof 30.1.2. □

We now have all the tools to reproduce Alpern and Schneider [1985]’s proof:

Proposition 4.1.5. *In any topological space $(\Sigma, \mathcal{T})$, every set $P \subseteq \Sigma$ is the intersection of a closed and a dense set.*

Proof. Let $P \subseteq \Sigma$. Let $L = (\overline{P} \setminus P)^c$. Then:

$$L \cap \overline{P} = (\overline{P} \setminus P)^c \cap \overline{P} = (\overline{P}^c \cup P) \cap \overline{P} = (\overline{P}^c \cap \overline{P}) \cup (P \cap \overline{P}) = \emptyset \cup P = P$$

By definition, $\overline{P}$ is closed ($\overline{\overline{P}} = \overline{P}$ by definition of closure and limit point). It remains to show L is a dense set. So let $U \in \mathcal{T}$ be a non-empty open set and assume for the sake of contradiction that $U \cap L = \emptyset$. Then, $U \subseteq \overline{P} \setminus P$ so that $P \subseteq \overline{P} \setminus U$. Note then that $\overline{P}$ is closed, and $\overline{P} \setminus U = \overline{P} \cap U^c$. But U^c is closed by definition of closed set, and closed sets

are closed under intersection, so $\overline{P} \setminus U$ is closed. Since U is non-empty, $\overline{P} \setminus U \subset \overline{P}$. But $\overline{P}$ is equivalently defined as the intersection of every closed set containing P , and hence $\overline{P} \subseteq \overline{P} \setminus U$, a contradiction. $\square$

Corollary 4.1.1. *Every property is the intersection of a safety and a liveness property.*

4.2 Safety and Liveness in the Scott Topology

What is remarkable about the Alpern-Schneider argument is that it breaks down into two orthogonal parts: (i) defining a topology on traces that agrees with our informal understanding of properties, and (ii) applying the topology-agnostic argument from Prop. 4.1.5. This means that we can easily vary our notions of traces and their topologies without affecting part (ii) of the argument.

While it is quite sensible to consider the product topology on infinite traces, in practice it results in some unintuitive encodings. For example, a finite execution needs to be represented by a sequence of states with an infinite suffix of the form σ^ω , where σ is the final state of the system. Note, as well, that our trace mapping from state transition systems $\mathcal{S}_{\#s_0}$ included both finite and infinite traces. Our first step is to improve this situation by taking inspiration from domain theory [Scott, 1970, 1972], an approach for programming language semantics with roots in topology.

Definition 4.2.1. Let $(\Sigma, \leq)$ be a partial order. The Scott-topology on $(\Sigma, \leq)$, denoted $\mathcal{S}((\Sigma, \leq))$, is the topology whose open sets $U \subseteq \Sigma$ are those satisfying:

Upward Closure: if $s \in U$ and $s \leq t$ then $t \in U$.

Directed: if $D \subseteq \Sigma$ is a directed set then if $\sup(D) \in U$ then $D \cap U \neq \emptyset$.

This leads to the following order-theoretic characterization of safety and liveness:

Proposition 4.2.1. *A set S is closed in $\mathcal{S}((\Sigma, \leq))$ if and only if:*

Downward Closure: *If $s \in S$ and $t \leq s$ then $t \in S$.*

Completeness: *If $D \subseteq S$ is directed and $\sup(D)$ exists then $\sup(D) \in S$.*

A set L is dense in $\mathcal{S}((S, \leq))$ if and only if:

Extensibility: For every $p \in \Sigma$, there exists $d \in L$ such that $p \leq d$.

Density: For every $p \in \Sigma$ and every $p' > p$, there exists $d \in L$ such that $p \leq d \leq p'$.

When we fix $\text{Traces} := (\Sigma^* \cup \Sigma^\omega, \sqsubseteq)$ where $- \sqsubseteq -$ is the finite prefix ordering $p \sqsubseteq s \iff \exists t. s = p \cdot t$, we obtain a slight generalization of the definitions of safety and liveness properties we saw in Def. 4.1.3.

Proposition 4.2.2.

$$\mathcal{B} := \{\uparrow s \subseteq \text{Traces} \mid s \in \Sigma^*\}$$

defines a basis for $\mathcal{S}((S, \leq))$.

Proposition 4.2.3. A property S is a safety property in $\mathcal{S}(\text{Traces})$ if and only if:

$$s \not\models S \implies \exists p \in \Sigma^*. p \sqsubseteq s \wedge \forall t. p \cdot t \not\models S$$

A property L is a liveness property in $\mathcal{S}(\text{Traces})$ if and only if:

$$\forall s \in \Sigma^*. \exists t \in \Sigma^* \cup \Sigma^\omega. s \cdot t \models L$$

Proof. See Proof 30.1.3. □

Remark 1. One might wonder if we can relate the Scott topology $\mathcal{S}(\text{Traces})$ to the product topology, similarly to how we characterized the topology used in Alpern-Schneider. Indeed, let $\Sigma_\tau := \Sigma + \{\tau\}$, i.e., the free pointed set associated with Σ . Then, we can consider the product topology $\prod_{i \in \omega} \top$ on Σ_τ^ω . Now, let $\text{Traces}' \subseteq \Sigma_\tau^\omega$ be defined as the subset:

$$\text{Traces}' := \{s \in \Sigma_\tau^\omega \mid s \in \Sigma^\omega \vee \exists p \in \Sigma^*. s = p \cdot \tau^\omega\}$$

It turns out that $\mathcal{S}(\text{Traces})$ is homeomorphic to the topology induced by $\prod_{i \in \omega} \top$ on Traces' ,

where the homeomorphism is given by the bijection $h : \text{Traces} \rightarrow \text{Traces}'$:

$$h(s) := \begin{cases} s, s \in \Sigma^\omega \\ p, p \in \Sigma^* \wedge s = p \cdot \tau^\omega \end{cases}$$

as is easily checked.

This characterization of safety and liveness should be kept in mind as we define the notions of strategy, which will correspond to safety properties, and win conditions, which will correspond to liveness properties. It also justifies certain distinctions between the two. For example, the completeness requirement on safety properties means that it is enough to specify the finite traces of a safety property, as the infinite traces would be implied by them.

Chapter 5

Plays and Games

We build our semantic model as a game semantics model. Game semantics should be understood, at least for the purposes of this dissertation, as an approach to computational semantics with two main aspects. First, it is essentially a compositional trace semantics, in the sense that everything in the model is a set of traces, and composition is given by operations on those traces. This is distinct from, say, having a model of programs and linking, and then proving properties about traces generated from those. In game semantics, instead, the compositional structure is defined directly on sets of traces. The second main idea is a clear separation between the environment, traditionally called the *Opponent*, and the system, called the *Player*.

Before we start the chapter, I provide a quick introduction to the game semantics vocabulary. An event in game semantics is called a *move*, which comes with some metadata called its *polarity*. This metadata includes whether it is an Opponent or Proponent move (environment or system), whether it is a *Question* (an invocation) or an *Answer* (a response), and, in our concurrent models, which agent is responsible for the event. In some of our models, special events such as crash events, divergence markers, and starvation events are given a distinct polarity to indicate their special status. This metadata (the polarity) is essential to define well-formedness for traces, such as sequentiality or local sequentiality. A well-formed trace is called a *play*. A *game*, which should be understood as a type, defines the “rules of the game”, and may be understood as a predicate over the set of all well-formed traces. Specifications, programs, and objects are all modeled by the notion of a *strategy*, which

consists of a non-empty set of plays, potentially with extra properties, and they constitute the central object of study for game semantics. They are called strategies for they specify how the Player reacts to possible moves by the Opponent. Additional notions, some of them developed for the sake of this dissertation, will be introduced when appropriate, but this should make for a good dictionary of terms.

This chapter pulls from several of my papers [Oliveira Vale et al., 2022, 2024a, 2023] as well as a couple of manuscripts and an accepted paper to appear. I note, however, that there are subtle differences between the game model here and that from my work on sequential compositional verification [Oliveira Vale et al., 2022]. Mainly, since I will not assume determinism here, to prepare for the concurrent game models, I change the notion of refinement used here to match the presence of non-determinism.

5.1 Moves and Plays

We move now to defining notions of events and well-formed traces, in the form of *moves* and *plays*.

Definition 5.1.1. We define the set of polarities $\text{Pol} := (\text{O}+\text{P}) \times (\text{Q}+\text{A}) = \{\text{OQ}, \text{OA}, \text{PQ}, \text{PA}\}$ and $\text{Pol}^{\circ\bullet} := \text{Pol} \uplus \{\text{O}\circ, \text{P}\circ\} \uplus \bullet$.

Moves are labeled by a pair of a player O/P and move type Q/A. O stands for Opponent and is understood as an environment move, while P stands for Proponent and is understood as a system move. They are further labeled as either a Q, standing for Question and understood as an invocation, or an A, standing for an Answer and understood as a return to an invocation. The special polarities $\bullet$ and $\circ$ are for special semantic tokens. $\bullet$ denotes that the environment has decided to terminate the computation, and because of this, no longer schedules moves. Meanwhile, $\circ$ will mean that either the system ($\text{P}\circ$) or environment ($\text{O}\circ$) has diverged. These tokens help us distinguish between non-termination and scheduler starvation, and are treated differently from regular moves. Their roles will be made clearer later.

Definition 5.1.2. A *move set* M consists of a pair $M = (|M|, \lambda_M : |M| \rightarrow \text{Pol})$ of a set of

moves $m \in M$ together with a polarity-assigning function λ_M , assigning a polarity $\lambda_M(m)$ to each move $m \in M$.

An *extended move set* consists of a pair $M = (|M|, \lambda_M : |M| \rightarrow \text{Pol}^{\odot\bullet})$. Note that every move set defines an extended move set.

We will almost always abuse notation by omitting $| - |$ and writing M for the set of moves $|M|$ as well as $\lambda_M : M \rightarrow \text{Pol}^{\odot\bullet}$. Extended move sets play only a technical role in our definitions; the main notion to consider is that of a move set.

Example 5.1.1. Most of the games we use in our examples are generated by effect signatures.

Definition 5.1.3. An *effect signature* consists of a set E of operations together with an assignment $\text{ar}(-) : E \rightarrow \mathbf{Set}$ of a set of responses $\text{ar}(e)$ to every effect. We encapsulate an effect signature by the notation:

$$E := \{e : \text{ar}(e) \mid e \in E\}$$

which denotes the effect signature with effects $e \in E$ each with arity $\text{ar}(e)$.

We find it useful to slightly generalize effect signatures to be of the form:

$$E = \{e : \text{par}(e) \rightarrow \text{ar}(e) \mid e \in E\}$$

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

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

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

For example, the following are signatures for a counter and a stack, respectively:

$$\text{Counter} := \{\text{inc} : \mathbf{1} \rightarrow \mathbf{1}, \text{get} : \mathbf{1} \rightarrow \mathbb{N}\} \quad \text{Stack} := \{\text{push} : \text{Val} \rightarrow \mathbf{1}, \text{pop} : \mathbf{1} \rightarrow (\text{Val} + \perp)\}$$

Here, $\mathbf{1}$ is the singleton set, and we denote its element as follows in this context: $\text{inc}()$ in the case of an effect or ok for a response.

Every effect signature defines a move set as follows:

$$M_E := \sum_{e \in E} \text{par}(e) + \sum_{e \in E} \text{ar}(e)$$

$$\lambda_E(e(a)) := \text{OQ}, \text{ for } e \in E \text{ and } a \in \text{par}(e) \quad \lambda_E(v) := \text{PA}, \text{ for } v \in \text{ar}(e) \text{ for some } e \in E$$

where we write $e(a) \in \sum_{e \in E} \text{par}(e)$ and $v \in \sum_{e \in E} \text{ar}(e)$ (omitting the event e to which v is a response).

For example, the signature **Counter** defines the move set $(M_{\text{Counter}}, \lambda_{\text{Counter}})$ with invocations $\text{inc}()$ and $\text{get}()$ and responses ok and n^1 .

We are now ready to define the notion of *play*, that is, of a well-formed trace. Recall that we write S^* for the set of finite sequences over S , S^∞ for the infinite countable sequences, and $S^* = S^* \cup S^\infty$ for all countable sequences. If $\lambda : S \rightarrow T$ is a function, we denote by $\lambda^* : S^* \rightarrow T^*$ its lift to countable sequences. Moreover, for $S \subseteq M^*$, we write $\downarrow S := \{p \in M^* \mid \exists s \in S. p \sqsubseteq s\}$ for the down-closure of S under finite prefixing ($- \sqsubseteq -$).

We now formally define well-formed plays. A well-formed play ensures that agents alternate between O- and P-moves starting with an O-move; that questions and answers are properly bracketed; that $\bullet$ tokens are terminating events; that $\circ$ moves are terminating events; and that divergence flows from $\text{O}\circ$ to $\text{P}\circ$.

Definition 5.1.4. First, given an extended move set, we define several well-formed positions

1. Formally, the responses should be denoted $\text{inc}().\text{ok}$ and $\text{get}().n$, but we often omit the indexing for space.

for a sequence $s \in M^*$:

$$\begin{array}{c}
\begin{array}{cc}
\frac{s \in \mathbb{SP}_M^{\text{PA}} \quad \lambda(m) = \text{OQ}}{s \cdot m \in \mathbb{SP}_M^{\text{OQ}}} & \frac{s \in \mathbb{SP}_M^{\text{OA}} \cup \mathbb{SP}_M^{\text{OQ}} \quad \lambda(m) = \text{PA}}{s \cdot m \in \mathbb{SP}_M^{\text{PA}}} \\
\epsilon \in \mathbb{SP}_M^{\text{PA}} &
\end{array} \\
\\
\begin{array}{cc}
\frac{s \in \mathbb{SP}_M^{\text{OQ}} \cup \mathbb{SP}_M^{\text{OA}} \quad \lambda(m) = \text{PQ}}{s \cdot m \in \mathbb{SP}_M^{\text{PQ}}} & \frac{s \in \mathbb{SP}_M^{\text{PQ}} \quad \lambda(m) = \text{OA}}{s \cdot m \in \mathbb{SP}_M^{\text{OA}}} \\
\end{array} \\
\\
\frac{s \in \mathbb{SP}_M^{\text{PA}} \cup \mathbb{SP}_M^{\text{OQ}} \cup \mathbb{SP}_M^{\text{PQ}} \cup \mathbb{SP}_M^{\text{OA}}}{s \in \mathbb{SP}_M^{\text{Pol}}} \\
\\
\begin{array}{cc}
\frac{s \in \mathbb{SP}_M^{\text{PQ}} \quad \lambda(m) = \text{O}\circ}{s \cdot m \in \mathbb{SP}_M^{\text{O}\circ}} & \frac{s \in \mathbb{SP}_M^{\text{OQ}} \cup \mathbb{SP}_M^{\text{OA}} \cup \mathbb{SP}_M^{\text{O}\circ} \quad \lambda(m) = \text{P}\circ}{s \cdot m \in \mathbb{SP}_M^{\text{P}\circ}} \\
\end{array} \\
\\
\frac{s \in \mathbb{SP}_M^{\text{Pol}} \wedge \lambda(m) = \bullet}{s \cdot m \in \mathbb{SP}_M^\bullet}
\end{array}$$

We say that $s \in M^*$ is in **pos**-position when $s \in \mathbb{SP}_M^{\text{pos}}$. We define an **O**-position (analogously for **P**-, **Q**-, **A**-, **○**-position) as the union of **OQ**, **OA** and **O○** positions. We similarly extend these conventions to moves, so that an **O**-move is either an **OQ**-move, or an **OA**-move, or an **O○**-move (and similarly for **P**-, **A**- and **Q**-moves). We take the freedom to write $\lambda_M(m) = \text{O}$ to mean $\lambda_M(m) \in \text{OQ} \cup \text{OA} \cup \text{O}\circ$ and analogously for **P**, **Q**, **A**, **○**. We take the freedom of referring to moves labelled by **○** and **•** as tokens.

We say a sequence $s \in M^*$ is a sequential play when

$$\downarrow s \subseteq \mathbb{SP}_M^{\text{Pol}} \cup \mathbb{SP}_M^{\text{O}\circ} \cup \mathbb{SP}_M^{\text{P}\circ} \cup \mathbb{SP}_M^\bullet$$

We say a sequential play is well-scheduled when it is either infinite or in **P**-position or **•**-position (i.e., one of $\mathbb{SP}_M^{\text{PQ}}$, $\mathbb{SP}_M^{\text{PA}}$, $\mathbb{SP}_M^{\text{P}\circ}$, $\mathbb{SP}_M^\bullet$).

We write $\mathbb{SP}_M$ for the set of finite sequential plays over M with no **○**-tokens (moves labelled by **○**), $\mathbb{SP}_M^{\text{ws}}$ for the set of all well-scheduled plays, and $\mathbb{SP}_M^{\leq \infty} := \downarrow \mathbb{SP}_M^{\text{ws}} \cup \mathbb{SP}_M^{\text{ws}}$ for

the set of all well-formed plays.

We write $\text{ops}(s) \in \mathbb{P}_M^{\leq \infty}$ for the largest subsequence of $s \in \mathbb{P}_M^{\leq \infty}$ containing only moves m such that $\lambda_M^\gamma(m) \in \text{Pol}$.

Note that well-formed plays are defined as plays whose sequences of polarities are generated by the following labeled transition system, which also shows the relationships between the respective positions:

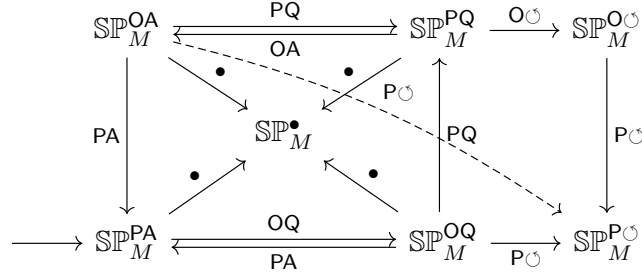


Figure 5.1: Position graph for well-formed plays.

Well-scheduled plays are essentially *total* executions. That is, those are executions where the system (Proponent) was given as much opportunity to execute as possible. Plays that are not well-scheduled (as well as some of the well-scheduled plays) are *partial* executions; they show some prefix of a well-scheduled play. The terminology *well-scheduled* comes from the fact that the way we model the dynamics of plays and strategies makes it so that the well-scheduled assumption corresponds to a scheduling assumption: the environment schedules the Proponent whenever it can, unless it has terminated the computation by issuing a starvation token $\bullet$.

Before giving some examples of plays, let us see how extended move sets are constructed from move sets:

Definition 5.1.5. Given a move set M , we define $M^\circ := M + \circ$ with $\lambda^\circ : M^\circ \rightarrow \text{Pol}^{\circ\bullet}$ defined as the extension of λ with $\lambda^\circ(\circ) = P\circ$, and $M^\bullet := M + \bullet$ with $\lambda^\bullet : M^\bullet \rightarrow \text{Pol}^{\circ\bullet}$ defined as the extension of λ with $\lambda^\bullet(\bullet) = \bullet$.

Example 5.1.2. Consider the signature $\text{Unit} := \{q : \mathbf{1} \rightarrow \mathbf{1}\}$. For this particular case, I will write the events of M_{Unit} as simply q and $a \in \text{ar}(q)$. Now, let me list and explain each of the possible plays over the extended move set $M_{\text{Unit}}^{\circ\bullet}$.

The simplest plays are those plays where neither $\bullet$ nor $\circ$ appear. For instance:

$$\epsilon, \quad q, \quad q \cdot a, \quad q \cdot a \cdot q, \quad \dots, \quad (q \cdot a)^*, \quad (q \cdot a)^* \cdot q, \quad (q \cdot a)^\infty$$

ϵ represents the empty play where the system waits for the environment to prompt it. Plays of the form $(q \cdot a) \cdot q$ are plays where the environment has asked the question q and is waiting for the system to provide an answer. Meanwhile, plays of the form $(q \cdot a)^+$ are plays where the system has answered such a question and waits to be prompted once more by the environment. $(q \cdot a)^\infty$ is the only possible infinite play, where the environment and system repeatedly exchanged questions and answers. Among these plays, plays in $(q \cdot a)^*$ and $(q \cdot a)^\infty$ are considered *well-scheduled*, while plays in $(q \cdot a)^* \cdot q$ are not considered well-scheduled.

At any point, the environment (or the scheduler) is allowed to terminate the computation by issuing a starvation token $\bullet$. This interrupts the computation permanently:

$$(q \cdot a)^* \cdot \bullet \qquad (q \cdot a) \cdot q \cdot \bullet$$

Finally, the Proponent is allowed to mark a divergence:

$$(q \cdot a)^* \cdot q \cdot \circ$$

which signals that the last q invocation will never receive a response. Note that all of these plays (the ones with $\bullet$ and $\circ$) are well-scheduled.

This provides a description of all the plays over $M_{\text{Unit}}^{\circ\bullet}$.

Note that in Ex. 5.1.2, no play in OA-, PQ- or O $\circ$ -position appears. The reason for this is that the move set M_E for any effect signature is what we call a *positive* move set, i.e., a move set with only OQ- and PA-moves. Because of this, when the Opponent issues a question, there is simply no way for the Player to retort, as there is no PQ move available. The ability to retort will be available to strategies that interact across two games.

For simplicity, I will later assume games have positive move sets for the most part. Most of the work of this dissertation was done without this assumption. The work on

liveness, however, makes this assumption to avoid defining certain higher-order versions of well-formedness, as it was not thought to be necessary for the verification targets we planned to tackle. I expect the results would readily translate to a setting without this positivity assumption. The game semantics expert will note that Fig. 5.1 already benefits from this assumption.

5.2 Sequential Games

Definition 5.2.1. We define a game $A = (M_A, P_A)$ to be a move set M_A satisfying that

$$\textbf{positivity: } \forall m \in M_A. \lambda_{M_A}(m) \in \{\text{OQ}, \text{PA}\}$$

together with a non-empty, prefix-closed set of finite well-formed plays $P_A \subseteq \mathbb{SP}_{M_A}$.

We say a play $s \in \mathbb{SP}_{M_A}^{\leq \infty}$ (from now on we often replace $M_A^{\circ \bullet}$ with A) is valid for the game A when $\downarrow \text{ops}(s) \subseteq P_A$. We write $P_A^\bullet \subseteq \mathbb{SP}_A$ for the set of finite $\circ$ -token free valid plays over A , P_A^{ws} for the set of valid well-scheduled plays, and $P_A^{\leq \infty} := P_A^\bullet \cup P_A^{\text{ws}}$.

Example 5.2.1. For an effect signature E , we define the game $[E]$ as $[E] = (M_E, P_E)$ where

$$P_E := \cup_{e \in E} \cup_{v \in \text{ar}(e)} \downarrow(e \cdot v)$$

The game $[E]$ allows for the Opponent to issue an effect of E *at most once* and then the Proponent might reply with v . As in Example 5.1.2, the game $[E]$ also includes the plays $\bullet$, $e \cdot \bullet$, $e \cdot \circ$ and $e \cdot v \cdot \bullet$. In almost all circumstances, we will write E where we should have written $[E]$.

But what if we wanted to use E more than once? Typically, if we have a counter, we would like to use it several times. For this, we can use the *replay modality*. We briefly introduce it for the sake of this example, but will provide a detailed discussion soon.

For an effect signature E , we define the game $\dagger E = (M_{\dagger E}, P_{\dagger E})$ with the move set defined by:

$$M_{\dagger E} := \mathbb{N} \times M_E \qquad \lambda_{\dagger E}(i.m) := \lambda_E(m)$$

in other words, a move of $M_{\dagger E}$ is simply a move of E with a copy index i . Moves inherit the polarity of the move in E . Then, its set of plays is given by:

$$P_{\dagger E} := \downarrow (\cup_{i \in \mathbb{N}} \cup_{e \in E} \cup_{a \in \text{par}(e)} \cup_{v \in \text{ar}(e)} i.e(a) \cdot i.v)^* \cap \{s \in \mathbb{SP}_{\dagger E} \mid \pi_0(s) \sqsubseteq 0 \cdot 0 \cdot 1 \cdot 1 \cdot 2 \cdot 2 \cdot \dots\}$$

In other words, plays of $P_{\dagger E}$ are essentially plays in the Kleene star of P_E , except that the copy indexes must strictly increase from each play of P_E .

The copy indexes are a technical requirement for $\dagger$ – to satisfy certain desired algebraic properties. It turns out that in the case of the game E , since interactions have length at most 2, the copy indexes are redundant information: we may always guess them from the length of the play. Because of this, I will always omit the copy indexes in the case of $\dagger E$. So, for instance, I will write:

$$\text{inc}() \cdot \text{ok} \cdot \text{get} \cdot 1 \cdot \text{inc}() \cdot \text{ok} \cdot \text{inc}() \cdot \text{ok} \cdot \text{get} \cdot 3 \cdot \text{get}$$

for the following play of $\dagger \text{Counter}$:

$$0.\text{inc}() \cdot 0.\text{ok} \cdot 1.\text{get} \cdot 1.1 \cdot 2.\text{inc}() \cdot 2.\text{ok} \cdot 3.\text{inc}() \cdot 3.\text{ok} \cdot 4.\text{get} \cdot 4.3 \cdot 5.\text{get}$$

Example 5.2.2. Given the effect signature Counter , the set of plays P_{Counter} is given by:

$$\begin{aligned} P_{\text{Counter}}^\bullet &= \{ \epsilon, \bullet, \text{inc}, \text{inc} \cdot \bullet, \text{inc} \cdot \text{ok}, \text{inc} \cdot \text{ok} \cdot \bullet, \\ &\quad \text{get}, \text{get} \cdot \bullet, \text{get} \cdot n, \text{get} \cdot \circlearrowleft, \text{get} \cdot n \cdot \bullet \} \\ P_{\text{Counter}}^{\text{ws}} &:= \{ \epsilon, \text{inc} \cdot \bullet, \text{inc} \cdot \circlearrowleft, \text{inc} \cdot \text{ok}, \text{inc} \cdot \text{ok} \cdot \bullet, \\ &\quad \text{get} \cdot \bullet, \text{get} \cdot n, \text{get} \cdot \circlearrowleft, \text{get} \cdot n \cdot \bullet \} \end{aligned}$$

Meanwhile, $P_{\dagger \text{Counter}}^{\leq \infty}$ includes all the plays above, but also:

$$\begin{aligned} &\text{inc}() \cdot \text{ok} \cdot \text{get} \cdot 1 \cdot \text{inc}() \cdot \text{ok} \cdot \text{inc}() \cdot \text{ok} \cdot \text{get} \cdot 3 \cdot \text{get}, \quad \text{get} \cdot 13 \cdot \text{inc} \cdot \text{ok} \cdot \text{get} \cdot 3 \cdot \text{get} \cdot \circlearrowleft, \\ &(\text{inc} \cdot \text{ok})^\infty, \quad (\text{inc} \cdot \text{ok})^*, \quad (\text{inc} \cdot \text{ok})^* \cdot \text{get} \cdot \circlearrowleft, \quad (\text{inc} \cdot \text{ok})^* \cdot \bullet, \quad \text{etc} \end{aligned}$$

Note that, at least in the way we use games, the type $\dagger \text{Counter}$ does not define a *correct*

counter type. It simply defines the type of any object that allows for operations defined by **Counter** to be called repeatedly, independently of what they might return or do. Specifying the correct behaviors of a counter will be the job of our notions of strategies, win conditions, and objects.

We also define a composition operation for games, which we call horizontal composition or the tensor of two games. Given a sequence in $((M_A^\circ + M_B^\circ)^\bullet)^*$, we write $s|_{A,-} \in (M_A^\circ)^\bullet$ for its projection to only moves from M_A° and similarly for $s|_{-,B}$. Note that a $\bullet$ -token in s still appears in *both* $s|_{A,-}$ and $s|_{-,B}$. Meanwhile, a $\circ$ -token appears only in the component whose question it replies to.

Definition 5.2.2. Given games A, B , we define the game $A \otimes B$ by the following data:

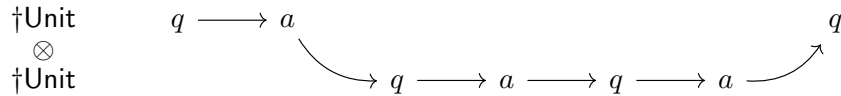
$$\begin{aligned} M_{A \otimes B} &:= M_A + M_B & \lambda_{A \otimes B} &:= \lambda_A + \lambda_B \\ P_{A \otimes B} &:= \{s \in \mathbb{SP}_{A \otimes B} \mid s|_{A,-} \in P_A \wedge s|_{-,B} \in P_B\} \end{aligned}$$

The unit for the $- \otimes -$ is given by the empty game **1** described by the following data:

$$M_1 := \emptyset \qquad \lambda_1 := \emptyset \qquad P_1 = \{\epsilon\}$$

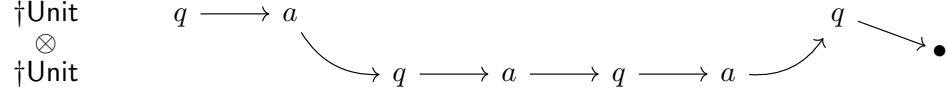
Note that there are actually two valid plays for **1**, ϵ and $\bullet$. Note too that $\dagger\emptyset \cong \mathbf{1}$, that is, if $\emptyset$ is the empty signature, then its set of moves is $M_\emptyset = \emptyset$, so that $M_{\dagger\emptyset} = \emptyset$, which then coincides with **1**.

Example 5.2.3. Let's consider some plays of $\dagger\text{Unit} \otimes \dagger\text{Unit}$. The basic examples are simple sequential interleavings of plays of $\dagger\text{Unit}$. For instance:

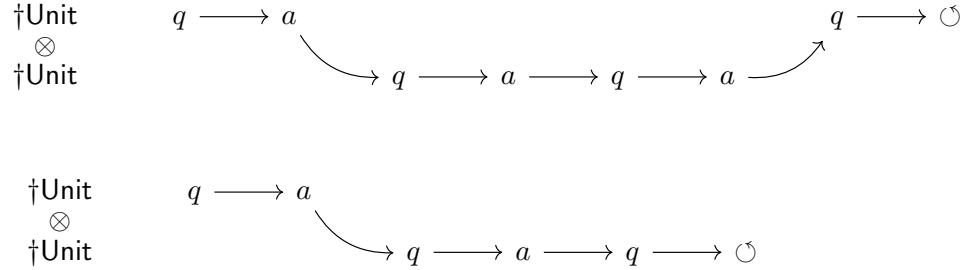


Note that the sequentiality assumption means that only Opponent is allowed to switch components. Indeed, Opponent must start the game, at which point only the Proponent may move, whose only option is then to move in the same component as the other component is in a P-position already. This is known as the switching condition for the tensor.

So far, the tensor seems like the typical game semantics tensor. The situation is more interesting, however, for the $\bullet$ - and $\circ$ -tokens. A first observation is that there is a unique such token for both components in the tensor $\dagger\text{Unit} \otimes \dagger\text{Unit}$, by definition of $P_A^{\leq \infty}$. A $\bullet$ -token may happen at any position. As the $\bullet$ -token is a signal of the environment to terminate the computation across the two components, we might depict it as:



However, while there is a unique $\circ$ -token, it naturally belongs to the component whose question was left open, for example:



where we attach the divergence to the component from which it stems.

5.3 The Implication “Game”

Toward defining notions of strategies and win conditions, which shall serve as our specifications, we first define the dual of a move set, which flips the roles of environment and system.

Definition 5.3.1. Given an extended move set $M = (M, \lambda)$, we define the extended move set $M^\perp := (M, \lambda^\perp)$ where

$$\begin{aligned}
 \lambda^\perp(m) = \text{OQ} &\iff \lambda(m) = \text{PQ} & \lambda^\perp(m) = \text{OA} &\iff \lambda(m) = \text{PA} \\
 \lambda^\perp(m) = \text{PQ} &\iff \lambda(m) = \text{OQ} & \lambda^\perp(m) = \text{PA} &\iff \lambda(m) = \text{OA} \\
 \lambda^\perp(m) = \text{P}\circ &\iff \lambda(m) = \text{O}\circ & \lambda^\perp(m) = \text{O}\circ &\iff \lambda(m) = \text{P}\circ \\
 \lambda^\perp(m) = \bullet &\iff \lambda(m) = \bullet
 \end{aligned}$$

Note that, below, when we have a play $s \in ((M_A^{\circ\perp} + M_B^{\circ})^\bullet)^*$, $s|_{A,-} \in (M_A^{\circ\perp\bullet})^*$ is the projection of s to only moves in $M_A^{\circ\perp\bullet}$, and similarly for $s|_{-,B}$, where $\bullet$ is considered as a move in both A and B .

Definition 5.3.2. Given games A and B , we define an extended move set $(M_{A\multimap B}, \lambda_{A\multimap B})$ and set of plays $P_{A\multimap B}^\bullet$ as follows:

$$\begin{aligned} M_{A\multimap B} &:= (M_A^{\circ\perp} + M_B^{\circ})^\bullet & \lambda_{A\multimap B} &:= (\lambda_A^{\circ\perp} + \lambda_B^{\circ})^\bullet \\ P_{A\multimap B}^\bullet &:= \{s \in \mathbb{SP}_{A\multimap B} \mid s|_{A,-} \in P_A \wedge s|_{-,B} \in P_B\} \end{aligned}$$

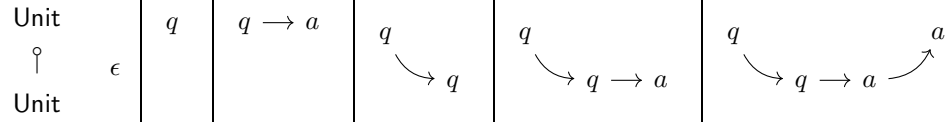
We take the same conventions as we took for plays of a game in defining $P_{A\multimap B}$, $P_{A\multimap B}^{\text{ws}}$ and $P_{A\multimap B}^{\leq\infty}$.

In $A \multimap B$, A is called the *source* and B is called the *target*. Note that, as usual in game models, $A \multimap B \cong A^\perp \otimes B$, which gives another way to parse the definition. Notably, the negation on A changes the dynamics of the tensor significantly. For instance, the Opponent in $A \otimes B$ can start either game at the beginning, while in $A \multimap B$, since $A^\perp$ only has P-starting plays (except for plays starting with $\bullet$), the Opponent can only open the game B at the start. This naturally encodes the flow of information in programs, where the environment first makes a call to B , which is then able to make library calls in A .

Note that our restriction that the move set of a game be positive means that $A \multimap B$ is not considered a game. In other words, we only model “first-order games” and strategies between them. We do this in order to maintain simplicity of our framework and parity with the usual notions of locally sequential traces and object specifications from the concurrency theory and linearizability literature [Herlihy and Wing, 1990]. It does not mean, however, that the ideas in this work cannot be extended to game models admitting $A \multimap B$ as a game.

In $M_{A\multimap B}$, due to positivity, there is a unique $\text{O}\circ$ -move, which comes from $M_A^{\circ\perp}$, and a unique $\text{P}\circ$ -move, which comes from M_B° . Hence, in the context of $M_{A\multimap B}$, we will often refer to the $\text{O}\circ$ -move by either $\circ_{\text{O}}$ or $\circ_A$ and the $\text{P}\circ$ -move by $\circ_{\text{P}}$ or $\circ_B$.

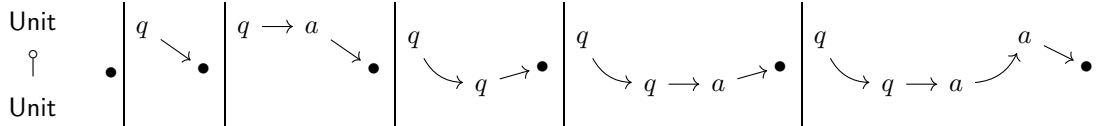
Example 5.3.1. Below are depicted all of the possible plays for the game $\text{Unit} \multimap \text{Unit}$. The regular plays are, separated by vertical lines:



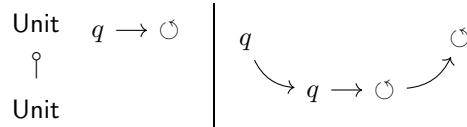
First, we note that there are two maximal interactions. The first is $q \cdot a$ where the environment asks q and Proponent immediately responds with a , the other one is $q \cdot q \cdot a \cdot a$ (note that we omit the indexes into the source and target components). In this play, upon being asked q by the Opponent, the Player retorts with the question q to the source component. The Opponent replies with a in the source, which the Player then uses to respond a in the target component. The best way to understand these plays is that Proponent uses the source component to decide how to answer questions to its target.

A second observation is that the negation in the source component means that this is the first instance in which we see OA- and PQ-moves which appear as negations of PA- and OQ-moves from the original game. This also means that the only OQ-moves available to open the game are moves in the target component. Following similar reasoning to the switching condition for the tensor $- \otimes -$, we observe that only Proponent is allowed to switch components in $\text{Unit} \multimap \text{Unit}$.

Now, for the plays involving $\bullet$ - and $\circ$ -tokens, first we have:



which are just the plays from before with a $\bullet$ -token terminating the computation. Like the $- \otimes -$, the $\bullet$ -token is considered to belong to both components. For the plays with $\circ$ -tokens, we have:



The first play is a case where after receiving the question from the Opponent, the Player silently diverges. The second play is the case where after being asked q by the Opponent, Proponent retorts by asking q to the source component, and then the Opponent silently diverges trying to answer Proponent's question. When this happens, the Player can only signal silent divergence as well, as the only way the Opponent could diverge is if it is silently interacting with the Player in an infinite interaction that has been hidden.

More interesting examples of plays of $A \multimap B$ will be given soon, after we define the notion of a strategy.

Chapter 6

Safety Specifications

6.1 Strategies

Strategies are the core object of study in game semantics. They make for the denotations of all computational entities we are interested in, including specifications, programs, modules, libraries, etc. Strictly speaking, we could call what ultimately will be our pair (S, L) of a safety and a liveness specification a strategy. We find this confusing both with my own previous works and with previous works in game semantics. I therefore opt to call a safety specification a *strategy* and a liveness specification a *win condition*. We call the pair (S, L) an object.

The safety game model is assembled by the usual notions of games and strategies. This is the same model appearing in my work Oliveira Vale et al. [2024a], differing crucially in the inclusion of $\bullet$ -moves.

Definition 6.1.1. A *strategy* $\sigma : A \multimap B$ between the games A and B is a non-empty, prefix-closed set $\sigma \subseteq P_{A \multimap B}^\bullet$. We also define a convention: we will call a strategy $\nu : \mathbf{1} \multimap A$ a strategy for the game A , or a safety specification over A , and write it simply as $\nu : A$.

Example 6.1.1. There is a unique strategy over $\mathbf{1}$ without special tokens, given by $\{\epsilon\}$. There is also a unique $\bullet$ -receptive strategy (a strategy where $\bullet$ is always enabled) over $\mathbf{1}$, which we call $\nu_1 : \mathbf{1}$, given by $\nu_1 := \downarrow\bullet = \{\epsilon, \bullet\}$.

Example 6.1.2. In our examples and practical verification, specifications are given by strategies $\nu_E : \dagger E$ over an effect signature E .

For example, a (safety) specification for a counter is given by $\nu_{\text{Counter}} : \dagger \text{Counter}$:

$$\nu_{\text{Counter}} := \{s \in P_{\dagger \text{Counter}}^\bullet \mid s = s_1 \cdot \text{get} \cdot n \cdot s_2 \implies n = \# \text{inc}(s_1)\}$$

where $\# \text{inc}(-) : \mathbb{P}_{\dagger \text{Counter}} \rightarrow \mathbb{N}$ is the function inductively defined by:

$$\begin{aligned} \# \text{inc}(\epsilon) &:= 0 & \# \text{inc}(s \cdot \text{get} \cdot n) &:= \# \text{inc}(s) & \# \text{inc}(s \cdot \text{get} \cdot \bullet) &:= \# \text{inc}(s) \\ \# \text{inc}(s \cdot \text{inc} \cdot \text{ok}) &:= \# \text{inc}(s) + 1 & \# \text{inc}(s \cdot \text{inc} \cdot \bullet) &:= \# \text{inc}(s) + 1 \end{aligned}$$

Alternatively, we can define ν_{Counter} using a state transition system

$$\mathcal{V}_{\text{Counter}} := ((\mathbb{N} \times (\text{Counter} + \text{idle} + \text{starved}), M_{\text{Counter}}^{\circ \bullet}, - \xrightarrow{\text{Counter}} -), (0, \text{idle}))$$

defined by the rules:

$$\begin{array}{c} \frac{}{(n, \text{idle}) \xrightarrow{\text{get}} (n, \text{get})} \qquad \frac{}{(n, \text{get}) \xrightarrow{n} (n, \text{idle})} \qquad \frac{}{(n, \text{idle}) \xrightarrow{\text{inc}} (n, \text{inc})} \\[10pt] \frac{}{(n, \text{inc}) \xrightarrow{\text{ok}} (n+1, \text{idle})} \qquad \frac{c \neq \text{starved}}{(n, c) \xrightarrow{\bullet} (n, \text{starved})} \end{array}$$

Then, we have that:

$$\nu_{\text{Counter}} = \mathcal{V}_{\text{Counter}} \# (0, \text{idle}) \cap (M_{\text{Counter}}^{\bullet})^*$$

Example 6.1.3. A simple memory cell can be modeled by the signature $\text{Cell}[\text{Val}] := \{\text{wrt} : \text{Val} \rightarrow \mathbf{1}, \text{rd} : \mathbf{1} \rightarrow \text{Val}\}$ and its specification is given by the traces generated from the initial state $(\text{init}, \text{idle})$, where init is some initial value for memory cells, by the transition system

defined by the rules:

$$\begin{array}{c}
\frac{}{(v, \text{idle}) \xrightarrow{\text{rd}} (v, \text{rd})} \quad \frac{}{(v, \text{rd}) \xrightarrow{v} (v, \text{idle})} \quad \frac{}{(v, \text{idle}) \xrightarrow{\text{wrt}(v')} (v, \text{wrt}(v'))} \\
\\
\frac{}{(v, \text{wrt}(v')) \xrightarrow{\text{ok}} (v', \text{idle})} \quad \frac{c \neq \text{starved}}{(v, c) \xrightarrow{\bullet} (v, \text{starved})}
\end{array}$$

We call the resulting specification ν_{Cell} .

Given a state transition system $\mathcal{V} := ((S, L, - \xrightarrow{\quad} -), s_0)$ well-formed in that $\mathcal{V}_{\#s_0}$ defines a strategy over some game A , we define its safety interpretation by:

$$[\![\mathcal{V}]\!]_S := \mathcal{V}_{\#s_0} \cap L^*$$

That is, the set of finite traces of $\mathcal{V}$.

We see in the above examples that it is common to simply make strategies $\bullet$ -receptive. That is, strategies will often be such that $\bullet$ -tokens are always enabled. Formally:

Definition 6.1.2. We say a move m is *enabled* in a play $s \in \sigma$ of a strategy $\sigma : A \multimap B$ when $s \cdot m \in \sigma$.

A strategy σ is pol -receptive, for some polarity pol , when for any play $s \in \sigma$ if $s \cdot m \in \mathbb{SP}_{A \multimap B}^{\text{pol}}$ then $s \cdot m \in \sigma$.

In cases where we wish to define a $\bullet$ -receptive strategy, it is useful to just define a specification without taking into account starvation and then simply take its $\bullet$ -receptive closure, computed as follows.

Definition 6.1.3. Let $S \subseteq \mathbb{SP}_{A \multimap B}^{\leq \infty}$. We define its $\bullet$ -receptive closure $S^\bullet$ as the set of plays defined by the rules:

$$\begin{array}{c}
\frac{s \in S}{s \in S^\bullet} \quad \frac{s \in S \wedge s \cdot \bullet \in \mathbb{SP}_{A \multimap B}^{\leq \infty}}{s \cdot \bullet \in S^\bullet}
\end{array}$$

Example 6.1.4. Consider the following library, which implements a counter using a memory cell:

```

MCounter : Counter
Import B : Cell[N]

inc() {
  i ← B.rd();
  B.wrt(i+1);
  ret ok;
}

get() {
  i ← B.rd();
  ret i;
}

```

The programs for `inc()` and `get()` are denoted as strategies

$$M_{\text{Counter}}^{\text{inc}} : \dagger \text{Cell}[\mathbb{N}] \multimap \text{Counter} \qquad M_{\text{Counter}}^{\text{get}} : \dagger \text{Cell}[\mathbb{N}] \multimap \text{Counter}$$

defined by:

$$M_{\text{Counter}}^{\text{inc}} := \downarrow \{ \text{inc}() \cdot \text{rd}() \cdot i \cdot \text{wrt}(i+1) \cdot \text{ok} \cdot \text{ok} \mid i \in \mathbb{N} \} \qquad M_{\text{Counter}}^{\text{get}} := \downarrow \{ \text{get} \cdot \text{rd}() \cdot i \cdot i \mid i \in \mathbb{N} \}$$

The correspondence between the code and the sets of plays should be apparent, as well as the fact that this conversion may be defined co-inductively (or inductively) over the structure of the program. The two programs may be combined into $(M^{\text{inc}} \cup M^{\text{get}})^{\bullet} : \dagger \text{Cell}[\mathbb{N}] \rightarrow \text{Counter}$. This provides the semantics of the implementation on individual calls to `Counter`. The full library semantics would allow for several calls to be made to the counter. This will be handled soon when we discuss the replay modality $\dagger -$ in detail.

Another important example of a strategy is the copycat strategy, which will play a fundamental role in our development.

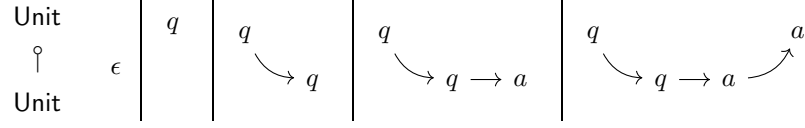
Definition 6.1.4. The *copycat strategy* $\text{seqcopy}_A : A \multimap A$ over the game A is the strategy defined by:

$$\text{seqcopy}_A := \{ s \in P_{A \multimap A} \mid \forall p \sqsubseteq_{\text{even}} s. p \upharpoonright_{A,-} = p \upharpoonright_{-,A} \}^{\bullet}$$

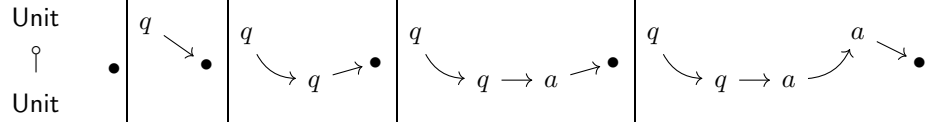
where $- \sqsubseteq_{\text{even}} -$ is the even-length prefix relation.

Example 6.1.5. The copycat strategy $\text{seqcopy} : A \multimap A$ plays the target game A by asking the same questions it receives in the target to the source component, and then redirecting

the responses it gets from the source back to the target. This is evident by considering the strategy $\text{seqcopy}_{\text{Unit}}$, which contains the following plays:



as well as the following plays implied by $\bullet$ -receptivity:



6.2 Strategy Composition

Strategy composition is defined, as expected, by the motto “interaction + hiding”. The only notable surprise is that $\bullet$ -tokens, which project to all components by our definition of composition, have a slightly different behavior from regular moves. Note that we will define the operation $-; -$ over plays of $P_{A \multimap B}^{\leq \infty}$ instead of just $P_{A \multimap B}^\bullet$ as we will use it later to define and analyze the composition for our liveness model.

Definition 6.2.1. For games A , B , and C , we define the set of interaction sequences:

$$M_{A \circ B \circ C} := (M_A^\circ + M_B^\circ + M_C^\circ)^\bullet$$

$$A \circ B \circ C := \{s \in M_{A \circ B \circ C}^\bullet \mid s \upharpoonright_{A,B,-} \in P_{A \multimap B}^{\leq \infty} \wedge s \upharpoonright_{-,B,C} \in P_{B \multimap C}^{\leq \infty}\}$$

where we extend the projections on plays over $A \multimap B$ into projections over the interactions $- \upharpoonright_{A,B,-}$, $- \upharpoonright_{-,B,C}$, and $- \upharpoonright_{A,-,C}$ restricting the plays to the displayed components.

We define the set of interactions of two plays, extended to sets of plays $S \subseteq P_{A \multimap B}^{\leq \infty}$ and $T \subseteq P_{B \multimap C}^{\leq \infty}$:

$$s \circ t := \{s' \in A \circ B \circ C \mid s' \upharpoonright_{A,B,-} = s \wedge s' \upharpoonright_{-,B,C} = t\} \quad S \circ T := \cup_{s \in S} \cup_{t \in T} s \circ t$$

Composition is then defined by interaction followed by hiding:

$$s ; t := (s \circ t) \upharpoonright_{A,-,C} \quad S ; T := \cup_{s \in S} \cup_{t \in T} s ; t$$

We call $- ; -$ composition by many names in this dissertation. Because of how layered system diagrams are usually depicted, we sometimes call it *vertical composition*. When focusing on the algebraic structure it is part of, we call it *categorical composition*. When distinguishing between the safety and liveness models, we call it *strategy composition*.

Example 6.2.1. Consider the strategy

$$\tau : \text{Counter} \multimap \{\text{getbool} : 1 \rightarrow \mathbf{Bool}\}$$

where $\mathbf{Bool} := \{\text{true}, \text{false}\}$ given by:

$$\tau := \downarrow \cup_{n \in \mathbb{N}} \text{getbool}() \cdot \text{get}() \cdot n \cdot (n \geq 0)$$

Here, what is meant by $n \geq 0$ is the $\mathbf{Bool}$ obtained by testing whether n is bigger or equal to 0. Then, let $\sigma : \dagger \text{Cell}[\mathbb{N}] \multimap \text{Counter}$ be exactly $\sigma := M_{\text{Counter}}^{\text{inc}} \cup M_{\text{Counter}}^{\text{get}}$ defined in Ex.

6.1.4. Then, the plays:

$$t = \begin{array}{c} \text{getbool}() \\ \searrow \\ \text{get}() \longrightarrow 0 \nearrow \text{false} \end{array} \in \sigma$$

and

$$s = \begin{array}{c} \text{get}() \\ \searrow \\ \text{rd}() \longrightarrow 0 \nearrow 0 \end{array} \in \tau$$

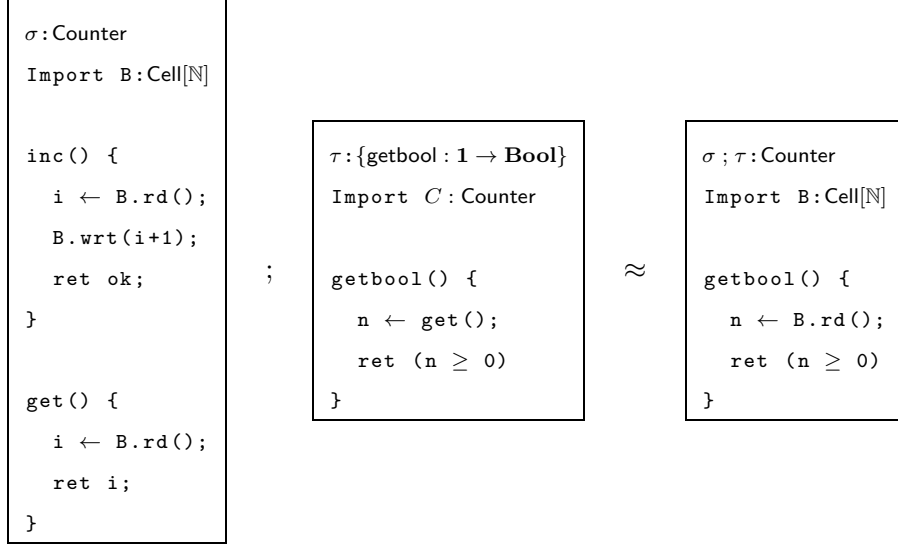
result in the interaction sequence:

$$s ; t = \begin{array}{c} \text{getbool}() \\ \searrow \\ \text{get}() \searrow \text{rd}() \longrightarrow 0 \nearrow \text{false} \end{array} \in \sigma ; \tau$$

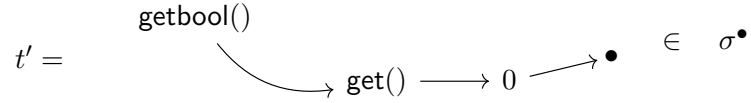
which after hiding generates the play:

$$s ; t = \begin{array}{c} \text{getbool}() \\ \searrow \\ \text{rd}() \longrightarrow 0 \nearrow \text{false} \end{array} \in \sigma ; \tau$$

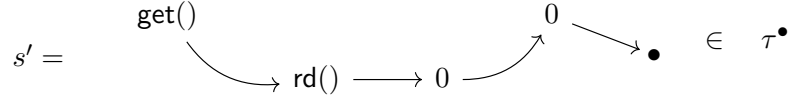
Intuitively, in terms of the respective programs these strategies represent, this corresponds to the inlining below (or to the semantics of the linking of the respective programs):



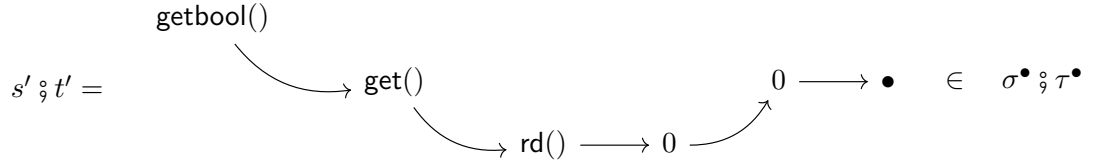
Now, if we consider $\sigma^\bullet$ and $\tau^\bullet$ then



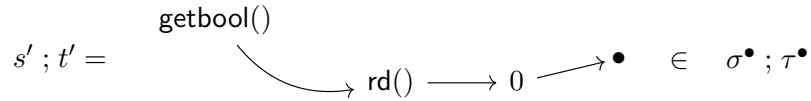
and



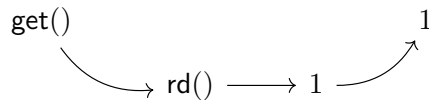
result in the interaction sequence:



where the $\bullet$ -token belongs to all three components involved. After hiding we obtain the play:



Meanwhile, s' does not compose with t , and s does not compose with t' , in the sense that $s' \circ t = s \circ t' = \emptyset$, because there is no $\bullet$ -token to synchronize with in s and t . Similarly, the play:



does not compose with t (or t' for that matter), in this case because the returns do not match on `Counter`, as $0 \neq 1$.

A key property for compositionality is that all the composition operations of our models be associative. Otherwise, the order in which we build certified systems will impact the resulting system, which is not desirable. The usual game semantics argument for associativity, with some extra straightforward reasoning caused by $\bullet$ -tokens and infinite plays, gives that:

Proposition 6.2.1. *For any sets of plays $S \subseteq P_{A \multimap B}^{\leq \infty}$, $T \subseteq P_{B \multimap C}^{\leq \infty}$, and $R \subseteq P_{C \multimap D}^{\leq \infty}$:*

$$S ; (T ; R) = (S ; T) ; R$$

In particular,

Corollary 6.2.1. *Strategy composition is associative.*

6.3 The Copycat and Neutrality

Intuitively, the copycat strategy $\text{seqcopy}_F : \dagger F \multimap \dagger F$ over an effect signature F , as defined in Def. 6.1.4, corresponds to the following program:

```

 $I_F : F$ 
Import Obj : F

f(a : par(f)) {
  v <- Obj.f(a);
  ret v
}
:

```

for every $f \in F$. Now, under the intuitive understanding of composition as inlining or linking, as described in Ex. 6.2.1, we see that inlining such a program into any program $M : F \rightarrow G$ or inlining a program $N : E \rightarrow F$ into I_F would result in the same program as M and N , respectively. We might expect the copycat strategy to behave as the neutral element for composition, considering these examples. This is indeed true for strategies generated by programs. It is, however, not true for the notion of strategy from Def. 6.1.1.

Indeed, consider the strategy $\perp : \text{Unit} \multimap \text{Unit}$ defined by $\perp := \downarrow \bullet = \{\epsilon, \bullet\}$. Note then that

$$\text{seqcopy}_{\text{Unit}} := (\downarrow q \cdot q \cdot a \cdot a)^\bullet$$

Therefore, we have:

$$\perp ; \text{seqcopy}_{\text{Unit}} = (\downarrow q)^\bullet \neq \perp$$

Similarly, for the strategy $\sigma : \text{Unit} \multimap \text{Unit}$ given by $\sigma := (\downarrow q \cdot q)^\bullet$, we have:

$$\text{seqcopy}_{\text{Unit}} ; \sigma = (\downarrow q \cdot q \cdot a)^\bullet \neq \sigma$$

It is folklore in game semantics that

Proposition 6.3.1. *For a sequential strategy $\sigma : A \multimap B$, $\text{seqcopy}_A ; \sigma ; \text{seqcopy}_B = \sigma$ if and only if σ is O-receptive.*

which gives as a corollary that:

Corollary 6.3.1. *The copycat strategy is the neutral element for strategy composition of O-receptive strategies.*

At the heart of the core result of this work will be a more fundamental explanation of this phenomenon. For now, this more traditional categorization will suffice. Note that this already provides an explanation for why in the case of strategies generated by programs the copycat will indeed be the neutral element for composition: the denotations of programs will always be O-receptive.

6.4 Independent Composition

We already saw how to compose two games so that they may be both played simultaneously. That is, given games A and B , we defined $A \otimes B$ which constructs all the sequential interleavings of plays of A and B . The intuition for what this does is quite clear when we

consider games given by effect signatures, where the tensor verifies the equation:

$$[E + F] \cong [E] \otimes [F]$$

Here, $E + F$ is the effect signature given by the following data:

$$|E + F| := E + F \quad \text{par}_{E+F} := [\text{par}_E, \text{par}_F] \quad \text{ar}_{E+F} := [\text{ar}_E, \text{ar}_F]$$

where given $f : S \rightarrow U$ and $g : T \rightarrow U$, $[f, g] : S + T \rightarrow U$ is the unique (up-to isomorphism) function implied by the universal property of the coproduct $- + -$ (in the category of sets and functions). In other words, $E + F$ has the effects of E and the effects of F , each with the same set of arguments and arity as they had in E and F .

Now, it remains to define what $- \otimes -$ does to strategies.

Definition 6.4.1. Given sets of traces $S \subseteq P_A^{\leq \infty}$ and $T \subseteq P_B^{\leq \infty}$, we define their horizontal composition $S \otimes T \subseteq P_{A \otimes B}^{\leq \infty}$ by $S \otimes T := \{s \in P_{A \otimes B}^{\leq \infty} \mid s \upharpoonright_{A,-} \in S \wedge s \upharpoonright_{-,B} \in T\}$

Note that, in particular, given strategies $\sigma : A \multimap B$ and $\tau : C \multimap D$, we may construct their tensor $\sigma \otimes \tau : (A \otimes B) \multimap (C \otimes D)$. What this operation does is consider the semantics of σ and τ as non-interferent with each other. That is, a behavior of $\sigma \otimes \tau$ is any (well-formed) trace s that behaves as σ on $s \upharpoonright_{A \multimap B, -}$ and as τ in $s \upharpoonright_{-, C \multimap D}$. Note that we are taking a convention that the projections $- \upharpoonright_{A \multimap B, -}$ and $- \upharpoonright_{-, C \multimap D}$ properly re-index the moves while projecting.

We call the operation $- \otimes -$ the tensor, or symmetric monoidal structure, when considering the algebraic structure it defines. We call it *horizontal composition* in the context of verification.

An important fact about the tensor is that it distributes over the categorical composition.

Proposition 6.4.1. *Whenever well-typed, the following equation of strategies holds:*

$$(\sigma \otimes \tau) ; (\sigma' \otimes \tau') = (\sigma ; \sigma') \otimes (\tau ; \tau')$$

Moreover, we say two games A and B are *isomorphic*, written $A \cong B$, whenever there

are O-receptive strategies $\phi : A \multimap B$ and $\phi^{-1} : B \multimap A$ (which are themselves called *isomorphisms*) such that:

$$\phi; \phi^{-1} = \text{seqcopy}_A \qquad \phi^{-1}; \phi = \text{seqcopy}_B$$

Then, $- \otimes -$ is both associative and unital up-to isomorphism:

$$(A \otimes B) \otimes C \cong A \otimes (B \otimes C) \qquad A \otimes \mathbf{1} \cong \mathbf{1} \otimes A \cong A$$

These isomorphisms moreover satisfy several coherence axioms that shall be formalized soon.

6.5 Background: Domains and Continuity

A key lesson of domain theory is a careful characterization of what kinds of sets make for good targets for the denotations of functions [Scott, 1970, 1972; Abramsky and Jung, 1994]. A central notion is that of a pointed DCPO, which turns out to be a mathematical structure supporting a computational notion of fixpoint by Kleene's fixpoint theorem. The fixpoint operator then gets used to define the denotations of important notions in computation, such as recursion and loops. It is therefore not surprising that we wish the set of strategies of a certain type to form some kind of domain, at the very least a pointed DCPO.

Definition 6.5.1. Let S be a set. A *preorder* $- \leq -$ over S consists of a relation $- \leq - \subseteq S \times S$ that is:

Reflexive: $\forall x \in S. x \leq x$.

Transitive: $\forall x, y, z \in S. x \leq y \wedge y \leq z \implies x \leq z$.

A preorder induces an equivalence relation $x \equiv y \iff x \leq y \wedge y \leq x$. A preorder is a *partial order* when it is moreover:

Antisymmetric: $\forall x, y \in S. x \leq y \wedge y \leq x \implies x = y$.

A preorder is *pointed* when it has a least element, i.e. there exists $\perp \in S$ such that $\perp \leq x$ for any $x \in S$. The least element is unique up to $\equiv$ (and unique when the preorder is a partial order).

An *upper bound* of two elements $x, y \in S$ is an element $z \in S$ such that $x \leq z$ and $y \leq z$. Dually, a *lower bound* is an element $z \in S$ such that $z \leq x$ and $z \leq y$.

A *directed set* $D \subseteq S$ in a preordered set S is a non-empty set (i.e. $D \neq \emptyset$) such that for any two $x, y \in D$, there exists an upper bound $z \in D$ of x and y .

A *directed-complete partial order* (DCPO) is a partial order such that every directed set D admits a supremum $\bigvee D \in S$. Recall a *supremum* is a *least upper bound*, that is, for any $x \in D$, $x \leq \bigvee D$ and for any z such that for any $x \in D$, $x \leq z$, it follows that $\bigvee D \leq z$.

A *pointed* DCPO is a DCPO with a least element.

Note that a pointed DCPO is a DCPO for which any directed or empty set admits a supremum. Now, a key requirement we have identified in Ch. 3 is that any composition operations we may define be monotonic. In fact, we have a stronger requirement.

Definition 6.5.2. Let $(S, - \leq_S -)$ and $(T, - \leq_T -)$ be DCPOs. A function $f : S \rightarrow T$ is a Scott-continuous function when it preserves directed suprema, that is, for every directed subset $D \subseteq S$ it follows that $f(\bigvee D) = \bigvee f(D)$.

It is well-known (and a fun exercise) that preservation of directed suprema implies monotonicity.

Now, fortunate for us, our domains of strategies (or other denotations for computation) are actually even more well-behaved than just pointed DCPOs.

Definition 6.5.3. A *join-semilattice* $(S, - \leq -)$ is a preordered set such that any two elements $x, y \in S$ have a least upper bound $x \vee y \in S$, unique up to $\equiv$.

A *complete* join-semilattice, or *sup-lattice*, is one such that *any* subset $X \subseteq S$ admits a supremum $\bigvee X$, unique up to $\equiv$.

Traditionally, join-semilattices and sup-lattices are defined over partial orders. We state them for preorders, as this generality will be needed in later chapters. When the underlying preorder is a partial order, these definitions recover the standard ones; in particular, a

complete join-semilattice over a partial order is a pointed DCPO (indeed, the “any” in the condition includes both the empty set and any directed set).

Definition 6.5.4. Let $(S, - \leq_S -)$ and $(T, - \leq_T -)$ be sup-lattices. A continuous function $f : S \rightarrow T$ between the sup-lattices is a function f that preserves all non-empty suprema, that is, for any subset $X \subseteq S$, we have $f(\bigvee X) = \bigvee f(X)$.

Typically, one would define homomorphisms of sup-lattices as those that preserve *all* joins. We diverge here from most approaches as we find that most of our models of systems do not admit continuous functions in this sense. For example, for strategies the bottom element would be $\perp := \{\epsilon\}$. Note then that while $\sigma ; \perp = \perp$, $\perp ; \sigma$ might be different from $\perp$. This is not unlike the difference between Scott-continuous functions and *strict* Scott-continuous functions (where *strict* Scott-continuous functions must preserve the bottom element of a pointed DCPO). Generally, we will characterize our sets of possible denotations, such as our domain of strategies, as a sup-lattice. In fact, this makes our domains complete lattices by a standard argument.

6.6 Refinement

The next key ingredient for our compositional models will be a notion of refinement, as discussed in Ch. 3. For strategies, this is defined quite simply as set containment.

Definition 6.6.1. We say a strategy $\sigma : A \multimap B$ *refines* a strategy $\tau : A \multimap B$, written $\sigma \sqsubseteq_S \tau$ when $\sigma \subseteq \tau$.

As promised, this makes strategies into a nice domain:

Proposition 6.6.1. *The set of all strategies of type $A \multimap B$ ordered by $- \sqsubseteq_S -$ forms a sup-lattice.*

Then, key to our abstraction results will be the following important facts:

Proposition 6.6.2. *$- ; -$ is a complete function on both arguments. That is, for any strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$, the functions $- ; \tau$ and $\sigma ; -$ are both complete.*

Similarly, $- \otimes -$ is a complete function on both arguments.

Note that the order on strategies given by $-\sqsupseteq_S-$ goes in the opposite direction from set inclusion: $\sigma \sqsupseteq_S \tau$ when $\sigma \subseteq \tau$, so that a strategy with fewer behaviours is considered more refined. For the abstract framework, it is convenient to write $\sigma \leq \tau$ (following the refinement order) when $\sigma \sqsupseteq \tau$ (the computation order), the direction opposite to set inclusion. Under this convention, the sup-lattice structure described above dualizes. This motivates the following dual definitions.

Definition 6.6.2. A *meet-semilattice* $(S, - \leq -)$ is a preordered set such that any two elements $x, y \in S$ have a greatest lower bound $x \wedge y \in S$, unique up to $\equiv$.

A *complete* meet-semilattice, or *inf-lattice*, is one such that *any* subset $X \subseteq S$ admits an infimum $\bigwedge X$, unique up to $\equiv$.

Definition 6.6.3. Let $(S, - \leq_S -)$ and $(T, - \leq_T -)$ be inf-lattices. A *cocontinuous* function $f : S \rightarrow T$ between the inf-lattices is a function f that preserves all non-empty infima, that is, for any non-empty subset $X \subseteq S$, we have $f(\bigwedge X) = \bigwedge f(X)$.

Under the refinement convention, the sup-lattice of strategies becomes an inf-lattice with infima given by union, and the continuous functions become cocontinuous. This is the enrichment we develop in the following chapters, where $-\leq-$ denotes the refinement order throughout.

Example 6.6.1. In Ex. 6.1.2, we defined the strategy ν_{Counter} specifying a counter object. We did this in two ways, through a predicate and through a state transition system $\mathcal{V}_{\text{Counter}}$. Of course, these two examples satisfy that:

$$\mathcal{V}_{\text{Counter}}^\#(0, \text{idle}) \subseteq \nu_{\text{Counter}}$$

(as well as the reverse containment in this case). In practice, we would only have access to integers up-to whatever the underlying hardware allows. For example, in a 32-bit machine, a counter may hold any value less than 2^{32} represented as an `uint`. So, we could define a transition system $\mathcal{V}_{\text{Counter}}^{\text{uint}}$ which instead has states in $(\text{uint} + \perp) \times (\text{Counter} + \text{idle} + \text{stv})$ and

transitions given by:

$$\begin{array}{c}
\frac{n \in \text{uint}}{(n, \text{idle}) \xrightarrow{\text{get}} (n, \text{get})} \qquad \frac{n \in \text{uint}}{(n, \text{get}) \xrightarrow{n} (n, \text{idle})} \\
\\
\frac{}{(\perp, \text{idle}) \xrightarrow{\text{get}} (\perp, \text{get})} \qquad \frac{n \in \mathbb{N}}{(\perp, \text{get}) \xrightarrow{n} (\perp, \text{idle})} \\
\\
\frac{n \in \text{uint}}{(n, \text{idle}) \xrightarrow{\text{inc}} (n, \text{inc})} \qquad \frac{n + 1 \in \text{uint}}{(n, \text{inc}) \xrightarrow{\text{ok}} (n + 1, \text{idle})} \qquad \frac{n + 1 \notin \text{uint}}{(n, \text{inc}) \xrightarrow{\text{ok}} (\perp, \text{idle})} \\
\\
\frac{}{(\perp, \text{idle}) \xrightarrow{\text{inc}} (\perp, \text{inc})} \qquad \frac{}{(\perp, \text{inc}) \xrightarrow{\text{ok}} (\perp, \text{idle})} \\
\\
\frac{c \neq \text{starved}}{(n, c) \xrightarrow{\bullet} (n, \text{starved})}
\end{array}$$

Essentially, if the counter overflows, we go into a state $\perp$ which represents undefined behavior. Then, we would have:

$$\llbracket \mathcal{V}_{\text{Counter}} \rrbracket_S \subseteq \llbracket \mathcal{V}_{\text{Counter}}^{\text{uint}} \rrbracket_S$$

representing the fact that the counter with infinite storage has more defined behaviors than the one with finite storage with potential overflows.

Chapter 7

Background: Enriched (Semi)Category Theory

We have now considered at least two types of strategies (strategies, and O-receptive strategies), as well as the composition operations $- ; -$ and $- \otimes -$, both satisfying several important properties, and a notion of refinement $- \sqsubseteq_S -$. These are common ingredients of our compositional models. Therefore, it is useful to develop a common language for all of these models, which then enables us to state results and constructions in a model-agnostic way, so that they may be re-used in any compositional model we develop.

This can be done quite elegantly in the language of category theory [Mac Lane, 1998], which we now review. To us, it will be essential to incorporate the refinement into the compositional structures we consider, as well as to consider categories without identities, for technical reasons. Because of this, the reader will not find a gentle introduction to category theory here. Instead, we will provide a quick overview of core concepts building up to enriched category theory [Kelly, 1982].

7.1 Categories, Functors, Natural Transformations

Definition 7.1.1. A *category* $\mathbf{C}$ consists of the following data:

Objects: A collection of *objects* $\mathbf{C}$,

Morphisms: An assignment $\mathbf{C}(A, B)$ of a collection of *morphisms* to each pair of objects $A, B \in \mathbf{C}$, where we write $f : A \rightarrow B \in \mathbf{C}$ whenever $f \in \mathbf{C}(A, B)$.

Composition: For all objects $A, B, C \in \mathbf{C}$, a mapping

$$- \circ_{A,B,C} - : \mathbf{C}(B, C) \times \mathbf{C}(A, B) \rightarrow \mathbf{C}(A, C),$$

moreover associative in the sense that:

$$(h \circ_{B,C,D} g) \circ_{A,B,C} f = h \circ_{B,C,D} (g \circ_{A,B,C} f)$$

Neutral Element: For every object $A \in \mathbf{C}$, a morphism $\mathbf{id}_A \in \mathbf{C}(A, A)$ such that for any object $B \in \mathbf{C}$ and morphisms $g \in \mathbf{C}(A, B)$ and $f \in \mathbf{C}(B, A)$, it follows that:

$$\mathbf{id}_A \circ g = g \qquad f \circ \mathbf{id}_A = f$$

It is common to write just $- \circ -$ for the composition operation. When several categories are involved, we might write $\circ_{\mathbf{C}}$ for the composition operation in category $\mathbf{C}$.

Example 7.1.1. The category **Set** has sets as objects and functions $f : X \rightarrow Y$ as morphisms, which are composed pointwise, i.e., $(g \circ f)(x) := g(f(x))$, admitting the identity function $\mathbf{id}_X(x) = x$ as neutral element.

Example 7.1.2. The structure defined by inf-lattices as objects, cocontinuous functions between them as morphisms, function composition as the categorical composition, and the identity function as the neutral element defines a category **Inf**.

It is notable that all the examples above have some set with extra structure as objects, and functions with side-conditions as morphisms, with composition given by the usual function composition. These set-based categories are called *concrete* categories. In contrast, the categories we will build for our models of computation have a different flavor to composition. For example, strategy composition is not given by functional composition.

Categories come with a natural notion of equivalence of objects up-to the categorical structure.

Definition 7.1.2. In a category $\mathbf{C}$ a morphism $f : A \rightarrow B \in \mathbf{C}$ is called an *isomorphism* when it admits a left and right inverse (necessarily the same). That is, there exists $f^{-1} : B \rightarrow A \in \mathbf{C}$ such that

$$f^{-1} \circ f = \mathbf{id}_A \qquad f \circ f^{-1} = \mathbf{id}_B$$

Two objects A and B are said *isomorphic*, written $A \cong B$, when there exists an isomorphism between them.

To accommodate an operation like the tensor, it is usual to consider extra structures on a category. To do this, we need several intermediary notions. Essentially, we wish to accommodate the tensor $- \otimes -$ as some kind of structure-preserving morphism between $\mathbf{C}$ and $\mathbf{C}$. However, the tensor takes two objects or morphisms as input, so, intuitively, it must be typed as $- \otimes - : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$. This requires both defining what it means to take the product of two categories and what “structure-preserving” morphisms between categories are.

Definition 7.1.3. Given categories $\mathbf{C}$ and $\mathbf{D}$, their *product category* $\mathbf{C} \times \mathbf{D}$ is the category given by the following data:

Objects are pairs (C, D) of objects $C \in \mathbf{C}$ and $D \in \mathbf{D}$,

Morphisms are pairs $(f, g) : (A, B) \rightarrow (C, D)$ of morphisms $f : A \rightarrow C \in \mathbf{C}$ and $g : B \rightarrow D \in \mathbf{D}$, and

Composition is defined coordinate-wise:

$$(f', g') \circ_{\mathbf{C} \times \mathbf{D}} (f, g) = (f' \circ_{\mathbf{C}} f, g' \circ_{\mathbf{D}} g)$$

The neutral element is then given by $(\mathbf{id}_C, \mathbf{id}_D) : (C, D) \rightarrow (C, D) \in \mathbf{C} \times \mathbf{D}$.

Definition 7.1.4. Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A *functor* $F : \mathbf{C} \rightarrow \mathbf{D}$ between $\mathbf{C}$ and $\mathbf{D}$ consists of

- an assignment of an object $F A \in \mathbf{D}$ for each object $A \in \mathbf{C}$, and

- an assignment of a morphism $F f \in \mathbf{D}(F A, F B)$ to any morphism $f \in \mathbf{C}(A, B)$ moreover satisfying that, whenever well-typed:

$$F (g \circ_{\mathbf{C}} f) = F g \circ_{\mathbf{D}} F f$$

A *bifunctor* between $\mathbf{C}$, $\mathbf{D}$ and $\mathbf{E}$ is a functor $F : \mathbf{C} \times \mathbf{D} \rightarrow \mathbf{E}$.

In fact, categories with functors as morphisms form themselves a category (ignoring some issues of size). A final step toward axiomatizing the properties of the tensor is to consider morphisms that preserve the structure of functors. These are called natural transformations.

Definition 7.1.5. A *natural transformation* $\eta : F \rightarrow G$ between two functors $F, G : \mathbf{C} \rightarrow \mathbf{D}$ consists of an assignment η_A of a morphism $\eta_A : F A \rightarrow G A \in \mathbf{D}$ to each object $A \in \mathbf{C}$ satisfying that the following diagram commutes for any morphism $f : A \rightarrow B \in \mathbf{C}$:

$$\begin{array}{ccc} F A & \xrightarrow{\eta_A} & G A \\ F f \downarrow & & \downarrow G f \\ F B & \xrightarrow{\eta_B} & G B \end{array}$$

in the sense that the equation below holds:

$$\forall f : A \rightarrow B \in \mathbf{C} \quad . \quad \eta_B \circ F f = G f \circ \eta_A$$

A *natural isomorphism* is a natural transformation $\eta : F \rightarrow G$ such that for every $A \in \mathbf{C}$, η_A is an isomorphism.

The reader will not be surprised at this point to find out that functors with natural transformations as morphisms themselves form a category (again, ignoring some size issues). Natural transformations in categorical semantics of programming languages correspond to parametric functions. To us, they will be mostly relevant in their use to state coherence laws for categorical structures.

The definition of a natural transformation features the first instance of a commutative diagram. A commutative diagram consists of a directed graph with objects of a category as nodes, and morphisms as edges. The morphisms must be well-typed with respect to

their source and target nodes. Every path in a commutative diagram defines a composite morphism, for instance the path

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$$

represents the composite $h \circ (g \circ f)$. Two paths in a commutative diagram with the same start and end node define an equation which holds when the diagram *commutes* or is *commutative*. For instance, the diagram:

$$\begin{array}{ccccc} A & \xrightarrow{f} & B & \xrightarrow{g} & C \\ h \downarrow & & \downarrow k & & \downarrow l \\ D & \xrightarrow{m} & E & \xrightarrow{n} & F \end{array}$$

commutes when both inner squares commute, i.e., when $k \circ f = m \circ h$ and $l \circ g = n \circ k$. There are then three paths from A to F , all equal:

$$l \circ g \circ f = n \circ k \circ f = n \circ m \circ h$$

7.2 Monoidal Categories

We are now ready to axiomatize the notion of a category equipped with a tensor operation.

Definition 7.2.1. A *monoidal* category consists of a category $\mathbf{C}$ equipped with a bifunctor:

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

and an identified unit object $\mathbf{1}$ admitting the following natural isomorphisms:

left unitor: $\lambda_A : \mathbf{1} \otimes A \cong A$,

right unitor: $\rho_A : A \otimes \mathbf{1} \cong A$,

associator: $\alpha_{A,B,C} : (A \otimes B) \otimes C \cong A \otimes (B \otimes C)$

moreover satisfying several coherence equations, as depicted by the following diagrams:

$$\begin{array}{ccc}
(A \otimes \mathbf{1}) \otimes B & \xrightarrow{\alpha_{A,\mathbf{1},B}} & A \otimes (\mathbf{1} \otimes B) \\
& \searrow \rho_A \otimes \text{id}_B & \swarrow \text{id}_A \otimes \lambda_B \\
& A \otimes B &
\end{array}$$

$$\begin{array}{ccc}
& ((A \otimes B) \otimes C) \otimes D & \\
\alpha_{A,B,C} \otimes \text{id}_D \swarrow & & \searrow \alpha_{A \otimes B,C,D} \\
(A \otimes (B \otimes C)) \otimes D & & (A \otimes B) \otimes (C \otimes D) \\
\downarrow \alpha_{A,B \otimes C,D} & & \downarrow \alpha_{A,B,C \otimes D} \\
A \otimes ((B \otimes C) \otimes D) & \xrightarrow{\text{id}_A \otimes \alpha_{B,C,D}} & A \otimes (B \otimes (C \otimes D))
\end{array}$$

A monoidal category is *symmetric* if it additionally admits a natural isomorphism:

braiding: $\beta_{A,B} : A \otimes B \cong B \otimes A$

such that the inverse $\beta_{A,B}^{-1}$ of $\beta_{A,B}$ is $\beta_{B,A}$ and satisfying the coherence equation depicted by the following diagram:

$$\begin{array}{ccccc}
(A \otimes B) \otimes C & \xrightarrow{\alpha_{A,B,C}} & A \otimes (B \otimes C) & \xrightarrow{\beta_{A,B \otimes C}} & (B \otimes C) \otimes A \\
\beta_{A,B} \otimes \text{id}_C \downarrow & & & & \downarrow \alpha_{B,C,A} \\
(B \otimes A) \otimes C & \xrightarrow{\alpha_{B,A,C}} & B \otimes (A \otimes C) & \xrightarrow{\text{id}_B \otimes \beta_{A,C}} & B \otimes (C \otimes A)
\end{array}$$

Example 7.2.1. The category **Set** with the cartesian product $(\mathbf{Set}, - \times -, \mathbf{1})$ defines a symmetric monoidal category (where $\mathbf{1}$ is some identified singleton set).

Example 7.2.2. A crucial example of a symmetric monoidal category for us will be the category **Inf**. The obvious way to endow it with a symmetric monoidal structure is by constructing the product lattice. That is, given complete meet-semilattices $(S, - \leq_S -)$ and $(T, - \leq_T -)$, we can construct their product lattice $(S \times T, - \leq_{S \times T} -)$ where $(x, y) \leq_{S \times T} (w, z)$ when $x \leq_S w$ and $y \leq_T z$. This results in a cocontinuous pairing operation $(-, -)$, which is however only so jointly on the two variables. That is, the cocontinuity is expressed by the equation:

$$f\left(\bigwedge Z\right) = \bigwedge_{(x,y) \in Z} f(x, y)$$

This, and several other abnormalities of the product of lattices make it not appropriate for modeling systems. Specifically, it is typical that in models of systems the composition operation $- \circ -$ be only cocontinuous on each variable separately.

This is what the tensor defined by Joyal and Tierney [1984] captures. The key idea is to *define* the tensor product exactly as the object satisfying the universal property of being the target of maps that are bimorphisms (i.e., cocontinuous on each variable).

Formally, observe that there exists a forgetful functor $U : \mathbf{Inf} \rightarrow \mathbf{Set}$ given by:

$$(S, - \leq_S -) \xrightarrow{U} S \quad f : (S, - \leq_S -) \rightarrow (T, - \leq_T -) \xrightarrow{U} f : S \rightarrow T$$

that is, U simply forgets the lattice structure and the cocontinuity of morphisms.

A bimorphism f of inf-lattices $(S, - \leq_S -)$, $(T, - \leq_T -)$ and $(R, - \leq_R -)$ is a function $f : S \times T \rightarrow R$ that is cocontinuous in each coordinate, in the sense that for every (non-empty) $X \subseteq S$ and $Y \subseteq T$, and every $x \in S$ and $y \in T$:

$$f\left(\bigwedge X, y\right) = \bigwedge_{x \in X} f(x, y) \quad f\left(x, \bigwedge Y\right) = \bigwedge_{y \in Y} f(x, y)$$

Note that such a morphism satisfies that:

$$f\left(\bigwedge X, \bigwedge Y\right) = \bigwedge_{x \in X} f(x, Y) = \bigwedge_{x \in X} \bigwedge_{y \in Y} f(x, y) = \bigwedge_{(x, y) \in X \times Y} f(x, y)$$

So, if we let $L_S = (S, - \leq_S -)$ and $L_T = (T, - \leq_T -)$, the tensor $L_S \otimes L_T$ is defined as the inf-lattice (up-to isomorphism) such that there exists a bimorphism

$$h : U L_S \times U L_T \rightarrow U (L_S \otimes L_T)$$

such that for any inf-lattice $L_U = (U, - \leq_U -)$ and any bimorphism $f : S \times T \rightarrow U$ there exists a unique cocontinuous function $\bar{f} : L_S \otimes L_T \rightarrow L_U$ such that

$$U \bar{f} \circ h = f$$

Or, diagrammatically:

$$\begin{array}{ccc} S \times T & \xrightarrow{h} & U (L_S \otimes L_T) \\ & \searrow f & \downarrow U \bar{f} \\ & & R \end{array} \quad \begin{array}{ccc} & & L_S \otimes L_T \\ & \xleftarrow{R} & \downarrow \exists! \bar{f} \\ & & L_R \end{array}$$

This actually ensures that there is a bijection between bimorphisms and cocontinuous functions out of the tensor.

Now, this is only really a definition if an object satisfying this property exists. Fortunately, it can be shown that the free inf-lattice generated by formal elements $x \otimes y$ (for $x \in S$ and $y \in T$), that is $\mathcal{P}(\{x \otimes y \mid x \in S \wedge y \in T\})$ ordered by $- \supseteq -$, subject to the equations (in that we take a quotient under these equations):

$$\begin{aligned} (\bigwedge X) \otimes y &= \bigwedge_{x \in X} (x \otimes y), & X \neq \emptyset \\ x \otimes (\bigwedge Y) &= \bigwedge_{y \in Y} (x \otimes y), & Y \neq \emptyset \end{aligned}$$

The concrete construction is involved and may be found in [Joyal and Tierney, 1984]. The unit for the tensor is the unique inf-lattice over the singleton set $\mathbf{1}$. This is enough to define all of the structural morphisms of the symmetric monoidal structure $(\mathbf{Inf}, - \otimes -, \mathbf{1})$. It remains to discuss the functorial structure of the tensor.

Indeed, for us, while we omit the construction of the tensor inf-lattice, we remark that to define a cocontinuous function out of the tensor, by the universal property, it suffices to provide a bilinear morphism. Then, the universal property guarantees that:

$$\bar{f} \left(\bigwedge_{i \in I} x_i \otimes y_i \right) = \bigwedge_{i \in I} f(x_i, y_i)$$

This means that to define the tensor $f \otimes g : L_S \otimes L_T \rightarrow L_U \otimes L_V$ it suffices to define:

$$(f \otimes g)(x \otimes y) := f(x) \otimes g(y)$$

Because it can be verified that $- \otimes -$ is a bimorphism by construction, it then follows that the formula above uniquely defines a cocontinuous function $f \otimes g : L_S \otimes L_T \rightarrow L_U \otimes L_V$.

7.3 Enriched (Semi)Categories

It is quite convenient to think of the collection of morphisms of a category as a collection, or, in fact, just a set for the most part. However, we wish to have good denotations for programs, modules, components, transducers, etc. Denotational semantics has identified precisely what structures sets must come with to provide good denotational domains for computation. Therefore, we desire to enforce and study the models we develop taking into account that they are denotational domains.

To do this, we must move into *enriched* category theory. The key idea is that where we had chosen for objects to be sets, where we had considered sets of morphisms between objects, assignments to be set-assigning functions, etc., we can generalize it so that the collection of morphisms of an enriched category $\mathbf{C}$ is instead an object in an (ordinary) category $\mathbf{V}$. Of course, to do this effectively, we must have some extra structure on $\mathbf{V}$. Particularly, we wish it to be a monoidal category $(\mathbf{V}, - \otimes -, \mathbf{1})$ as this allows us to define the composition operation as a morphism:

$$- \circ_{A,B,C} - : \mathbf{C}(B, C) \otimes \mathbf{C}(A, B) \rightarrow \mathbf{C}(A, C) \in \mathbf{V}$$

Before we give the formal definition, it is important to first generalize the notion of being an *element* of a set.

Definition 7.3.1. In a monoidal category $(\mathbf{C}, - \otimes -, \mathbf{1})$ an *element* of an object $A \in \mathbf{C}$ is a morphism $e : \mathbf{1} \rightarrow A$.

This is more traditionally, especially when $\mathbf{1}$ is a terminal object, called a global element, a terminology coming from sheaf theory. However, in arbitrary monoidal categories, the analogy breaks down and it might instead be closer to the notions of a *point*, a *scalar*, or a *state* depending on the particular monoidal category. We opt to call it simply an element for simplicity.

Example 7.3.1. The intuition for why a morphism $e : \mathbf{1} \rightarrow X$ might be called an element comes from the category $(\mathbf{Set}, - \times -, \mathbf{1})$, and many categories derived from $\mathbf{Set}$ or where $\mathbf{1}$ is a *terminal* object, where morphisms $\llbracket x \rrbracket : \mathbf{1} \rightarrow X$ are in bijection with elements $x \in X$.

Indeed, given an element $x \in X$ we simply define $\lfloor x \rfloor$ as the function $\lfloor x \rfloor(*) = x$ where $* \in \mathbf{1}$.

Example 7.3.2. Our model of strategies provides a quite distinct flavor of “element” than what is seen in **Set**. Indeed, we have seen several examples so far of strategies $\nu : \mathbf{1} \multimap A$ as trace-based specifications of safety properties for transition systems.

Example 7.3.3. Similarly, in our monoidal category $(\mathbf{Inf}, - \otimes -, \mathbf{1})$, a cocontinuous function $x : \mathbf{1} \rightarrow L$ simply picks an element of the lattice L .

This provides all the background we need to define the notion of an enriched category. We give the definition in two parts, for reasons that shall become clear later.

Definition 7.3.2. Given a monoidal category $(\mathbf{V}, - \otimes -, \mathbf{1})$, a $\mathbf{V}$ -enriched semicategory $\underline{\mathbf{C}}$ consists of the following data:

Objects: A collection of objects $\underline{\mathbf{C}}$,

Morphisms: An assignment $\underline{\mathbf{C}}(-, -)$ of an object $\underline{\mathbf{C}}(A, B) \in \mathbf{V}$ to any pair of objects $A, B \in \underline{\mathbf{C}}$,

Composition: For all objects $A, B, C \in \underline{\mathbf{C}}$, a morphism

$$- \circ_{A,B,C} - : \underline{\mathbf{C}}(B, C) \otimes \underline{\mathbf{C}}(A, B) \rightarrow \underline{\mathbf{C}}(A, C) \in \mathbf{V},$$

moreover associative in that the following diagram commutes:

$$\begin{array}{ccccc}
& & (\underline{\mathbf{C}}(C, D) \otimes \underline{\mathbf{C}}(B, C)) \otimes \underline{\mathbf{C}}(A, B) & & \\
& \swarrow \circ_{B,C,D} \otimes \text{id} & & \searrow \alpha & \\
\underline{\mathbf{C}}(B, D) \otimes \underline{\mathbf{C}}(A, B) & & & & \underline{\mathbf{C}}(C, D) \otimes (\underline{\mathbf{C}}(B, C) \otimes \underline{\mathbf{C}}(A, B)) \\
\downarrow \circ_{A,B,D} & & & & \downarrow \text{id} \otimes \circ_{A,B,C} \\
\underline{\mathbf{C}}(A, D) & \xleftarrow{\circ_{A,C,D}} & & & \underline{\mathbf{C}}(C, D) \otimes \underline{\mathbf{C}}(A, C)
\end{array}$$

A $\mathbf{V}$ -enriched category $\mathbf{C}$ is a $\mathbf{V}$ -enriched semicategory $\underline{\mathbf{C}}$ together with

Neutral element: An element $\text{id}_A : \mathbf{1} \rightarrow \mathbf{C}(A, A) \in \mathbf{V}$ for every $A \in \mathbf{C}$ such that the following diagrams commute:

$$\begin{array}{ccc}
\mathbf{1} \otimes \mathbf{C}(A, B) & & \mathbf{C}(A, B) \otimes \mathbf{1} \\
\text{id}_B \otimes \mathbf{C}(A, B) \downarrow & \searrow \lambda & \text{C}(A, B) \otimes \text{id}_A \downarrow \\
\mathbf{C}(B, B) \otimes \mathbf{C}(A, B) & \xrightarrow{\circ_{A, B, B}} & \mathbf{C}(A, B) & \quad \quad & \mathbf{C}(A, B) \otimes \mathbf{C}(A, A) \xrightarrow{\circ_{A, A, B}} \mathbf{C}(A, B) \\
& & & \nearrow \rho &
\end{array}$$

The diagrams encode the associativity of the composition operation and the neutrality of the neutral elements up-to the structural isomorphisms of the monoidal category $\mathbf{V}$. Note that a **Set**-enriched category is just an ordinary category (more accurately, a locally small category). The notion of an enriched semicategory was originally developed by Moens et al. [2002]. Since it will be imperative to consider models of specifications which do not assemble into categories once we consider concurrent computation, we will develop some of our constructions in the absence of neutral elements for composition. This does create the annoyance that we are unable to define the notion of an enriched monoidal semicategory, as there are no neutral elements to state the coherence laws with. We will handle this in due time.

Enriched categories naturally admit a notion of an enriched functor.

Definition 7.3.3. A $\mathbf{V}$ -enriched semifunctor $F : \underline{\mathbf{C}} \rightarrow \underline{\mathbf{D}}$ between two $\mathbf{V}$ -enriched semicategories consists of:

- an assignment of an object $F A \in \underline{\mathbf{D}}$ for each object $A \in \underline{\mathbf{C}}$, and
- for each pair of objects $A, B \in \underline{\mathbf{C}}$, a morphism $F_{A, B} : \underline{\mathbf{C}}(A, B) \rightarrow \underline{\mathbf{D}}(F A, F B)$ in $\mathbf{V}$

such that the following diagram commutes:

$$\begin{array}{ccc}
\underline{\mathbf{C}}(B, C) \otimes \underline{\mathbf{C}}(A, B) & \xrightarrow{\circ_{A, B, C}^{\underline{\mathbf{C}}}} & \underline{\mathbf{C}}(A, C) \\
F_{B, C} \otimes F_{A, B} \downarrow & & \downarrow F_{A, C} \\
\underline{\mathbf{D}}(F B, F C) \otimes \underline{\mathbf{D}}(F A, F B) & \xrightarrow{\circ_{F A, F B, F C}^{\underline{\mathbf{D}}}} & \underline{\mathbf{D}}(F A, F C)
\end{array}$$

A $\mathbf{V}$ -enriched functor $F : \mathbf{C} \rightarrow \mathbf{D}$ between two $\mathbf{V}$ -enriched categories is a $\mathbf{V}$ -enriched semifunctor between the underlying semicategories that additionally preserves identities, in that the following diagram commutes:

$$\begin{array}{ccc}
& \mathbf{1} & \\
\text{id}_A^{\mathbf{C}} \swarrow & & \searrow \text{id}_{F A}^{\mathbf{D}} \\
\mathbf{C}(A, A) & \xrightarrow{F_{A, A}} & \mathbf{D}(F A, F A)
\end{array}$$

Then, in its essence, an enriched monoidal category is given by the following:

Definition 7.3.4. A *monoidal $\mathbf{V}$ -enriched category* consists of a $\mathbf{V}$ -enriched category $\mathbf{C}$ equipped with:

tensor: an assignment of an object $A \otimes B \in \mathbf{C}$ to each pair of objects $A, B \in \mathbf{C}$, together with morphisms in $\mathbf{V}$:

$$- \otimes_{A,B,C,D} - : \mathbf{C}(A, C) \otimes_{\mathbf{V}} \mathbf{C}(B, D) \rightarrow \mathbf{C}(A \otimes B, C \otimes D)$$

compatible with composition and identities,

unit: an identified unit object $\mathbf{1} \in \mathbf{C}$,

admitting (enriched) natural isomorphisms:

left unitor: $\lambda_A : \mathbf{1} \otimes A \cong A$,

right unitor: $\rho_A : A \otimes \mathbf{1} \cong A$,

associator: $\alpha_{A,B,C} : (A \otimes B) \otimes C \cong A \otimes (B \otimes C)$

satisfying the same triangle and pentagon coherence equations as in the non-enriched case.

A monoidal $\mathbf{V}$ -enriched category is *symmetric* if it additionally admits an (enriched) natural isomorphism:

braiding: $\beta_{A,B} : A \otimes B \cong B \otimes A$

such that $\beta_{A,B}^{-1} = \beta_{B,A}$ and satisfying the hexagon coherence equation.

These compatibility conditions are defined similarly to how equations on categories translate to diagrams in $\mathbf{V}$ in the definitions of category and functor. Since we can treat our models for the most part as having these structures strictly, we take the liberty of avoiding giving all the diagrams. The key novelty of the definition of an *enriched* monoidal category is simply that the tensor $- \otimes -$ must now be a morphism in $\mathbf{V}$. In the case of $\mathbf{Inf}$, this is simply the claim that it is cocontinuous.

Example 7.3.4. We can encapsulate all of the structure of sequential strategies we have discussed so far in the following proposition:

Proposition 7.3.1. *Let $\underline{\mathbf{Safe}}^{\text{seq}}$ be the structure described by the following data:*

Objects: *its objects are games A, B, C ;*

Morphisms: *as morphisms $\underline{\mathbf{Safe}}^{\text{seq}}(A, B)$ the inf-lattice of strategies $\sigma : A \multimap B$, $\tau : B \multimap C$, ordered by the refinement preorder, with infima given by arbitrary union;*

Morphism Composition: *with the categorical composition of morphisms given by*

$$- ; - : \underline{\mathbf{Safe}}^{\text{seq}}(A, B) \times \underline{\mathbf{Safe}}^{\text{seq}}(B, C) \rightarrow \underline{\mathbf{Safe}}^{\text{seq}}(A, C)$$

*Then, this structure defines a **Inf**-enriched semicategory.*

Moreover, by our observations in Prop. 6.3.1:

Proposition 7.3.2. *The **Inf**-enriched sub-semicategory $\mathbf{Safe}^{\text{seq}}$ of $\underline{\mathbf{Safe}}^{\text{seq}}$ defined by restricting strategies to **O**-receptive ones defines an **Inf**-enriched category where neutral elements are given by seqcopy_A .*

Being able to encapsulate this sophisticated compositional structure succinctly is an immediate benefit of our review of enriched category theory. We will see, however, that later we can re-use many of the notions and structures I define over sequential games by supplying a comparable construction categorically. Moreover, when it comes to my handling of linearizability, the categorical ideas introduced here will be fundamental.

Chapter 8

An Interactive Linear Logic Model of State

We have now a model for safety properties and a few handcrafted interpretations of programs in this model. There is a fundamental gap, however. First, we should be able to *compute* the interpretations of programs quite directly as some kind of morphism in **Safe**^{seq}. But beyond that, we have only constructed interpretations for code of the form $M : \dagger E \multimap F$. Such a morphism only allows at most a single operation of F to be invoked. But when we import a module or library such as the counter implementation in Ex. 6.1.4, we expect to make several calls to it in the process of implementing some other object.

What we are missing is both a precise account of the denotations for these libraries and of how we might actually interpret them as strategies. In a functional setting, it is common to interpret programs into the Kleisli category of a monad $!-$. Reddy’s pioneering work on object-based semantics [Reddy, 1993, 1996, 1994] provides an alternate modality for interpreting imperative programs instead, exemplified by developing a semantics for ALGOL. This is in sharp contrast to the tradition, where models of imperative programming languages are described as functions on global states. Reddy’s approach instead models imperative programming as a compositional calculus of objects, which are small components encapsulating state but fully characterized in terms of their observable behavior. The similarity with our goals is apparent. He calls his approach *object-based semantics* in contrast,

perhaps, to state-based approaches.

The main model of object-based semantics developed by Reddy was carried out on coherence spaces, a relational model of Linear Logic. In addition, Reddy’s work was seminal in the later development of game semantics. Despite that, the ideas originally developed by Reddy have been largely relegated to the semantics of reference cells [Melliès and Tabareau, 2009], and other such storage primitives. Meanwhile, the novel linear logic modality ($\dagger$) introduced in Reddy’s work has been largely dismissed in favor of the now predominant *of course* exponential modality ($!$) of linear logic, and is therefore understudied.

In this chapter, we re-introduce and extend Reddy’s ideas in order to develop an interactive account of object-based semantics.

8.1 Background: Object-Based Semantics

Reddy [Reddy, 1996] introduces a semantic domain for imperative languages based on the model of state introduced in [Reddy, 1993], itself based on the coherence spaces semantics for linear logic [Girard, 1987b] with the before operator [Retoré, 1997]. He calls this semantic domain object-based semantics. This section gives a quick introduction to coherence spaces and object-based semantics.

8.1.1 The Basic Principles

Reddy [Reddy, 1996] advocates the idea that a program with an internal state can be entirely characterized in terms of its interactive behavior, as what he calls an *object*. This object-based semantics is based on four basic principles:

1. An object can in general be used only sequentially,
2. The behavior of an object is in general affected by its past history of operations,
3. Object functions must be linear maps,
4. Object functions are regular maps.

Principle 1 means that objects can be described by the linear trace they produce, and Principle 2 that the prefix of a trace influences what comes next. Principles 3 and 4 are

then elegantly formalized using the notion of coherence spaces originally introduced by Girard [Girard, 1987b].

8.1.2 Coherence Spaces and Linear Maps

Definition 8.1.1. A *coherence space* $A = (|A|, \supseteq_A)$ is a set of tokens $|A|$ together with a reflexive and symmetric coherence relation $\supseteq_A \subset |A| \times |A|$.

We often use A to mean its web $|A|$ when it causes no confusion.

Example 8.1.1. The coherence space $\text{Cell}[\mathbb{N}]$ encodes the operations over a memory cell. Its web $|\text{Cell}[\mathbb{N}]|$ is given by

$$|\text{Cell}[\mathbb{N}]| := \{\text{rd}.n \mid n \in \mathbb{N}\} \uplus \{\text{wrt}(n).* \mid n \in \mathbb{N}\}$$

Each token of $\text{Cell}[\mathbb{N}]$ encodes both a call and return event: the token $\text{wrt}(n).*$ encodes a call to wrt with argument n returning $*$; the token $\text{rd}.n$ encodes a call to rd with no arguments and returning n . The coherence relation of $\text{Cell}[\mathbb{N}]$ can be given as

$$\text{op}.v \supseteq_{\text{Cell}[\mathbb{N}]} \text{op}'.v' \iff \text{op} = \text{op}' \Rightarrow v = v'$$

It can be understood as stating that operations in $\text{Cell}[\mathbb{N}]$ are deterministic. Note that $\text{op} = \text{op}'$ means that, in particular, the arguments to the operation are the same.

Similarly, we can define the coherence space Counter which encodes the operations of a counter.

$$|\text{Counter}| := \{\text{get}.n \mid n \in \mathbb{N}\} \cup \{\text{inc}.*\}$$

with coherence relation

$$\text{op}.v \supseteq_{\text{Counter}} \text{op}'.v' \iff \text{op} = \text{op}' \Rightarrow v = v'$$

Functions between coherence spaces are required to be linear maps, defined in the following way:

Definition 8.1.2. A map $f \subseteq |A| \times |B|$ is a *linear map* $f : A \multimap B$ when for all $a_1 \mapsto b_1, a_2 \mapsto b_2 \in f$

1. $a_1 \supset_A a_2 \Rightarrow b_1 \supset_B b_2$
2. $a_1 \supset_A a_2 \wedge b_1 = b_2 \Rightarrow a_1 = a_2$

Recall that a clique in a coherence space A is a clique in its underlying graph, or, in other words, a subset of the web $|A|$ whose tokens are pairwise coherent. Note that if we define the coherence space $\mathbf{1}$ with web $|\mathbf{1}| := \{*\}$ and coherence $* \supset_{\mathbf{1}} *$, every clique f in A corresponds to a linear map $f : \mathbf{1} \multimap A$. So elements of coherence spaces are cliques.

8.1.3 Constructors on Coherence Spaces

We define several standard operators on coherence spaces, which we will see later form a model of intuitionistic linear logic. Let A and B be coherence spaces in the following definitions.

$\otimes$ The tensor product $A \otimes B$ is defined by

$$|A \otimes B| := |A| \times |B|$$

$$(a, b) \supset_{A \otimes B} (a', b') \iff a \supset_A a' \wedge b \supset_B b'$$

$\mathbf{1}$ The tensor unit $\mathbf{1}$ is the coherence space whose web $|\mathbf{1}| = \{*\}$ is the singleton set and coherence is given by equality.

$\multimap$ The linear implication $A \multimap B$ is defined by

$$|A \multimap B| := |A| \times |B|$$

$$a \mapsto b \supset_{A \multimap B} a' \mapsto b' \iff (a \supset_A a' \Rightarrow b \supset_B b') \wedge (b \asymp_B b' \Rightarrow a \asymp_A a')$$

$\&$ The *with* of A and B , $A \& B$, is defined by

$$|A \& B| := |A| + |B|$$

$$\text{in}_X x \subset_{A \& B} \text{in}_Y y \iff X = Y \Rightarrow x \subset_X y$$

$\top$ The unit for $\&$ is the empty coherence space $\top$.

$!$ The exponential conjunction $!A$, known as *of course*, is the coherence space

$$|!A| := \text{all finite subsets of } |A|$$

$$s \subset_{!A} s' \iff \forall a, a' \in s \cup s'. a \subset_A a'$$

8.1.4 The Before Operator

The before operator was introduced in [Retoré, 1997] as a non-symmetric product operator capturing a notion of sequential composition. In terms of coherence spaces, a product operator is such that its web is the cartesian product of the webs of its components. Up to direction, it is in fact the only possible positive product operator other than $\wp$ and $\otimes$ on coherence spaces. This is verified by noting that the table defining the coherence relation for a product operator $A \odot B$ has the form seen in table 8.1.

Table 8.1: Coherences table for a positive product operator $A \odot B$

$A \odot B$	$\frown$	$=$	$\smile$
$\frown$	$\frown$	$\frown$	$\frown \odot \smile$
$=$	$\frown$	$=$	$\smile$
$\smile$	$\smile \odot \frown$	$\smile$	$\smile$

The choices of entries $\frown \odot \smile$ and $\smile \odot \frown$ are the only ones that are not fixed by the positivity assumption. Each choice of those entries yields a different product operator, as seen in table 8.2.

The operators $\otimes$ and $\wp$ are the standard tensor product and par of linear logic. Operators $\triangleright$ and $\triangleleft$, read *before* and *after*, were introduced by Retor and can be noted to be non-symmetric. In fact, they only differ in terms of the direction of propagation of coherence. $\triangleright$ is such that if the left side is strictly coherent/incoherent then so is the pair, while $\triangleleft$ has the reverse property which justifies their naming. The choice of which one to use is

Table 8.2: Possible positive product operators

$\neg \odot \neg$	$\neg \odot \neg$	$A \odot B$
$\neg$	$\neg$	$A \otimes B$
$\neg$	$\neg$	$A \wp B$
$\neg$	$\neg$	$A \triangleright B$
$\neg$	$\neg$	$A \triangleleft B$

entirely arbitrary, given the reverse symmetry between the two. We find it more natural to use $\triangleright$ for our purposes.

The $\triangleright$ operator enjoys many interesting properties, especially in how it relates to $\otimes$ and $\wp$. It can be seen as a hybrid of the two operators. This is witnessed by the fact that there is a series of linear injections given by the identity of types

$$A \otimes B \multimap A \triangleright B \multimap A \wp B$$

A detailed study of the $\triangleright$ operator can be found in [Retoré, 1997] and [Reddy, 1993]. For our purposes, we will simply note its non-symmetric monoidal structure as noted by the isomorphisms

$$A \triangleright (B \triangleright C) \cong (A \triangleright B) \triangleright C$$

$$\mathbf{1} \triangleright A \cong A \cong A \triangleright \mathbf{1}$$

and the non-isomorphism

$$A \triangleright B \not\cong B \triangleright A$$

and the fact that it is self-dual

$$(A \triangleright B)^\perp = A^\perp \triangleright B^\perp$$

8.1.5 The Replay Modality

The main thesis of [Reddy, 1996] is that the informal notion of object is appropriately captured by the notion of clique of a coherence space of the form $\dagger A$, whose tokens are finite sequences of tokens of A . A regular map f from A to B is then defined as a linear map $f : \dagger A \multimap B$.

Definition 8.1.3. Given a coherence space A , the *object space* $\dagger A$ uses tokens in $|\dagger A| := |A|^*$ and the coherence relation $\supset_{\dagger A}$ which relates $\langle a_1, \dots, a_n \rangle \supset_{\dagger A} \langle b_1, \dots, b_m \rangle$ if and only if:

$$\forall i \in [\min(n, m)]. \langle a_1, \dots, a_{i-1} \rangle = \langle b_1, \dots, b_{i-1} \rangle \Rightarrow a_i \supset_A b_i$$

An *object* is a clique in $\dagger A$.

The coherence relation over $\dagger A$ ensures that at the first point where the two sequences differ, they differ coherently. More informally, it can be read as “at the first point in time when two computations take different paths, they must do so coherently”.

Example 8.1.2. The object space associated to $\text{Cell}[\mathbb{N}]$ is the space of all sequences of call and return events that can be performed on a memory cell. Note that $\dagger A$ does not enforce a particular semantics for the returns across this sequence. For instance,

$$\langle \text{rd}.3, \text{wrt}(7).*, \text{rd}.5 \rangle \in |\dagger \text{Cell}[\mathbb{N}]|$$

$$\langle \text{rd}.0, \text{wrt}(1).*, \text{rd}.1 \rangle \in |\dagger \text{Cell}[\mathbb{N}]|$$

On the other hand, an object in $\dagger \text{Cell}[\mathbb{N}]$ specifies a particular semantics by restricting those sequences. For instance, the following set of sequences forms a clique in $\dagger \text{Cell}[\mathbb{N}]$:

$$\begin{aligned} V_{\text{Cell}} := \{ & s \in |\dagger \text{Cell}[\mathbb{N}]| \mid (s = \text{rd}.i \cdot s' \Rightarrow i = 0) \\ & \wedge (s = p \cdot \text{rd}.i \cdot \text{rd}.i' \cdot t \Rightarrow i = i') \\ & \wedge (s = p \cdot \text{wrt}(i).* \cdot \text{rd}.i' \cdot t \Rightarrow i = i') \} \end{aligned}$$

The properties over the sequences in V_{Cell} enforce that:

- The cell initially responds to a `rd` with a 0
- Consecutive calls to `rd` return the same value
- A call to `rd` following a `wrt(i)` must return i

This defines a prefix-closed clique encoding all possible behaviors starting from a certain state. Similarly, we can define:

$$V_{\text{Counter}} := \{s \in |\dagger\text{Counter}| \mid (s = p \cdot \text{get}.i \cdot t \Rightarrow i = \#\text{inc}(p))\}$$

where $\#\text{inc}(p)$ counts occurrences of `inc.*` in the sequence p .

8.1.6 Regular Maps between Object Spaces

Maps between object spaces are required to satisfy a regularity requirement. In [Reddy, 1996], regular functions are first defined using a structural property of the map, then an equivalence is proven with the co-Kleisli category of $\dagger$ on coherence spaces.

Definition 8.1.4. For object spaces $\dagger A$ and $\dagger B$, a *regular function* $f : \dagger A \rightarrow_{\text{Reg}} \dagger B$ is a linear map satisfying

1. $(s_1 \mapsto t_1), \dots, (s_n \mapsto t_n) \in f \Rightarrow (s_1 \cdots s_n \mapsto t_1 \cdots t_n) \in f$
2. If $(s \mapsto t_1 \cdots t_n) \in f$, then there exists $s_1, \dots, s_n \in \dagger A$ such that $s = s_1 \cdots s_n$ and $s_1 \mapsto t_1 \wedge \dots \wedge s_n \mapsto t_n$

The equivalence is then given by the following theorem.

Proposition 8.1.1 (Reddy). *There is an order-isomorphism*

$$\dagger A \multimap B \cong \dagger A \rightarrow_{\text{Reg}} \dagger B$$

Given $f : \dagger A \multimap B$ we use $f^\dagger$ to denote the corresponding regular function. Notice that this characterization shows that regular functions are exactly linear functions between object spaces that are generated by some underlying $f : \dagger A \multimap B$ linear function. One direction

is particularly important for us. Namely, given a linear function $f : \dagger A \multimap B$ Reddy shows we can construct a regular extension $f^\dagger : \dagger A \rightarrow_{\text{Reg}} \dagger B$ that acts on $\dagger B$ pointwise according to f by

$$f^\dagger := \{s_1 \cdots s_n \mapsto \langle b_1, \dots, b_n \rangle \mid n \geq 0 \wedge \forall i, s_i \mapsto b_i \in f\}$$

Example 8.1.3. We can define a regular map $M : \dagger \text{Cell}[\mathbb{N}] \multimap \text{Count}$ by first defining two maps M^{get} , and M^{inc} which each define the sequences that map to a `get.i` token and `inc.*` token in `Count`, respectively.

$$M^{\text{get}} = \{\langle \text{rd}.i \rangle \mapsto \text{get}.i \mid i \in \mathbb{N}\}$$

$$M^{\text{inc}} = \{\langle \text{rd}.i, \text{wrt}(i+1).* \rangle \mapsto \text{inc}.* \mid i \in \mathbb{N}\}$$

which intuitively corresponds to the following pieces of code one might write to implement a counter using a memory cell

```
inc() {                               get() {
    i ← rd();                          i ← rd();
    wrt(i+1);                          return i;
    return *;                          }
}
```

Then, the map M defined as $M := M^{\text{get}} \uplus M^{\text{inc}}$ is indeed a linear map $M : \dagger \text{Cell}[\mathbb{N}] \multimap \text{Count}$ which can be extended to a regular map

$$M^\dagger : \dagger \text{Cell}[\mathbb{N}] \rightarrow_{\text{Reg}} \dagger \text{Count}$$

mapping objects in $\dagger \text{Cell}[\mathbb{N}]$ to objects in $\dagger \text{Count}$.

Composition of linear functions is given by the usual relational composition of the underlying maps.

Definition 8.1.5. Given linear maps $f : A \multimap B$ and $g : B \multimap C$, we define the composition $g \circ f : A \multimap C$ as

$$g \circ f := \{a \mapsto c \mid \exists b \in |B|. a \mapsto b \in f \wedge b \mapsto c \in g\}$$

The identity $\mathbf{id}_A : A \multimap A$ is defined as

$$\mathbf{id}_A := \{a \mapsto a \mid a \in |A|\}$$

This means we can encapsulate coherence spaces as a category.

Definition 8.1.6. The category **Coh** has coherence spaces A, B, C as objects and linear maps $f : A \multimap B, g : B \multimap C$ as morphisms. Composition and identity are inherited from the overlying category **Rel** of sets and relations.

We can define the composition of linear maps generating regular functions $f : \dagger A \multimap B$ and $g : \dagger B \multimap C$ as $g \circ f^\dagger$.

In summary, object spaces are modeled as coherence spaces $\dagger A$ and morphisms between these object spaces are regular maps $\dagger A \rightarrow_{\text{Reg}} \dagger B$. Regular maps can instead be described by linear maps $\dagger A \multimap B$, and such linear maps can be composed in accordance to regular map composition. A single object of type A is a clique in the graph of the relation $\subset_{\dagger A}$.

8.1.7 Background: Comonads and the Co-Kleisli Category

We just saw a surprising equivalence:

$$\dagger A \multimap B \cong \dagger A \rightarrow_{\text{Reg}} \dagger B$$

This equivalence is actually an instance of a more general and deeper construction.

Definition 8.1.7. A *comonad* on a category **C** consists of an endofunctor $M : \mathbf{C} \rightarrow \mathbf{C}$ together with natural transformations:

counit: $\epsilon_A : M A \rightarrow A$,

comultiplication: $\delta_A : M A \rightarrow M (M A)$

such that the following diagrams commute:

$$\begin{array}{ccc} M A & \xrightarrow{\delta_A} & M (M A) \\ \delta_A \downarrow & & \downarrow \delta_{M A} \\ M (M A) & \xrightarrow{M \delta_A} & M (M (M A)) \end{array} \qquad \begin{array}{ccccc} & & M A & & \\ & \swarrow & \downarrow \delta_A & \searrow & \\ M A & \xleftarrow{\epsilon_{M A}} & M (M A) & \xrightarrow{M \epsilon_A} & M A \end{array}$$

Example 8.1.4. In the category **Set** we may define the functor $(-)^{\infty}$:

$$A \longmapsto A^{\infty} \quad f : A \rightarrow B \longmapsto f^{\infty}, \text{ where } f^{\infty}(a_0 \cdot a_1 \cdot a_2 \cdot \dots) = f(a_0) \cdot f(a_1) \cdot f(a_2) \cdot \dots$$

Then, the counit and comultiplication are defined by:

$$\epsilon_A : A^{\infty} \rightarrow A, \text{ where } \epsilon_A(a_0 \cdot a_1 \cdot a_2 \cdot \dots) = a_0$$

$$\delta_A : A^{\infty} \rightarrow (A^{\infty})^{\infty}, \text{ where } (\delta_A(s))[n] = (a_n, a_{n+1}, a_{n+2}, \dots) \text{ for } s = (a_0, a_1, a_2, \dots)$$

That is, ϵ_A returns the head of the stream and δ_A produces the stream of all tails: its n -th element is the stream s shifted by n . It is straightforward to verify that the necessary equations are satisfied.

Example 8.1.5. The operation $\dagger-$ in the category **Coh** defines a comonad. Indeed, the counit and comultiplication are given by the linear maps:

$$\begin{aligned} \delta_A : \dagger A &\multimap \dagger \dagger A & \epsilon_A : \dagger A &\multimap A \\ \delta_A &:= \{s_1 \cdot \dots \cdot s_n \mapsto \langle s_1, \dots, s_n \rangle \mid s_1, \dots, s_n \in |\dagger A|\} & \epsilon_A &:= \{\langle a \rangle \mapsto a \mid a \in |A|\} \end{aligned}$$

As a functor, the action of $\dagger-$ on a linear map $f : A \multimap B$ is

$$\dagger f := \{\langle a_1, \dots, a_n \rangle \mapsto \langle b_1, \dots, b_n \rangle \mid \forall i \leq n. a_i \mapsto b_i \in f\}$$

which applies f element-wise through the input sequences.

In any category **C**, given a comonad (M, ϵ, δ) it is possible to define its co-Kleisli category.

Definition 8.1.8. The *co-Kleisli category* $\mathbf{C}_M$ of a comonad (M, ϵ, δ) on **C** is defined by:

Objects: the same objects as **C**,

Morphisms: a morphism $f : A \rightarrow B$ in $\mathbf{C}_M$ is a morphism $f : M A \rightarrow B$ in **C**,

Identity: the identity on A is $\epsilon_A : M A \rightarrow A$,

Composition: given $f : M A \rightarrow B$ and $g : M B \rightarrow C$, their composite is:

$$g \circ_M f := g \circ M f \circ \delta_A$$

Or, diagrammatically:

$$M A \xrightarrow{\delta_A} M (M A) \xrightarrow{M f} M B \xrightarrow{g} C$$

An alternative presentation of the co-Kleisli category arises from the Eilenberg-Moore category of coalgebras.

Definition 8.1.9. Given a comonad (M, ϵ, δ) on $\mathbf{C}$, an M -coalgebra is a pair (A, κ) of an object $A \in \mathbf{C}$ and a morphism $\kappa : A \rightarrow M A$ in $\mathbf{C}$ such that the following diagrams commute:

$$\begin{array}{ccc} A & \xrightarrow{\kappa} & M A \\ \parallel & & \downarrow \epsilon_A \\ & & A \end{array} \quad \begin{array}{ccc} A & \xrightarrow{\kappa} & M A \\ \kappa \downarrow & & \downarrow \delta_A \\ M A & \xrightarrow{M \kappa} & M (M A) \end{array}$$

A morphism of M -coalgebras $f : (A, \kappa_A) \rightarrow (B, \kappa_B)$ is a morphism $f : A \rightarrow B$ in $\mathbf{C}$ such that:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \kappa_A \downarrow & & \downarrow \kappa_B \\ M A & \xrightarrow{M f} & M B \end{array}$$

The *Eilenberg-Moore category* $\mathbf{C}^M$ has M -coalgebras as objects and coalgebra morphisms as morphisms, with composition and identities inherited from $\mathbf{C}$.

Every object $A \in \mathbf{C}$ gives rise to a *free coalgebra* $(M A, \delta_A)$. The coalgebra diagrams are verified by the comonad laws. A key result then is that:

Proposition 8.1.2. *The co-Kleisli category $\mathbf{C}_M$ is equivalent to the full subcategory of $\mathbf{C}^M$ spanned by the free coalgebras.*

Concretely, the equivalence identifies a co-Kleisli morphism $f : M A \rightarrow B$ as the coalgebra morphism

$$M A \xrightarrow{\hat{f}} M B \quad = \quad M A \xrightarrow{\delta_A} M M A \xrightarrow{M f} M B$$

Example 8.1.6. The category of free coalgebras of $\dagger-$ in **Coh** is exactly the category of regular maps. Reddy’s equivalence:

$$\dagger A \multimap B \cong \dagger A \rightarrow_{\text{Reg}} \dagger B$$

is exactly the substance of Prop. 8.1.2.

The “decomposition” map δ_A plays an essential role in lifting a map $f : \dagger A \multimap B$ to the regular map $\widehat{f} : \dagger A \rightarrow \dagger B$ defined as the composite

$$\dagger A \xrightarrow{\widehat{f}} \dagger B = \dagger A \xrightarrow{\delta_A} \dagger \dagger A \xrightarrow{\dagger f} \dagger B$$

With the usual counter implementation $M : \dagger \text{Cell}[\mathbb{N}] \rightarrow \text{Counter}$ we observe the composition

$$\langle \text{rd}.a, \text{rd}.b, \text{wrt}(b+1).\text{ok}, \text{rd}.c \rangle \xrightarrow{\delta_{\text{Cell}[\mathbb{N}]}} \langle \langle \text{rd}.a \rangle, \langle \text{rd}.b, \text{wrt}(b+1).\text{ok} \rangle, \langle \text{rd}.c \rangle \rangle \xrightarrow{\dagger M} \langle \text{get}.a, \text{inc}.\text{ok}, \text{get}.c \rangle$$

where $\delta_{\text{Cell}[\mathbb{N}]}$ plays the role of decomposing the input trace before M can be replicated to map the input trace of $\dagger \text{Cell}[\mathbb{N}]$ to a trace of $\dagger \text{Counter}$. Note also that there are many decompositions of the input trace that do not get mapped through $\dagger M$ and therefore do not appear in $\widehat{M}$.

8.1.8 Background: Products

A category **C** has *finite products* when it admits a terminal object and binary products.

Definition 8.1.10. A *terminal object* in a category **C** is an object $\top \in \mathbf{C}$ such that for every object $A \in \mathbf{C}$ there exists a unique morphism $!_A : A \rightarrow \top$.

Definition 8.1.11. A *binary product* of two objects $A, B \in \mathbf{C}$ is an object $A \& B \in \mathbf{C}$ equipped with morphisms, called *projections*:

$$\pi_1 : A \& B \rightarrow A \quad \pi_2 : A \& B \rightarrow B$$

satisfying the universal property that for every object C and morphisms $f : C \rightarrow A$ and $g : C \rightarrow B$ there exists a unique morphism $\langle f, g \rangle : C \rightarrow A \& B$ such that:

$$\begin{array}{ccccc}
& & C & & \\
& \swarrow f & \downarrow \langle f, g \rangle & \searrow g & \\
A & \xleftarrow{\pi_1} & A \& B & \xrightarrow{\pi_2} B
\end{array}$$

That is, $\pi_1 \circ \langle f, g \rangle = f$ and $\pi_2 \circ \langle f, g \rangle = g$.

Finite products give rise to a symmetric monoidal category $(\mathbf{C}, \&, \top)$. In particular, there are natural isomorphisms:

$$A \& (B \& C) \cong (A \& B) \& C \quad A \& \top \cong A \cong \top \& A \quad A \& B \cong B \& A$$

constructed from projections and pairings. This monoidal structure is moreover *cartesian* in that every object is a comonoid via the diagonal $\Delta_A = \langle \text{id}_A, \text{id}_A \rangle : A \rightarrow A \& A$ and the unique map $!_A : A \rightarrow \top$.

Example 8.1.7. In **Coh**, finite products are given by $(\&, \top)$ as defined in §8.1.3. The projections $\pi_1 : A \& B \multimap A$ and $\pi_2 : A \& B \multimap B$ are given by:

$$\pi_1 := \{\text{in}_A \ a \mapsto a \mid a \in |A|\} \quad \pi_2 := \{\text{in}_B \ b \mapsto b \mid b \in |B|\}$$

and the pairing $\langle f, g \rangle : C \multimap A \& B$ of linear maps $f : C \multimap A$ and $g : C \multimap B$ is:

$$\langle f, g \rangle := \{c \mapsto \text{in}_A \ a \mid c \mapsto a \in f\} \cup \{c \mapsto \text{in}_B \ b \mid c \mapsto b \in g\}$$

The terminal object $\top$ is the empty coherence space, and for any A the unique map $!_A : A \multimap \top$ is the empty relation $\emptyset$.

8.1.9 Background: Comonoids

A comonoid is the categorical dual of a monoid. While a monoid has a multiplication and a unit, a comonoid has a comultiplication and a counit.

Definition 8.1.12. A *comonoid* in a monoidal category $(\mathbf{C}, - \otimes -, \mathbf{1})$ is an object $A \in \mathbf{C}$ equipped with morphisms:

comultiplication: $d_A : A \rightarrow A \otimes A$,

counit: $e_A : A \rightarrow \mathbf{1}$

such that the following diagrams commute:

$$\begin{array}{ccc}
 (A \otimes A) \otimes A & \xrightarrow{\alpha_{A,A,A}} & A \otimes (A \otimes A) \\
 \uparrow d_A \otimes \text{id}_A & & \uparrow \text{id}_A \otimes d_A \\
 A \otimes A & & A \otimes A \\
 \swarrow d_A & & \searrow d_A \\
 & A &
 \end{array}
 \qquad
 \begin{array}{ccccc}
 \mathbf{1} \otimes A & \xleftarrow{e_A \otimes \text{id}_A} & A \otimes A & \xrightarrow{\text{id}_A \otimes e_A} & A \otimes \mathbf{1} \\
 & \swarrow \lambda_A^{-1} & \uparrow d_A & \searrow \rho_A^{-1} & \\
 & & A & &
 \end{array}$$

We write (A, d_A, e_A) to denote a comonoid.

Definition 8.1.13. A *comonoid morphism* $f : (A, d_A, e_A) \rightarrow (B, d_B, e_B)$ between two comonoids in $(\mathbf{C}, \otimes, \mathbf{1})$ is a morphism $f : A \rightarrow B$ in $\mathbf{C}$ preserving the comonoid structure, i.e., the following diagrams commute:

$$\begin{array}{ccc}
 A & \xrightarrow{f} & B \\
 d_A \downarrow & & \downarrow d_B \\
 A \otimes A & \xrightarrow{f \otimes f} & B \otimes B
 \end{array}
 \qquad
 \begin{array}{ccc}
 A & \xrightarrow{f} & B \\
 e_A \searrow & & \downarrow e_B \\
 & & \mathbf{1}
 \end{array}$$

Example 8.1.8. In a category with finite products $(\&, \top)$, every object A is canonically a comonoid in $(\mathbf{C}, \&, \top)$ via the diagonal $\Delta_A = \langle \text{id}_A, \text{id}_A \rangle : A \rightarrow A \& A$ and the unique map $!_A : A \rightarrow \top$. Moreover, every morphism $f : A \rightarrow B$ is a comonoid morphism with respect to these canonical comonoid structures.

Example 8.1.9. The replay modality $\dagger -$ actually induces the cofree $\triangleright$ -comonoid and the resulting adjunction is monadic in that the Eilenberg-Moore category $\mathbf{Coh}^\dagger$ of $\dagger -$ is equivalent to $\mathbf{CoMon}_{\triangleright}$. This is a result I showed as part of my work [Oliveira Vale et al., 2022].

Indeed, to a coherence space A we associate the $\triangleright$ -comonoid:

$$(\dagger A, d_A : \dagger A \multimap \dagger A \triangleright \dagger A, e_A : \dagger A \multimap \mathbf{1})$$

where the comultiplication and counit are given by:

$$s_1 \cdot s_2 \mapsto s_1 \triangleright s_2 \quad \epsilon \mapsto *$$

The comonad $\dagger$ on **Coh** equips each object $\dagger A$ with a comonoid structure in $(\mathbf{Coh}, \triangleright, \mathbf{1})$, given by the comultiplication $\delta_A : \dagger A \rightarrow \dagger A \triangleright \dagger A$ and counit $\epsilon_A : \dagger A \rightarrow \mathbf{1}$. Note however that not every linear map $f : \dagger A \multimap \dagger B$ is a comonoid morphism with respect to this structure: preserving the comonoid structure is exactly the regularity condition of Definition 8.1.4.

8.1.10 Linear Logic Model of State

Coherence spaces can be seen to give a model of intuitionistic linear logic (and in fact of linear logic as a whole). This is done by interpreting each operator of linear logic as one of the constructors defined in section 8.1.3. See [Girard, 1987b; Melliès, 2009, 2005b].

For our purposes, we use the following as our definition of a categorical model of intuitionistic linear logic.

Definition 8.1.14. A categorical model of intuitionistic linear logic (without exponentials) is a symmetric monoidal closed category $(\mathbf{C}, \otimes, \mathbf{1})$ with finite products $(\&, \top)$.

It is usual to require there to be a categorical exponential modality $!$ witnessing some coherence properties with the other structures of a categorical model of linear logic. We omit the $!$ exponential modality because it is mostly orthogonal to our development.

Proposition 8.1.3. $(\mathbf{Coh}, - \otimes -, \mathbf{1})$ with finite products given by $(- \& -, \top)$ is a model of intuitionistic linear logic.

The question then arises of exactly when we have a category suitable for an object-based semantics. Reddy answers that in [Reddy, 1993] by introducing the notion of a linear logic model of state. We present below a simplified version of a linear logical model of state where we omit the relation between $\dagger -$ and $! -$. On the other hand, we refine the model by adding extra structure around $\dagger -$.

Definition 8.1.15. A linear logic model of state is a model of intuitionistic linear logic as seen by the symmetric monoidal closed category $(\mathbf{C}, - \otimes -, \mathbf{1})$, finite products $(- \& -, \top)$. In addition,

- There is a monoidal structure $(\triangleright, \mathbf{1})$ such that $(- \otimes -, \mathbf{1})$ is a submonoidal structure of $(\triangleright, \mathbf{1})$, in the sense that there is a natural monic $\text{ser}_{-, -} : - \otimes - \multimap - \triangleright -$ preserving

the associator and left and right unital structures

- There is a comonad $\dagger$ which is a comonoid of $(\mathbf{C}, \triangleright, \mathbf{1})$.
- For every object A there is a morphism $\text{der}_A^\dagger : \dagger A \rightarrow A$ such that for every morphism $f : \dagger A \rightarrow B$ there exists a unique comonoidal morphism

$$f^\dagger : (\dagger A, d_A^\dagger, e_A^\dagger) \rightarrow (\dagger B, d_B^\dagger, e_B^\dagger)$$

making the diagram

$$\begin{array}{ccc} \dagger A & \xrightarrow{f^\dagger} & \dagger B \\ & \searrow f & \downarrow \text{der}_B^\dagger \\ & & B \end{array}$$

- There is a comonoidal isomorphism

$$(\mathbf{1}, \rho_1^{-1} = \lambda_1^{-1}, \mathbf{id}_1) \cong (\dagger \top, d_\top, e_\top)$$

The domain of discourse for object-based semantics is then the co-Kleisli category of $\dagger$ in $\mathbf{C}$ (or even the Eilenberg-Moore category), a linear logic model of state.

The categorical definition above makes sure that $\triangleright$ is related to $\otimes$ in the way we outlined in §8.1.4 and §8.1.5 and that $\dagger$ —“internalizes” $\triangleright$ in an appropriate sense. The two last properties, which we add to Reddy’s original formulation [Reddy, 1993], make sure that the lifting of a map $\dagger A \rightarrow B$ to a map $\dagger A \rightarrow \dagger B$ is comonoidal, and the last property that we may see a map $\mathbf{1} \rightarrow \dagger A$ as a map $\dagger \top \rightarrow \dagger A$.

Example 8.1.10. **Coh** with the structure we have built on it defines a linear logic model of state. In particular, the dereliction $\text{der}_A^\dagger$ is given by the counit $\epsilon_A : \dagger A \multimap A$.

8.2 An Interactive Linear Logic Model of State

We now define the structure necessary to develop a linear logic model of state in our game model for (sequential) safety properties.

8.2.1 Finite Products

We already have much of the structure defined, such as the categorical structure and the $-\otimes -$. Notably missing are the product structure and the non-symmetric monoidal structure.

Definition 8.2.1. Let A and B be games. We define the game $A \& B$ by

$$M_{A\&B} := M_A + M_B \quad \lambda_{A\&B} := \lambda_A + \lambda_B \quad P_{A\&B} = \iota_A^*(P_A) + \iota_B^*(P_B)$$

where ι_A and ι_B are the injections into the left and right components of $M_A + M_B$.

The game $\top$ is defined as the game with no moves $M_\top = \emptyset$ and a single play $P_\top = \{\epsilon\}$.

Hence, $A \& B$ allows for either a play entirely in A or a play entirely in B . Note as well that as is common with models of linear logic, the constants are degenerate in the sense that $\mathbf{1} = \top$ (just like in coherence spaces, in fact).

The projections behave just like the copycat strategy, but with some re-indexing. This is a common pattern for structural morphisms in game models.

$$\begin{aligned} \pi_A : A \& B \multimap A &:= \{s \in P_{(A\&B) \multimap A} \mid \forall p \sqsubseteq_{\text{even}} s. p \upharpoonright_{A\&B, -} = \iota_A^*(p \upharpoonright_{-, A})\}^\bullet \\ \pi_B : A \& B \multimap B &:= \{s \in P_{(A\&B) \multimap B} \mid \forall p \sqsubseteq_{\text{even}} s. p \upharpoonright_{A\&B, -} = \iota_B^*(p \upharpoonright_{-, B})\}^\bullet \end{aligned}$$

Meanwhile, the unique terminal morphisms $!_A : A \multimap \top$ are just $!_A := \downarrow \bullet = \{\epsilon\}^\bullet$.

Proposition 8.2.1. $\mathbf{Safe}^{\text{seq}}$ with $(\mathbf{Safe}^{\text{seq}}, -\otimes -, \mathbf{1})$ as monoidal structure and $(\mathbf{Safe}^{\text{seq}}, -\& -, \top)$ as finite products is a model of intuitionistic linear logic (without exponentials).

8.2.2 Before Operator

Now, toward the stateful side of the linear logic model of state we start with the before operator:

Definition 8.2.2. Let A and B be games. We define the game $A \triangleright B$ by the following data

$$\begin{aligned} M_{A \triangleright B} &:= M_A + M_B & \lambda_{A \triangleright B} &:= \lambda_A + \lambda_B \\ P_{A \triangleright B} &:= \{s \cdot t \in \mathbb{SP}_{A \triangleright B} \mid s \in \iota_A^*(P_A) \wedge t \in \iota_B^*(P_B)\} \end{aligned}$$

The action on strategies $\sigma : A \multimap C$ and $\tau : B \multimap D$ is given by

$$\sigma \triangleright \tau := \{(\iota_A + \iota_C)^*(s) \cdot (\iota_A + \iota_C)^*(t) \in \mathbb{SP}_{A \triangleright B \multimap C \triangleright D} \mid s \in \sigma \wedge t \in \tau\}$$

The intuition for $- \triangleright -$ is that a play of $A \triangleright B$ is a play of A followed by a play of B . Since the operator is positive, polarities are preserved. This has as consequence the fact that it is Opponent who decides when to move from the first copy to the second copy. In particular, in a play $s \in P_{A \triangleright B}$ it is always the case that if the B game is opened, meaning $s|_B \neq \epsilon$, then the play in A is a P-position. This is also part of the reason why $- \triangleright -$ cannot be symmetric. These remarks get us most of the way to establish:

Proposition 8.2.2. $- \triangleright -$ endows **Safe**^{seq} with a non-symmetric monoidal structure.

Proof. See Proof 30.2.1. □

8.2.3 The $\dagger$ -Replay Modality

With the $\triangleright$ operator on games defined and shown to be a monoidal structure, we are ready to define the replay modality $\dagger -$.

Definition 8.2.3. Let A be a game. We define the game $\dagger A = (M_{\dagger A}, P_{\dagger A})$ as

$$M_{\dagger A} := \sum_{n \in \mathbb{N}} M_A \quad \lambda_{\dagger A}(n.m) = \lambda_A(m) \quad P_{\dagger A} := \{s_0 \cdot \dots \cdot s_n \in \mathbb{SP}_{\dagger A} \mid \forall i. s_i \in \iota_i^*(P_A)\}$$

In other words, the definition ensures that a play of $\dagger A$ is a well-formed concatenation of plays of A with a copy index $i \in \mathbb{N}$. Moreover, whenever a new play of A is started, its index must be incremented by exactly 1, so that in a play with n copies of A , all the indexes from 0 to $n - 1$ appear in sequence.

The intuition for Definition 8.2.3 is that Opponent can open as many copies of A as they need, but once a new copy is opened they can't go back to the older copies. This is in stark contrast with the *of course* ! modality, which allows Opponent to play on older copies even after new copies have been opened. This leads to the intuition that $\dagger$ subsumes any

finitely iterated A under $\triangleright$, like so

$$\underbrace{A \triangleright A \triangleright \dots \triangleright A}_{n \text{ times}}$$

This discussion is formalized in the following claim:

Proposition 8.2.3. $\dagger A$ in **Game** is the limit of the diagram

$$1 \longleftarrow A \longleftarrow A \triangleright A \longleftarrow A \triangleright A \triangleright A \longleftarrow \dots \longleftarrow A^{\triangleright n} \longleftarrow \dots$$

where each morphism $A^{\triangleright n} \longleftarrow A^{\triangleright n+1}$ is the partial copycat strategy such that Opponent never opens the $n+1$ -th component in $A^{\triangleright n+1}$. The legs $\lambda_n : \dagger A \multimap A^{\triangleright n}$ are given by the strategy

$$\lambda_n := \{s \in P_{\dagger A \multimap A^{\triangleright n}} \mid \forall t \sqsubseteq_{\text{even}} s. t \upharpoonright_{\dagger A} = t \upharpoonright_{A^{\triangleright n}}\}$$

defined up-to some re-indexing.

Proof. See Proof 30.2.2. □

The characterization of $\dagger -$ as a limit leads to simpler proofs of its categorical properties. In particular, as a corollary we obtain that:

Corollary 8.2.1. $\dagger - : \mathbf{Safe}^{\text{seq}} \rightarrow \mathbf{Safe}^{\text{seq}}$ is an endofunctor.

It also simplifies the task of showing $\dagger -$ is a comonad.

Proposition 8.2.4. $\dagger -$ defines a comonad in $\mathbf{Safe}^{\text{seq}}$.

Proof. See Proof 30.2.3. □

Proposition 8.2.5. $\dagger A$ is a comonoid over $(\mathbf{Safe}^{\text{seq}}, - \triangleright -, 1)$ for every game A .

Proof. Define the strategies

$$e_A := \lambda_1^A$$

$$d_A := \{s \in P_{\dagger A \multimap \dagger A \triangleright \dagger A} \mid \exists s_1, s_2. \forall t \sqsubseteq_{\text{even}} s. t \upharpoonright_{\dagger A} \sqsubseteq_{\text{even}} s_1 \cdot s_2 \wedge t \upharpoonright_{\dagger A \triangleright \dagger A} \sqsubseteq_{\text{even}} \iota_1^*(\langle s_1 \rangle) \cdot \iota_2^*(\langle s_2 \rangle)\}$$

It is a straightforward, although lengthy, computation to check that the associative and left and right unit laws hold. □

Chapter 9

Modules and Their Interpretation

Now, this long excursion through the semantics of linear logic and models of imperative programming languages serves one main purpose for our final goals: precisely characterizing the denotations of the modules and programs we use in our practical examples. In this chapter, we define a simple programming language providing a Kleene algebra with tests and operating over primitive commands of an effect signature E . A module or library then consists of a signature F specifying the operations the module implements, and then for each such operation a program providing its implementation. Once we define the syntax of such modules, we discuss two approaches for computing their denotations. One is obtained using much of the infrastructure of the co-Kleisli category $\mathbf{Safe}^{\text{seq}}_{\dagger}$. The second approach uses an *open* operational semantics to compute a set of plays.

9.1 Syntax

We assume

- a universe of values $\mathbf{Val}$,
- a set of variable names $\mathbf{Var}$, which should admit at least constants $v \in \mathbf{Val}$ and variable names $x \in \mathbf{Var}$ as expressions,
- a set of variable environments $\Delta \in \mathbf{Env}$ defined as partial maps $\Delta : \mathbf{Var} \rightharpoonup \mathbf{Val}$,
- a grammar of expressions $\mathbf{Expr}$,

- an evaluation function $\Downarrow_- : \text{Env} \rightarrow \text{Expr} \rightarrow \text{Val}$ which, given an environment $\Delta \in \text{Env}$ and an expression $t \in \text{Expr}$, computes a value $\Downarrow_\Delta t \in \text{Val}$.

We then define a language for commands over an effect signature E with returns in R by the following grammar:

$$\begin{aligned} \text{Prim}_E^R &:= x \leftarrow e(t) \mid \text{assume}(\phi) \mid \text{ret } t \\ \text{Com}_E^R &:= \text{Prim}_E^R \mid \text{Com}_E^R; \text{Com}_E^R \mid \text{Com}_E^R + \text{Com}_E^R \mid \text{Com}_E^{R*} \mid \text{skip} \end{aligned}$$

Prim_E^R stands for primitive commands, while Com_E^R is the grammar of commands. The most important commands work as follows:

- $x \leftarrow e(t)$ executes the effect $e \in E$ after evaluating its argument $t \in \text{Expr}$;
- $\text{ret } v$ stores in a reserved variable the result of the evaluation, and may only be called once in any control flow path of a program;
- $\text{assume}(\phi)$ takes a boolean function over the local environment and terminates computation if ϕ evaluates to **false**. $\text{assume}(-)$ can be used to implement a while loop and if conditionals in the usual way.

The remaining commands are as usual in a Kleene algebra.

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

Remark 2. *Where we defined a module to be a (mathematical) collection of programs, we could have instead formalized more of the syntax of the module implementations we have described, say in Ex. 6.1.4. That is, a module would start with a module signature that is syntactically defined. Formalizing this would then require defining a simple notion of well-typing of the programs below the signature, so that when the module is well-typed, we would be able to separate the programs into a collection as above. To avoid expounding on these details, we simply assume this part of the process has already been carried out, and focus on interpreting the programs from this intermediary representation.*

9.2 Interpreting Modules Denotationally

Fix a type for modules $E \rightarrow F$. Then, given a command $C \in \text{par}(f) \rightarrow \text{Com}_E^{\text{ar}(f)}$, we compute a set

$$\llbracket C \rrbracket_\Delta : \mathcal{P}(\text{Env} \times \mathbb{SP}_{\dagger E \multimap \{f : \text{par}(f) \rightarrow \text{ar}(f)\}}^{\text{OQ}}) + \perp$$

of pairs of environments together with a play that produces that environment starting from Δ . Here, $\perp$ means that there is a typing issue at some point of the execution. The interpretation function is then defined by:

$$\llbracket x \leftarrow e(a) \rrbracket_\Delta := \cup_{v \in \text{ar}(e)} \{(\{\Delta[x \mapsto v]\}, \downarrow e(\downarrow_\Delta a) \cdot v) \mid \downarrow_\Delta a \in \text{par}(e)\} \cup \{\perp \mid \downarrow_\Delta a \notin \text{par}(e)\}$$

$$\llbracket \text{ret } v \rrbracket_\Delta := \{(\Delta[\text{ret} \mapsto \downarrow_\Delta v], \downarrow_\Delta v) \mid \downarrow_\Delta v \in \text{ar}(f)\} \cup \{\perp \mid \downarrow_\Delta v \notin \text{ar}(f)\}$$

$$\llbracket \text{assume}(\phi) \rrbracket_\Delta := \begin{cases} \{(\Delta, \{\epsilon\})\}, & \phi(\Delta) = \text{true} \\ \emptyset, & \phi(\Delta) = \text{false} \end{cases}$$

$$\begin{aligned} \llbracket C; C' \rrbracket_\Delta &:= \{(\Delta'', s \cdot s') \mid \exists \Delta'. \exists \Delta''. (\Delta', s) \in \llbracket C \rrbracket \wedge (\Delta'', s') \in \llbracket C' \rrbracket_{\Delta'} \wedge s \cdot s' \in \mathbb{SP}^{\text{OQ}}\} \\ &\cup \{\perp \mid \perp \in \llbracket C \rrbracket_\Delta\} \\ &\cup \{\perp \mid \exists \Delta'. (\Delta', -) \in \llbracket C \rrbracket_\Delta \wedge \perp \in \llbracket C' \rrbracket_{\Delta'}\} \end{aligned}$$

$$\llbracket C + C' \rrbracket_\Delta := \llbracket C \rrbracket_\Delta \cup \llbracket C' \rrbracket_\Delta \quad \llbracket C^* \rrbracket_\Delta := \cup_{n \in \mathbb{N}} \llbracket C^n \rrbracket_\Delta \quad \llbracket \text{skip} \rrbracket_\Delta := \{(\Delta, \epsilon)\}$$

where C^n is defined recursively by $C^0 := \text{skip}$ and $C^{n+1} := C; C^n$.

Now, given $v \in \text{par}(f)$ for $f \in F$, we define $\Delta_v := \emptyset[\text{arg} \mapsto v]$. Then, given $M^f : \text{par}(f) \rightarrow \text{Com}_E^{\text{ar}(f)}$, its interpretation is:

$$\llbracket M^f \rrbracket := \cup_{v \in \text{par}(f)} f(v) \cdot \llbracket M^f(v) \rrbracket_{\Delta_v}$$

Assuming $\perp \notin \llbracket M^f \rrbracket$, we have then that $\llbracket M^f \rrbracket : \dagger E \multimap \{f : \text{par}(f) \rightarrow \text{ar}(f)\} \in \mathbf{Safe}^{\text{seq}}$, up-to some re-indexing. In this situation (i.e., assuming $\perp \notin \llbracket M^f \rrbracket$ for any f), we define

the interpretation of M by:

$$\llbracket M \rrbracket := (\cup_{f \in F} \llbracket M^f \rrbracket^\bullet)^\dagger : \dagger E \multimap \dagger F$$

We see here that we then exploit the entire structure of the linear logic model of state to achieve this denotation. We remind the reader of our key assumptions: (1) a return statement appears in any control flow path of each M^f and (2) $\perp$ is never computed, in other words, every M^f is well-typed, and (3) for every $f \in F$, $\llbracket M^f \rrbracket \neq \emptyset$. We will assume this to be the case for any program we verify (and we will only define programs for which this is true).

9.3 Interpreting Modules Operationally

In practical verification, it is useful to instead process a program operationally. This both enables reasoning about state, when convenient, and reasoning about programs “step-by-step” with a reduced need for structural recursion on a program. We therefore present an open operational semantics for our modules, in the sense that environment operations are not assumed to have any particular semantics.

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

$$\text{UndState} = \text{Env} \times P_{\dagger E}$$

so that a state $(\Delta, s) \in \text{UndState}$ contains a local environment $\Delta \in \text{Env}$ (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 $s \in \dagger E$. Concretely, s is the history of operations on the underlying object.

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

$$\llbracket B \rrbracket : \text{UndState} \rightarrow \mathcal{P}(\text{UndState})$$

satisfying moreover that for all $(\Delta, s) \in \text{UndState}$, if $(\Delta', s') \in \llbracket B \rrbracket(\Delta, s)$ then:

- $\text{Dom}(\Delta) \subseteq \text{Dom}(\Delta')$

- $\exists t.s' = s \cdot t$

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.

The semantics for the primitives we have considered is given by the following state transformers. First, we have a command **id** with interpretation given by:

$$\llbracket \mathbf{id} \rrbracket(\Delta, s) = \{(\Delta, s)\}$$

which behaves exactly like **skip**.

A command **ret** – with interpretation:

$$\llbracket \mathbf{ret} \ t \rrbracket(\Delta, s) = \begin{cases} (\Delta[\mathbf{res} : v], s), & \Delta(\mathbf{res}) = \perp \wedge \Downarrow_{\Delta} t = v \\ \emptyset, & \Delta(\mathbf{res}) \neq \perp \end{cases}$$

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 $x \leftarrow e(t)$ with interpretation given by:

- If $\mathbf{even}(s)$ then $\llbracket x \leftarrow e(a) \rrbracket(\Delta, s) := \begin{cases} \{(\Delta, s \cdot e(a))\}, & a \in \mathbf{par}(e) \wedge \Downarrow_{\Delta} t = a \\ \emptyset, & \text{otherwise} \end{cases}$
- If $\pi_{\alpha}(s) = p \cdot e(a)$ where $\Downarrow_{\Delta} t = a$ then

$$\llbracket x \leftarrow e(a) \rrbracket(\Delta, s) := \{(\Delta[x \mapsto v], s \cdot v) \mid v \in \mathbf{ar}(e)\}$$

- Otherwise, $\llbracket x \leftarrow e(a) \rrbracket(\Delta, s) := \emptyset$.

This models the fact that the implementation may call effects from its source component E . $x \leftarrow e(t)$ executes the effect $e \in E$ with argument a obtained by evaluating t under the local environment Δ .

Finally, to implement branching, we have a command **assume**(ϕ) where $\phi : \mathbf{Env} \rightarrow \mathbf{Bool}$

is interpreted by

$$\llbracket \text{assume}(\phi) \rrbracket(\Delta, s) := \begin{cases} \{(\Delta, s)\}, \phi(\Delta) = \text{true} \\ \emptyset, \text{ Otherwise} \end{cases}$$

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

The local semantics is then given by

$$\begin{aligned} \rightsquigarrow &\subseteq \text{Com}_E^R \times \text{Prim}_E^R \times \text{Com}_E^R \\[10pt] \frac{}{B \rightsquigarrow_B \text{skip}} \quad & \frac{C_1 \rightsquigarrow_B C'_1}{C_1; C_2 \rightsquigarrow_B C'_1; C_2} \quad \frac{}{\text{skip}; C \rightsquigarrow_{\text{id}} C} \quad \frac{}{C^* \rightsquigarrow_{\text{id}} C; C^*} \\[10pt] \frac{}{C^* \rightsquigarrow_{\text{id}} \text{skip}} \quad & \frac{}{C_1 + C_2 \rightsquigarrow_{\text{id}} C_1} \quad \frac{}{C_1 + C_2 \rightsquigarrow_{\text{id}} C_2} \quad \frac{}{\text{ret } v; C \rightsquigarrow_{\text{id}} \text{ret } v} \\[10pt] \longrightarrow &\subseteq (\text{Cont} \times \text{UndState}) \times M_E \times (\text{Cont} \times \text{UndState}) \\[10pt] \frac{(\Delta', s \cdot m) \in \llbracket B \rrbracket(\Delta, s) \quad C \rightsquigarrow_B C'}{\langle (C, \perp), \Delta, s \rangle \xrightarrow{m} \langle (C', B), \Delta', s \cdot m \rangle} \quad & \frac{(\Delta', s \cdot m) \in \llbracket B \rrbracket(\Delta, s) \quad B \neq \perp}{\langle (C, B), \Delta, s \rangle \xrightarrow{m} \langle (C, \perp), \Delta', s \cdot m \rangle} \end{aligned}$$

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 may take two steps to reduce. In the first rule, when the pending primitive is $\perp$, a command reduction fires and produces a primitive B , whose interpretation advances the state and sets B as the new pending primitive. In the second rule, when there is a pending primitive $B \neq \perp$, the interpretation of B resolves it and clears the pending primitive to $\perp$. This reflects that invocations and responses are treated as separate atomic events.

To move to the full module semantics, we use the set of states

$$\text{ModState} = \text{Env} \times P_{\dagger E \multimap \dagger F}$$

There is an obvious lifting of $\llbracket - \rrbracket$ to ModState by advancing the play state further only by

concatenation in the same way.

$$\begin{aligned}
& \longrightarrow \subseteq (\mathbf{Cont} \times \mathbf{ModState}) \times (M_{\dagger E \multimap \dagger F} \times \mathbf{Mod}) \times (\mathbf{Cont} \times \mathbf{ModState}) \\
& \frac{f \in F \quad a \in \mathbf{par}(f) \quad \Delta_a = \Delta[\mathbf{arg} : a]}{\langle \mathbf{idle}, \Delta, s \rangle \xrightarrow{f(a)}_M \langle (M[\alpha]^f, \perp), \Delta_a, s \cdot f \rangle} \\
& \frac{\langle (C, B), \Delta, s \upharpoonright_{\dagger E} \rangle \xrightarrow{m} \langle (C', B'), \Delta', s' \upharpoonright_{\dagger E} \rangle}{\langle (C, B), \Delta, s \rangle \xrightarrow{m}_M \langle (C', B'), \Delta', s' \rangle} \\
& \frac{s \upharpoonright_{\dagger F} = p \cdot f \quad v \in \mathbf{ar}(f)}{\langle (\mathbf{ret} \ v, \perp), \Delta, s \rangle \xrightarrow{v}_M \langle \mathbf{idle}, \emptyset, s \cdot v \rangle} \qquad \frac{c \neq \mathbf{stv}}{\langle c, \Delta, s \rangle \xrightarrow{\bullet}_M \langle \mathbf{stv}, \Delta, s \rangle}
\end{aligned}$$

Here, a continuation $c \in \mathbf{Cont}$ consists of $c := \mathbf{idle} + \mathbf{stv} + \mathbf{Com} \times (\mathbf{Prim} + \{\perp\})$. The initial continuation is $\mathbf{idle}$. The initial environment is the empty mapping $\emptyset$. The four rules correspond, in order, to a target component invocation, a step in the source component, a return in the target component, and being deterministically starved.

We obtain a regular strategy $\llbracket M \rrbracket_S : \dagger E \multimap F$ from a module $M \in \mathbf{Mod}$ by:

$$\llbracket M \rrbracket_S = \{s \mid \exists c \in \mathbf{Cont}. \exists \Delta \in \mathbf{Env}. \langle \mathbf{idle}, \emptyset, \epsilon \rangle \xrightarrow{s^*}_M \langle c, \Delta, s \rangle\}$$

A program M can be linked with a specification for its library E , which we denote by $\mathbf{Link} \ \mathcal{V}_E; M$. The purpose of this is to reason about the correctness of the implementation M once we have a specification for its library, as will be clear soon. So, say we are given a state transition system $\mathcal{V}_E := ((S_E, M_{\dagger E}, \multimap_E, s_0), \text{well-formed in that } \llbracket \mathcal{V}_E \rrbracket_S : \dagger E$. The operational semantics of $\mathbf{Link} \ \mathcal{V}_E; M$ is given by:

$$\longrightarrow_{\mathcal{V}_E} \subseteq (\text{Cont} \times \text{ModState} \times S_E) \times (M_{\dagger F} \times \text{Mod}) \times (\text{Cont} \times \text{ModState} \times S_E)$$

$$\frac{f \in F \quad a \in \text{par}(f) \quad \Delta_a = \Delta[\text{arg} : a]}{\langle \text{idle}, \Delta, s, \text{st}_E \rangle \xrightarrow{\mathcal{V}_E}^{f(a)^M} \langle (M[\alpha]^f, \perp), \Delta_a, s \cdot f, \text{st}_E \rangle}$$

$$\frac{\langle \langle (C, B), \Delta, s \restriction_{\dagger E} \rangle \xrightarrow{m} \langle \langle (C', B'), \Delta', s' \restriction_{\dagger E} \rangle \quad \text{st}_E \xrightarrow{m}_E \text{st}'_E}{\langle \langle (C, B), \Delta, s, \text{st}_E \rangle \xrightarrow{\mathcal{V}_E}^M \langle \langle (C', B'), \Delta', s', \text{st}'_E \rangle}$$

$$\frac{s \restriction_{\dagger F} = p \cdot f \quad v \in \text{ar}(f)}{\langle \langle \text{ret } v, \perp \rangle, \Delta, s, \text{st}_E \rangle \xrightarrow{\mathcal{V}_E}^{v \cdot M} \langle \text{idle}, \emptyset, s \cdot v, \text{st}_E \rangle} \quad \frac{c \neq \text{stv}}{\langle c, \Delta, s, \text{st}_E \rangle \xrightarrow{\mathcal{V}_E}^{\bullet M} \langle \text{stv}, \Delta, s, \text{st}_E \rangle}$$

From a linked program $\text{Link } \mathcal{V}_E; M$, we can obtain a corresponding strategy $\llbracket \text{Link } \mathcal{V}_E; M \rrbracket_S : \dagger F$ similarly to before:

$$\llbracket \text{Link } \mathcal{V}_E; M \rrbracket_S = \{s \mid \exists c \in \text{Cont}. \exists \Delta \in \text{Env}. \langle \text{idle}, \emptyset, \epsilon \rangle \xrightarrow{\mathcal{V}_E}^s \langle c, \Delta, s \rangle\}$$

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

Proposition 9.3.1. *For any well-typed $M \in \text{Mod}$, $\llbracket M \rrbracket_S$ defines a regular strategy $\llbracket M \rrbracket_S : \dagger E \multimap \dagger F$ and given $\mathcal{V}_E := ((S_E, M_{\dagger E}, - \xrightarrow{E} -), s_0)$ well-formed in that $\llbracket \mathcal{V}_E \rrbracket_S : \dagger E$,*

$$\llbracket \text{Link } \mathcal{V}_E; M \rrbracket_S = \llbracket \mathcal{V}_E \rrbracket_S ; \llbracket M \rrbracket_S$$

Chapter 10

A Refinement Calculus of Certified Components

We now have enough ingredients to define a compositional notion of correctness for components. Our approach is inspired by the refinement calculus [Back, 1981]. The main definition of refinement as defined by Back [1981] goes as follows. First, one assumes given a set **Prog** of programs and a set **Spec** of specifications. Then, a correctness criterion for **Prog** consists of a satisfaction relation $- \models - \subseteq \mathbf{Spec} \times \mathbf{Prog}$, where $S \models P$ formalizes the notion that program P satisfies specification S . Then, refinement is defined by:

Definition 10.0.1 ([Back, 1981]). A program P' is a refinement of P , written $P' \sqsupseteq P$ when

$$\forall S \in \mathbf{Spec}. S \models P \implies S \models P'$$

We will see how this definition works for us almost exactly.

10.1 The Refinement Calculus Across Models

We start with the definition of our compositional refinement calculus in an arbitrary symmetric monoidal **Inf**-enriched category.

Definition 10.1.1. Let $(\mathbf{C}, - \otimes -, \mathbf{1})$ be a symmetric monoidal **Inf**-enriched category. We define the **Inf**-category $\mathcal{R}(\mathbf{C})$ as the category whose objects are morphisms $v : \mathbf{C}(\mathbf{1}, A)$, and

morphisms $f \in \mathcal{R}(\mathbf{C})(v_A, v_B)$, where $v_A : \mathbf{1} \rightarrow A$ and $v_B : \mathbf{1} \rightarrow B$, are given by morphisms $f \in \mathbf{C}(A, B)$ satisfying that:

$$v_B \leq f \circ v_A$$

Composition is exactly as in $\mathbf{C}$ and the identity $\mathbf{id}_A \in \mathcal{R}(\mathbf{C})(v_A, v_B)$ is given by the identity $\mathbf{id}_A \in \mathbf{C}(A, A)$.

Here, $- \leq -$ denotes the enrichment order on hom-objects; concretely, for strategies, $\sigma \leq \tau$ iff $\sigma \supseteq \tau$. This definition refines the original definition of refinement by Back [1981] in several ways. First, we take into account a notion of typing for our components. Moreover, specifications are two-sided. Essentially, a component $f \in \mathcal{R}(\mathbf{C})(v_A, v_B)$ is typed $f : A \rightarrow B$ first in $\mathbf{C}$, which can be understood as meaning that f describes some interaction between the types A and B . When given a specification $v_A : A$ of type A , and a specification $v_B : B$, we can ask that f satisfy the verification condition:

$$v_B \leq f \circ v_A$$

and if it holds, we say that f satisfies the specification $v_A \Rightarrow v_B$, written $f : v_A \Rightarrow v_B$.

This idea is nicely captured by the formalism of [Melliès and Zeilberger, 2015], by noting that there exists a forgetful functor

$$U : \mathcal{R}(\mathbf{C}) \rightarrow \mathbf{C} \quad (v : \mathbf{1} \rightarrow A) \xrightarrow{U} A \quad f : v_A \Rightarrow v_B \xrightarrow{U} f : A \rightarrow B$$

Considering the remarks above, we think of $\mathcal{R}(\mathbf{C})$ as the category of certified $\mathbf{C}$ components.

We spend the rest of this section developing most of the argument that $\mathcal{R}(\mathbf{C})$ is indeed a symmetric monoidal **Inf**-enriched category, and then we finish by presenting this fact in terms of a nice calculus for verification.

First, notice that the fact that composition in $\mathcal{R}(\mathbf{C})$ remains associative, and that $\mathbf{id}$ is still a neutral element follow trivially. What is not guaranteed is that $\mathbf{id}_A : v_A \Rightarrow v_A$ is indeed verified.

Proposition 10.1.1. *For any $v_A : \mathbf{1} \rightarrow A$, $\mathbf{id}_A : v_A \Rightarrow v_A$.*

Proof. By neutrality: $\mathbf{id}_A \circ v_A = v_A$, and then the refinement holds by reflexivity. $\square$

This establishes the ordinary categorical structure of $\mathcal{R}(\mathbf{C})$. Now, we show it inherits the enrichment as well.

Proposition 10.1.2. *For any $v_A : \mathbf{1} \rightarrow A$ and $v_B : \mathbf{1} \rightarrow B$, $\mathcal{R}(\mathbf{C})(v_A, v_B)$ is a sub-lattice of $\mathbf{C}(A, B)$.*

Proof. The key claim to be shown here is that for any collection $F = (f_i : v_A \Rightarrow v_B)_{i \in I}$, $f := \bigwedge_{i \in I} f_i : v_A \Rightarrow v_B$. Indeed, for any $f_i \in F$, $v_B \leq f_i \circ v_A$ since $f_i : v_A \Rightarrow v_B$. Hence, v_B is a lower bound of $\{f_i \circ v_A\}_{i \in I}$, and by cocontinuity of $- \circ v_A$:

$$v_B \leq \bigwedge_{i \in I} (f_i \circ v_A) = \left(\bigwedge_{i \in I} f_i \right) \circ v_A = f \circ v_A$$

as desired. $\square$

Note that since $\mathcal{R}(\mathbf{C})(v_A, v_B)$ is a sub-lattice of $\mathbf{C}(A, B)$, both composition and the tensor are still cocontinuous functors. What remains to be shown is that they are still well-defined.

Proposition 10.1.3.

- For any $v_A : \mathbf{1} \rightarrow A$, $v_B : \mathbf{1} \rightarrow B$, $v_C : \mathbf{1} \rightarrow C$, it holds that if

$$f : v_A \Rightarrow v_B \wedge g : v_B \Rightarrow v_C \implies g \circ f : v_A \Rightarrow v_C$$

- For any $v_A : \mathbf{1} \rightarrow A$, $v_B : \mathbf{1} \rightarrow B$, $v_C : \mathbf{1} \rightarrow C$, $v_D : \mathbf{1} \rightarrow D$,

$$f : v_A \Rightarrow v_C \wedge g : v_B \Rightarrow v_D \implies f \otimes g : v_A \otimes v_B \Rightarrow v_C \otimes v_D$$

Proof. For the first claim:

$$\begin{aligned}
v_C &\leq g \circ v_B && (\text{Def. } g : v_B \Rightarrow v_C) \\
&\leq g \circ (f \circ v_A) && (\text{Def. } f : v_A \Rightarrow v_B \text{ and monotonicity}) \\
&= (g \circ f) \circ v_A && (\text{Associativity})
\end{aligned}$$

And for the second claim:

$$\begin{aligned}
v_C \otimes v_D &\leq (f \circ v_A) \otimes (g \circ v_B) && (\text{Def. } f : v_A \Rightarrow v_C \text{ \& } g : v_B \Rightarrow v_D) \\
&= (f \otimes g) \circ (v_A \otimes v_B) && (\text{Bifunctoriality})
\end{aligned}$$

□

All the coherence diagrams of the composition and tensor operations still hold, as they are exactly the same.

Now, this categorical treatment also helps reveal that typical properties in verification frameworks just follow from more basic categorical properties.

Proposition 10.1.4 (Consequence Rule).

- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v_A \leq v'_A \iff \mathbf{id}_A : v'_A \Rightarrow v_A$
- $v'_A \geq v_A \wedge f : v_A \Rightarrow v_B \wedge v_B \geq v'_B \implies f : v'_A \Rightarrow v'_B$

Proof. For the first claim, we have in the forward direction that:

$$v_A \leq v'_A = \mathbf{id}_A \circ v'_A$$

and in the backward direction:

$$v_A \leq \mathbf{id}_A \circ v'_A = v'_A$$

For the second property, we simply note that: $f = \mathbf{id}_B \circ f \circ \mathbf{id}_A$. But from the first property we have:

$$\mathbf{id}_A : v'_A \Rightarrow v_A \qquad \mathbf{id}_B : v_B \Rightarrow v'_B$$

so that

$$f = v'_A \xRightarrow{\mathbf{id}_A} v_A \xRightarrow{f} v_B \xRightarrow{\mathbf{id}_B} v'_B$$

□

We have already noted in Ch. 3 how contextual refinement is just monotonicity. We reinforce this here by the following proposition.

Proposition 10.1.5 (Contextual Refinement).

- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v'_A \geq v_A \iff \forall B \in \mathbf{C}. \forall f : A \rightarrow B. f \circ v'_A \geq f \circ v_A$
- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v'_A \geq v_A \iff \forall B, C \in \mathbf{C}. \forall f : B \rightarrow C. v'_A \otimes f \geq v_A \otimes f$
- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v'_A \geq v_A \iff \forall B, C \in \mathbf{C}. \forall f : B \rightarrow C. f \otimes v'_A \geq f \otimes v_A$

Proof. The forward direction is just monotonicity of $- \circ -$ (and $- \otimes -$) under $- \leq -$. The backward direction of the first claim holds by instantiating $B = A$ and $f = \mathbf{id}_A$:

$$v'_A = \mathbf{id}_A \circ v'_A \geq \mathbf{id}_A \circ v_A = v_A$$

For the backward direction of the second claim, we instantiate $B = C = \mathbf{1}$ and $f = \mathbf{id}_1$, then, by assumption:

$$v'_A \otimes \mathbf{id}_1 \geq v_A \otimes \mathbf{id}_1$$

Now, we compose the right unitor on both sides and use monotonicity of composition to get:

$$\rho_A \circ (v'_A \otimes \mathbf{id}_1) \geq \rho_A \circ (v_A \otimes \mathbf{id}_1)$$

But by naturality:

$$v'_A \circ \rho_1 = \rho_A \circ (v'_A \otimes \mathbf{id}_1) \geq \rho_A \circ (v_A \otimes \mathbf{id}_1) = v_A \circ \rho_1$$

But since ρ_1 is an isomorphism it admits an inverse ρ_1^{-1} , so that by monotonicity:

$$v'_A = v'_A \circ \mathbf{id}_1 = v'_A \circ \rho_1 \circ \rho_1^{-1} \geq v_A \circ \rho_1 \circ \rho_1^{-1} = v_A \circ \mathbf{id}_1 = v_A$$

as desired.

The third claim follows from the second by symmetry. $\square$

We conclude this section by collecting the properties of $\mathcal{R}(\mathbf{C})$ into a compositional calculus of certified implementations. In the rules below, all specifications $v : \mathbf{1} \rightarrow A$ and morphisms $f : A \rightarrow B$ are assumed well-typed in $\mathbf{C}$.

$$\begin{array}{c}
\frac{}{\mathbf{id}_A : v_A \Rightarrow v_A} \text{ID} \qquad \frac{f : v_A \Rightarrow v_B \quad g : v_B \Rightarrow v_C}{g \circ f : v_A \Rightarrow v_C} \text{COMP} \\
\\
\frac{f : v_A \Rightarrow v_C \quad g : v_B \Rightarrow v_D}{f \otimes g : v_A \otimes v_B \Rightarrow v_C \otimes v_D} \text{TENSOR} \quad \frac{v'_A \geq v_A \quad f : v_A \Rightarrow v_B \quad v_B \geq v'_B}{f : v'_A \Rightarrow v'_B} \text{CONSEQ} \\
\\
\frac{f \leq g \quad f : v_A \Rightarrow v_B}{g : v_A \Rightarrow v_B} \text{MONO} \qquad \frac{\forall i \in I. f_i : v_A \Rightarrow v_B}{\bigwedge_{i \in I} f_i : v_A \Rightarrow v_B} \text{INF}
\end{array}$$

We could also include rules stating that $\beta_{A,B}$, $\alpha_{A,B,C}$, λ_A , and ρ_A are all certified. But, since they are isomorphisms in $\mathbf{C}$, their inverses also yield certified implementations in the reverse direction. We could then include all of the coherence diagrams of a symmetric monoidal category. But then, why not include the fact that any diagram including only structural morphisms of a symmetric monoidal category is also certified? This quickly gets overwhelming. The right language for this calculus is perhaps string diagrams, to avoid introducing all of these extra rules. This is actually what we do for the most part. Once we verify each component of a system, we build a string diagram depicting its correctness. Our string diagrams are written instead as more traditional systems diagrams, as opposed to strings and nodes like in the category theory tradition. See Fig. 22.2 for an example.

10.2 The Refinement Calculus for Safety of Sequential Objects

In §10.1, we saw how any symmetric monoidal **Inf**-enriched category **C** defines a compositional refinement calculus which can be used to compose correct components into larger systems of components while guaranteeing the correctness of the composed system. We saw in Ch. 7 that our model of safety properties defines such a structure, which we called **Safe**^{seq}. Therefore, we can immediately consider the resulting compositional refinement calculus $\mathcal{R}(\mathbf{Safe}^{\text{seq}})$. In particular, given strategies $\nu_A : A$ and $\nu_B : B$, a strategy $\sigma : A \multimap B$ is said *certified* (for safety), written $\sigma : \nu_A \Rightarrow \nu_B$, when:

$$\nu_A; \sigma \sqsupseteq_S \nu_B$$

Essentially, ν_B is a safety property for the object implemented by σ when running “on top of ν_A ”. Frequently in practical examples, what we want to show is that $\llbracket M \rrbracket_S : \llbracket \mathcal{V}_E \rrbracket_S \Rightarrow \llbracket \mathcal{V}_F \rrbracket_S$ for some module M (see Ch. 9) and for state-transition systems $\mathcal{V}_E$ and $\mathcal{V}_F$. This setup reconstructs much of the structure of the certified abstraction layers framework (CAL) [Gu et al., 2015] used to verify the single-processor version of the CertiKOS [Gu et al., 2011] certified operating system.

Remark 3. *In CAL the layers are assumed to be deterministic. This means that by flipping the refinement from $- \sqsupseteq_S - := - \subseteq -$ to $- \sqsubseteq -$, one is able to also enforce certain liveness properties including must termination. We could have done this here, and in fact this is the approach my collaborators and I took in Oliveira Vale et al. [2022]. However, this is incompatible with most of the later developments on concurrency. So we present here, instead, the model we have built so far which includes non-determinism. Liveness will be dealt with in due time through an alternative approach that more easily generalizes to the concurrent setting.*

There is no benefit in re-digesting the compositionality calculus obtained by the $\mathcal{R}(\mathbf{Safe}^{\text{seq}})$ construction: it is readily obtained by replacing $- \leq -$ with $- \sqsupseteq_S -$ in §10.1. So instead, we elaborate on what verifying an individual module $\llbracket M \rrbracket : \nu_E \Rightarrow \nu_F$ entails. This could

be done through a variety of well-understood techniques, from Hoare logic to simulation techniques. Here, we focus on a simple notion of simulation.

We discuss an extended example due to Jérémie Koenig in Koenig and Shao [2020], and extended by him to the setting of Oliveira Vale et al. [2022]. We make minor adaptations for this dissertation, and most of the text in this section is due to Jérémie Koenig and Zhong Shao. Moreover, we note that while many of the specifications involved would involve the starvation tokens in the framework we developed, all of them are $\bullet$ -receptive. Therefore, it is enough to show refinement ignoring the starvation tokens, which is the approach we take below.

We start with the state-based presentation of CAL. We see in Fig. 10.1 a certified layer that implements a bounded queue with at most N elements using a circular buffer. In the underlay interface L_{rb} , its abstract state contains an array $f \in \mathbf{Val}^N$ with N values of type $\mathbf{Val}$ and two counters which will take values in the interval $0 \leq c_1, c_2 < N$. The array can be accessed through the primitives `get` and `set`; the primitives `inc1` and `inc2` increment the corresponding counter and return the counter's old value.

The overlay L_{bq} features two primitives `enq` and `deq` which respectively add a new element to the queue and remove the oldest element. If we attempt to add an element which would overflow the queue's capacity N , or remove an element from an empty queue, the result is $\emptyset$ (i.e., the primitive aborts).

The layer implementation M_{bq} stores the queue's elements into the array, between the indices given by the counters' values. This is expressed by the simulation relation R , which explains how overlay states are realized by M_{bq} in terms of underlay states. The code of M_{bq} can be interpreted in the monad $S_{rb} \rightarrow \mathcal{P}^1(- \times S_{rb})$, with calls to primitives of L_{rb} replaced by their specifications. We write $L_{rb}[M_{bq}]$ to denote the result. We define a certified layer $L_{rb} \vdash_R M_{bq} : L_{bq}$ if for each operation $\text{op} \in \{\text{enq}(v), \text{deq}() \mid v \in \mathbf{Val}\}$, a simulation between $L_{bq}.\text{op}$ and $L_{rb}[M_{bq}].\text{op}$ holds under R .

We now consider the situation in our setting. The operations of the layer interfaces

$$\begin{array}{c}
\boxed{\mathcal{V}_{\text{bq}}} \quad S_{\text{bq}} := \text{Val}^* \quad \text{BQueue} := \{\text{enq} : \text{Val} \rightarrow \mathbf{1} \quad , \quad \text{deq} : \mathbf{1} \rightarrow \text{Val}\} \\
\frac{| \vec{q} | < N}{(\vec{q}, \text{idle}) \xrightarrow{\text{enq}(v)} (\vec{q}, \text{enq}(v))} \quad \frac{}{(\vec{q}, \text{enq}(v)) \xrightarrow{\text{ok}} (\vec{q}v, \text{idle})} \\
\frac{}{(v\vec{p}, \text{idle}) \xrightarrow{\text{deq}} (v\vec{p}, \text{deq})} \quad \frac{}{(v\vec{p}, \text{deq}) \xrightarrow{v} (\vec{p}, \text{idle})} \\
\boxed{M_{\text{bq}}} \quad R \subseteq S_{\text{bq}} \times S_{\text{rb}} \\
M_{\text{bq}}^{\text{enq}(v)} := i \leftarrow \text{inc}_2(); \text{set}(i, v); \text{ret ok} \quad \vec{q} R (f, c_1, c_2) \Leftrightarrow (c_1 \leq c_2 < N \wedge \vec{q} = f_{c_1} \cdots f_{c_2-1}) \vee \\
M_{\text{bq}}^{\text{deq}()} := i \leftarrow \text{inc}_1(); v \leftarrow \text{get}(i); \text{ret } v \quad (c_2 \leq c_1 < N \wedge \vec{q} = f_{c_1} \cdots f_{N-1} f_0 \cdots f_{c_2-1}) \\
\boxed{\mathcal{V}_{\text{rb}}} \quad S_{\text{rb}} := \text{Val}^N \times \mathbb{N} \times \mathbb{N} \\
\text{RBuffer} := \{\text{set} : \mathbb{N} \times \text{Val} \rightarrow \mathbf{1} \quad , \quad \text{get} : \mathbb{N} \rightarrow \text{Val} \quad , \quad \text{inc}_1 : \mathbf{1} \rightarrow \mathbb{N} \quad , \quad \text{inc}_2 : \mathbf{1} \rightarrow \mathbb{N}\} \\
\frac{i < N}{((f, c_1, c_2), \text{idle}) \xrightarrow{\text{set}(i, v)} ((f, c_1, c_2), \text{set}(i, v))} \quad \frac{}{((f, c_1, c_2), \text{set}(i, v)) \xrightarrow{\text{ok}} ((f[i:=v], c_1, c_2), \text{idle})} \\
\frac{i < N}{((f, c_1, c_2), \text{idle}) \xrightarrow{\text{get}(i)} ((f, c_1, c_2), \text{get}(i))} \quad \frac{}{((f, c_1, c_2), \text{get}(i)) \xrightarrow{f_i} ((f, c_1, c_2), \text{idle})} \\
\frac{}{((f, c_1, c_2), \text{idle}) \xrightarrow{\text{inc}_1} ((f, c_1, c_2), \text{inc}_1)} \quad \frac{c'_1 = (c_1+1) \bmod N}{((f, c_1, c_2), \text{inc}_1) \xrightarrow{c_1} ((f, c'_1, c_2), \text{idle})} \\
\frac{}{((f, c_1, c_2), \text{idle}) \xrightarrow{\text{inc}_2} ((f, c_1, c_2), \text{inc}_2)} \quad \frac{c'_2 = (c_2+1) \bmod N}{((f, c_1, c_2), \text{inc}_2) \xrightarrow{c_2} ((f, c_1, c'_2), \text{idle})}
\end{array}$$

Figure 10.1: A certified abstraction layer $L_{\text{rb}} \vdash_R M_{\text{bq}} : L_{\text{bq}}$ implementing a bounded queue of size N using a ring buffer. The transition systems $\mathcal{V}_{\text{bq}}$ and $\mathcal{V}_{\text{rb}}$ define the overlay and underlay interfaces. The implementation M_{bq} and the simulation relation R are used in the correctness proof.

presented in Fig. 10.1 can be described by the following effect signatures:

$$E_{\text{bq}} := \{\text{enq}(v) : \mathbf{1}, \text{deq} : \mathbf{Val} \mid v \in \mathbf{Val}\}$$

$$E_{\text{rb}} := \{\text{set}(i, v) : \mathbf{1}, \text{get}(i) : \mathbf{Val}, \text{inc}_1 : \mathbb{N}, \text{inc}_2 : \mathbb{N} \mid i \in \mathbb{N}, v \in \mathbf{Val}\}$$

Note that a primitive of type $A \rightarrow B$ is described in the signature as an A -indexed family of operations of arity B . For example, $\text{get} : \mathbb{N} \rightarrow \mathbf{Val}$ corresponds to one operation $\text{get}(i) : \mathbf{Val}$ for each possible index $i \in \mathbb{N}$.

The strategy associated with the implementation $M_{\text{bq}} : E_{\text{rb}} \rightarrow E_{\text{bq}}$ outlined in Fig. 10.1 can be described as:

$$\llbracket M_{\text{bq}} \rrbracket^{\text{enq}(v)} = \downarrow \{\text{inc}_2 \cdot n \cdot \text{set}(n, v) \cdot \text{ok} \cdot \text{ok} \mid n \in \mathbb{N}\}$$

$$\llbracket M_{\text{bq}} \rrbracket^{\text{deq}} = \downarrow \{\text{inc}_1 \cdot n \cdot \text{get}(n) \cdot v \cdot v \mid n \in \mathbb{N} \wedge v \in \mathbf{Val}\}$$

The empty queue is a natural initial state, so we define the layer interface:

$$L_{\text{bq}} := (E_{\text{bq}}, \nu_{\text{bq}}) \quad \nu_{\text{bq}} := \mathcal{V}_{\text{bq}}^\#(\epsilon, \text{idle})$$

For example, for all $u, v \in \mathbf{Val}$, L_{bq} allows the play

$$\text{enq}(u) \cdot \text{ok} \cdot \text{enq}(v) \cdot \text{ok} \cdot \text{deq} \cdot u \cdot \text{deq} \cdot v \in \nu_{\text{bq}},$$

as witnessed by the sequence of states $\epsilon, u, uv, v, \epsilon$.

For ring buffers, we prefer not to make any assumptions on initial contents, so that get on a location which has not yet been set is undefined. The corresponding layer interface is:

$$L_{\text{rb}} := (E_{\text{rb}}, \nu_{\text{rb}}) \quad \nu_{\text{rb}} := \bigcap_{f \in \mathbf{Val}^N} \mathcal{V}_{\text{rb}}^\#((f, 0, 0), \text{idle})$$

In this case, $\text{set}(i, v) \cdot \text{ok} \cdot \text{get}(i) \cdot v$ is a play in ν_{rb} for all $i < N$, $v \in \mathbf{Val}$, but $\text{get}(i) \cdot v$ on its own is never accepted when $|\mathbf{Val}| > 1$.

The correctness of M_{bq} can be established using the simulation relation R given in

Fig. 10.1. We can show by induction on plays that:

$$\vec{q} R (f, c_1, c_2) \Rightarrow \llbracket M_{\text{bq}} \rrbracket \circ \mathcal{V}_{\text{rb}} \sharp ((f, c_1, c_2), \text{idle}) \sqsupseteq_S \mathcal{V}_{\text{bq}} \sharp (\vec{q}, \text{idle})$$

For the empty play ϵ , we have $\llbracket M_{\text{bq}} \rrbracket \circ \{\epsilon\} = \{\epsilon\}$. For a transition $(\vec{q}, \text{idle}) \xrightarrow{m} \dots \xrightarrow{n} (\vec{q}', \text{idle})$ in $\mathcal{V}_{\text{bq}}$, we only need to witness a related state $\vec{q}' R (f', c'_1, c'_2)$ of $\mathcal{V}_{\text{rb}}$ reached by the corresponding sequence of operations in M_{bq} . Since the initial states $\epsilon R (f, 0, 0)$ are related for all $f \in \text{Val}^N$, we can conclude $\llbracket M_{\text{bq}} \rrbracket \circ \nu_{\text{rb}} \sqsupseteq_S \nu_{\text{bq}}$.

Part II

Safety

Chapter 11

Introduction

We have now discussed compositional verification of sequential components in detail. What emerges is that symmetric monoidal **Inf**-enriched categories provide a natural domain in which to develop a compositional refinement calculus. We wish to develop such a category for safety and liveness of concurrent components. This brings several challenges, however.

First, concurrency brings significantly more flexibility to how and when different parties can add events to an execution. When several agents execute concurrently, plays are no longer alternating. For instance, n threads might make invocations before any of them receives a response, so even relatively simple proofs, such as associativity of strategy composition, become significantly more challenging. Moreover, in the sequential setting divergences and starvation can only happen as the last events of a play, whereas in a concurrent setting an agent can be starved or diverge while others are still able to run. This makes liveness a significantly more interesting issue in concurrent computation.

Second, concurrent computation is inherently non-deterministic. This is mostly due to the non-determinism introduced by scheduling between different agents. In a multi-threaded setting, scheduling is provided by an OS. This is typically modeled by a completely non-deterministic scheduler, even if the actual scheduling algorithm is deterministic. Properties of the scheduler, such as fairness, may then be assumed in order to prove stronger properties about the system.

But why not just assume a deterministic scheduler? As soon as we compose two components, we hide some events at their boundary. This makes certain scheduling decisions

disappear, so that composition causes the scheduling algorithm to no longer match the scheduling of visible events. Moreover, on a multicore processor or any other distributed system, non-deterministic scheduling is simply ground truth. Each core has its own independent clock regulating its cycles, typically a crystal oscillator. Due to physical properties of these clocks, they suffer fluctuations and drift from variations in temperature, voltage, etc. This makes modern computing devices thermodynamically chaotic systems: it is physically impossible to synchronize them perfectly to obtain a global clock across processors. The memory hierarchy adds further non-determinism, as we cannot know in advance whether cache accesses will hit or miss, resulting in unpredictable variations in cycles across processors, compounded by OS-level interrupts and interaction with other devices through device drivers.

The third complication has to do with abstraction. In a sequential setting, the basic kind of abstraction is data refinement [Hoare, 1989]: two implementations have the same (or more) behaviors up to a change in the representation of state. But in a concurrent setting, there is an entirely new opportunity: what if we could also abstract the concurrency itself? If we think of data refinement as abstraction along *space*, concurrency makes a new axis of abstraction very natural: *time*.

This is not just a question for verification; it is foundational to answering what we mean by a correct concurrent data structure. For instance, consider the trace

$$s_0 = \mathbf{0}:\text{get}()$$

representing that thread 0 calls the `get` operation of a concurrent counter. In a sequential setting, the only possible next event is `0:0`. But in a concurrent setting, any number of `inc()` events by other threads might interleave with this `get()`:

$$s_k = \mathbf{0}:\text{get}() \cdot \underbrace{\mathbf{1}:\text{inc}() \cdot \mathbf{1}:\text{ok} \cdot \dots \cdot \mathbf{k}:\text{inc}() \cdot \mathbf{k}:\text{ok}}_{k \text{ times}} \cdot \mathbf{0}:n$$

and then the question is: what should n be? The immediate answer would be k , but it turns out that in a concurrent setting it is sensible to allow $n \leq k$. The reason is that

the `get` might have read the value of the counter at any point between its invocation and response, so it might have seen any intermediate state of the counter that occurred during the interval of execution of the `get`.

It took decades to formalize this notion of functional correctness for concurrent objects, which was finally done by Herlihy and Wing [1990] through a correctness criterion known as *linearizability*. Linearizability improves data refinement by also allowing for abstraction *along the degree of concurrency*, so that a concurrent object is correct if, up to some allowable re-orderings of events across time, the executions of the concurrent counter can be related to the executions of a sequential counter. This reduces the challenge of specifying the functional correctness of a concurrent data structure to stating that it is linearizable to a corresponding functionally correct sequential data structure. Interestingly, the key insight of linearizability lies in the preservation of *happens-before ordering*, a technique for ordering events in time more loosely than a linear order. This notion itself, stemming from Lamport [1978], was directly inspired by the causal structure of spacetime in special relativity, showing that we are really talking about abstraction along time.

In this part, I provide an account of linearizability based on compositionality. I show that linearizability is in some sense intrinsic to compositional models of concurrency, meaning that any reasonable compositional model of concurrency supports a compatible notion of linearizability. This both provides a foundational explanation for linearizability, as opposed to the pragmatic justification given by Herlihy and Wing [1990], and allows its theory to be more readily generalized and adapted to new settings. Over this part as a whole, I provide a thorough account of safety of concurrent objects in a compositional setting.

11.1 The State of the Theory of Linearizability

Linearizability is a notion of correctness for concurrent objects introduced 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 captures a large class of

useful concurrent objects; it allows linearizable concurrent objects to be horizontally composed while preserving linearizability, which Herlihy and Wing [1990] call locality; it aids in the derivation of other safety properties [Herlihy and Wing, 1990]; and it is intuitive: a linearizable concurrent object essentially behaves as if its operations happened atomically under any concurrent execution, a property formalized by the notion of linearization point by Herlihy and Wing [1990] and by an observational refinement property by Filipovic et al. [2010].

Linearizability is commonly used to define correctness of concurrent objects and to aid in verification of concurrent code. It suffers from a few biases, however.

Atomicity: Because the classic definition of linearizability is based on linearizing to an atomic specification, most subsequent work has focused on atomicity. Even though Filipovic et al. [2010] noticed that the insight of linearizability lies not in atomicity but in preservation of the happens-before order, most subsequent work still focuses on atomicity. This is so even though many useful concurrent objects do not linearize, leading to numerous variations on the theme [Haas et al., 2016; Neiger, 1994; Castañeda et al., 2015; Goubault et al., 2018]. When aiming for compositionality, atomicity becomes a hindrance: even if an object linearizes to an atomic specification, the components used to implement it may not themselves be atomically linearizable.

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 $N = N_1 \parallel \dots \parallel N_k$, specified in the programming language, to form a syntactic term $\text{Link } L'_B; N$. A trace semantics $\llbracket - \rrbracket$ yields the traces for the resulting interface $\llbracket \text{Link } L'_B; N \rrbracket$, and an observational refinement property allows one 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 $\text{Link } L'_A; M$, to implement L'_B . There is *no* obvious way to compose M and N so as to re-use their proofs of linearizability to obtain a linearizable object $\text{Link } L'_A; (N \circ M)$. At

$$\frac{N}{\frac{L'_B}{\frac{M}{L'_A}}}$$

best, one has to either syntactically link them together and re-do the proofs, or inline M in N and re-verify the resulting code.

Syntax: There is also a bias towards syntax when approaches with some degree of compositionality are presented, 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 (e.g., hardware and software components, or components written in different programming languages). This situation occurs in real systems. For instance, Gu et al. [2016, 2018, 2015]’s verified OS contains components in both **C** and **Asm**. They manage to make the two interact by composing components only after compiling **C** code into **Asm** using CompCert [Leroy, 2009], a solution still reliant on syntactic linking. Without a compiler to aid with this, 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 [Matthews and Findler, 2007; Perconti and Ahmed, 2014; Patterson et al., 2017]. In a large heterogeneous system this becomes unwieldy, as there could be several computational models involved. A compositional abstract model could instead embed each heterogeneous component and reason about them at a coarser 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 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 was no clean theory addressing the semantic and computational content of linearizability or providing foundations for properties such as locality and observational refinement. As a side-effect, the proofs of these properties are rather complicated. This is a substantial problem: many of these proofs are much simpler assuming atomicity, and the lack of good tools to develop the theory beyond atomicity precludes developing expressive correctness criteria with a robust theory. For example, while there are generalizations of linearizability beyond atomicity, such as set linearizability [Neiger,

1994; Hemed et al., 2015] and interval linearizability [Castañeda et al., 2015], both lacked an observational refinement proof. A more general and abstract theory of linearizability, such as the one developed throughout this dissertation, could not only simplify these issues but also be more easily adapted to novel settings.

11.2 Compositional Verification in Practice

Compositionality plays a fundamental role in the semantics of programming languages, but it can also be powerful for scalability in formal verification. We now provide an example where compositionality helps organize a verification effort in the context of safety verification.

11.2.1 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:

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

meaning that $\mathbf{1} = \{\text{ok}\}$ is the set of return values for both the `acq` and `rel` operations. By $\dagger\text{Lock}$ we now mean a concurrent version of $\dagger\text{Lock}$, where each thread is able to play the game $\dagger\text{Lock}$ in parallel. Then, $P_{\dagger\text{Lock}}^\bullet$ is the set of traces of type $\dagger\text{Lock}$. An example of a concurrent trace in $P_{\dagger\text{Lock}}^\bullet$ is (the arrows keep track of individual threads and are merely a visual aid):

$$s = \quad \alpha_1:\text{acq} \quad \alpha_2:\text{acq} \rightarrow \alpha_2:\text{ok} \quad \xrightarrow{\alpha_3:\text{acq}} \alpha_2:\text{rel} \quad \xrightarrow{\alpha_3:\text{ok}} \alpha_2:\text{ok}$$

This trace s linearizes to the following atomic trace t , also in $P_{\dagger\text{Lock}}^\bullet$, called atomic because every invocation immediately receives its response:

$$t = \quad \alpha_2:\text{acq} \rightarrow \alpha_2:\text{ok} \rightarrow \alpha_2:\text{rel} \rightarrow \alpha_2:\text{ok} \quad \alpha_3:\text{acq} \rightarrow \alpha_3:\text{ok}$$

In particular, linearizability enforces that any operation that “happens before” another in s (an operation happens before another if the return of the first precedes the invocation

of the latter) still happens before that operation in t . This is usually formalized by defining a partial order on the events of a trace, called the happens-before order. We call this aspect of linearizability “preservation of happens-before order.”

As usual, concurrent objects are specified by sets of traces. A concurrent lock object is specified as a prefix-closed set of traces $\nu'_{\text{lock}} \subseteq P_{\dagger\text{Lock}}^\bullet$. To be correct, this specification ν'_{lock} should linearize to the atomic specification $\nu_{\text{lock}} \subseteq P_{\dagger\text{Lock}}^\bullet$ given by the set of traces $s \in P_{\dagger\text{Lock}}^\bullet$ such that:

if $s = s_1 \cdot \alpha_1:m_1 \cdot \alpha_2:m_2 \cdot \alpha_3:m_3 \cdot \alpha_4:m_4 \cdot s_2$ then

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

and, if s is non-empty, then its first invocation is **acq**. We adopt the convention that a primed specification (like ν'_{lock}) is more *concurrent* than its unprimed counterpart (like ν_{lock}).

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:

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

Its concrete concurrent specification ν'_{queue} can be specified as the largest set of traces $s \in P_{\dagger\text{Queue}}$ such that

if $s = p \cdot \alpha:\text{deq} \cdot \alpha:k \cdot s'$ and p is atomic
then either $\text{qstate}(p) = k :: q'$ or $\text{qstate}(p) = []$ and $k = \emptyset$,

where qstate is an inductively defined function taking an atomic trace p and returning the state $\text{qstate}(p)$ of the queue after executing p from the empty queue $[]$. Note that as soon as any non-atomic interleaving happens in a trace of ν'_{queue} , the behaviors of **enq** and **deq** are unspecified and therefore completely non-deterministic. This reflects the assumption

<pre> M_{queue}:Queue Import Q:Queue Import L:Lock enq(k) { L.acq(); r <- Q.enq(k); L.rel(); ret r } deq() { L.acq(); r <- Q.deq(); L.rel(); ret r } </pre>	<pre> M_{lock}:Lock Import F:FAI Import C:Counter Import Y:Yield acq() { my_tick <- F.fai(); while (cur_tick ≠ my_tick) { Y.yield(); cur_tick <- C.get() } ret ok } rel() { C.inc(); ret ok } </pre>

Figure 11.1: Shared Queue implementation (left), and Lock implementation (right)

that this Queue object is a sequential implementation not resilient to concurrent execution, i.e., a racy object.

Such a Queue object can be shared across several agents by locking around all the operations of Queue, as demonstrated in the following implementation $M_{\text{queue}} : \text{Lock} \otimes \text{Queue} \multimap \text{Queue}$ implementing a shared queue using a lock and a sequential queue implementation (see Fig. 11.1). Note that when several independent objects must be used together, we use the linear logic tensor $- \otimes -$ to compose them horizontally into a new object, such as in the source type of M_{queue} .

The queue object ν'_{queue} implemented by M_{queue} is linearizable to the usual atomic specification ν_{queue} of a Queue. But observe that ν'_{queue} is not linearizable to ν_{queue} . This means that the composition of ν'_{lock} and ν'_{queue} into an object of type $\text{Lock} \otimes \text{Queue}$, specified as $\nu'_{\text{lock}} \otimes \nu'_{\text{queue}}$ (the set of all sequentially consistent interleavings of ν'_{lock} and ν'_{queue}), is also not linearizable to an atomic specification. This is enough for approaches that are over-reliant on atomicity (and determinism) to be unable to handle this situation cleanly.

One solution is to remove the dependence on the non-linearizable queue by inlining its implementation in terms of programming language primitives. This is unfortunate, as intuitively what M_{queue} does is turn a non-linearizable queue into a linearizable one. In fact, when we formalize this example, we can show that this works for any racy object. Inlining its implementation removes the connection between the sequential implementation and the code implementing this sharing pattern.

Instead, one would like to use off-the-shelf sequential components freely, as in the code in Fig. 11.1. By divorcing linearizability from atomicity, we will still have that $\nu'_{\text{lock}} \otimes \nu'_{\text{queue}}$

is linearizable to $\nu_{\text{lock}} \otimes \nu'_{\text{queue}}$ according to a generalized notion of linearizability. We connect our model with a program logic to show that the code in Fig. 11.1 does implement a linearizable Queue object correctly.

11.2.2 Implementing a Lock

A typical implementation for Lock is the ticket lock implementation (see Fig. 11.1), relying on a sequential counter and a fetch-and-increment object with signatures

$$\text{Counter} := \{\text{inc} : \mathbf{1} \rightarrow \mathbf{1}, \text{get} : \mathbf{1} \rightarrow \mathbb{N}\} \quad \text{FAI} := \{\text{fai} : \mathbb{N}\}$$

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 ν'_{fai} object specification is linearizable to an atomic one ν_{fai} .

The Counter object ν'_{srcntr} has a subtler specification. It models a semi-racy sequential counter implementation similarly to the queue from §11.2.1. Unlike the racy queue, the counter must be slightly more defined, as the lock implementation requires the sequential implementation to be resilient to concurrent `get` calls and 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, ν'_{srcntr} , is *linearizable* (in our generalized sense) with respect to a *less* concurrent one, ν_{srcntr} , given by the largest set of traces $s \in P_{\dagger\text{Counter}}$ satisfying:

$$\begin{aligned} \text{If } s = p \cdot \alpha\text{:get} \cdot m \cdot s' \text{ then } m = \alpha\text{:}k \text{ and if moreover } p \upharpoonright_{\{\text{inc:ok}\}} \text{ is atomic and even-length} \\ \text{then } k = \# \text{inc}(p), \end{aligned}$$

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

Occasionally, one implements the ticket lock so that it yields while spinning so as to let other agents get access to the underlying computational resource (such as processor time).

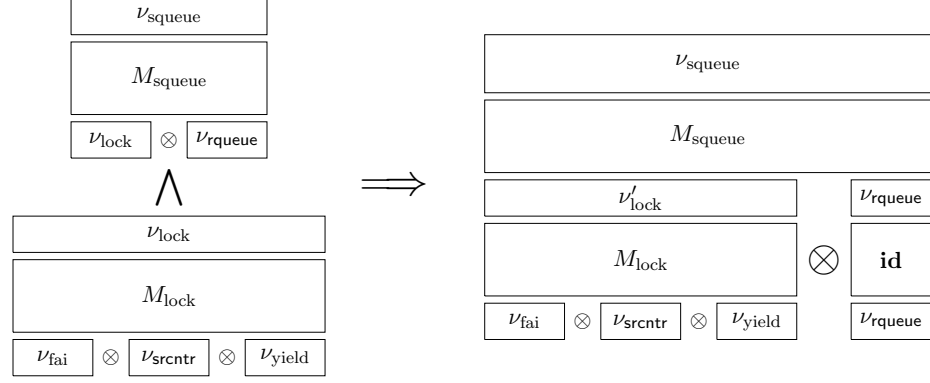


Figure 11.2: Correctness of individual components implies correctness of the composed system. The lock implementation M_{lock} (bottom-left) is verified against the linearized specifications ν_{fai} , ν_{srcntr} , and ν_{yield} , while the shared queue implementation M_{queue} (top-left) is verified using ν_{lock} and the racy queue ν_{rqueue} . By compositionality, these two individual results ($\wedge$) imply ($\Rightarrow$) that the vertically composed system (right) correctly implements ν_{queue} . Horizontal composition uses the linear logic tensor $- \otimes -$, and copy denotes the neutral element for composition.

For some purposes, this is crucial to obtain better liveness properties. For this, we define a signature

$$\text{Yield} := \{\text{yield} : 1\}$$

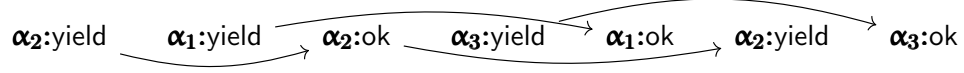
with concurrent specification ν'_{yield} given by

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

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 ν'_{yield} looks like:

$$\alpha_1 : \text{yield} \quad \alpha_2 : \text{yield} \rightarrow \alpha_2 : \text{ok} \quad \alpha_3 : \text{yield} \rightarrow \alpha_1 : \text{ok} \quad \alpha_2 : \text{yield} \rightarrow \alpha_3 : \text{ok}$$

Now, observe that by definition ν'_{yield} contains no atomic traces, as `yield` only returns if another `yield` happens concurrently with it. This means that no *atomic* linearized specification for ν'_{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:



That is, `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 client side of `Yield`. With the `Yield` object at hand, we verify that the implementation in Fig. 11.1 for the ticket lock is linearizable using a program logic. Once M_{lock} and M_{queue} 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 linearizability. We depict this example in Fig. 11.2.

11.2.3 Overview

Our work addresses the issues discussed throughout this introduction by showing that linearizability is already baked into 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 **Safe**, 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 **Safe** a compositional model **Safe** which does have neutral elements. This new model **Safe** differs from **Safe** in that its strategies σ of type $A \multimap B$ are strategies of **Safe** that moreover are invariant upon composition with a certain strategy called `copy-`. This strategy corresponds to the traces of a program where each agent in the concurrent system runs `seqcopy` in parallel. In other words, taking the code in §6.3 for `seqcopy`, then the `copy` corresponds to the code in Fig. 11.3.

This construction comes with some infrastructure: a saturation operation K_{copy} and a forgetful operation `Emb`, depicted in Fig. 11.3. Importantly, $K_{\text{copy}} \sigma$ is *defined* to be `copyA;  $\sigma$ ; copyB` while `Emb  $\sigma$` is *by definition* just σ itself. The central but simple result of this part is that

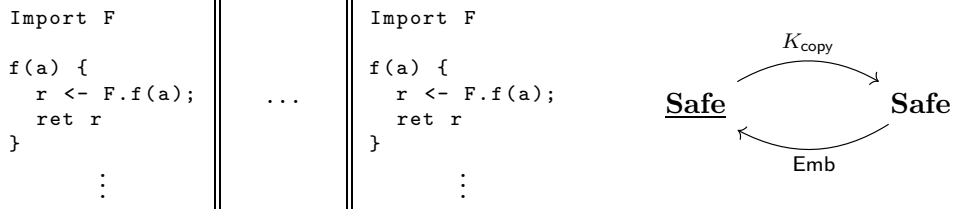


Figure 11.3: Code corresponding to `copy_` (left); Diagram depicting the operations K_{copy} and Emb (right)

Proposition 11.2.1 (Compositional Linearizability). *A strategy $\sigma : A \in \underline{\text{Safe}}$ is linearizable to a strategy $\tau : A \in \underline{\text{Safe}}$ if and only if*

$$\sigma \sqsubseteq_S K_{\text{copy}} \tau$$

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

We give a novel characterization of linearizability by showing that the strategy `copy_` corresponds to proofs of linearizability, giving a computational interpretation to such proofs. We call this a *computational* interpretation because `copy_` is the denotation of the concrete program in Fig. 11.3.

Proposition 11.2.2 (Computational Interpretation). *s linearizes to t, both plays of type A, if and only if there exists a play $s' \in \text{copy}_A : A \multimap A$ such that*

$$s' \upharpoonright_{A,-} = t \qquad s' \upharpoonright_{-,A} = s$$

where A and A denote the source and target components of $A \multimap A$.

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 11.2.3 (Observational Refinement). *$\nu'_A : A \in \text{Safe}$ is linearizable to $\nu_A : A \in$*

Safe if and only if for all concurrent games B and $\sigma : A \multimap B$ it holds that

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

After that, we define a tensor $A \otimes B$ amounting to all the sequentially consistent interleavings of traces of type A with traces of type B , that is, interleavings such that each agent behaves sequentially locally. We then use the insight given by the computational interpretation of linearizability proofs to show that for any A and B :

$$\text{copy}_{A \otimes B} = \text{copy}_A \otimes \text{copy}_B$$

This equation can be interpreted to say that proofs of linearizability for objects of type $A \otimes B$ correspond to a pair of a proof of linearizability for the 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 11.2.4 (Locality). *Let $\nu'_A : A$, $\nu'_B : B$ and $\nu_A : A$, $\nu_B : B$. Then*

$$\nu' = \nu'_A \otimes \nu'_B \text{ is linearizable w.r.t. } \nu = \nu_A \otimes \nu_B$$

if and only if

$$\nu'_A \text{ is linearizable w.r.t. } \nu_A \text{ and } \nu'_B \text{ is linearizable w.r.t. } \nu_B$$

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. 11.2.1.

This success in developing the fundamental theory of linearizability from this angle motivates a straightforward categorification of the notion of linearizability in models based on this kind of Karoubization. We also closely compare our definition with other well-established notions of linearizability in locally sequential models: the original formulation in terms of atomic specifications [Herlihy and Wing, 1990] and the more recent and expressive interval-sequential linearizability [Castañeda et al., 2015].

We then build an axiomatic approach for formulating linearizability proofs that unifies the possibilities approach of Herlihy and Wing [1990] with our methodology. In the process, we use the computational interpretation to show that each possibility axiom corresponds to a different kind of move that the copycat strategy might make in a valid execution. We refine this into a principled way to annotate an implementation strategy with proofs of linearizability, which ultimately results in a framework for possibility-based axiomatic proofs and an alternative way to characterize linearization points (and their generalization to linearization intervals). We then elaborate our axiomatic approach into a rely-guarantee program logic inspired by [Khyzha et al., 2017].

At this point, we will have all the ingredients to compose concurrent objects into larger systems, as in the example in Fig. 11.2. We refine the compositional refinement calculus from Ch. 10 into one where compositional linearizability replaces the underlying notion of refinement.

Chapter 12

Concurrent Games

We define our concurrent game model in terms of the sequential game model defined in §5. To do so, we assume as a parameter a countable set Υ of agent names $\alpha \in \Upsilon$. All the notions we define for concurrent games specialize to the sequential model when we take Υ to be a singleton set. Because of this, we will take the freedom to overload much of the notation across the two models, and, moreover, any new notions we define for concurrent plays, games, and strategies will be assumed to be defined as well for sequential plays, games, and strategies by specializing as described above.

Definition 12.0.1. Let M be a move set. We define $M^\Upsilon := \sum_{\alpha \in \Upsilon} M$ and define $\lambda^\Upsilon : M^\Upsilon \rightarrow \{\text{OQ}, \text{OA}, \text{PQ}, \text{PA}, \bullet, \circ\}$ by the formula $\lambda^\Upsilon(\alpha:m) = \lambda(m)$.

A concurrent move set lifts a sequential move set by adding an agent name α to every move m of M , so that a move $\alpha:m \in M^\Upsilon$ consists of the agent $\alpha \in \Upsilon$ that issues it together with the move $m \in M$ issued. When $\Upsilon = \{\alpha\}$ we may write m where we should have written $\alpha:m$.

Example 12.0.1. Recall the move set $M_{\dagger E}$. A move of $M_{\dagger E}^{\circ \bullet \Upsilon}$, for each $\alpha \in \Upsilon$, has events of the form $\alpha:e(a)$ for an effect $e \in E$ and parameter $a \in \text{par}(a)$, or $\alpha:v$ where $v \in \text{ar}(e)$ for some $e \in E$, or one of the special tokens $\alpha:\bullet$, or $\alpha:\circ$.

Now, a well-formed concurrent play is simply a play that locally, i.e., for each agent, is well-formed as a sequential play. For that, we denote by $\pi_\alpha(s)$ the projection of s to only moves that are labeled by α in s .

Definition 12.0.2. We say $s \in (M^\Upsilon)^*$ is a well-formed (concurrent) play over M when it is locally well-formed for every $\alpha \in \Upsilon$, i.e.,

$$\forall \alpha \in \Upsilon. \pi_\alpha(s) \in \mathbb{SP}_M^{\leq \infty}$$

We define, as expected, $\mathbb{P}_M$ and $\mathbb{P}_M^{\text{ws}}$ as the sets of concurrent plays, respectively:

$$s \in \mathbb{P}_M \iff \forall \alpha \in \Upsilon. \pi_\alpha(s) \in \mathbb{SP}_M \qquad s \in \mathbb{P}_M^{\text{ws}} \iff \forall \alpha \in \Upsilon. \pi_\alpha(s) \in \mathbb{SP}_M^{\text{ws}}$$

We write $\text{ops}(s) \in \mathbb{P}_M^{\leq \infty}$ for the largest subsequence of $s \in \mathbb{P}_M^{\leq \infty}$ containing only moves m such that $\lambda_M^\Upsilon(m) \in \text{Pol}$.

Just as with sequential games, we define a concurrent game simply by specifying a sequential game. Concretely, a concurrent game is determined by specifying which sequential plays each individual agent is allowed to make. The well-formedness conditions on concurrent plays (Def. above) then ensure that a concurrent play is valid precisely when each agent's projection is a valid sequential play. Since every agent plays the same game, specifying the sequential plays for one agent suffices.

Definition 12.0.3. A (concurrent) game $A = (M_A, P_A)$ is a sequential game: a positive move set M_A , together with a non-empty, prefix-closed set of finite well-formed sequential plays $P_A \subseteq \mathbb{SP}_{M_A^\circ}^\bullet$. The set of concurrent plays of A is the set of self-interleavings:

$$P_A^\Phi := \|\alpha \in \Upsilon \ \iota_\alpha(P_A) \subseteq \mathbb{P}_{M_A^\circ}^\bullet$$

where $\iota_\alpha(S)$ labels every move of every play of S with the agent name α , and $s \parallel t$ denotes the set of interleavings of s and t . We say a play $s \in \mathbb{P}_{M_A^\circ}^{\leq \infty}$ is valid for the game A when $\downarrow \text{ops}(s) \subseteq P_A^\Phi$. We take the same conventions around $P_A^\bullet$, P_A^{ws} and $P_A^{\leq \infty}$ as we did in the sequential case.

Remark 4. By $s \parallel t$ we mean the operation (coinductively) defined by the rules:

$$\epsilon \parallel t := \{t\} \qquad s \parallel \epsilon := \{s\} \qquad m \cdot s \parallel n \cdot t := m \cdot (s \parallel n \cdot t) \cup n \cdot (m \cdot s \parallel t)$$

Since concurrent games are just sequential games, all the operations defined over sequential games, namely $-\otimes-$, $\mathbf{1}$, $-\multimap-$, and $\dagger-$, carry over directly and without modification to concurrent games. For instance, the concurrent game $A\otimes B$ is the sequential game $A\otimes B$. Similarly, $\mathbf{1}$ in the concurrent setting is the same $\mathbf{1}$ as in the sequential setting. There is no need to lift or transport structure, as the games themselves are the same; only the notion of play changes.

Example 12.0.2. For an effect signature E , the concurrent game $\dagger E = (M_{\dagger E}, P_{\dagger E})$ is simply the sequential game $\dagger E$ defined in Ex. 5.2.1, with the set of moves $M_{\dagger E}$ as in Ex. 12.0.1. The concurrent plays of $\dagger E$ are the self-interleavings $P_{\dagger E}^\Phi$, i.e. the interleavings of the sequential plays of $\dagger E$ across different agents. More generally, the dagger of a concurrent game A is just the sequential game $\dagger A$; no additional construction is needed.

Remark 5 (Side-conditions on concurrent plays). *One could equivalently define a concurrent game directly as a pair (M_A, Q_A) where $Q_A \subseteq \mathbb{P}_{M_A^\circ}$ is a set of concurrent plays satisfying:*

- $Q_A = \parallel_{\alpha \in \Upsilon} \pi_\alpha(Q_A)$ (the set of plays is closed under interleaving), and
- for all $\alpha, \alpha' \in \Upsilon$, $\pi_\alpha(Q_A) = \pi_{\alpha'}(Q_A)$ (every agent plays the same game).

The self-interleaving mapping $(M_A, P_A) \mapsto (M_A, P_A^\Phi)$ is a bijection between sequential games and concurrent games satisfying these side-conditions. Therefore, working directly with sequential games loses no generality.

Remark 6. *Our model of concurrent games effectively makes every agent play the same game. It is straightforward to generalize this so that each agent $\alpha \in \Upsilon$ has its own sequential game $A[\alpha]$, and then the corresponding concurrent game is the (non-uniform) parallel composition of each of these sequential games, for instance $P_A := \parallel_{\alpha \in \Upsilon} \iota_\alpha(P_{A[\alpha]})$. Such a non-uniform model creates a lot of extra indexing and friction without much benefit, as we can just assume each α has access to its own name, or to an object specifying its name. With this feature, we can then use the sequential game $\&_{\alpha \in \Upsilon} A[\alpha]$ and then enforce that strategies only allow α to open the game $A[\alpha]$.*

12.1 Concurrent Strategies

Definition 12.1.1. A *strategy* $\sigma : A \multimap B$ between the games A and B is a non-empty, prefix-closed set $\sigma \subseteq P_{A \multimap B}^\bullet$. We also define a convention: we will call a strategy $\nu : \mathbf{1} \multimap A$ a strategy for the game A , and write it simply as $\nu : A$.

We say a strategy $\sigma : A \multimap B$ *refines* a strategy $\tau : A \multimap B$, written $\sigma \sqsubseteq_S \tau$ when $\sigma \subseteq \tau$.

Example 12.1.1. There is a unique strategy over $\mathbf{1}$ without special tokens given by: $\{\epsilon\}$. We define the strategy $\nu_{\mathbf{1}} : \mathbf{1}$ as $\|\alpha \in \Upsilon \ \iota_\alpha(\downarrow \bullet)$.

Strategy composition is defined, as expected, by the motto “interaction + hiding”. Recall that $\bullet$ -tokens, which project to all components by our definition of composition, have a slightly different behavior from regular moves. Note that we will define the operation $-; -$ over plays of $P_{A \multimap B}^{\leq \infty}$ instead of just $P_{A \multimap B}^\bullet$ as we will use it later to define and analyze the composition for our liveness model.

Definition 12.1.2. For games A , B , and C , we define the set of interaction sequences:

$$M_{A \circ B \circ C} := ((M_A^\circ + M_B^\circ + M_C^\circ)^\bullet)^\Upsilon$$

$$A \circ B \circ C := \{s \in M_{A \circ B \circ C}^\star \mid s \upharpoonright_{A,B,-} \in P_{A \multimap B}^{\leq \infty} \wedge s \upharpoonright_{-,B,C} \in P_{B \multimap C}^{\leq \infty}\}$$

where we extend the projections on plays over $A \multimap B$ into projections over the interactions $- \upharpoonright_{A,B,-}$, $- \upharpoonright_{-,B,C}$, and $- \upharpoonright_{A,-,C}$ restricting the plays to the displayed components.

and the set of interactions of two plays, extended to sets of plays $S \subseteq P_{A \multimap B}^{\leq \infty}$ and $T \subseteq P_{B \multimap C}^{\leq \infty}$:

$$s \circ t := \{s' \in A \circ B \circ C \mid s' \upharpoonright_{A,B,-} = s \wedge s' \upharpoonright_{-,B,C} = t\} \quad S \circ T := \cup_{s \in S} \cup_{t \in T} s \circ t$$

As in the sequential case, we define the standard composition by:

$$s ; t := (s \circ t) \upharpoonright_{A,-,C} \quad S ; T := \cup_{s \in S} \cup_{t \in T} s ; t$$

Just like in the sequential case, it then holds that:

Proposition 12.1.1. *For any sets of plays $S \subseteq P_{A \multimap B}^{\leq \infty}$, $T \subseteq P_{B \multimap C}^{\leq \infty}$, and $R \subseteq P_{C \multimap D}^{\leq \infty}$:*

$$S ; (T ; R) = (S ; T) ; R$$

Proof. See Proof 31.1.1. □

In particular, we have that:

Proposition 12.1.2. *Let Safe be the structure described by the following data:*

Objects: *its objects are games A, B, C ;*

Homs: *as morphisms Safe(A, B) the inf-lattice of strategies $\sigma : A \multimap B$, $\tau : B \multimap C$, ordered by the refinement preorder, with infima given by arbitrary union;*

Morphism Composition: *with the categorical composition of morphisms given by $- ; -$:*

$$\mathbf{Safe}(A, B) \times \mathbf{Safe}(B, C) \rightarrow \mathbf{Safe}(A, C);$$

*Then, this structure defines a **Inf**-enriched semicategory.*

12.2 Concurrent Games and Synchronization

We have just defined a concurrent game semantics modeling potentially blocking locally sequential computation, and we noted that we obtain a semicategorical structure. In this section, we discuss the issue with neutral elements (§12.2.1) and present a solution by constructing from the semicategory Safe a category K_{copy} Safe of concurrent games (§12.2.2), presented abstractly, and discuss some infrastructure around it (§10.1, §12.2.3). We finalize by adapting a result of Ghica and Murawski [2004] which allows us to give a concrete characterization of this category (§12.3).

12.2.1 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 us first ground the discussion on sequential computation. As we saw in Ch. 6, the neutral element in **Safe**^{seq} is the copycat

strategy seqcopy_- . 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 $\text{seqcopy}_{\text{Unit}} : \text{Unit} \multimap \text{Unit}$ there is only one possible interaction without starvation tokens (displayed on the left):

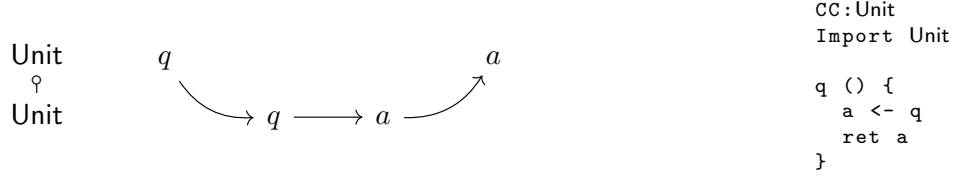


Figure 12.1: Maximal play of $\text{seqcopy}_{\text{Unit}}$ (left) and corresponding pseudocode (right)

All other plays of $\text{seqcopy}_{\text{Unit}}$ are prefixes of this play. This strategy corresponds to the implementation displayed on the right of Fig. 12.1, for the method q using a library that already implements the method q . Note that in the figures from now on, the arrows indicate the flow of each thread of computation (in other words, the events of each agent).

Suppose we compose the copycat strategy with itself, that is, we build the strategy $\text{seqcopy}_{\text{Unit}} ; \text{seqcopy}_{\text{Unit}}$, and recall the motto “interaction + hiding”. The resulting interaction prior to hiding is:

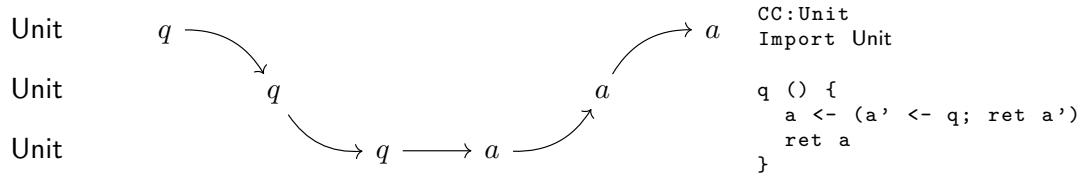
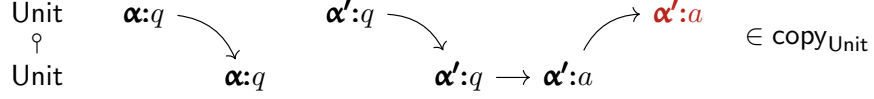


Figure 12.2: Maximal play of $\text{int}(\text{seqcopy}_{\text{Unit}}, \text{seqcopy}_{\text{Unit}})$ (left) and corresponding pseudocode (right).

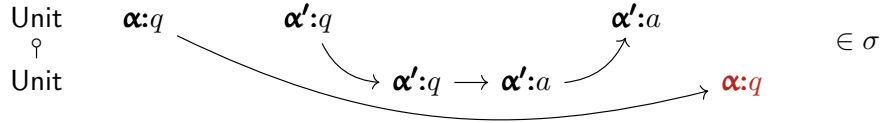
The middle row of the interaction is the one that is then hidden. It simultaneously plays the role of the source of the play in the top two rows, and the target in the play in the bottom two rows. The resulting interaction, after hiding, is the interaction from Fig. 12.1, as expected. In terms of the corresponding implementations, composing the two strategies amounts to inlining the code of one into the other, as depicted in the right of Fig. 12.2.

In the concurrent version $\text{Unit} \in \underline{\text{Safe}}$ of Unit , each agent of Υ locally plays Unit . The

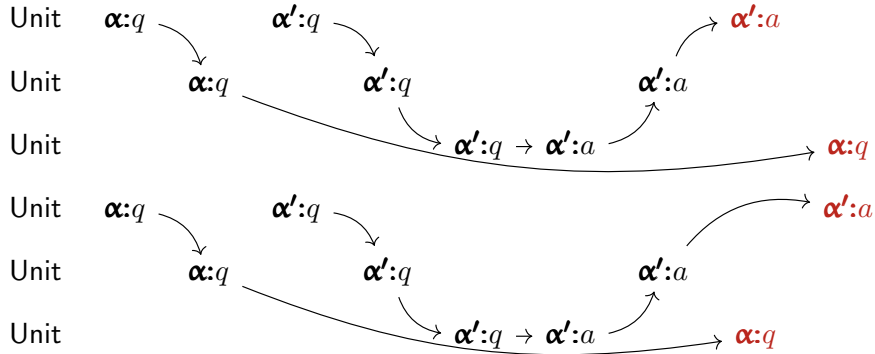
obvious neutral element in this situation would be to have each agent $\alpha \in \Upsilon$ locally run $\text{seqcopy}_{\text{Unit}}$, a strategy we call $\text{copy}_{\text{Unit}} : \text{Unit} \multimap \text{Unit}$. $\text{copy}_{\text{Unit}}$ therefore comprises all plays which are interleavings of $\text{seqcopy}_{\text{Unit}}$. One such play is the play t displayed below:



Now, consider a strategy $\sigma : \text{Unit} \multimap \text{Unit}$ 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 $\sigma; \text{copy}_{\text{Unit}}$:



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

$$\alpha:q \cdot \alpha':q \cdot \alpha':q \cdot \alpha':a \cdot \alpha':a \cdot \alpha:q, \quad \alpha:q \cdot \alpha':q \cdot \alpha':q \cdot \alpha':a \cdot \alpha:q \cdot \alpha':a \in \sigma; \text{copy}_{\text{Unit}}$$

This is despite the fact that the second play is not in σ . Therefore, $\text{copy}_{\text{Unit}}$ 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 Laird [2001] and Ghica and Murawski [2004] faced and is a common issue in compositional models of concurrent computation.

12.2.2 Saturated Strategies

We start by formally defining the concurrent copycat strategy **copy**:

Definition 12.2.1. The concurrent copycat strategy $\text{copy}_A : A \multimap A$ is defined as the self-interleaving of the sequential copycat strategy $\text{copy}_A : A \multimap A$:

$$\text{copy}_A := \text{copy}_A^\Phi$$

or, equivalently:

$$\text{copy}_A := \{s \in P_{A \multimap A} \mid \forall \alpha \in \Upsilon. \forall p \sqsubseteq_{\text{even}} \pi_\alpha(s). p \upharpoonright_{A,-} = p \upharpoonright_{-,A}\}^\bullet$$

While the copycat strategy is not the identity for all strategies, it, crucially, is the identity for itself:

Proposition 12.2.1. copy_A is idempotent, i.e. $\forall A. \text{copy}_A ; \text{copy}_A = \text{copy}_A$.

Proof. See Proof 31.2.2. □

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 copy_- does act as the neutral element for strategy composition, as we will treat in detail in §14.1. This construction allows us to construct a category **Safe** that specializes **Safe** to strategies that are well-behaved upon composition with the family of idempotents copy_- . Concretely, **Safe** is defined as follows:

Definition 12.2.2. The category **Safe** has as objects concurrent games A, B and as morphisms strategies $\sigma : A \multimap B \in \underline{\text{Safe}}$ saturated in that

$$\text{copy}_A ; \sigma ; \text{copy}_B = \sigma$$

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

It is straightforward to check that saturated strategies compose, as the copycat is idempotent, and that the copycat is the neutral element for saturated strategies, by definition of saturation. We will not do this now, as we will later prove this generically.

12.2.3 The Semifunctors K_{copy} and Emb

The abstract treatment in §14.1 will also show that the abstract construction giving rise to **Safe** comes with some infrastructure around it for free. For instance, it readily gives a forgetful semifunctor from **Safe** (seen here as a semicategory by forgetting the fact it has neutral elements) to **Safe**

$$\text{Emb} : \mathbf{Safe} \longrightarrow \underline{\mathbf{Safe}}$$

acting as the identity semifunctor. We will omit applications of Emb 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

$$K_{\text{copy}} : \underline{\mathbf{Safe}} \rightarrow \mathbf{Safe}$$

as defined in §14.1, and explicitly given by:

$$A \xrightarrow{K_{\text{copy}}} A \qquad \sigma : A \multimap B \xrightarrow{K_{\text{copy}}} \text{copy}_A; \sigma; \text{copy}_B$$

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

Proposition 12.2.2. *For any $\sigma : A \multimap B$ and $\tau : B \multimap C$:*

$$K_{\text{copy}}(\sigma; \tau) \sqsupseteq_S K_{\text{copy}}(\sigma); K_{\text{copy}}(\tau)$$

Proof. See Proof 31.5.1. □

It is straightforward to check that K_{copy} is cocontinuous, that is, it is monotonic and meet-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 §14.1.

12.3 Fine-Grained Synchronization in Concurrent Games

In §12.2.2, we gave a rather abstract definition for the strategies in **Safe**. Ghica [2023], in a slightly different setting, observed that this abstract definition is equivalent to a concrete one, originally appearing in Ghica and Murawski [2004], involving a rewrite system which we now adapt to our setting.

Definition 12.3.1. For a game A , we define the rewrite system $- \rightsquigarrow_A -$ over finite plays of $P_A^{\leq \infty}$ with local rewrite rules of the form:

$$s \cdot \alpha : m \cdot \alpha' : m' \cdot t \rightsquigarrow_A s \cdot \alpha' : m' \cdot \alpha : m \cdot t$$

when $\alpha \neq \alpha'$ and one of the following holds:

$$\begin{array}{ll} [\text{OO/PP}] : \lambda_A(m) \in \{\text{O}, \text{P}\} \wedge \lambda_A(m) = \lambda_A(m') & [\text{OP}] : \lambda_A(m) = \text{O} \wedge \lambda_A(m') = \text{P} \\ [\text{O}\bullet] : \lambda_A(m) = \text{O} \wedge \lambda_A(m') = \bullet & [\bullet\text{P}] : \lambda_A(m) = \bullet \wedge \lambda_A(m') = \text{P} \end{array}$$

as well as the local rewrite rules below, allowing for dropping or completing pending operations:

$$[(\text{P})] : \lambda_A(m) = \text{P} \implies s \cdot \epsilon \cdot t \rightsquigarrow_A s \cdot \alpha : m \cdot t \quad [(\text{O})] : \lambda_A(m) = \text{O} \implies s \cdot \alpha : m \cdot t \rightsquigarrow_A s \cdot \epsilon \cdot t$$

and say a play s over A is *safely linearizable* w.r.t. to a play t over A when $s \rightsquigarrow_A t$. When A is implied, we omit it in the rewrite relation, writing only $\rightsquigarrow$. We extend this to safety specifications in the usual way by saying that $\nu' : A$ is *safely linearizable* w.r.t. to $\nu : A$, when $\forall s \in \nu'. \exists t \in \nu. s \rightsquigarrow t$.

We write $- \rightsquigarrow -$ for the rewrite relation which includes only the first four rules (OO/PP, OP, O $\bullet$, $\bullet$ P).

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

Proposition 12.3.1. *A strategy $\sigma : A \multimap B$ is saturated if and only if it is:*

O-receptive $\forall s \in \sigma. \forall m \in M_{A \multimap B}^\Upsilon. \lambda^\Upsilon(m) = O \wedge s \cdot m \cdot t \in P_{A \multimap B} \implies s \cdot m \cdot t \in \sigma$

P-delaying $\forall s \in \sigma. \forall m \in M_{A \multimap B}^\Upsilon. \lambda^\Upsilon(m) = P \wedge s \cdot m \cdot t \in \sigma \implies s \cdot t \in \sigma$

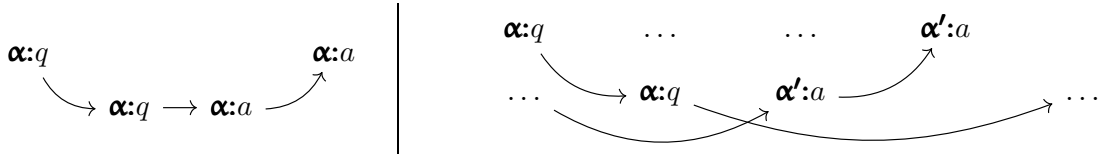
$\rightsquigarrow$ -closed $\forall s \in \sigma. \forall s' \in P_{A \multimap B}. s' \rightsquigarrow_{A \multimap B} s \implies s' \in \sigma$

The key lemma to show this alternative characterization is the synchronization lemma, as coined by Ghica [2023]. 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.

Proposition 12.3.2 (Synchronization Lemma). *Let $s = p \cdot \alpha : m \cdot \alpha' : m' \cdot p' \in P_{A \multimap B}^{\leq \infty}$. Then, s' is a witness that $p \cdot \alpha' : m' \cdot \alpha : m \cdot p' \in K_{\text{copy}} s$ with $s' \upharpoonright_{-,A,B,-} = s$ if and only if $\alpha \neq \alpha'$ and $\alpha' : m' \cdot \alpha : m \rightsquigarrow_A \alpha : m \cdot \alpha' : m'$.*

Proof. See Proof 31.6.1. □

The core of the proof of Prop. 12.3.2 lies in the dynamics of copy_- . If we focus on an agent $\alpha \in \Upsilon$, a typical play in copy_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 $s \in P_B$ such that $s = p \cdot \alpha : q \cdot \alpha' : a \cdot p'$ any of its interactions with copy_B , such as in $K_{\text{copy}} s; \text{copy}_B$, look something like the play displayed above on the right.

After hiding the interaction in the source, the resulting play can at most make $\alpha : q$ appear earlier and $\alpha' : a$ 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. 12.3.2 is a case analysis of the polarities of $\alpha : m$ and $\alpha' : m'$.

Revisiting the example in §12.2, if strategies were required to be saturated under the rewrite system $\rightarrow \rightsquigarrow$ (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.

Chapter 13

Safe Linearizability

In this section, we argue that linearizability emerges from the Karoubi construction used to define **Safe** and establish several of the main results of this paper. In §13.1, we establish that K_{copy} — leads to a general notion of linearizability which is improved in §13.2, while in §13.3 we observe that plays of copy_- correspond to proofs of linearizability. In §13.4 we show a property analogous to the usual observational refinement property, and in §13.5 we show the locality property.

13.1 Linearizability

We start by defining linearizability.

Definition 13.1.1 (Safe Linearizability). We say a play $s \in P_A^\bullet$ is *safely linearizable* w.r.t. $t \in P_A^\bullet$ when $s \rightsquigarrow_A t$.

Similarly, we say a strategy $\nu' : A$ is *safely linearizable* w.r.t. to a strategy $\nu : A$ when $\nu' \rightsquigarrow_A \nu$, that is, when for any $s \in \nu'$ there is $t \in \nu$ such that $s \rightsquigarrow_A t$.

This definition may seem quite different from the original definition of linearizability, but it is not too far from it. That linearizability could be defined by using a rewrite system instead of partial order containment had already been noticed [Filipovic et al., 2010; Goubault et al., 2018]. Moreover, when s does not contain $\bullet$ -tokens, the definition reduces to:

$s \in P_A$ is linearizable to a play $t \in P_A$ if there exists a sequence of Opponent moves $s_O \in (M_A^O)^*$ and a sequence of Proponent moves $s_P \in (M_A^P)^*$ such that

$$s \cdot s_P \rightsquigarrow_A t \cdot s_O$$

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 $\rightsquigarrow_A$ plays the role of preservation of happens-before order. In this locally sequential formulation of concurrent games, this generalized definition of linearizability is closely related to interval-sequential linearizability [Castañeda et al., 2015], which we address in more detail in §15.2. When the linearized strategy is specialized to atomic strategies only, we obtain Herlihy-Wing linearizability. In §15.1, we give a thorough account of the specialization to atomic games.

It is then possible to show that $K_{\text{copy}} -$ may be defined in terms of linearizability:

Proposition 13.1.1. *For any $\nu : A \in \mathbf{Safe}$*

$$K_{\text{copy}} \nu = \{s \in P_A \mid s \text{ is linearizable with respect to } \nu\}$$

A lot of this proposition is accounted for by Prop. 12.3.1. Observe that O -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 P -delaying and from prefix-closure. We also find it important to remind the reader that K_{copy} is defined in terms of its role in the relationship between a semicategory and its Karoubi envelope, as will be treated in detail in §14.1. In this way, Prop. 13.1.1 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. 13.1.1 is an alternative definition of linearizability.

Corollary 13.1.1 (compositional linearizability). *A strategy $\nu' : A \in \mathbf{Safe}$ is linearizable*

to a strategy $\nu : A$ if and only if

$$\nu' \sqsubseteq_S K_{\text{copy}} \nu$$

As K_{copy} appears 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 $- \rightsquigarrow -$. Moreover, it is defined entirely based on three key ingredients: composition (as internalized by the definition of $K_{\text{copy}} -$), the choice of the idempotent **copy**, and refinement $- \sqsubseteq_S -$. This will be central to what is perhaps the most important result of this whole dissertation, developed in §14.1.

As a 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. 13.1.1 and developed in detail in §14.1 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 §13.4 and locality in §13.5 than was previously known.

13.2 Unary Linearizability

In the literature and informally it is very common to see statements such as “X implementation of a stack is linearizable” or “X is not linearizable”. A careful reader should be skeptical about such statements: X implementation is (not) linearizable with respect to what? There is a fundamental problem with such assertions: they do not qualify over what it is linearizable with respect to. This is a fault of even the original work on linearizability [Herlihy and Wing, 1990]. At first, this might seem like a minor issue; unfortunately, it is not. A classical result by Szpilrajn [1930] shows that any partial order may be extended into a total order. In the context of classical linearizability, it means that any trace s is linearizable with respect to *some* atomic trace t . So given a set of traces of a concrete implementation ν' , we may just appeal to this result to obtain a set ℓ of traces containing a linearization to each $s \in \nu'$. Taking its prefix-closure $\nu = \downarrow \ell$ then makes it into a strategy. So every implementation admits some linearization, making such a unary statement of

linearizability the trivial property!

Moreover, the computed set of linearizations might not be very useful. The classical example of an object that “is not linearizable” is an exchanger. We can model an exchanger object which exchanges natural numbers by the following signature:

$$\text{Exch} := \{\text{exch} : \mathbb{N} \rightarrow \mathbb{N}\}$$

Intuitively, `exch` allows two agents to synchronously exchange a value, so that the following is a prototypical trace of an exchanger:

$$\alpha:\text{exch}(n) \cdot \alpha':\text{exch}(n') \cdot \alpha:n' \cdot \alpha':n$$

where we see that α and α' receive as returns the values that each other passed as argument. A more complex trace of the exchanger is:

$$s = \alpha_1:\text{exch}(n_1) \cdot \alpha_2:\text{exch}(n_2) \cdot \alpha_3:\text{exch}(n_3) \cdot \alpha_3:n_2 \cdot \alpha_4:\text{exch}(n_4) \cdot \alpha_1:n_4$$

Note that in s agents α_1 and α_4 have already committed to exchange values with each other, and so have α_2 and α_3 (even though α_2 and α_4 have not returned yet). We refrain from giving a formal specification ν_{Exch} for conciseness, as the intuition will suffice for our argument here. Now, observe that s is linearizable (in fact, Herlihy-Wing linearizable) to:

$$\alpha_1:\text{exch}(n_1) \cdot \alpha_1:n_4 \cdot \alpha_3:\text{exch}(n_3) \cdot \alpha_3:n_2$$

This is not a very useful trace, as the values n_4 and n_2 appear out-of-thin-air.

We might then give up, and just accept that linearizability is fundamentally a binary property: one must provide both the concrete set of behaviors ν' and the specification ν to state some linearizability statement. But it is possible in at least two ways to make linearizability into a unary property that is both non-trivial and intuitive.

For the first one, our alternative and abstract characterization suggests the following variation of linearizability:

Definition 13.2.1. We say $\sigma : A \in \mathbf{Safe}$ is self-linearizable w.r.t. $\tau : A$ when σ is linearizable with respect to τ and $\tau \sqsubseteq_S \sigma$.

This obviously is stronger than the notion of linearizability in Def. 13.1.1. As the restriction of that notion of linearizability to atomic plays implies linearizability, it immediately follows that atomic self-linearizability implies Herlihy-Wing linearizability. Note that when σ is self-linearizable with respect to τ we obtain that:

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

Together with Corollary 13.1.1 it follows that $\sigma = K_{\text{copy}} \tau$ so that σ is fully characterized by its linearization. Therefore, a self-linearizable specification σ is a strategy which is in the image of K_{copy} . Our abstract treatment in §14.1 allows σ to be seen as an element for which a certain weak adjunction derived from K_{copy} and $\mathbf{Emb}$ behaves as a strict adjunction, which is how the definition arises.

Concretely, self-linearizability matches the intuitive understanding of usual 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 not self-linearizable to a specification, it means that the specification is not accurate: it is an over-approximation. This is what happens in our example with the exchanger above. No trace of a concrete exchanger implementation is atomic. So standard linearizability does not rule out such bad specifications, while self-linearizability does. Our formalism shows exactly in which sense non self-linearizable objects yield an *over-approximation*: If σ is self-linearizable to τ then $\sigma = K_{\text{copy}} \tau$, as we showed above. Meanwhile, when σ is linearizable to τ but not self-linearizable, we have a strict containment $\sigma \subset K_{\text{copy}} \tau$.

A critic to this argument may say that instead, when one says that an object is “not linearizable” they mean that for a specific atomic specification. Note though that no atomic specification for the exchanger object makes sense, as its behaviors are, at least intuitively,

intrinsically concurrent. But again, self-linearizability makes this precise: it requires that the linearized specification be a “sub-object” of the concurrent object, in that all the linearized behaviors were already possible concretely.

Self-linearizability, therefore, clarifies in which sense objects are “not linearizable”. The exchanger object is not Herlihy-Wing linearizable because no atomic strategy $\nu_{\text{Exch}}^{\text{atomic}}$ of type $\dagger\text{Exch}$ satisfying $\nu'_{\text{Exch}} \subseteq K_{\text{copy}} \nu_{\text{Exch}}^{\text{atomic}}$ will satisfy $\nu_{\text{Exch}}^{\text{atomic}} \subseteq \nu'_{\text{Exch}}$. It is important to stress that as our framework subsumes set-linearizability, it is possible to characterize ν'_{Exch} as self-linearizable to a less concurrent (in fact, set-sequential) strategy. We will see later on that self-linearizability also helps clarify an apparent fault with the usual refinement theorem around linearizability (§13.4).

Now, if we know our object is supposed to be atomically linearizable, then we may take “X implementation is linearizable” to mean that there exists an atomic specification such that the implementation is self-linearizable to. Or, formally:

Definition 13.2.2. A strategy $\nu' : A$ is atomically linearizable when there exists an *atomic* strategy $\nu : A$ such that ν' is self-linearizable w.r.t. to ν .

The existential then makes the property into a unary property, which is moreover non-trivial as it rejects the exchanger. By our remarks above, it is actually equivalent to the statement that $\nu' : A$ is linearizable w.r.t. $U_{\text{Atom}} \nu'$, where $U_{\text{Atom}} \nu'$ is the restriction of ν' to its atomic plays. Again, the property is unary because $U_{\text{Atom}} \nu'$ is determined by ν' . This provides an equivalent characterization of atomic self-linearizability.

Now, the second way to make linearizability into a unary property is to require that

Definition 13.2.3. A strategy $\nu' : A$ is deterministically linearizable when there exists a deterministic strategy $\nu : A$ such that $\nu' \rightsquigarrow_A \nu$.

We avoid defining determinism in general until Ch. 27, as the definition is more challenging for non-atomic specifications. We note that no atomic specification for an exchanger will be deterministic, essentially because by the argument above it must allow for any trace:

$$\alpha:\text{exch}(v) \cdot \alpha:v'$$

no matter the v' , as any value can be given out-of-thin-air in such a specification. The exchanger is nonetheless deterministically linearizable because its set-linearizable specification is deterministic. While this results in a non-trivial unary linearizability property that is intuitive, we do not study in as much depth as self-linearizability, since for the sake of verification it is quite useful to study objects that do not admit any deterministic linearized specification.

13.3 Computational Interpretation of Linearizability

We just saw that linearizability can be characterized by the transformation $K_{\text{copy}} -$. We now offer yet another perspective on linearizability by providing a computational interpretation of linearizability proofs. Recall that in our discussion in §12.2.1, we observed that copy_- is the denotation of a concrete program. Interestingly, the plays of copy_- correspond to proofs of linearizability, which gives some explanation to the surprising coincidence that composition with the copy_- somehow generates the set of traces linearizable w.r.t. some strategy.

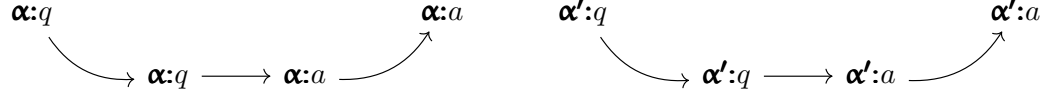
Proposition 13.3.1. *$s \in P_A$ linearizes to $t \in P_A$ if and only if there is a play $s' \in \text{copy}_A$ such that*

$$s' \upharpoonright_{-,A} = s \quad \quad s' \upharpoonright_{A,-} = t$$

Proof. For this, one first proves that every play $s \in \text{copy}_A$ whose projections to the target and source components are locally sequential with respect to each other (their projection to each agent is the same) satisfies $s \upharpoonright_{-,A} \rightsquigarrow_A s \upharpoonright_{A,-}$. Then, prefix-closure and receptivity of copy_A allow for linearizability to be used instead of $- \rightsquigarrow_- -$. See Proof 31.7.1 for a complete argument. $\square$

What Prop. 13.3.1 essentially establishes is that proofs of linearizability encode executions of the code in Fig. 11.3, and that executions of the code in Fig. 11.3 encode proofs of linearizability. Intuitively, the reason for this is that in a play of copy_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 $\text{copy}_{\text{Unit}}$:



13.4 Observational 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 $\sigma : A \multimap B$. Now, the game A appears in a negative position in the type $A \multimap B$. Because of this, there is a contravariant effect to linearizability on $- \multimap -$ in that if $s \rightsquigarrow_{A \multimap B} t$ then, while $s|_{-,B}$ is “more concurrent” than $t|_{-,B}$, $s|_{A,-}$ is “less concurrent” than $t|_{A,-}$. So while a saturated strategy must be able to produce any “more concurrent” play in B it must be able to interact with any “less concurrent” play of A . This intuition leads to the following result, analogous to the observational refinement equivalence of Filipovic et al. [2010].

Proposition 13.4.1 (Observational Refinement). *$\nu'_A : A \in \mathbf{Safe}$ is linearizable to $\nu_A : A \in \mathbf{Safe}$ if and only if for all concurrent games B and $\sigma : A \multimap B \in \mathbf{Safe}$ it holds that*

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

Proof. By Corollary 13.1.1, monotonicity of composition, and saturation of σ :

$$\nu'_A; \sigma \subseteq K_{\text{copy}} \nu_A; \sigma = (\text{copy}_1; \nu_A; \text{copy}_A); \sigma = (\text{copy}_1; \nu_A); (\text{copy}_A; \sigma) = \nu_A; \sigma$$

For the reverse direction, it can be shown that $\nu \subseteq \nu; \text{copy}_A$ for any $\nu : A \in \mathbf{Safe}$. Then, instantiating the hypothesis at $\sigma = \text{copy}_A$:

$$\nu'_A \subseteq \nu'_A; \text{copy}_A \subseteq \nu_A; \text{copy}_A = \text{copy}_1; \nu_A; \text{copy}_A = K_{\text{copy}} \nu_A$$

□

An interesting remark at this point is that the direction of the refinement in Prop. 13.4.1 is not intuitive. Usually, one would believe that having a client interact with less concurrent traces should lead to fewer behaviors, not more, as Prop. 13.4.1 leads one to believe by the direction of the refinement. Note that this artifact is already present in the original result by Filipovic et al. [2010]. Again, our treatment clarifies the source for this. As we have discussed in §13.2, linearizability provides only an over-approximation. This means that by substituting ν'_A by ν_A one may introduce behaviors that are not included in ν'_A (consider replacing a concurrent specification such as ν'_{Exch} , which has no atomic traces, by one with only atomic traces as per our argument in §13.2). This discussion suggests that the more precise criterion given by self-linearizability should guarantee a stronger result. Indeed, the following is a straightforward corollary of Prop. 13.4.1:

Corollary 13.4.1. $\nu'_A : A \in \mathbf{Safe}$ is self-linearizable w.r.t. to $\nu_A : A \in \underline{\mathbf{Safe}}$ if and only if for all B and $\sigma : A \multimap B \in \mathbf{Safe}$:

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

13.5 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 tensor $- \otimes -$. For this, we start with a *faux* definition of tensor of strategies.

Definition 13.5.1. Given sets of traces $S \subseteq P_{A \multimap A'}^{\leq \infty}$ and $T \subseteq P_{B \multimap B'}^{\leq \infty}$, we define their horizontal composition $S \otimes T \subseteq P_{A \otimes B \multimap A' \otimes B'}^{\leq \infty}$ by $S \otimes T := \{s \in P_{A \otimes B \multimap A' \otimes B'}^{\leq \infty} \mid s \upharpoonright_{A \multimap A'}, - \in S \wedge s \upharpoonright_{-, B \multimap B'} \in T\}$

Given strategies $\sigma_A : A$ and $\sigma_B : B$ we define the strategy $\sigma_A \otimes \sigma_B : A \otimes B$ as the set $\sigma_A \otimes \sigma_B$, the set of locally sequential interleavings of σ_A and σ_B .

Note that:

$$\sigma_A \otimes \sigma_B = (\sigma \parallel \tau) \cap P_{A \otimes B}$$

We call this a *faux* tensor 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 does define a bi-semifunctor in **Safe**, which becomes a proper tensor when specialized to **Safe**.

Proposition 13.5.1. *(Safe, $- \otimes -, 1$) assembles into a symmetric monoidal category.*

Proof. See Proof 31.4.3. □

This structure is obtained by mapping the corresponding structural maps in **Seq_{det}** through the interleaving functor $-\Phi$. In particular, Prop. 13.5.1 says that $- \otimes -$ is a bifunctor in **Safe**, so that

Proposition 13.5.2. *For all concurrent games A, B :*

$$\text{copy}_{A \otimes B} = \text{copy}_A \otimes \text{copy}_B$$

Proof. See Proof 31.4.1. □

This rather simple result is auspicious given the computational interpretation of copy_- in terms of linearizability proofs seen in §13.3. Indeed, if copy plays are in bijection with linearizability proofs, then this statement tells us that proofs of linearizability in $A \otimes B$ are in bijection with proofs of linearizability in A and B *separately*. This property, together with the fact that $- \otimes -$ is a bi-semifunctor, readily implies that K_{copy} distributes over the tensor.

Proposition 13.5.3. *Let $\sigma_A : A \multimap A'$ and $\sigma_B : B \multimap B'$. Then:*

$$K_{\text{copy}} (\sigma_A \otimes \sigma_B) = K_{\text{copy}} \sigma_A \otimes K_{\text{copy}} \sigma_B$$

Proof.

$$\begin{aligned} K_{\text{copy}} (\sigma_A \otimes \sigma_B) &= \text{copy}_{A \otimes B}; (\sigma_A \otimes \sigma_B); \text{copy}_{A' \otimes B'} && \text{(Def.)} \\ &= (\text{copy}_A \otimes \text{copy}_B); (\sigma_A \otimes \sigma_B); (\text{copy}_{A'} \otimes \text{copy}_{B'}) && \text{(Prop. 13.5.2)} \\ &= (\text{copy}_A; \sigma_A; \text{copy}_{A'}) \otimes (\text{copy}_B; \sigma_B; \text{copy}_{B'}) && \text{(bi-semifunctoriality of } - \otimes -) \\ &= K_{\text{copy}} \sigma_A \otimes K_{\text{copy}} \sigma_B && \text{(Def.)} \end{aligned}$$

□

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

Corollary 13.5.1 (Locality). *Let $\nu'_A : A$, $\nu'_B : B \in \mathbf{Safe}$ and $\nu_A : A$, $\nu_B : B \in \underline{\mathbf{Safe}}$. Then*

$$\nu' = \nu'_A \otimes \nu'_B \text{ is linearizable w.r.t. } \nu = \nu_A \otimes \nu_B$$

if and only if

$$\nu'_A \text{ is linearizable w.r.t. } \nu_A \text{ and } \nu'_B \text{ is linearizable w.r.t. } \nu_B$$

Proof. By Prop. 13.5.3 and Prop. 13.1.1

$$\nu' = \nu'_A \otimes \nu'_B \subseteq K_{\text{copy}} (\nu_A \otimes \nu_B) = K_{\text{copy}} \nu_A \otimes K_{\text{copy}} \nu_B$$

in particular,

$$\nu'_A = (\nu'_A \otimes \nu'_B) \upharpoonright_A \subseteq (K_{\text{copy}} \nu_A \otimes K_{\text{copy}} \nu_B) \upharpoonright_A = K_{\text{copy}} \nu_A$$

$$\nu'_B = (\nu'_A \otimes \nu'_B) \upharpoonright_B \subseteq (K_{\text{copy}} \nu_A \otimes K_{\text{copy}} \nu_B) \upharpoonright_B = K_{\text{copy}} \nu_B$$

For the reverse direction, we have:

$$\nu' = \nu'_A \otimes \nu'_B \subseteq K_{\text{copy}} \nu_A \otimes K_{\text{copy}} \nu_B = K_{\text{copy}} (\nu_A \otimes \nu_B)$$

□

We would like to observe that not only does our methodology yield a stronger result in Prop. 13.5.3 and 13.5.1, 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. Our result is also stronger in another way. The usual statement of locality relies on a projection: one assumes an object with many independent sub-objects and says that this large object is linearizable if and only if the sub-objects are as well. Our treatment instead relies on a pre-defined operation for composition objects together into larger objects (the tensor) and states the locality theorem in terms of this operation. This biases the statement toward composing objects together rather than decomposing

them. The benefits of this become evident when one notes that, because we show that the tensor makes our model into an enriched symmetric monoidal category, our locality theorem smoothly interacts with vertical composition and refinement, essentially extending the symmetric monoidal structure of the model to linearizable objects.

Chapter 14

Compositional Linearizability

In this chapter, we exploit the ideas and discoveries we developed as part of Ch. 13 to provide a fully generic account of linearizability in any **Inf**-enriched semicategory. This provides both an explanation of why linearizability is the *right* correctness criterion and the ability to derive linearizability-like criteria in new models.

14.1 Karoubi Envelope

14.1.1 The Karoubi Envelope

Typically, given a semicategory $\mathbf{C}$, we can construct its Karoubi envelope as the category $\text{Kar } \mathbf{C}$ which has as objects pairs:

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

of an object C and an idempotent e of C . Recall that an idempotent of an object is simply an idempotent endomorphism, in the sense that:

$$e \circ e = e$$

A morphism

$$f : (C, e) \rightarrow (C', e')$$

in $\mathbf{Kar} \mathbf{C}$ is a morphism $f : C \rightarrow C'$ 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 $\mathbf{Kar} \mathbf{C}$ 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

$$\mathbf{Semi} : \mathbf{Cat} \rightarrow \mathbf{SemiCat}$$

which given a category $\mathbf{C}$ assigns a semicategory $\mathbf{Semi} \mathbf{C}$ 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, $\mathbf{Semi}$ – admits a right adjoint

$$\mathbf{Kar} : \mathbf{SemiCat} \rightarrow \mathbf{Cat}$$

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

$$F : \mathbf{C} \rightarrow \mathbf{D} \xrightarrow{\mathbf{Kar}} \mathbf{Kar} F : \mathbf{Kar} \mathbf{C} \rightarrow \mathbf{Kar} \mathbf{D}$$

is defined by:

$$\begin{aligned} (C, e) &\xrightarrow{\mathbf{Kar} F} (F C, F e) \\ f : (C, e_C) \rightarrow (C', e') &\xrightarrow{\mathbf{Kar} F} F f : (F C, F e) \rightarrow (F C', F e') \end{aligned}$$

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:

$$\mathbf{SemiKar} : \mathbf{SemiCat} \rightarrow \mathbf{SemiCat}$$

Note that this comonad assigns to a semicategory $\mathbf{C}$ the semicategory $\mathbf{SemiKar} \mathbf{C}$, 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:

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

which immediately makes any morphism $f : C \rightarrow C'$ into a morphism $f : (C, \mathbf{id}_C) \rightarrow (C', \mathbf{id}_{C'})$ due to the unital laws. Note that this functor corresponds to selecting a family $(e_C : C \rightarrow C)_{C \in \mathbf{C}}$ 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 $f : C \rightarrow D$. 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 $f : C \rightarrow D$ and $g : D \rightarrow E$ 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 $(e_C : C \rightarrow C)_{C \in \mathbf{C}}$ there is no canonical such semifunctor. Despite that, there is always a forgetful semifunctor:

$$\mathbf{Emb} : \mathbf{SemiKar} \mathbf{C} \rightarrow \mathbf{C}$$

given by the mapping

$$(C, e) \xrightarrow{\text{Emb}} C \quad f : (C, e) \rightarrow (D, e') \xrightarrow{\text{Emb}} f : C \rightarrow D$$

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

Let $\underline{\mathbf{C}}$ be a semicategory enriched over $\mathbf{Inf}$ (in the sense of Moens et al. [2002] as explained in Ch. 7). Then, recall that we have a semifunctor:

$$\text{Emb} : \text{SemiK } \underline{\mathbf{C}} \rightarrow \underline{\mathbf{C}}$$

We can then ask for there to be a right inverse S :

$$\text{Emb} \circ S = \text{id}_{\underline{\mathbf{C}}}$$

where $\text{id}_{\underline{\mathbf{C}}}$ is the identity semifunctor. This is a *section* in the category of semicategories.

It turns out this is generally not possible, but still instructive. The reason for this was already covered, although not in this language. Essentially, such a section consists of an assignment of an idempotent $e_C : C \rightarrow C$ to each object $C \in \underline{\mathbf{C}}$, as well as a semifunctorial action mapping a morphism $f : C \rightarrow D$ to a saturated morphism $S f : (C, e_C) \rightarrow (D, e_D)$. Generally, it is not possible to find such a mapping, because the semifunctoriality is hard to achieve, as discussed at the end of the previous section.

However, if we focus on the section only along the objects, which we represent by a

family of idempotents:

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

Any such family defines a full subcategory $\mathbf{C}_e$ of the Karoubi envelope $\text{Kar } \mathbf{C}$ of $\mathbf{C}$, obtained by restricting the objects to precisely the idempotents in e_- . We call such a subcategory of $\text{Kar } \mathbf{C}$ a *section* of the Karoubi envelope of $\mathbf{C}$.

It is immediate that for any section $\mathbf{C}_e$, the restriction

$$\text{Emb}_e : \text{Semi } \mathbf{C}_e \rightarrow \mathbf{C}$$

of the forgetful functor Emb defines an embedding. There is also a candidate for a semi-functor in the reverse direction:

$$K_e : \mathbf{C} \rightarrow \text{Semi } \mathbf{C}_e$$

given by

$$C \mapsto^{K_e} (C, e_C) \quad f : C \rightarrow D \mapsto^{K_e} e_D \circ f \circ e_C$$

K_e often fails to be a semifunctor, as we have noted. Despite that, semifunctoriality, even weak, is not required for our purposes.

We are now ready to define compositional linearizability, where we recall that we work over an **Inf**-enriched semicategory.

Definition 14.1.1. Let $\mathbf{C}$ be an enriched semicategory equipped with a bi-semifunctor

$$- \otimes - : \underline{\mathbf{C}} \times \underline{\mathbf{C}} \rightarrow \underline{\mathbf{C}}$$

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

We say a morphism $f : C \rightarrow D \in \underline{\mathbf{C}}$ is (*compositionally*) *linearizable* to a morphism $g : C \rightarrow D \in \underline{\mathbf{C}}$ when

$$f \geq K_e g$$

In addition, when

$$f = K_e g$$

we say f is *self-linearizable* w.r.t. g .

Since our proofs of locality and interaction refinement on **Safe** were abstract, relying on Prop. 13.1.1, we can collect the necessary assumptions to obtain those results.

Proposition 14.1.1. *In the following, let $\underline{\mathbf{C}}$ and $\mathbf{C}_e$ satisfy the conditions of Def. 14.1.1.*

Observational Refinement *Suppose for all $C \in \underline{\mathbf{C}}$ and $f : \mathbf{1} \rightarrow C \in \underline{\mathbf{C}}$ it holds that*

$$f \circ e_1 = f$$

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

$$h \circ f \geq h \circ g$$

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

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

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

$$- \otimes_{\underline{\mathbf{C}}} - : \mathbf{C}_e(\mathbf{1}, C) \otimes \mathbf{C}_e(\mathbf{1}, D) \rightarrow \mathbf{C}_e(\mathbf{1}, C \otimes D)$$

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

Proof. These are essentially the same proofs as the corresponding proofs we presented in §13.4 and §13.5.

Observational Refinement

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

For the reverse direction, by reductivity of K_e (Prop. 14.4.1) and the hypothesis instantiated at $h = e_C$:

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

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

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

for the reverse direction, observe that

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

by the assumption that the tensor defines an order embedding, we have:

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

□

Note that Prop. 14.1.1 does not require that K_e be functorial in any way. In practice, K_e is often a(n) (op)lax semifunctor in that either

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

in which case K_e is called oplax, or,

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

in which case K_e is lax.

14.2 Sketches of Compositional Linearizability

We now discuss a few examples of compositional linearizability.

Example 14.2.1 (The Degenerate Case). When the underlying semicategory $\mathbf{C}$ is a category already, computing the section $(\mathbf{C})_{\text{id}}$ yields an equivalent category to $\mathbf{C}$, as, by definition of a category, every morphism is saturated under the identity morphisms. Compositional linearizability then amounts to

$$f \geq K_{\text{id}} g = g$$

so that compositional linearizability coincides with the underlying enrichment.

Example 14.2.2 (Sequential Games). Consider our model of sequential computation, $\mathbf{Safe}^{\text{seq}}$. Although we did not emphasize that there, we defined it as the section $(\underline{\mathbf{Safe}}^{\text{seq}})_{\text{copy}}$. As is well known, and as we discussed in §6.3, strategies saturated with respect to seqcopy are characterized precisely as the O -receptive strategies. Considering this, compositional linearizability for our sequential model means that $\sigma : A \in \mathbf{Safe}^{\text{seq}}$ is linearizable to $\tau : A \in \mathbf{Safe}^{\text{seq}}$ when:

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

As before, it is not hard to see that in this context, given a (not necessarily O -receptive) strategy $\tau : A \multimap B \in \underline{\mathbf{Safe}}^{\text{seq}}$,

$$K_{\text{copy}} \tau = O\text{-recep}(\tau)$$

that is, the O -receptive closure of τ . Note that in the presence of starvation tokens, $O\text{-recep}(\tau)$ actually allows for O -moves to be added before a $\bullet$ token. For example, for

$\nu = \downarrow \bullet : \text{Unit}$, $\text{O-recep}(\nu) = \downarrow q \cdot \bullet \cup \bullet$. Moreover, compositional linearizability states that $\sigma : A$ is linearizable to $\tau : A$ when

$$\sigma \subseteq \text{O-recep}(\tau)$$

It is folklore in game semantics that receptivity can be largely disregarded in the theory, which is precisely what our compositional linearizability formalism retrieves, as we obtain locality and interaction refinement between O -receptive sequential strategies and not necessarily receptive sequential strategies. It is also easy to see that the compatible notion of linearizability on traces is that a sequential play s is linearizable to t when either: (1) $s = t$, or (2) $s = t \cdot m_O$ for some O -move m_O , or (3) there is a P move m_P such that $t = s \cdot m_P$, potentially with a $\bullet$ -token at the end, as in $s = t \cdot m_O \cdot \bullet$ or $t = s \cdot m_P \cdot \bullet$. That is to say, sequential linearizability allows a trace s with a pending operation (necessarily unique when it exists) to be either removed or completed.

This means that using compositional linearizability in the sequential setting allows us to specify the counter from Ex. 6.1.2 using the following state transition system instead:

$$\mathcal{V}_{\text{Counter}}^{\text{lin}} := ((\mathbb{N} \times (\text{Counter} + \text{idle} + \text{starved}), M_{\text{Counter}}^{\circ \bullet}, - \xrightarrow{-}_{\text{Counter}} -), (0, \text{idle}))$$

defined by the rules:

$$\begin{array}{c} \frac{}{(n, \text{idle}) \xrightarrow{\text{get}} (n, \text{get})} \qquad \frac{}{(n, \text{get}) \xrightarrow{n} (n, \text{idle})} \qquad \frac{}{(n, \text{idle}) \xrightarrow{\text{inc}} (n, \text{inc})} \\[1em] \frac{}{(n, \text{inc}) \xrightarrow{\text{ok}} (n+1, \text{idle})} \qquad \frac{}{(n, \text{idle}) \xrightarrow{\bullet} (n, \text{starved})} \end{array}$$

The key difference is that in this linearized specification, starvation now only happens when the system is idle.

Example 14.2.3 (Concurrent Games). The core of this dissertation, through sections §12.1, §12.2, §13 provides the setting of original linearizability as seen through the lens of compositional linearizability (except for the addition of starvation tokens). In the language of compositional linearizability we have introduced in §14.1 so far, we started by defining a

concurrent game model **Safe** encoding locally sequential concurrent computation, and then showed it defines a semicategory $(- ; -)$ enriched with a notion of refinement $(- \sqsubseteq_S -)$. Then, we proved that copy_A is an idempotent for every game A . This enabled us to define the category **Safe** as the section $(\text{Safe})_{\text{copy}}$, which comes with a forgetful semifunctor Emb_{copy} and a saturation operation K_{copy} . After showing that there is a bi-semifunctor

$$- \otimes - : \text{Safe} \times \text{Safe} \rightarrow \text{Safe}$$

whose lift defines a symmetric monoidal category $(\text{Safe}, - \otimes -, \mathbf{1})$ we obtain a notion of compositional linearizability by Def. 14.1.1, which supports observational refinement and a locality property by Prop. 14.1.1. These abstract notions were shown to agree with the usual conception of linearizability in §13, a matter we discuss further in §15.1 and §15.2.

Example 14.2.4 (Delay-Insensitive Circuits). Ghica [2013] defines a compositional trace model **DIAsy** of asynchronous delay-insensitive circuits, which assembles into a compact closed category. To get to this model, Ghica first defines a category **IdAsy** providing a circuit model with idealized wires **W** (wires that are *not* physically realizable) as the identity morphisms. To obtain a more realistic model, it becomes necessary to replace these idealized wires by a more realistic notion of a capacitive wire **K** which models potential glitches caused by the fact that real wires have capacitance. These capacitive wires **K** are idempotent, justifying computing the Karoubi envelope of **IdAsy** to obtain a model for which **K** can behave as the identity. In the remainder of this section, we briefly present the construction and argue that it serves as an example of a setting where the techniques here could apply.

First, we define a *port* as a pair of a set A representing the signals that travel through the wire and a polarity $p \in \{+, -\}$ representing the “direction” of the wire. We depict these pairs by pA , e.g. $+A$ and $-A$. An object of **IdAsy** consists of an array of ports, which we depict as $p_1A_1 \otimes \dots \otimes p_nA_n$. Given such a port array, we define:

$$\llbracket p_1A_1 \otimes \dots \otimes p_nA_n \rrbracket := A_1 + \dots + A_n$$

Toward defining morphisms (which represent circuits with input and output ports), recall

first that we defined in §12 an interleaving $S \parallel T$ operation on sets of sequences $S, T \subseteq A^*$. Denote, for the sake of this section, by $!S$ the set $\cup_{n \in \mathbb{N}} S^{\parallel n}$ where $S^{\parallel n}$ is the interleaving of S with itself n times. Then, a morphism $f : X \rightarrow Y$ in $\mathbf{IdAsy}$ between port arrays X and Y consists of a set of traces $f \subseteq (\llbracket X \rrbracket + \llbracket Y \rrbracket)^*$ closed under self-interleaving in that $!f = f$. To define the identity morphism, we first write $w_{pA} := \text{inl}^*(A) \cdot \text{inr}^*(A)$ (where $\cdot$ is serial composition as in the usual Kleene algebra on languages). Then, for a port array $X = p_1 A_1 \otimes \dots \otimes p_n A_n$ we write

$$w_X := w_{p_1 A_1} \parallel \dots \parallel w_{p_n A_n}$$

Finally, the identities for $\mathbf{IdAsy}$ are the idealized wires $W_X := !w_X$. We refer the reader to Ghica [2013] for the definition of morphism composition $-\circ-$, which follows a similar motto to game semantics composition but works differently in how it computes interactions prior to hiding.

$\mathbf{IdAsy}$ defines a compact closed category, the structure of which we do not fully define here, merely defining the tensor structure and the dual on objects. The tensor on port arrays is given by $X \otimes Y = X \otimes Y$ with unit $\mathbf{1} = +\emptyset$, and by $f \otimes g := \text{inl}^*(f) \parallel \text{inr}^*(g)$ on morphisms $f : A \rightarrow B$ and $g : C \rightarrow D$. The dual $(pA)^\perp$ of the port (pA) is $p^\perp A$ (where $+\perp = -$, and $-\perp = +$). The dual $X^\perp$ of a port array X is obtained by dualizing all the ports in X . Even though we do not define it here, we write $f^\perp : Y^\perp \rightarrow X^\perp$ for the dual of a morphism $f : X \rightarrow Y$ (defined by the usual formula on a compact closed category).

With this in place, the more realistic capacitive wire $k_{+A} : +A \rightarrow +A$ on a port $+A$ is given by

$$k_{+A} := !((\text{inl}^*(A) \cdot \text{inr}^*(A)) \cup (\text{inl}^*(A) \cdot \text{inl}^*(A)))$$

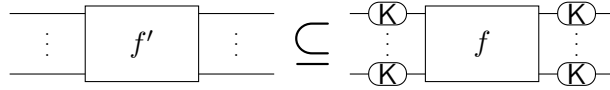
The first term is a successful signal transmission, and the second term models a glitch where two input signals cancel each other out due to capacitive inertia. For a port $-A$ the capacitive wire is given by the dual $k_{-A} := k_{+A}^\perp : -A \rightarrow -A$. Then, for a port array $X = p_1 A_1 \otimes \dots \otimes p_n A_n$ we define $K_X : X \rightarrow X := k_{p_1 A_1} \otimes \dots \otimes k_{p_n A_n}$.

Ghica [2013] proves that K is idempotent, allowing for defining the category $\mathbf{DIAsy}$ of

asynchronous delay-insensitive circuits as the section $(\text{IdAsy})_K$. Intuitively, IdAsy models circuits where wires are assumed idealized. Saturation under K in DIAsy means that these idealized wires in the input and output ports of a circuit come with the component K attached to it to model the effect of unknown bounded capacitance on the wire, which non-deterministically causes glitches of the type described above. This means that, using the language of string diagrams, a wire in DIAsy corresponds to the following circuit in IdAsy :



We can enrich IdAsy over trace containment, so we obtain that a circuit with capacitive wires $f' : X \rightarrow Y \in \text{DIAsy}$ is linearizable to a circuit with idealized wires $f : X \rightarrow Y \in \text{IdAsy}$ when the trace containment $f' \subseteq K_X; f; K_Y$ holds, nicely depicted by the string diagrams below:



The fact that DIAsy and IdAsy have compatible compact closed structures should mean that circuits with input and output ports can be formally “bent” so they only have wires facing one way. This should mean that our formulation of linearizability in terms of morphisms $\mathbf{1} \rightarrow A$ would still be enough. We formulate it directly on morphisms $A \rightarrow B$ in this way for it produces a more appealing and intuitive theory.

Almost the same technique as in Prop. 14.1.1 allows us to show that:

Proposition 14.2.1. *A circuit $f' : X \rightarrow Y \in \text{DIAsy}$ is linearizable to a circuit $f : X \rightarrow Y \in \text{IdAsy}$ if and only if, for all circuits $g : X' \rightarrow X$ and $h : Y \rightarrow Y'$ in DIAsy , it holds that $g; f'; h \subseteq g; f; h$.*

So one can reduce reasoning about individual components within a larger circuit with capacitive wires to a circuit where that component is substituted by an idealized component.

Locality is only slightly more complicated than in our concurrency model. Suppose we denote by $\perp_{X,Y} : X \rightarrow Y$ the circuit $\emptyset$, which should be understood as a buggy or failing circuit, and say that a circuit $f : X \rightarrow Y$ is *safe* when $f \neq \perp_{X,Y}$. While for safe circuits $f : X \rightarrow Y, g : X' \rightarrow Y'$ the required conditions for locality do hold, they fail for the failing

circuit $\perp_{X,Y}$. Indeed, for any f it holds that $f \otimes \perp = \perp$. Because of this, $f \otimes \perp \subseteq f' \otimes \perp$ does not imply $f \subseteq f'$. Intuitively, the failing circuit is refined by any circuit, and parallel composition with a failing circuit should result in a failing circuit, causing the issue. On the other hand, the same techniques we used before can establish the locality property for safe circuits.

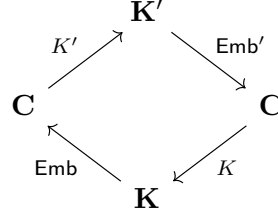
Proposition 14.2.2. *Let $f, g : X \rightarrow Y$ and $f', g' : X' \rightarrow Y'$ be safe circuits. Then, $f \otimes g$ is linearizable to $f' \otimes g'$ if and only if f is linearizable to g and f' is linearizable to g' .*

This provides an interesting transfer of technology. Linearizability was originally formulated to reduce reasoning about concurrent objects to reasoning about atomic components. The discovery of this abstract formulation of linearizability allows us to smoothly provide a compelling criterion to reduce reasoning about circuits with capacitive wires to circuits with idealized wires, a notably different setting. This setting is also interesting for `IdAsy` is already a category; the shift to the section `DIAsy` allows for a change of interpretation for its objects, along the lines of our explanation in §14.1.2, which in this case is of material relevance to physical circuits.

14.3 The Linearizability Galois Connection

The main result of Goubault et al. [2018] is that if $\text{Lin} -$ is the operation taking some atomic object specification S to the set of traces $\text{Lin } S$ linearizable w.r.t. S , and $U -$ is the operation taking concurrent object specifications S' (in particular, $\rightsquigarrow$ -closed) to the set of atomic traces $U \ S' \subseteq S'$ contained in it, then $\text{Lin} \dashv U$ is a Galois connection (in fact, an insertion). In this section, we show how our treatment accommodates this result at the level of compositional linearizability, which we will later instantiate in §15.1 to obtain the result between our concurrent games and a notion of atomic games.

We are interested in comparing two sections $\mathbf{C}_e$ and $\mathbf{C}_{e'}$ of the same semicategory $\mathbf{C}$. A particular example of this will be when $\mathbf{C}_e$ corresponds to concurrent games and $\mathbf{C}_{e'}$ corresponds to atomic games. 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 operations L and R defined as:

$$L : \mathbf{K} \rightarrow \mathbf{K}' := K' \circ \text{Emb} \quad R : \mathbf{K}' \rightarrow \mathbf{K} := K \circ \text{Emb}'$$

These should be interpreted as canonical conversions from one concurrency model to the other. Our key claim is that whenever it holds that for all $A \in \mathbf{C}$, $e_A \geq e'_A$, the operations L and R assemble into a pair of adjoint functors $R \dashv L : \mathbf{K}(A, B) \rightarrow \mathbf{K}'(A, B)$, which in the context of the assumed posetal enrichment essentially says they form a Galois connection. In addition, under the same assumption, we also obtain that $L : \mathbf{K} \rightarrow \mathbf{K}'$ is a lax semifunctor and that $R : \mathbf{K}' \rightarrow \mathbf{K}$ is an oplax semifunctor.

Proposition 14.3.1. *If*

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

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

$$e_A \geq e'_A$$

for every $A \in \mathbf{C}$, then the mappings L and R defined by

$$L : \mathbf{C}_e \rightarrow \mathbf{C}_{e'} := K' \circ \text{Emb} \quad R : \mathbf{C}_{e'} \rightarrow \mathbf{C}_e := K \circ \text{Emb}'$$

define a lax functor and an oplax functor, respectively.

Moreover, for every pair of $A, B \in \mathbf{C}$, the associated functors of hom-categories:

$$L : \mathbf{C}_e(A, B) \rightarrow \mathbf{C}_{e'}(A, B) \quad R : \mathbf{C}_{e'}(A, B) \rightarrow \mathbf{C}_e(A, B)$$

form an adjunction.

Proof. For convenience, we let

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

and

$$K = K_e \quad \text{Emb} = \text{Emb}_e \quad K' = K_{e'} \quad \text{Emb}' = \text{Emb}_{e'}$$

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

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

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

First, we show the weak functoriality results. We start with L . Note first that

$$L e = e' \circ e \circ e' \geq e' \circ e' \circ e' = e'$$

moreover

$$L (g \circ f) = e' \circ g \circ f \circ e' = e' \circ g \circ e \circ f \circ e' \geq e' \circ g \circ e' \circ f \circ e' = e' \circ g \circ e' \circ e' \circ f \circ e' = L g \circ L f$$

For R , we have

$$e = e \circ e \circ e \geq e \circ e' \circ e = R e'$$

moreover

$$R g \circ R f = e \circ g \circ e \circ e \circ f \circ e = e \circ g \circ e \circ f \circ e \geq e \circ g \circ e' \circ f \circ e = e \circ g \circ f \circ e = R (g \circ f)$$

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

Now, for the adjunction result, we simply note that for any $f : A \rightarrow B \in \mathbf{K}$ and any

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

$$f = e_B \circ f \circ e_A = e_B \circ e_B \circ f \circ e_A \circ e_A \geq e_B \circ e'_B \circ f \circ e'_A \circ e_A = R \, L \, f$$

and

$$L \, R \, f' = e_B \circ e'_B \circ f' \circ e'_A \circ e_A \geq e'_B \circ e'_B \circ f' \circ e'_A \circ e'_A = e'_B \circ f' \circ e'_A = f'$$

the naturality squares and triangle identities follow simply from being well-typed as $\mathbf{K}(A, B)$ and $\mathbf{K}'(A, B)$ are both thin categories. $\square$

This might seem like a rather weak result, but it readily gives as a corollary the main result of Goubault et al. [2018], as we will see in §15.1, moreover refining it by providing an account of the effect of linearizability on composition. Note also that in the above proposition, we do not require K and K' to be even (op)lax semifunctors. Indeed, although in all our models they will be, this is not required to show Proposition 14.3.1.

14.4 The Compositional Linearizability Refinement Calculus

To finish this chapter, we show that compositional linearizability is a robust and appropriate notion of refinement in $\mathbf{C}_e$. Following our observations in §10, equipping $\mathbf{C}_e$ with compositional linearizability as its enrichment would suffice to obtain a compositional refinement calculus. However, there is a subtlety: compositional linearizability does not define a partial order in general. So, say we write $f \rightsquigarrow g$ when $f \geq K_e g$. Then:

$$f \rightsquigarrow g \wedge g \rightsquigarrow f \implies K_e f = K_e g$$

which is a different claim from $f = g$. This is precisely the reason we defined inf-lattices over preorders (Def. 6.6.2): compositional linearizability defines only a preorder, not a partial order, but the inf-lattice structure still applies with constructs unique up to the induced equivalence $f \equiv g \iff f \geq g \wedge g \geq f$. We focus then on establishing the enrichment of $\mathbf{C}_e$ over compositional linearizability.

First, we establish several properties of K_e .

Proposition 14.4.1.

- K_e is cocontinuous, and
- K_e is idempotent, and
- K_e is surjective, and
- $K_e -$ is a kernel operator

Proof.

- Follows immediately from the definition and the fact that categorical composition is cocontinuous.
- This follows from idempotence of e_- . Indeed, for $f : C \rightarrow D$:

$$K_e K_e f = e_D \circ e_D \circ f \circ e_C \circ e_C = e_D \circ f \circ e_C = K_e f$$

- Let $f \in \mathbf{C}_e(C, D)$ then $K_e f = e_D \circ f \circ e_C = f$.
- Recall that a kernel operator is a monotonic, idempotent, reductive operator. Note that the proof of surjectivity also implies that $f \geq K_e f$ by reflexivity, showing reductivity, which is the missing property to show $K_e -$ is a kernel operator.

□

Proposition 14.4.2. *For any $C, D \in \underline{\mathbf{C}}$, $(\underline{\mathbf{C}}(C, D), - \leftarrow -)$ is a pre-ordered inf-lattice, which restricts into a (pre-ordered) sub-inf-lattice $(\mathbf{C}_e(C, D), - \leftarrow -)$.*

This is enough to make $\underline{\mathbf{C}}$ and $\mathbf{C}_e$ into an enriched semicategory and an enriched symmetric monoidal category, respectively.

Proof. First, that it is a pre-order follows readily from the fact that $K_e -$ is a cocontinuous kernel operator. Indeed, reflexivity follows from the fact that $K_e -$ is reductive:

$$\forall f. f \geq K_e f \implies f \rightsquigarrow f$$

and transitivity follows from:

$$f \curvearrowright g \wedge g \curvearrowright h \iff g \geq K_e f \wedge h \geq K_e g \implies h \geq K_e g \geq K_e K_e f = K_e f \iff f \curvearrowright h$$

Finally, that we can compute all meets follows from cocontinuity. Indeed, given a collection $(f_i)_{i \in I}$ let their infimum be computed as in the pre-existing enrichment of $\underline{\mathbf{C}}$. We show this is still an infimum under compositional linearizability. That it is a lower bound is readily seen from cocontinuity:

$$\forall i. f_i \geq K_e f_i \geq \bigwedge_{i \in I} K_e f_i = K_e \bigwedge_{i \in I} f_i$$

so that $f_i \curvearrowright \bigwedge_{i \in I} f_i$. Now, suppose for all f_i , $f_i \curvearrowright f$. Then:

$$\forall i. f_i \geq K_e f \implies \bigwedge_{i \in I} f_i \geq K_e f$$

so $\bigwedge_{i \in I} f_i \curvearrowright f$.

Now, that the restriction $\mathbf{C}_e(C, D)$ is pre-ordered is immediate. So say the collection $(f_i)_{i \in I}$ includes only saturated morphisms. We argue that $\bigwedge_{i \in I} f_i$ is saturated. Indeed, by cocontinuity:

$$e_D \circ \left(\bigwedge_{i \in I} f_i \circ e_C \right) = e_D \circ \bigwedge_{i \in I} (f_i \circ e_C) = \bigwedge_{i \in I} (e_D \circ f_i \circ e_C) = \bigwedge_{i \in I} K_e f_i = \bigwedge_{i \in I} f_i$$

where the last equality holds by saturation.

Now, that both categorical composition and tensoring are still cocontinuous follows immediately from the fact that meets are computed in the same way, as per the argument above. $\square$

Were we to run now the construction $\mathcal{R}(-)$ from Ch. 10 we would certainly obtain a calculus. However, the notion of specification would be given by $\mathbf{C}_e(\mathbf{1}, C)$ which forces specifications to be saturated. But the whole point of compositional linearizability is to present saturated morphisms $f' \in \mathbf{C}_e(\mathbf{1}, C)$ in terms of morphisms $f \in \underline{\mathbf{C}}(\mathbf{1}, C)$ by iden-

tifying $f' = K_e f$. So we make a key modification to the construction to accommodate linearizability.

Definition 14.4.1. Given a section $\mathbf{C}_e$ satisfying the conditions in §14.1.2, we define $\mathcal{L}(\mathbf{C}_e)$ the category given by the following data:

Objects $v \in \mathcal{L}(\mathbf{C}_e)$ are morphisms $v \in \underline{\mathbf{C}}(\mathbf{1}, C)$ for some object $C \in \underline{\mathbf{C}}$,

Morphisms $f : v_C \Rightarrow v_D$ in $\mathcal{L}(\mathbf{C}_e)(v_C, v_D)$ are given by saturated morphisms $f \in \mathbf{C}_e(C, D)$ satisfying that:

$$f \circ v_C \rightsquigarrow v_D$$

Categorical composition is given by the categorical composition in $\mathbf{C}_e$. The neutral elements $\mathbf{id}_v : v \Rightarrow v$, for $v : \mathbf{1} \rightarrow C$, are exactly $\mathbf{id}_C : C \rightarrow C$. The tensor operation is inherited from $\mathbf{C}_e$.

There should be no surprises in the definition of $\mathcal{L}(\mathbf{C}_e)$, as it corresponds to $\mathcal{R}(\mathbf{C}_e)$, except that we take specifications in $\underline{\mathbf{C}}$ instead of $\mathbf{C}_e$. In particular, as long as categorical composition and tensoring are well-defined, they will satisfy all the necessary coherence laws, similar to Ch. 10. However, several of the proofs there need small adaptations.

Proposition 14.4.3.

- For any $v_A : \mathbf{1} \rightarrow A$, $v_B : \mathbf{1} \rightarrow B$, $v_C : \mathbf{1} \rightarrow C$, it holds that if

$$f : v_A \Rightarrow v_B \wedge g : v_B \Rightarrow v_C \implies g \circ f : v_A \Rightarrow v_C$$

- For any $v_A : \mathbf{1} \rightarrow A$, $v_B : \mathbf{1} \rightarrow B$, $v_C : \mathbf{1} \rightarrow C$, $v_D : \mathbf{1} \rightarrow D$,

$$f : v_A \Rightarrow v_C \wedge g : v_B \Rightarrow v_D \implies f \otimes g : v_A \otimes v_B \Rightarrow v_C \otimes v_D$$

Proof. For the first claim:

$$K_e v_C \leq g \circ v_B = K_e g \circ v_B$$

where the last equality holds by saturation. Then:

$$e_C \circ g \circ e_B \circ v_B = e_C \circ g \circ e_B \circ e_B \circ v_B = e_C \circ g \circ e_B \circ e_B \circ v_B \circ e_1$$

note that the last equality holds by our assumption that for any $f \in \underline{\mathbf{C}}(\mathbf{1}, C)$, $f \circ e_1 = f$.

But then:

$$e_C \circ g \circ e_B \circ e_B \circ v_B \circ e_1 = g \circ K_e v_B \leq g \circ (f \circ v_A) = (g \circ f) \circ v_A$$

Combining all the inequalities, we get:

$$K_e v_C \leq (g \circ f) \circ v_A$$

Now, for the tensor, note that we have already shown that K_e – distributes over it in Prop. 14.1.1. So we have that:

$$K_e (v_C \otimes v_D) = K_e v_C \otimes K_e v_D \leq (f \circ v_A) \otimes (g \circ v_B) = (f \otimes g) \circ (v_A \otimes v_B)$$

□

Proving the consequence rule also requires some adaptations:

Proposition 14.4.4 (Consequence Rule).

- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v'_A \rightsquigarrow v_A \iff e_A : v'_A \Rightarrow v_A$.
- $v'_A \rightsquigarrow v_A \wedge f : v_A \Rightarrow v_B \wedge v_B \rightsquigarrow v'_B \implies f : v'_A \Rightarrow v'_B$

Proof. For the first claim, note that $K_e v'_A = e_A \circ v'_A$ for $v'_A : \mathbf{1} \rightarrow A$. In the forward direction,

$$v'_A \geq e_A \circ v_A \implies e_A \circ v'_A \geq e_A \circ e_A \circ v_A = e_A \circ v_A = K_e v_A$$

Conversely,

$$e_A \circ v'_A \geq K_e v_A \implies v'_A \geq K_e v'_A = e_A \circ v'_A \geq K_e v_A$$

by reductivity.

For the second claim, the first claim gives:

$$e_A : v'_A \Rightarrow v_A \qquad e_B : v_B \Rightarrow v'_B$$

By composition and saturation ($e_B \circ f \circ e_A = f$):

$$f = v'_A \xRightarrow{e_A} v_A \xRightarrow{f} v_B \xRightarrow{e_B} v'_B$$

□

We have already noted in Ch. 3 how contextual refinement is just monotonicity. We reinforce this here by the following proposition.

Proposition 14.4.5 (Contextual Refinement).

- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v'_A \leadsto v_A \iff \forall B \in \mathbf{C}. \forall f \in \mathbf{C}_e(A, B). f \circ v'_A \leadsto f \circ v_A$
- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v'_A \leadsto v_A \iff \forall B, C \in \mathbf{C}. \forall f \in \mathbf{C}_e(B, C). v'_A \otimes f \leadsto v_A \otimes f$
- For any $v'_A, v_A : \mathbf{1} \rightarrow A$, $v'_A \leadsto v_A \iff \forall B, C \in \mathbf{C}. \forall f \in \mathbf{C}_e(B, C). f \otimes v'_A \leadsto f \otimes v_A$

Proof. For the forward direction, notice that:

$$v'_A \leadsto v_A \iff v'_A \geq e_A \circ v_A \implies f \circ v'_A \geq f \circ e_A \circ v_A = f \circ v_A$$

where the last equality holds by saturation. Then, we see that:

$$f \circ v'_A = e_B \circ f \circ v'_A \geq e_B \circ f \circ v_A = K_e (f \circ v_A)$$

For the backward direction, note that by instantiating $f = e_A$ we have:

$$e_A \circ v'_A \geq K_e e_A \circ v_A = K_e K_e v_A = K_e v_A$$

Then, by reductivity,

$$v'_A \geq K_e v'_A = e_A \circ v'_A \geq K_e v_A$$

For the tensor, the forward direction follows from monotonicity of $- \otimes -$ and the fact that $K_e -$ distributes over $- \otimes -$:

$$v'_A \otimes f \geq K_e v_A \otimes f = K_e v_A \otimes K_e f = K_e (v_A \otimes f)$$

where the second equality holds by saturation of f .

For the backward direction of the second claim, we instantiate $B = C = \mathbf{1}$ and $f = e_1$, then, by assumption:

$$v'_A \otimes e_1 \geq K_e (v_A \otimes e_1) = K_e v_A \otimes e_1$$

Now, we compose the right unitor on both sides and use monotonicity of composition to get:

$$\rho_A \circ (v'_A \otimes e_1) \geq \rho_A \circ (K_e v_A \otimes e_1)$$

By naturality and the assumption that $g \circ e_1 = g$ for $g : \mathbf{1} \rightarrow A$:

$$v'_A \circ \rho_1 = \rho_A \circ (v'_A \otimes e_1) \geq \rho_A \circ (K_e v_A \otimes e_1) = K_e v_A \circ \rho_1$$

Since ρ_1 is an isomorphism with inverse ρ_1^{-1} , by monotonicity:

$$v'_A = v'_A \circ \rho_1 \circ \rho_1^{-1} \geq K_e v_A \circ \rho_1 \circ \rho_1^{-1} = K_e v_A$$

as desired.

The third claim follows from the second by symmetry. □

Now, quite importantly, we show that linearizability generalizes the underlying notion of refinement:

Proposition 14.4.6. *If $f \geq g$ then $f \rightsquigarrow g$*

Proof. By reductivity:

$$f \geq g \geq K_e g$$

□

We can then restate the calculus developed in Ch. 10, where now correctness is understood as in $\mathcal{L}(\mathbf{C}_e)$. That is, the claims of correctness use the compositional linearizability pre-order instead of the underlying abstraction pre-order. This provides a calculus for compositional proofs of linearizability in any model of compositional linearizability. Of course, as before, we have omitted many rules that follow from the categorical structure as well, such as all the facts following from the coherence laws of a symmetric monoidal category.

$$\begin{array}{c}
\frac{}{e_A : v_A \Rightarrow v_A} \text{ ID} \qquad \frac{f : v_A \Rightarrow v_B \quad g : v_B \Rightarrow v_C}{g \circ f : v_A \Rightarrow v_C} \text{ COMP} \\
\\
\frac{f : v_A \Rightarrow v_C \quad g : v_B \Rightarrow v_D}{f \otimes g : v_A \otimes v_B \Rightarrow v_C \otimes v_D} \text{ TENSOR} \quad \frac{v'_A \leadsto v_A \quad f : v_A \Rightarrow v_B \quad v_B \leadsto v'_B}{f : v'_A \Rightarrow v'_B} \text{ CONSEQ} \\
\\
\frac{f \leadsto g \quad f : v_A \Rightarrow v_B}{g : v_A \Rightarrow v_B} \text{ MONO} \qquad \frac{\forall i \in I. f_i : v_A \Rightarrow v_B}{\bigwedge_{i \in I} f_i : v_A \Rightarrow v_B} \text{ INF}
\end{array}$$

Chapter 15

Equivalence with Linearizability

15.1 Atomicity

In this section, we show that our framework provides a conservative generalization of the theory surrounding Herlihy-Wing linearizability, which always assumes the linearized specification is atomic. In the process, we generalize the result by Goubault et al. [2018] that Herlihy-Wing linearizability forms a Galois connection between concurrent and atomic specifications. We start by defining a category of sequential atomic games **Atomic** in §15.1.1. Then, we show it can be seen as a section of the Karoubi envelope in §15.1.2 which exhibits linearizability as an approximation of concurrent specifications. We then specialize the theory of linearizability developed in Ch. 13 to Herlihy-Wing linearizability.

15.1.1 Sequential Atomic Games

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

Definition 15.1.1. Let $A = (M_A, P_A)$ be a sequential game. We define its associated atomic game $!A = (M_{!A}, P_{!A})$ as follows:

$$M_{!A} := \sum_{\alpha \in \Upsilon} M_A \quad \lambda_{!A} := [\lambda_A]_{\alpha \in \Upsilon} \quad P_{!A} := \{s \in \mathbb{SP}_{!A} \mid \forall \alpha \in \Upsilon. \pi_\alpha(s) \in P_A\}$$

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:

$$\alpha_1:m_1 \rightarrow \alpha_1:n_1 \rightarrow \alpha_2:m_2 \rightarrow \alpha_2:n_2 \rightarrow \alpha_3:m_3 \rightarrow \alpha_3:n_3 \rightarrow \dots \rightarrow \alpha_k:m_k \rightarrow \alpha_k:n_k$$

where the m_i are O moves and the n_i are P moves. We may take $!A$ as an alternating version of the concurrent game corresponding to A^Φ , as any play of $!A$ may be seen as an alternating play of A^Φ , a fact we frequently make use of. The notation $!-$ comes from the similarity of the definition with the exponential modality defined in Hyland [1997], which is closely related to our definition.

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 $!\text{Unit} \multimap !\text{Unit}$

$$\begin{array}{c} \alpha:q \\ \searrow \\ \alpha':q \end{array}$$

even when $\alpha \neq \alpha'$. This disagrees with our agent naming discipline in 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 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:

$$\begin{array}{ccccccc} \alpha_1:m_1 & & \alpha_1:n_1 \rightarrow \alpha_2:m_2 & & \alpha_2:n_2 \rightarrow \dots \\ \searrow & & \nearrow & & \searrow & & \nearrow \\ & \alpha_1:m_1^1 \rightarrow \dots \rightarrow \alpha_1:n_{k_1}^1 & & \alpha_2:m_1^2 \rightarrow \dots \rightarrow \alpha_2:n_{k_2}^2 & & \end{array}$$

We call plays of this form *atomic plays*.

It is important to note that the copycat strategy

$$\text{seqcopy}_{!A} : !A \multimap !A$$

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

Definition 15.1.2. The category **Atomic** has atomic games $!A, !B$ as objects and O -receptive atomic strategies as morphisms. Composition is given by the usual sequential strategy composition and the identity is the sequential copycat $\text{copy}_{!A}$.

15.1.2 Concurrent Atomic Games

Interestingly, **Atomic** can be seen as a section of the Karoubi envelope of **Safe**. Let

$$\text{atocopy}_A : A \multimap A := \text{seqcopy}_{!A}$$

that is, atocopy_A is the concurrent strategy obtained by identifying the plays in $\text{seqcopy}_{!A}$ as plays of type $(A \multimap A)^\Phi$ as discussed in §15.1.1. It is straightforward to check that

Proposition 15.1.1. $\text{atocopy}_A : A \multimap A$ is idempotent.

This means we can construct the section

$$\mathbf{K}_{\text{Atom}} := (\mathbf{Safe})_{\text{atocopy}}$$

with associated (strict) semifunctors

$$\text{Emb}_{\text{Atom}} : \text{Semi } \mathbf{K}_{\text{Atom}} \longrightarrow \mathbf{Safe} \qquad K_{\text{Atom}} : \mathbf{Safe} \longrightarrow \text{Semi } \mathbf{K}_{\text{Atom}}$$

by following the construction in §14.1. Now, as

$$\text{atocopy}_A \subseteq \text{copy}_A$$

it immediately follows by Prop. 14.3.1 that if we define

$$\text{Lin}_{\text{Atom}} : \mathbf{K}_{\text{Atom}} \rightarrow \mathbf{Safe} := K_{\text{copy}} \circ \text{Emb}_{\text{Atom}}$$

$$U_{\text{Atom}} : \mathbf{Safe} \rightarrow \mathbf{K}_{\text{Atom}} := K_{\text{Atom}} \circ \text{Emb}$$

then we obtain a family of Galois connections:

$$\begin{array}{ccc}
& \text{Lin}_{\text{Atom}} & \\
\swarrow & & \searrow \\
\mathbf{Safe}(A, B) & \top & \mathbf{K}_{\text{Atom}}(A, B) \\
\searrow & & \swarrow \\
& \text{U}_{\text{Atom}} &
\end{array}$$

Note that, explicitly:

$$\tau : A \multimap B \mapsto^{\text{Lin}_{\text{Atom}}} \text{copy}_A; \tau; \text{copy}_B \qquad \sigma : A \multimap B \mapsto^{\text{U}_{\text{Atom}}} \text{atocopy}_A; \sigma; \text{atocopy}_B$$

By the results in §12.3, Lin_{Atom} is a closure operator computing the receptive-closure, P-delaying, and then the $\rightsquigarrow$ -closure. Meanwhile, U_{Atom} , it turns out, is equivalent to taking the largest substrategy of σ that plays only atomic plays:

$$\text{U}_{\text{Atom}} \sigma = \sigma \cap P_{!(A \multimap B)}$$

We can establish that **Atomic** is equivalent to $\mathbf{K}_{\text{Atom}}$.

Proposition 15.1.2. *There is an equivalence of categories:*

$$\mathbf{Atomic} \cong \mathbf{K}_{\text{Atom}}$$

Proof. See Proof 31.8.3. □

The equivalence is witnessed essentially by the identity functors, up to the conversion from atomic plays in $A \multimap B$ to plays in $!A \multimap !B$, and the reverse conversion.

So we have established a Galois connection (in fact, insertion) between atomic strategies and saturated concurrent strategies:

$$\begin{array}{ccc}
& \text{Lin}_{\text{Atom}} & \\
\swarrow & & \searrow \\
\mathbf{Safe}(A, B) & \top & \mathbf{Atomic}(!A, !B) \\
\searrow & & \swarrow \\
& \text{U}_{\text{Atom}} &
\end{array}$$

This faithfully enhances the results of Goubault et al. [2018]. There, they showed linearizability may be seen as an approximation operation by proving a certain Galois connection between specifications of concurrent objects. Here, we provide a compositional variant of

their result. Note that, explicitly, this means that for any saturated strategy $\sigma : A \multimap B$ and atomic strategy $\tau : !A \multimap !B$ the following equivalence holds:

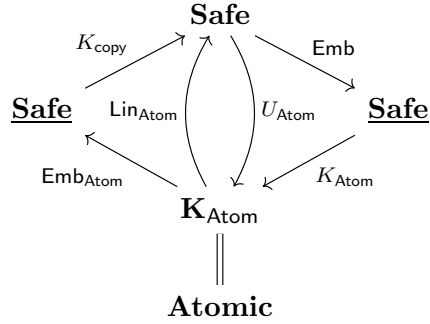
$$\text{Lin}_{\text{Atom}} \tau \sqsubseteq_S \sigma \iff \tau \sqsubseteq_S U_{\text{Atom}} \sigma$$

and moreover

$$U_{\text{Atom}} \text{Lin}_{\text{Atom}} \tau = \tau$$

Note that we have established this result by completely abstract means using our formalism in §14.1. Meanwhile, Goubault et al. [2018]’s original argument is based on the concrete formulation of linearizability in terms of their version of the rewrite relation $- \rightsquigarrow -$.

We find it useful to depict the results of this section along the lines of §14.1:



We note that in particular, Lin_{Atom} decomposes as an embedding of atomic games into the semicategory of concurrent games followed by closure under self-synchronization. An interesting fact is that the forgetful functor

$$U_{\text{Atom}} : \mathbf{Safe} \rightarrow \mathbf{Atomic}$$

admits a characterization in terms of the rewrite system $- \rightsquigarrow -$.

Proposition 15.1.3. *The irreducibles of $\rightsquigarrow_A$ are precisely the alternating plays of P_A^Φ .*

Proof. See Proof 31.8.4. □

This justifies the following definition.

Definition 15.1.3. Given a concurrent strategy $\sigma : A$, we denote by $\Downarrow \sigma : A$ its set of

irreducibles:

$$\Downarrow \sigma := \{s \in \sigma \mid s \text{ is alternating}\}$$

With which we obtain the desired characterization of U_{Atom} .

Proposition 15.1.4. *For any saturated $\sigma : A \multimap B$:*

$$U_{\text{Atom}} \sigma = \{s \in \sigma \mid s \upharpoonright_B \in \Downarrow (\sigma \upharpoonright_B)\}$$

namely, the set of all $s \in \sigma$ such that it plays an irreducible play in B .

Proof. See Proof 31.8.5. □

Developing a characterization of Lin_{Atom} along these lines is what we endeavor in §15.1.3.

15.1.3 Atomic Linearizability

We now endeavor to show that our definition of linearizability is equivalent to Herlihy-Wing linearizability [Herlihy and Wing, 1990]. Since Herlihy-Wing linearizability did not involve $\bullet$ -tokens, we ignore them in this section. Parts of our proof of this equivalence are adapted from Goubault et al. [2018] and Ghica and Murawski [2004]. 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 15.1.4. Indeed, we define an *operation* of a play $s = m_1 \cdot \dots \cdot m_k \in P_A$ as a pair $(p, q?)$ such that m_p is an O move, and, moreover, either $q? = q$, $\Upsilon(m_q) = \Upsilon(m_p)$ and

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

or $q? = \infty$ and

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

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 $\Upsilon(m_p) = \alpha$. We denote the set of operations of a play s by $\text{op}(s)$.

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

Definition 15.1.5. We define the happens-before order associated to a play s as the pair $(\text{op}(s), \prec_s)$ where

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

Definition 15.1.6. We say two plays $s, s' \in P_A$ 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 $\text{op}(s) = \text{op}(s')$ when convenient. We are now able to define Herlihy-Wing linearizability.

Definition 15.1.7. For a play $s \in P_A$, we call $\text{complete}(s)$ the largest subsequence of s such that

$$\forall \alpha \in \Upsilon. \pi_\alpha(\text{complete}(s)) = p \cdot m \Rightarrow \lambda_A(m) = P$$

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

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

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

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

Definition 15.1.8. The relation $- \equiv_A -$ on plays P_A is the smallest relation satisfying:

$$s \equiv_A t \iff s \rightsquigarrow_A t \text{ using only } OO \text{ and } PP \text{ swaps}$$

Observe that by Prop. 12.3.1, if $\sigma : A$ is saturated and $s \in \sigma$ then $[s]_{\equiv_A} \subseteq \sigma$, where $[s]_{\equiv_A}$ is the equivalence class of s under $\equiv_A$.

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

Proposition 15.1.5. *If $s, t \in P_A$ then $s \equiv_A t$ if and only if s and t are compatible and $\prec_s = \prec_t$.*

Proof. See Proof 31.9.1. □

Proposition 15.1.6. *For plays $s, t \in P_A$, there is a derivation*

$$s \rightsquigarrow_A t$$

if and only if s is compatible with t and

$$\prec_s \subseteq \prec_t$$

Proof. See Proof 31.9.2. □

These give the following important corollary.

Corollary 15.1.1. *A play $s \in P_A$ is linearizable to a play $t \in P_A$ if and only if s is Herlihy-Wing linearizable to t .*

Proof. See Proof 31.9.3. □

Our characterization of K_{copy} in terms of general linearizability also yields a characterization of the functor Lin_{Atom} .

Corollary 15.1.2. *For any atomic strategy $\tau : !A$*

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

Note that we arrived at the functor Lin_{Atom} through the abstract construction of the Karoubi envelope, which can be understood as closing a computational model, represented

by the semicategory **Safe**, by a synchronization pattern, represented by the choice of `copy_` or `atocopy_` as the unit. In this way, formally, Herlihy-Wing presents a solution to the problem of finding a concurrent strategy in **Safe** matching a certain atomic strategy in **Atomic**.

Proposition 15.1.2 also gives an alternative definition for Herlihy-Wing linearizability in terms of the image of the functor Lin_{Atom} .

Corollary 15.1.3. *A strategy $\sigma : A$ is Herlihy-Wing linearizable to an atomic strategy $\tau : !A$ if and only if*

$$\sigma \sqsubseteq_S \text{Lin}_{\text{Atom}} \tau$$

15.1.4 Observational Refinement and Locality

Herlihy-Wing linearizability admits its own computational interpretation of linearizability proofs, as a corollary of §13.3. Prop. 13.3.1 suggests defining a strategy

$$\text{intcopy}_A : A \multimap A := \{s \in \text{copy}_A \mid s \upharpoonright_{A,-} \in \Downarrow P_A\}$$

That is, intcopy_A is the substrategy of copy_A that plays atomically in the source component of $A \multimap A$. By Prop. 13.3.1 and Prop. 15.1.2, the plays of intcopy_A correspond to proofs of Herlihy-Wing linearizability. Interestingly, intcopy_- is idempotent, so that it admits its own theory along the lines of §14.1. In §16, we make use of the angle provided by Prop. 13.3.1 to analyze possibilities and other proof methodologies for linearizability.

Corollary 15.1.4 (Computational Interpretation of Herlihy-Wing Linearizability). *$s \in P_A$ is Herlihy-Wing linearizable to $t \in P_A$ if and only if there exists a play $s' \in \text{intcopy}_A$ such that*

$$s' \upharpoonright_{A,-} = t \qquad s' \upharpoonright_{-,A} = s$$

It is easy to see that the conditions of Prop. 14.1.1 are met by **Atomic**. In particular, we have that

Proposition 15.1.7. *(**Atomic**, $K_{\text{Atom}} \circ (- \otimes -)$, **1**) assembles into a symmetric monoidal category.*

We take the freedom of overloading $- \otimes -$ for the atomic tensor 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 15.1.8 (Observational Refinement). *$\nu'_A : A \in \mathbf{Safe}$ is Herlihy-Wing linearizable to $\nu_A : A \in \mathbf{Atomic}$ if and only if for all concurrent games B and $\sigma : A \multimap B \in \mathbf{Safe}$ it holds that*

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

Locality is also obtained by the same method as in §13.5, except that the source category is now **Atomic** and the lax functor Lin_{Atom} plays the role of K_{copy} .

Proposition 15.1.9 (Locality). *Let $\nu'_A : A$, $\nu'_B : B$ in **Safe** and $\nu_A : A$, $\nu_B : B$ in **Atomic**. Then*

$$\nu' = \nu'_A \otimes \nu'_B \text{ is linearizable w.r.t. } \nu = \nu_A \otimes \nu_B$$

if and only if

$$\nu'_A \text{ is linearizable w.r.t. } \nu_A \text{ and } \nu'_B \text{ is linearizable w.r.t. } \nu_B$$

15.2 Interval-Sequential Linearizability

In this section, we compare interval-sequential linearizability with our notion of linearizability. Goubault et al. [2018] noted, without proof, that their definition of linearizability is equivalent to interval-sequential linearizability (although it is the restriction of interval-sequential linearizability to total objects, in addition requiring self-linearizability). Here, we show that our definition of linearizability in the context of our model of locally sequential concurrent computation corresponds to a generalization of interval-sequential linearizability to handle blocking objects, which the original definition cannot handle [Castañeda et al., 2015].

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

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

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 $s \in P_A$ under $- \equiv_A -$ correspond precisely to plays of the form

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

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 15.2.1. A play $s \in P_A$ is interval-sequential linearizable to an equivalence class $[t]_{\equiv}$ of $- \equiv_A -$ if for every $t' \in [t]_{\equiv}$, s is linearizable to t' .

The discussion above promptly lets us prove that

Proposition 15.2.1. $s \in P_A$ is linearizable to $t \in P_A$ if and only if s is interval-sequential linearizable to $[t]_{\equiv}$, the equivalence class of t under $\equiv_A$.

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

$$s \cdot s_P \rightsquigarrow_A t \cdot s_O$$

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

$$t \rightsquigarrow_A t'$$

and therefore

$$s \cdot s_P \rightsquigarrow_A t \cdot s_O \rightsquigarrow_A t' \cdot s_O$$

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 . $\square$

Observe that we already showed the equivalence with happens-before order formulations of linearizability in §15.1.3. 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

$$\alpha:\text{yield} \cdot \alpha':\text{yield} \cdot \alpha:\text{ok}$$

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

1. $\alpha:\text{yield} \cdot \alpha':\text{yield} \cdot \alpha:\text{ok} \cdot \alpha':\text{ok}$ or any equivalent trace under $\equiv_{\dagger\text{Yield}}$;
2. $\alpha:\text{yield} \cdot \alpha:\text{ok}$;
3. $\alpha:\text{yield} \cdot \alpha:\text{ok} \cdot \alpha':\text{yield} \cdot \alpha':\text{ok}$ or $\alpha':\text{yield} \cdot \alpha':\text{ok} \cdot \alpha:\text{yield} \cdot \alpha:\text{ok}$

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.

Chapter 16

An Analysis of Herlihy-Wing Possibilities

Herlihy and Wing [1990] present a methodology for showing objects are Herlihy-Wing linearizable, inspired by ideas from abstract interpretation. Their methodology has been influential in later approaches for verifying linearizable objects, notably Khyzha et al. [2016, 2017]. The key idea behind possibilities is to associate to a concurrent object $\nu' : A$ an abstraction function $a : \nu' \rightarrow \mathcal{P}(S_A)$ which assigns to a play $s \in \nu'$ a corresponding set of possible linearized values taken from a set S_A . These values may be understood as an approximation for the set of states that the object ν' can reach by executing the play s .

A possibility for a play s is then defined as a triple $\langle v, I, R \rangle$ such that $v \in a(s)$, R is a set of responses to some of the pending invocations of s , and I are all the pending invocations in s that would not be completed by any of the responses in R . The proof methodology then consists of four axioms that allow one to derive that $\langle v, I, R \rangle$ is a possibility for a play $s \in \nu'$ by induction on the play s .

These axioms are justified by showing that there exists a derivation that $\langle v, I, R \rangle$ is a valid possibility for a play s if and only if $v \in a(s)$ ¹. In particular, when ν_A is the desired linearized specification, then one can define $S_A = \nu_A$ and $a(s)$ as the set of plays $t \in \nu_A$ such that s is linearizable to t . In this case, one obtains that s is linearizable with respect

1. More work is needed to justify why v being a linearized value is enough to obtain a linearizability proof, which we do not go into detail here and refer the reader to the source for the full account.

to ν_A if and only if there exists a derivation that s admits some possibility. That is to say, there is an equivalence between possibility derivations and proofs of linearizability.

The computational interpretation angle from §13.3 provides an alternative equivalence of linearizability proofs with plays of copy_A . This motivates analyzing Herlihy-Wing’s possibility methodology from the computational interpretation angle.

16.1 Copycat Positions

The key idea of our approach is to define a transition system $\mathcal{CC}_A$ for any game A such that:

$$\llbracket \mathcal{CC}_A \rrbracket = \text{copy}_A : A \multimap A$$

Crucially, this transition system is engineered to be quite minimal. In particular, we could have taken the copycat program (§6.3) instead and used an operational semantics to do this. Instead, we handcraft a transition system that is useful for reasoning.

Indeed, we let $\mathcal{CC}_A := (\text{Pos}_A, M_{A \multimap A}^{\bullet \Upsilon}, - \rightarrow_{\mathcal{CC}} -)$. Here, a copycat *position* is defined as follows. First, each agent will be assigned a linearization state linstate_A over A consisting of one of the following:

$$\text{linstate}_A := \text{idle} \mid \text{inv}(q) \mid \text{lin}(q) \mid \text{lin}(q, a) \mid \text{stv}$$

What this entails is that an agent can either be:

- *idle*, when it is inactive;
- *inv*(q), when it has a pending invocation q in the target component;
- *lin*(q), when it has a pending invocation q in the source component;
- *lin*(q, a), when it has both the invocation q and response a in the source component;
- *stv*, after it has been starved by the scheduler.

The set of positions is then given by:

$$\text{Pos}_A := P_A^{\bullet\Upsilon} \times (\Upsilon \rightarrow \text{lstate}_A)$$

and we write (s, π) for a position in Pos_A . The play $s \in P_A^{\bullet\Upsilon}$ is only used to make sure that moves are valid. Were we using a game such as $\dagger E$, where the information provided by the signature and the mapping π would be enough to determine validity, we could drop the play s from the positions.

The transition system for positions is then given by the following rules:

$$\begin{array}{c} \text{T-INVOKE} \frac{\pi[\alpha] = \text{idle} \quad \pi_\alpha(s) \cdot q \in P_A}{(s, \pi) \xrightarrow{\alpha:\text{tgt } q} (s, \pi[\alpha \mapsto \text{inv}(q)])} \qquad \text{T-RETURN} \frac{\pi[\alpha] = \text{lin}(q, a)}{(s, \pi) \xrightarrow{\alpha:\text{tgt } a} (s, \pi[\alpha \mapsto \text{idle}])} \\[10pt] \text{S-INVOKE} \frac{\pi[\alpha] = \text{inv}(q)}{(s, \pi) \xrightarrow{\alpha:\text{src } q} (s \cdot \alpha:q, \pi[\alpha \mapsto \text{lin}(q)])} \qquad \text{S-RETURN} \frac{\pi[\alpha] = \text{lin}(q) \quad s \cdot a \in P_A}{(s, \pi) \xrightarrow{\alpha:\text{src } a} (s \cdot a, \pi[\alpha \mapsto \text{lin}(q, a)])} \\[10pt] \text{STARVE} \frac{}{(s, \pi) \xrightarrow{\alpha:\bullet} (s \cdot \alpha:\bullet, \pi[\alpha \mapsto \text{stv}])} \end{array}$$

It is straightforward to check that

$$\llbracket \mathcal{CC}_A \rrbracket = \text{copy}_A : A \multimap A$$

16.2 Possibilities

It is useful, in practice, to assume the specification $\nu : A$, which we wish to linearize to, is generated by some transition system $\mathcal{V} = (S_{\mathcal{V}}, M_A^{\bullet\Upsilon}, - \xrightarrow{\mathcal{V}} -)$, in the sense that $\llbracket \mathcal{V} \rrbracket_S = \nu : A$. Note that if we wanted to just use a strategy ν , we could reproduce the construction below by using residuals of ν as state, as argued in Ch. 3.

The transition system for possibilities is then given by $\text{Poss}(\mathcal{V}) := \mathcal{V} \odot \mathcal{CC}_A$ (see Ch. 3).

Concretely, states are given by triples

$$\varrho = (\mathbf{st} \in S_{\mathcal{V}}, s, \pi), \quad \text{where } (s, \pi) \in \text{Pos}_A$$

And transitions are given by the extension of the rules for positions in §16.1 with a check for whether or not the transition is valid according to $\mathcal{V}$ (note that after the hiding we have silent transitions $-\dashrightarrow -$):

$$\begin{array}{c} \text{T-INVOKE} \frac{\pi[\alpha] = \text{idle} \quad \pi_{\alpha}(s) \cdot q \in P_A}{(\mathbf{st}, s, \pi) \xrightarrow{\alpha:\text{tgt } q} (\mathbf{st}, s, \pi[\alpha \mapsto \text{inv}(q)])} \quad \text{T-RETURN} \frac{\pi[\alpha] = \text{lin}(q, a)}{(\mathbf{st}, s, \pi) \xrightarrow{\alpha:\text{tgt } a} (\mathbf{st}, s, \pi[\alpha \mapsto \text{idle}])} \\[10pt] \text{S-INVOKE} \frac{\pi[\alpha] = \text{inv}(q) \quad \mathbf{st} \xrightarrow{\alpha:q}_{\mathcal{V}} \mathbf{st}'}{(\mathbf{st}, s, \pi) \dashrightarrow (\mathbf{st}', s \cdot \alpha:q, \pi[\alpha \mapsto \text{lin}(q)])} \\[10pt] \text{S-RETURN} \frac{\pi[\alpha] = \text{lin}(q) \quad s \cdot a \in P_A \quad \mathbf{st} \xrightarrow{\alpha:a}_{\mathcal{V}} \mathbf{st}'}{(\mathbf{st}, s, \pi) \dashrightarrow (\mathbf{st}', s \cdot a, \pi[\alpha \mapsto \text{lin}(q, a)])} \\[10pt] \text{STARVE} \frac{}{(\mathbf{st}, s, \pi) \xrightarrow{\alpha:\bullet} (\mathbf{st}', s \cdot \alpha:\bullet, \pi[\alpha \mapsto \text{stv}])} \end{array}$$

Figure 16.1: Possibility update rules.

The rest of the way toward linearizability proofs is made straightforward by the theory we have developed. Specifically, given a concrete transition system $\mathcal{V}'$ computing to a strategy $\mathcal{V}' : A$, we have:

$$\begin{aligned} \llbracket \mathcal{V}' \rrbracket_S \rightsquigarrow \llbracket \mathcal{V} \rrbracket_S &\iff \\ \llbracket \mathcal{V}' \rrbracket_S \sqsupseteq_S K \llbracket \mathcal{V} \rrbracket_S = \text{copy}_A ; \llbracket \mathcal{V} \rrbracket_S &= \llbracket \mathcal{CC}_A \rrbracket_S ; \llbracket \mathcal{V} \rrbracket_S = \llbracket \mathcal{CC}_A \odot \mathcal{V} \rrbracket_S = \llbracket \text{Poss}(\mathcal{V}) \rrbracket_S \end{aligned}$$

In particular, collecting the remarks above, we obtain that:

Corollary 16.2.1. *$t \in \llbracket \mathcal{V} \rrbracket_S$ is a linearization for $s \in \llbracket \mathcal{V}' \rrbracket_S$ if and only if there exists a path $(\mathbf{st}_0, \epsilon, \emptyset) \xrightarrow{s^*}_{\text{Poss}(\mathcal{V})} (\mathbf{st}, t, \pi)$.*

This can be made into a complete and forward axiomatic proof method, like Herlihy and Wing [1990]’s original approach, to write proofs of compositional linearizability. However, a more practical technique is to turn it into a complete forward simulation technique.

This is achieved as follows. First, it is well-known that trace containment is equivalent to the existence of a forward-backward simulation between transition systems [Lynch and Vaandrager, 1996]. In other words:

$$\llbracket \mathcal{V}' \rrbracket_S \rightsquigarrow \llbracket \mathcal{V} \rrbracket_S \iff \llbracket \mathcal{V} \rrbracket_S \subseteq \llbracket \text{Poss}(\mathcal{V}) \rrbracket_S \iff \exists \mathcal{V}'' . \mathcal{V}' \lesssim \mathcal{V}'' \gtrsim \text{Poss}(\mathcal{V})$$

But it is also well-known [Lynch and Vaandrager, 1996] that a backward simulation $\mathcal{S} \gtrsim \mathcal{S}'$ exists if and only if there is a forward simulation $\mathcal{S} \lesssim \mathcal{P}(\mathcal{S}')$ where $\mathcal{P}(\mathcal{S}')$ is the standard determinization construction due to Rabin and Scott [1959].² But this means that:

$$\llbracket \mathcal{V}' \rrbracket_S \rightsquigarrow \llbracket \mathcal{V} \rrbracket_S \iff \exists \mathcal{V}'' . \mathcal{V}' \lesssim \mathcal{V}'' \lesssim \mathcal{P}(\text{Poss}(\mathcal{V}))$$

Since forward simulations compose (that is, forward similarity is transitive) it follows that:

$$\llbracket \mathcal{V}' \rrbracket_S \rightsquigarrow \llbracket \mathcal{V} \rrbracket_S \iff \mathcal{V}' \lesssim \mathcal{P}(\text{Poss}(\mathcal{V}))$$

This argument as a whole then shows that:

Proposition 16.2.1. *Forward simulation to sets of possibilities is a complete proof technique for compositional linearizability.*

16.3 Linearization Points

The possibilities of Herlihy and Wing [1990] are likely where the intuition for linearization points originally came from. In the original possibilities framework, the **commit** rule is applied precisely when the operation takes effect in the linearization of the concrete trace, which is then called a linearization point. It is folklore that a trace can be linearized if and only if one can find linearization points happening in the interval of each operation which, moreover, can be totally ordered.

2. Succinctly, $\mathcal{P}(\mathcal{S}')$ has as set of states sets of states of $\mathcal{S}'$ and transitions $S \xrightarrow{l} S'$ whenever S' is the set of states reachable from a single transition labelled by l from a state in S . The initial state is the singleton set $\{s_0\}$ with the initial state of $\mathcal{S}$.

Despite this understanding of linearization points which relates to the `commit` rule for possibilities, the apocryphal formalization of the concept relies on showing that a trace is linearizable if and only if there is a monotonic mapping from the operations of a play ordered by happens-before ordering (as formalized in §15.1) to a dense total order such as $\mathbb{R}$ or $\mathbb{Q}$. While this does provide a sort of real time-based understanding of linearization points, it does not characterize them in terms of the computational model itself.

In fact, there are other issues with the notion of linearization point. On one hand, a single trace often admits many choices of linearization points. Once one considers a set of prefix-closed traces that need to be annotated with their linearization points, there is no choice of linearization points for p that guarantees that it is consistent with any s extending that trace ($p \sqsubseteq s$). This is because the new operations in s may invalidate the choice of linearization points made for p . This fact is noted in a different form by Herlihy and Wing [1990] and justifies the use of *sets* of linearized values in their proof methodology. Another issue comes when generalizing from atomicity. It turns out that the *point* intuition is intrinsic to atomicity. In generalized linearizability, one has to settle for a *linearization interval* instead, which may overlap with other *linearization intervals*.

The realization that `copy_` corresponds to linearizability proofs (§13.3) allows us to give a straightforward characterization of linearization intervals. The key idea is that given a strategy $\sigma : A \multimap B$, we can annotate it with the linearization intervals by adding to its source the ability to perform the operation from B in the source, thereby obtaining a strategy $\sigma_{lp} : A \otimes B \multimap B$ which behaves as σ when projected to $A \multimap B$, but behaves as `copyB` when projected to $B \multimap B$. The equivalence between plays of `copyB` and linearizability proofs then allows us to see such extended strategies σ_{lp} as a form of proof-carrying strategy.

This idea works just fine in the atomic case, where we are only allowed to annotate with plays of `intcopyB`. In the general linearizability case, local sequentiality becomes a hindrance: sometimes an agent has to both perform a move in A and B of the same polarity at the same time, thus breaking local sequentiality. So we cannot obtain a proper strategy σ_{lp} in general. We can, however, use labeled transition systems over positions instead of formalizing this idea. We moreover specialize our construction to the assumption

that $\sigma : A \multimap B$ will be running on top of specification $\nu_A : A$ in an attempt to implement a specification $\nu_B : B$, which models the typical verification problem for linearizability. This perspective may be found in my paper Oliveira Vale et al. [2024a]. Here, we present only the account of linearization points for atomic linearizability.

16.3.1 Punctual Extensions under Atomicity

Definition 16.3.1. A *punctual extension* of a strategy $\sigma : (A, \text{atocopy}_A) \multimap (B, \text{copy}_B) \in \text{Kar}$ **Safe** is a strategy

$$\sigma_{\text{lp}} : A \otimes B \multimap B$$

such that

$$\sigma_{\text{lp}} \upharpoonright_{A \otimes -, B} = \sigma \qquad \sigma_{\text{lp}} \upharpoonright_{-, \otimes B, B} \subseteq \text{intcopy}_B$$

The key idea behind a punctual extension is that σ marks in its source component the linearization point of the current target operation by atomically reproducing the corresponding *O* move (which has already happened in the target) and the *P* move (that will happen later in the target). The following two results give a novel formulation of the usual equivalence between Herlihy-Wing linearizability and linearization points. Notably, unlike typical approaches, we do not have to introduce a notion of time (such as by considering intervals in the reals) to characterize linearization points.

Proposition 16.3.1. Let $\nu_A : A \in K_{\text{Atom}}$ and $\nu_B : B \in K_{\text{Atom}}$ and

$$\sigma : A \multimap B \in \text{Safe}$$

Then,

$$\nu_A; \sigma \subseteq \text{Lin}_{\text{Atom}} \nu_B$$

if and only if there exists a punctual extension σ_{lp} of $\text{atocopy}_A; \sigma$ such that

$$((\nu_A \otimes \text{copy}_B); \sigma_{\text{lp}}) \upharpoonright_{-, \otimes B, -} \subseteq \nu_B$$

Proof. See Proof 31.10.2. □

The following corollary provides the usual equivalence of linearization points with atomic linearizability. It states that a concurrent strategy ν'_A is linearizable to an atomic specification ν_A if and only if one can find a certain punctual extension ρ . By the definition of punctual extension, by necessity, it must be that ρ is in fact a substrategy of intcopy .

Corollary 16.3.1. *A strategy $\nu'_A : A \in \mathbf{Safe}$ is linearizable to a strategy $\nu_A : A \in K_{\text{Atom}}$ if and only if ν'_A supports a punctual extension*

$$\rho : \mathbf{1} \otimes A \multimap A$$

such that

$$\rho \upharpoonright_{-\otimes A, -} \subseteq \nu_A$$

Chapter 17

A Complete Program Logic for Compositional Linearizability

In this chapter I present a complete program logic for compositional (safe) linearizability. For this, we use the programming language defined in Ch. 9. One straightforward way to obtain a concurrent semantics for such a programming language is to interpret the module $M : E \rightarrow F$ as:

$$\llbracket M \rrbracket_S := \parallel_{\alpha \in \Upsilon} \iota_\alpha(\llbracket M \rrbracket)$$

This makes the interpretation of these modules into *parallel implementations* in the sense of Ch. 26. In addition, we can also generalize it so that each agent α has its own module implementation $M[\alpha] : E \rightarrow F$. In other words, a concurrent module $M : E \rightarrow F$ consists of a collection $(M[\alpha])_{\alpha \in \Upsilon}$ of a sequential module $M[\alpha] : E \rightarrow F$ for each $\alpha \in \Upsilon$. The formula above then becomes:

$$\llbracket M \rrbracket_S := \parallel_{\alpha \in \Upsilon} \iota_\alpha(\llbracket M[\alpha] \rrbracket)$$

For the sake of reasoning in the program logic, we prefer instead to lift the operational semantics presented in Ch. 9 into a concurrent operational semantics, which is how we start the chapter. Then, we present our rely-guarantee program logic. At its inception, it was thought that the program logic was complete [Oliveira Vale et al., 2023], although this

$$\begin{array}{c}
\longrightarrow \subseteq (\text{Cont} \times \text{ModState}) \times \text{CMod} \times (\text{Cont} \times \text{ModState}) \\
\\
\frac{f \in F \quad a \in \text{par}(f) \quad \Delta' = \Delta[\alpha \mapsto [\arg \mapsto a]] \quad \iota \xrightarrow{+}_\alpha \iota'}{\langle c[\alpha \mapsto \text{idle}], \Delta, \iota, s \rangle \xrightarrow{\alpha:f}_M \langle c[\alpha \mapsto (M[\alpha]^f, \perp)], \Delta', \iota', s \cdot \alpha:f \rangle} \text{INV} \\
\\
\frac{\iota \xrightarrow{+}_\alpha \iota' \quad \langle (C, B), \Delta, \pi_\alpha(s) \upharpoonright_{\dagger E} \rangle \xrightarrow{m} \langle (C', B'), \Delta', \pi_\alpha(s) \upharpoonright_{\dagger E} \cdot m \rangle}{\langle c[\alpha \mapsto (C, B)], \Delta, \iota, s \rangle \xrightarrow{\alpha:m}_M \langle c[\alpha \mapsto (C', B')], \Delta', \iota', s \cdot \alpha:m \rangle} \text{STEP} \\
\\
\frac{\pi_\alpha(s \upharpoonright_F) = p \cdot f \quad v \in \text{ar}(f) \quad \iota \xrightarrow{-}_\alpha \iota'}{\langle c[\alpha \mapsto (\text{ret } v, \perp)], \Delta, \iota, s \rangle \xrightarrow{\alpha:v}_M \langle c[\alpha \mapsto \text{idle}], \Delta, \iota', s \cdot \alpha:v \rangle} \text{RET} \\
\\
\frac{c[\alpha] \neq \bullet \quad \iota \xrightarrow{-}_\alpha \iota'}{\langle c, \Delta, \iota, s \rangle \xrightarrow{\alpha:\bullet}_M \langle c[\alpha \mapsto \bullet], \Delta, \iota', s \cdot \alpha:\bullet \rangle} \text{STARVE}
\end{array}$$

Figure 17.1: Module Small-step semantics ($\longrightarrow$)

was not proven. It has since been mechanized and proven complete by Hatti et al. [2026]. The program logic essentially wraps the possibility reasoning technique described in Ch. 16 into a rely-guarantee program logic, so it is recommended that the reader first check the chapters referenced above before reading this section.

17.1 Concurrent Operational Semantics

Lifting operational semantics from Ch. 9 into a concurrent operational semantics is mostly straightforward, by just keeping track of a continuation for each $\alpha \in \Upsilon$ and having each whole-module semantics rule in Ch. 9 non-deterministically pick an agent to execute for the next step. A key complication that arises for liveness, however, is that we must guarantee that over an infinite execution any agent that has not been starved by the scheduler eventually takes a step. For this, we add a notion of a scheduler counter. It plays no role in generating finite execution traces though (in the sense that were we to remove them from the rules they would generate the same set of finite executions). We mark these aspects in blue and they may be ignored for the time being.

Fig. 17.1 shows the concurrent version of the module small-step semantics. Crucially, a continuation c is now given by a map $c : \Upsilon \rightarrow \text{Cont}$ (where Cont is the same Cont defined in

Ch. 9), and similarly, a local environment Δ is given by a map $\Delta : \Upsilon \rightarrow \text{Env}$, where Env is as was defined in Ch. 9. Finally, the state now is a quadruple (c, Δ, ι, s) of a continuation mapping c , a local environment mapping Δ , a scheduler counter (to be explained later in Part IV), and a state s represented by a trace $s \in P_{\dagger E \multimap \dagger F}^\bullet$. This is enough to define an alternative way to compute the denotation:

$$\llbracket M \rrbracket_S : \dagger E \multimap \dagger F := \{s \mid \exists c \in \text{Cont}. \exists \Delta \in (\Upsilon \rightarrow \text{Env}). \langle c_0, \Delta_0, \iota_0, \epsilon \rangle \xrightarrow{s}^M \langle c, \Delta, \iota, s \rangle\}$$

A concurrent program M can be linked with a specification for its library E , which we denote by $\text{Link } \mathcal{V}_E; M$. Say we are given a state transition system $\mathcal{V}_E := ((S_E, M_{\dagger E}^\bullet, - \xrightarrow{\quad}_E -), s_0)$, well-formed in that $\nu_E = \llbracket \mathcal{V}_E \rrbracket_S$ defines a strategy $\nu_E : \dagger E$. The operational semantics of $\text{Link } \mathcal{V}_E; M$ is given by:

$$\begin{aligned} & \xrightarrow{\quad}_{\mathcal{V}_E} \subseteq (\text{Cont} \times \text{ModState} \times S_E) \times \text{CMod} \times (\text{Cont} \times \text{ModState} \times S_E) \\ & \frac{f \in F \quad a \in \text{par}(f) \quad \Delta' = \Delta[\alpha \mapsto [\arg \mapsto a]] \quad \iota \xrightarrow{+}_{\alpha} \iota'}{\langle c[\alpha \mapsto \text{idle}], \Delta, \iota, s, \text{st}_E \rangle \xrightarrow{\alpha:f}^M_{\mathcal{V}_E} \langle c[\alpha \mapsto (M[\alpha]^f, \perp)], \Delta', \iota', s \cdot \alpha:f, \text{st}_E \rangle} \text{INV} \\ & \frac{\iota \xrightarrow{+}_{\alpha} \iota' \quad \text{st}_E \xrightarrow{\alpha:m}_E \text{st}'_E}{\langle (C, B), \Delta, \pi_\alpha(s) \upharpoonright_{\dagger E} \rangle \xrightarrow{m} \langle (C', B'), \Delta', \pi_\alpha(s) \upharpoonright_{\dagger E} \cdot m \rangle} \text{STEP} \\ & \frac{\langle c[\alpha \mapsto (C, B)], \Delta, \iota, s, \text{st}_E \rangle \xrightarrow{\alpha:m}^M_{\mathcal{V}_E} \langle c[\alpha \mapsto (C', B')], \Delta', \iota', s \cdot \alpha:m, \text{st}'_E \rangle}{\langle c[\alpha \mapsto (\text{ret } v, \perp)], \Delta, \iota, s, \text{st}_E \rangle \xrightarrow{\alpha:v}^M_{\mathcal{V}_E} \langle c[\alpha \mapsto \text{idle}], \Delta, \iota', s \cdot \alpha:v, \text{st}_E \rangle} \text{RET} \\ & \frac{c[\alpha] \neq \bullet \quad \iota \xrightarrow{-}_{\alpha} \iota'}{\langle c, \Delta, \iota, s, \text{st}_E \rangle \xrightarrow{\alpha:\bullet}^M_{\mathcal{V}_E} \langle c[\alpha \mapsto \bullet], \Delta, \iota', s \cdot \alpha:\bullet, \text{st}_E \rangle} \text{STARVE} \end{aligned}$$

From a linked program $\text{Link } \mathcal{V}_E; M$, we can obtain a corresponding strategy

$\llbracket \text{Link } \mathcal{V}_E; M \rrbracket_S : \dagger F$ similarly to before:

$$\llbracket \text{Link } \mathcal{V}_E; M \rrbracket_S = \{s \mid \exists c \in \mathbf{Cont}. \exists \Delta \in (\Upsilon \rightarrow \mathbf{Env}). \langle c_0, \Delta_0, \ell_0, \epsilon, s_0 \rangle \xrightarrow[\mathcal{V}_E]{s}^M \langle c, \Delta, \ell, s, \mathbf{st}_E \rangle\}$$

This, similarly to §9, satisfies that:

$$\llbracket \text{Link } \mathcal{V}_E; M \rrbracket_S = \llbracket \mathcal{V}_E \rrbracket_S ; \llbracket M \rrbracket_S$$

17.2 Program Logic

The program logic accepts the following three parameters: a library specification $\mathcal{V}_E$ (called the underlay) satisfying that $\nu_E := \llbracket \mathcal{V}_E \rrbracket_S : \dagger E$, a module specification $\mathcal{V}_F$ (called the overlay) satisfying that $\nu_F := \llbracket \mathcal{V}_F \rrbracket_S : \dagger F$, and the client policy $\xi \subseteq P_{\dagger F}^1$. The objective of the program logic is to establish the judgement

$$\xi \vdash M : \nu_E \Rightarrow \nu_F$$

which entails that M uses the underlay ν_E to implement an object, which safely linearizes to ν_F under restrictions on both client and scheduler given by ξ . Formally:

$$\xi \cap (\nu_E ; \llbracket M \rrbracket) \rightsquigarrow \nu_F$$

Example 17.2.1 (Client policies). For example, ξ_{lock} defined as follows only considers traces where successful acquires are followed by release invocations and without any starvation token.

$$t \in \xi_{\text{lock}} \iff \bullet \notin t \wedge \forall \alpha, t', t''. \pi_\alpha(t \upharpoonright_{\text{Lock}}) = t' \cdot \text{acq} \cdot \text{ok} \cdot t'' \Rightarrow \text{rel} \sqsubseteq t''$$

1. Formally, this should be a costrategy. The requirements are not quite important, however, so we ignore them.

The program logic uses as proof configurations tuples

$$(\Delta, \iota, s, \text{st}_E, \rho) \in \text{Config} := \text{ModState} \times S_E \times \mathcal{P}(\text{Poss}(\mathcal{V}_F))$$

This means that pre-conditions are given by sets $P \subseteq \text{Config}$, post-conditions by binary predicates $Q \subseteq \text{Config} \times \text{Config}$, as well as rely and guarantee conditions $\mathcal{R}, \mathcal{G} \subseteq \text{Config} \times \text{Config}$. We define stability requirements on pre-conditions P and post-conditions Q as usual:

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

As explained in Ch. 16, we use a set of possibilities to allow for future-dependent linearizations. Explicitly, this means that we lift the rules seen in Fig. 16.1 to sets of possibilities in the following way:

$$\begin{aligned} \rho \dashrightarrow \rho' &\iff \forall (\text{st}'_E, p', \pi') \in \rho'. \exists (\text{st}_E, p, \pi) \in \rho. (\text{st}_E, p, \pi) \dashrightarrow (\text{st}'_E, p', \pi') \\ \rho \xrightarrow{e} \rho' &\iff \forall (\text{st}'_E, p', \pi') \in \rho'. \exists (\text{st}_E, p, \pi) \in \rho. (\text{st}_E, p, \pi) \xrightarrow{e} (\text{st}'_E, p', \pi') \end{aligned}$$

In addition, the user needs to maintain the invariant I1 that at all points the set of possibilities is non-empty:

$$\rho \neq \emptyset \quad (\text{I1})$$

There are three ways through which a configuration can be modified: through a relational predicate $\text{invoke}_\alpha(-)$ which makes an invocation in F , and simultaneously adds it to the state and the possibility; a *commit* rule $\mathcal{G} \models_\alpha \{P\} B \{Q\}$, where $B \in \text{Prim}$, which allows one to modify the state by executing primitive commands over E , but also to commit events to ρ and add or remove possibilities from the current set of valid possibilities; and a post-condition $\text{return}(-)$ that checks if at the end of execution there is a valid return in the possibility, and then adds it to the state.

The key rule of the program logic, which verifies an atomic step (either an invocation

or a response by O or P), is the following commit rule.

$$\begin{aligned}
& \mathcal{G} \models_{\alpha} \{P\} B \{Q\} \iff \\
& \forall \Delta, \iota, s, \text{st}_E, \rho, \Delta', \iota', s', \text{st}_E'. \\
& \left((\Delta, \iota, s, \rho) \in P \wedge \text{st}_E \xrightarrow{m}_{\nu_E} \text{st}_E' \wedge s' = s \cdot m \wedge (\Delta', s') \in \llbracket B \rrbracket(\Delta, s) \wedge \iota \xrightarrow{+}_{\alpha} \iota' \right) \\
& \implies \exists \rho'. (\Delta, \iota, s, \text{st}_E, \rho) Q (\Delta', \iota', s', \text{st}_E', \rho') \\
& \wedge (\Delta, \iota, s, \text{st}_E, \rho) \mathcal{G} (\Delta', \iota', s', \text{st}_E', \rho') \wedge \rho \dashrightarrow^* \rho'
\end{aligned}$$

Most of the rule is common to any Hoare logic or rely-guarantee program logic. The key difference is that we allow the user to choose a new set of possibilities at this point. This allows them to update the set of possibilities that are currently considered valid. Of course, at this point, they must re-establish invariant I1 (and later, any other invariants we add).

When applying the effect of the primitive command, the prover can choose to update the possibility through possibility updates. The commit rule is used in the UNDERLAY rule for verifying underlay operations and the SILENT rule for all other primitive rules. The UNDERLAY rule applies the commit rule twice because we treat the underlay invocation event and return event as two atomic events. Interactions may take place in between if the underlay specification admits non-atomic behaviors. For atomic underlays, the underlay part of Q_O will not be changed by the rely and the prover can use a much stronger intermediate post-condition, e.g., the pre-condition P , as if only one commit rule is applied. The silent rule provides a more convenient rule for operations such as `skip` and `assume(-)`, or other silent rules that might be added to the programming language.

$$\begin{array}{c}
\frac{\text{stable}(\mathcal{R}, P) \quad \text{stable}(\mathcal{R}, Q^O) \quad \text{stable}(\mathcal{R}, Q^P) \quad \mathcal{G} \models_{\alpha} \{P\} x \leftarrow e(v) \{Q^O\} \quad \mathcal{G} \models_{\alpha} \{Q^O \circ P\} x \leftarrow e(v) \{Q^P\}}{\xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} x \leftarrow e(v) \{Q^P \circ Q^O\}} \text{ UNDERLAY} \\
\\
\frac{\text{stable}(\mathcal{R}, P) \quad \text{stable}(\mathcal{R}, Q) \quad \mathcal{G} \models_{\alpha} \{P\} B \{Q\} \quad \text{silent}(B)}{\xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} B \{Q\}} \text{ SILENT}
\end{array}$$

Another important rule is the RETURN rule. It first checks if all possibilities have the current thread's invocation and response linearized. Then, it updates the concrete trace and performs the system update S-RETURN to reset the state of the possibility.

$$\begin{array}{c}
\frac{\forall(-, -, -, -, \rho) \in P, (-, -, \pi) \in \rho. \pi[\alpha] = \text{lin}(q, a) \quad \text{return}(\alpha, v, P, Q, \mathcal{G})}{\xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} \text{ret } v \{Q\}} \text{ RETURN} \\
\\
\text{return}(\alpha, v, P, Q, \mathcal{G}) \iff \\
\left(\begin{array}{l} \forall(\Delta, \iota, s, \text{st}_E, \rho) \in P. \forall \Delta', \iota', s', \text{st}'_E, \rho'. \\ \iota \xrightarrow{\alpha} \iota' \wedge (\Delta', s') \in \llbracket \text{ret } v \rrbracket(\Delta, s) \wedge \rho \xrightarrow{\alpha: \text{ret } v} \rho' \end{array} \right) \\
\implies \left(\begin{array}{l} (\Delta, \iota, s, \text{st}_E, \rho) \ Q \ (\Delta', \iota', s', \text{st}'_E, \rho') \wedge \\ (\Delta, \iota, s, \text{st}_E, \rho) \ \mathcal{G} \ (\Delta', \iota', s', \text{st}'_E, \rho') \end{array} \right)
\end{array}$$

Other proof rules that induct over the grammar of commands are shown in Fig. 17.2.

$$\begin{array}{c}
\frac{\xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} C_1 \{Q_1\} \quad \xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{Q_1 \circ P\} C_2 \{Q_2\}}{\xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} C_1; C_2 \{Q_2 \circ Q_1\}} \text{ SEQ} \\
\\
\frac{\text{stable}(\mathcal{R}, P)}{\xi, \mathcal{R}, \text{ID} \vdash_{\alpha} \{P\} \text{skip} \{\text{ID}\}} \text{ SKIP} \quad \frac{\xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} C_1 \{Q\} \quad \xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} C_2 \{Q\}}{\xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} C_1 + C_2 \{Q\}} \text{ CHOICE} \\
\\
\frac{P' \subseteq P \quad Q \subseteq Q' \quad \text{stable}(\mathcal{R}', P') \quad \mathcal{R}' \subseteq \mathcal{R} \quad \text{stable}(\mathcal{R}', Q') \quad \mathcal{G} \subseteq \mathcal{G}' \quad \xi, \mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} C \{Q\}}{\xi, \mathcal{R}', \mathcal{G}' \vdash_{\alpha} \{P'\} C \{Q'\}} \text{ CONSEQ}
\end{array}$$

Figure 17.2: Other Proof Rules of HLHL

Finally, the LOCAL IMPL and CONC IMPL rules are two top-level rules showing the

correctness of the entire module.

$$\begin{array}{c}
\forall f \in F. \xi, \mathcal{R}[\alpha], \mathcal{G}[\alpha] \vdash_{\alpha}^f \{P[\alpha]\} M[\alpha]^f \{Q[\alpha]\} \\
\text{stable}(\mathcal{R}[\alpha], I[\alpha]) \quad \text{stable}(\mathcal{R}[\alpha], P[\alpha]) \quad \text{stable}(\mathcal{R}[\alpha], Q[\alpha]) \\
\text{invoke}(\alpha, f) \circ I[\alpha] \Rightarrow P[\alpha] \quad Q[\alpha] \circ P[\alpha] \Rightarrow \text{returned}(\alpha) \wedge I[\alpha] \\
\hline
\xi, \mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vdash_{\alpha} M_F[\alpha] \quad \text{LOCAL IMPL} \\
\\
(\Delta_0, \iota_0, \epsilon, \text{st}_{E,0}, \rho_0) \in I[\alpha] \quad \forall \alpha \in \Upsilon. \xi, \mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \models_{\alpha} M[\alpha] \\
\forall \alpha \neq \alpha' \in \Upsilon. \mathcal{G}[\alpha] \cup (\cup_{f \in F} \text{invoke}(\alpha, f)) \subseteq \mathcal{R}[\alpha'] \\
\hline
\xi \vdash M : \nu_E \Rightarrow \nu_F \quad \text{CONC IMPL}
\end{array}$$

$$\begin{aligned}
& (\Delta, \iota, s, \text{st}_E, \rho) \text{ invoke}(\alpha, f) (\Delta', \iota', s', \text{st}_E, \rho') \iff \\
& \exists a \in \text{par}(f). \left(\begin{array}{l} \Delta' = \Delta[\alpha \mapsto [\arg \mapsto a]] \wedge \iota \xrightarrow{+}_{\alpha} \iota' \wedge \\ s' = s \cdot \alpha : f(a) \wedge \rho \xrightarrow{\alpha : f(a)} \rho' \end{array} \right)
\end{aligned}$$

$$(\Delta, \iota, s, \text{st}_E, \rho) \in \text{returned}(\alpha) \iff \forall (p, \pi) \in \rho. \pi[\alpha] = \text{idle}$$

Of note are the premises

$$\text{invoke}(\alpha, f) \circ I[\alpha] \Rightarrow P[\alpha] \quad Q[\alpha] \circ P[\alpha] \Rightarrow \text{returned}(\alpha) \wedge I[\alpha]$$

of the rule 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 premise

$$\mathcal{G}[\alpha] \cup (\cup_{f \in F} \text{invoke}(\alpha, f)) \subseteq \mathcal{R}[\alpha']$$

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 17.2.1 (Soundness). *If $M : E \rightarrow F$ and $\mathcal{V}_E$ and $\mathcal{V}_F$ are well-formed, then*

$$\xi \vdash M : \nu_E \Rightarrow \nu_F \implies \xi \cap (\llbracket \mathcal{V}_E \rrbracket_S ; \llbracket M \rrbracket_S) \leadsto \llbracket \mathcal{V}_F \rrbracket_S$$

The program logic has been proven complete as well for compositional linearizability, formally:

Proposition 17.2.2 (Completeness). *For well-formed $\mathcal{V}_E$, $\mathcal{V}_F$ and $M : E \rightarrow F$, and given a client policy ξ ,*

$$\xi \cap (\llbracket \mathcal{V}_E \rrbracket_S ; \llbracket M \rrbracket_S) \leadsto \llbracket \mathcal{V}_F \rrbracket_S \implies \xi \vdash M : \nu_E \Rightarrow \nu_F$$

This is proven by Hatti et al. [2026], and the proof as well as a mechanization of many of the results of the program logic described here in Part II may be found there.

Part III

Crash-Safety

Chapter 18

Introduction

Ch. 14 develops a generic account of linearizability. We validated this account in a model that (except for the presence of starvation tokens) matches the original setting of linearizability by showing it conservatively generalizes the classical notion of atomic linearizability [Herlihy and Wing, 1990], despite the theory having fundamentally different definitions. This not only provides a way to study and work with traditional notions of linearizability, but also to develop novel notions of linearizability. The key advantage is that we do not need to invent a new criterion for every new model of computation, we simply prove the model is compositional (i.e., forms an enriched semicategory with certain properties) and compositional linearizability provides the recipe of what the corresponding linearizability-like criterion should be.

To showcase that this is sensible, in that it produces reasonable correctness criteria in new settings, in this chapter, we develop a compositional account of linearizability under full-system crashes by following the compositional linearizability approach. By a full-system crash, we mean a crash that results in all agents of a system failing or being reset. This could result from a power outage, a user holding the power button on their computer, a fatal crash in an OS, a critical component failure, etc. By compositional, we mean that verified components can be freely composed vertically and horizontally so that the composed system is *correct by construction*, in that no side conditions are necessary to derive its correctness from the correctness of its components. As a result, we obtain a framework for verifying large-scale crash-aware systems against linearizability. To see why

compositionality is important, consider one of our main examples: the FLiT library [Wei et al., 2022].

18.1 The FLiT Library

Implementing persistent data structures, even when non-volatile memory (NVM) is available, is notoriously challenging. For instance, one of the challenges when programming with NVM is that it provides a buffered interface **BCell**. We can encapsulate the operations of a buffered memory cell in the following signature, where $\mathbf{1}$ stands for some singleton set (we will write $() \in \mathbf{1}$ if it is an argument, and $\text{ok} \in \mathbf{1}$ if it is a return) and $\mathbf{Val}$ is a set of memory values:

$$\mathbf{BCell}[\mathbf{Val}] := \{\text{rd} : \mathbf{1} \rightarrow \mathbf{Val}, \text{wrt} : \mathbf{Val} \rightarrow \mathbf{1}, \text{flush} : \mathbf{1} \rightarrow \mathbf{1}\}$$

What this signature expresses is that **BCell** provides three operations: $\text{rd}()$, which takes unit $() \in \mathbf{1}$ as argument and returns some value in $\mathbf{Val}$; $\text{wrt}(v)$, which takes a value $v \in \mathbf{Val}$ as argument, and returns the unit $\text{ok} \in \mathbf{1}$; and $\text{flush}()$, which takes unit $()$ as argument and returns a unit ok . The signature **BCell** provides the syntax of the operations of a buffered memory cell. It must be paired with a specification defined later, which provides the semantics of the operations. Such a specification would state that wrts are not guaranteed to persist immediately; instead, they are buffered and persist only when the buffer is non-deterministically flushed or explicitly flushed by a $\text{flush}()$ invocation [Raad et al., 2019b,a]. In other words, once a crash happens, a rd is only guaranteed to read a value no older than the latest flush. The explicit flush operation guarantees a buffer flush at a significant performance cost, so in practice, one would like to minimize its usage. For instance, in the trace (where $\alpha_0, \alpha_1, \alpha_2$, and α_3 are the names of the agents performing the operations):

$$\alpha_0:\text{wrt}(0) \cdot \alpha_0:\text{ok} \cdot \alpha_0:\text{flush}() \cdot \alpha_0:\text{ok} \cdot \alpha_1:\text{wrt}(1) \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{rd}() \cdot \alpha_2:v \cdot \frac{1}{2} \cdot \alpha_3:\text{rd}() \cdot \alpha_3:v'$$

the value v must be $v = 1$, as 1 is currently the buffered value. Meanwhile, either $v' = 0$ (the value at the latest flush), or $v' = 1$ (which could have been non-deterministically flushed from the buffer). This non-determinism of the value of a load after a crash complicates programming with NVM.

Some works attempt to facilitate programming persistent data structures by providing more robust persistent objects than those available directly from the underlying NVM, which usually only provides buffered memory cells. One such work is FLiT, a C++ library which provides a wrapper for the BCell operations. Specifically, in its essence, FLiT provides an object with signature

$$\text{FLiT} = \text{Cell}[\text{Val}] := \{\text{rd} : \mathbf{1} \rightarrow \text{Val}, \text{wrt} : \text{Val} \rightarrow \mathbf{1}\}.$$

Traditionally with linearizability, the set of concrete traces ν' differs from its linearized specification ν only in that ν consists of fewer concurrent traces (often atomic, i.e., traces where every invocation is immediately followed by its response). In the context of durable linearizability [Izraelevitz et al., 2016], a correctness criterion for persistent objects, ν' also differs from its linearized specification ν in that ν' has explicit crashes while ν does not. More precisely, durable linearizability requires that $\text{ops}(\nu')$, the crash-less specification obtained by removing all crash events from traces in ν' , is linearizable (in the usual sense) w.r.t. ν .

We specify the FLiT object ν'_{FLiT} to be durably linearizable to ν_{FLiT} , the usual crash-less atomic memory cell. This should be understood as stating that the FLiT operations are persistent in ν'_{FLiT} , meaning that a load after a crash does read the most recently written value, up to happens-before reordering. FLiT's implementation M_{FLiT} , which runs on top of a buffered memory cell object ν'_{BCell} and of a volatile counter object ν'_{Counter} , does this by: (1) always flushing wrts; (2) using the counter to keep track of when flushes are necessary; (3) having rds only flush when the counter marks that a flush is necessary. The counter specified by ν'_{Counter} is volatile in that it lives in volatile memory, so after a crash, a new instance is created with the initial value of 0. The code M_{FLiT} for our simplified formulation of FLiT is found below in Fig. 18.1.

For instance, a buffered memory cell allows for the following trace:

$$\alpha_0:\text{wrt}(1) \cdot \alpha_1:\text{rd}() \cdot \alpha_1:1 \cdot \frac{1}{2} \cdot \alpha_2:\text{rd}() \cdot \alpha_2:v$$

where either $v = 0$ (when the buffer containing 1 has not flushed before the crash) or $v = 1$ (when the buffer is flushed before the crash). If $v = 0$, the trace is not durably linearizable to the usual memory cell specification because a 0 is read after 1 is read with no $\text{wrt}(0)$ to justify it.

```

MFLiT:FLiT
Import B:BCell
Import C:Counter

load()
{
  v ← B.load();
  if(C.get() != 0)
  { B.flush(); }
  return v;
}

store(v)
{
  C.inc();
  B.store(v);
  B.flush();
  C.dec();
  return; }

```

Figure 18.1: FLiT Memory Cell Implementation M_{FLiT}

Meanwhile, when using FLiT, the call to `wrt(1)` must execute at least up to the `B.wrt(1)` invocation (as `rd()` manages to read 1). This means that the call to `wrt(1)` will have executed `C.inc()`. Assuming it only executes up to receiving the response `ok` to its `B.wrt(1)` call (otherwise, it executes a flush), the α_1 :`rd()` call will execute to completion, so it will call `B.rd()` and receive 1 as response. Then, it will read 1 from `C.get()`, and will execute `B.flush()` before returning 1. Hence, when the crash $\downarrow$ happens, the buffered memory cell has been flushed, guaranteeing that any `rd()` calls after the crash will read 1. Therefore, calling the memory operations using FLiT guarantees that $v = 1$.

The FLiT paper claims that: “Using the library’s default mode makes any linearizable data structure durable [...]”, which they do not prove. In fact, it is challenging to state this theorem without a compositional model of crash-aware computation, as it concerns discussing the composition of arbitrary clients with FLiT. In addition, even if such a compositional model were available, it must provide good support for durable linearizability and be closely connected with a concurrent compositional model without crashes, also providing good support for usual linearizability [Herlihy and Wing, 1990]. The reason for this is that this statement relates an implementation that assumes the usual concurrent memory and implements a linearizable object, with an implementation that runs on top of the crash-aware FLiT library and implements a durably linearizable object. In this chapter, I build a framework for the verification of concurrent systems with crashes which allows for the correctness of FLiT to be stated in full formality, and for it to be proved and used to build provably correct durable components using a crash-less component (i.e., one whose specification does not involve crashes) which has been previously verified against a linearizability specification.

Using our compositional account of linearizability with crashes, we prove the following FLiT correctness theorem (ν'_{Cell} is any crash-less object Herlihy-Wing linearizable to $\nu_{\text{FLiT}} = \nu_{\text{Cell}[\text{Val}]}$ defined in Ex. 6.1.3).

Proposition 18.1.1 (FLiT Correctness). *For any object signature E , writing $\nu'_{\text{Mem}[\text{Val}]} := \otimes_{i \in I} \nu'_{\text{Cell}[\text{Val}]}$ for the horizontal composition of several memory cells, if $\llbracket M \rrbracket : \nu'_{\text{Mem}[\text{Val}]} \Rightarrow \nu_E \in \mathbf{Safe}$ then, writing $\nu'_{\text{BMem}[\text{Val}]} := \otimes_{i \in I} \nu'_{\text{BCell}[\text{Val}]}$, it follows that $\nu'_{\text{BMem}[\text{Val}]}; \llbracket M_{\text{FLiT}} \oplus M \rrbracket^\sharp$ is durably linearizable to ν_E .*

Note that, following our understanding of the $- \otimes -$, ν'_{Mem} defines a memory array. M is code implementing a new object with signature E using the memory array ν'_{Mem} . Similarly, ν'_{BMem} is a buffered memory array. $\llbracket M \rrbracket^\sharp$ is a version of $\llbracket - \rrbracket$ for modules that adds crash semantics to M by running it in each epoch (the period in between crashes), so that $\llbracket M_{\text{FLiT}} \oplus M \rrbracket^\sharp$ is the interpretation with crash semantics of the result of linking the two modules M_{FLiT} and M .

We prove this by developing a compositional theory of durable linearizability which supports both locality and observational refinement (where we write $\nu' \text{dur} \nu$ for “ ν' is durably linearizable to ν ”), proving locality and observational refinement properties for durable linearizability, and then introducing a program logic for verifying individual components to be durably linearizable. Using our program logic, we show that (depicted diagrammatically in Fig. 18.2 (a)):

Proposition 18.1.2. *$(\nu'_{\text{Counter}} \otimes \nu'_{\text{BCell}}); \llbracket M_{\text{FLiT}} \rrbracket^\sharp$ is durably linearizable with respect to ν_{FLiT} .*

$(\nu'_{\text{Counter}} \otimes \nu'_{\text{BCell}}); \llbracket M_{\text{FLiT}} \rrbracket^\sharp$ is the object obtained by running the code in Fig. 18.1 on top of the volatile counter ν'_{Counter} and the buffered memory cell ν'_{BCell} . By using an observational refinement property for *crash-aware linearizability*, a novel linearizability criterion we introduce, we prove this using instead the linearized specifications for the counter and the buffered memory cell, greatly simplifying its proof by only considering atomic traces. By verifying that M_{FLiT} is durably linearizable, we can use locality (Prop. 18.1.4) and observational refinement (Prop. 18.1.3) to prove FLiT’s correctness.

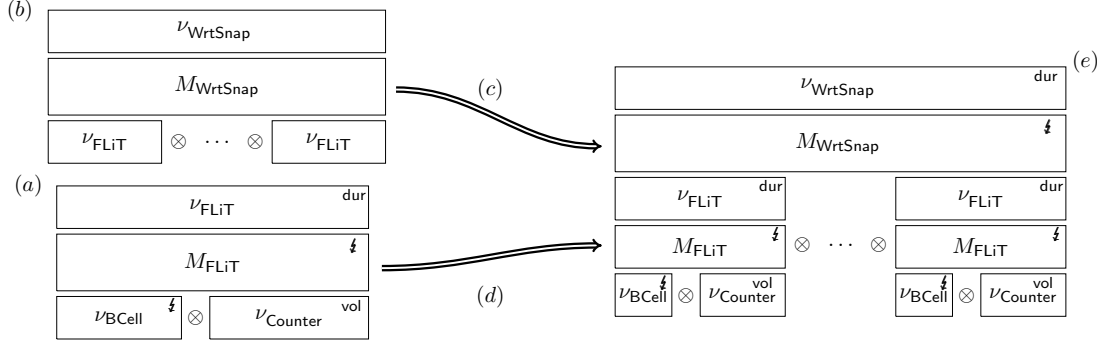


Figure 18.2: (a) Using our program logic for durable linearizability, we verify the FLiT implementation; (b) Using the program logic for compositional linearizability we verify the crash-less snapshot object; (c) Using the FLiT correctness theorem, we lift the crash-less snapshot object into a durably linearizable snapshot object running on top of a FLiT array; (d) Using vertical and horizontal composition, we obtain (e) a durably linearizable snapshot object running on top of an array of buffered memory cells and volatile counters.

Proposition 18.1.3 (Observational Refinement). *An object $\nu'_A : A$ is durably linearizable to ν_A if and only if whenever an implementation M implements a concurrent object linearizable to ν_B using ν_A , $\llbracket M \rrbracket^{\sharp}$ implements an object durably linearizable to ν_B using ν'_A .*

Proposition 18.1.4 (Locality). *For $\nu'_A : A, \nu'_B : B$ and $\nu_A : A, \nu_B : B$:*

$$\nu'_A \xrightarrow{\text{dur}} \nu_A \text{ and } \nu'_B \xrightarrow{\text{dur}} \nu_B \text{ if and only if } \nu'_A \otimes \nu'_B \xrightarrow{\text{dur}} \nu_A \otimes \nu_B$$

While the original paper on durable linearizability [Izraelevitz et al., 2016] claims it satisfies locality, it does not do so by formalizing horizontal composition. Meanwhile, our locality statement is directly formulated within our compositional model of computation with crashes, which is defined independently of any notion of linearizability. This makes our locality theorem much stronger as it interacts well with refinement and vertical composition. Observational refinement, however, had never been shown for any linearizability criterion with crashes. Our program logic is the first to verify any linearizability criterion with crashes. Moreover, as it is necessary to verify the FLiT library, our program logic can reason about external linearization points and helpings [Liang and Feng, 2013], even across crashes, just like the original logic developed for compositional linearizability (Ch. 17).

We further showcase the benefits of compositionality and of the FLiT correctness theorem by showing that we can lift a crash-less interval-sequential linearizable snapshot object

[Castañeda et al., 2015] into a durable one (Fig. 18.2 (b)). We do this by first verifying the write-snapshot implementation M_{WrtSnap} from Borowsky and Gafni [1993] using the program logic of Ch. 17. The implementation uses a (crash-less) memory cell object ν'_{Cell} (which is Herlihy-Wing linearizable to ν_{FLIT}) to implement an object with interface **WrtSnap** with a single operation **wrtsnap**:

$$\text{WrtSnap} := \{\text{wrtsnap} : \text{Val} \rightarrow \mathcal{P}(\text{Val})\}$$

The operation **wrtsnap** writes the current value to the memory and returns a set of values that have been written to the object before. The implementation M_{WrtSnap} uses one memory cell per agent $\alpha \in S$ in the snapshot system to implement the **WrtSnap** object.

Using the soundness theorem for our program logic (Ch. 17), we obtain a crash-less interval-sequential linearizable object. Because we formally connect our model and durable linearizability definition to their model, we can then use the FLiT correctness theorem to obtain that the write-snapshot object is interval-sequential *durably* linearizable in the model with crashes. Connecting the two models is only tractable because they share the same underlying structure, which was discovered as part of the compositional linearizability theory we discussed in Part II. Note that this also showcases that our linearizability criteria and program logic are all generalized to handle interval-sequential objects. We display this setup in Fig. 18.2 (e).

While durable linearizability is a good criterion for specifying persistent objects, it is inept at expressing objects with less persistent behaviors, such as volatile objects, buffered objects, or objects with hybrid crash behaviors (e.g., horizontal compositions of objects with different persistency guarantees). Therefore, we use the methodology of compositional linearizability (Ch. 14) to derive the inherent notion of linearizability for our compositional model, which we call *crash-aware linearizability*. This criterion, though simple, is novel and it satisfies locality and observational refinement. Then, we show that durable linearizability and strict linearizability factor through crash-aware linearizability as different ways to translate crash-aware linearizable objects to the crash-less model from compositional linearizability. We showcase that crash-aware linearizability is a robust verification criterion by verifying a fragment of a transactional file system interface featuring recovery and objects

with many different persistency guarantees.

Chapter 19

A Concurrent Game Semantics with Crashes

In this chapter, we define our game model with crashes. As durable linearizability involves crash-aware objects as well as objects without crashes, which live in different models of computation, it will be essential to connect this crash model with our model from Part II. Since we will not discuss liveness of components with crashes, I will silently omit all aspects of our original model that involve the $\bullet$ -token (and the $\circ$ -token, for that matter).

19.1 Games with Full-System Crashes

We are now ready to define our model of concurrent computation with full-system crashes. We follow the same structure as Part II.

Definition 19.1.1 (Polarities with Crashes). We define the set of polarities for crash-aware games $\text{Pol}^{\sharp}$ as the set $\text{Pol}^{\sharp} := \text{Pol} + \{\sharp\}$.

Note that crash-aware games will effectively have an extra player which is responsible for issuing crash signals, which are represented by moves labelled by $\sharp$ and will be treated differently from O and P moves.

Definition 19.1.2 (Move Set and Well-Formed Plays). We define a crash-aware move set to be a move set $(M, \lambda : M \rightarrow \text{Pol}^{\sharp})$. Then, given a move set $M = (|M|, \lambda)$, we define $M^{\sharp}$

with set of moves $|M^\sharp| := |M| + \sharp$ and polarity function $\lambda^\sharp := \lambda + \sharp$.

Note that a sequence $s \in (M^\Upsilon)^\sharp$ is of the form

$$s_1 \cdot \sharp \cdot s_2 \cdot \sharp \cdot \dots \cdot \sharp \cdot s_{n+1}$$

where every $s_i \in (M^\Upsilon)^\sharp$. We define $\|s\|$ to be $n + 1$. We also define the operation $\mathbf{epo}_i(-)$ which assigns to s the sequence $\mathbf{epo}_i(s) := s_i$, called the i -th *epoch* of s . If $i > \|s\|$ then we take the convention that $\mathbf{epo}_i(s) := \epsilon$.

We say a sequence $s \in M^*$ is a well-formed crash-aware play when, for every $i \in \mathbb{N}$, $\mathbf{epo}_i(s) \in \mathbb{P}_{M^\Upsilon}$. We denote by $\mathbb{P}_M^\sharp$ the set of all well-formed crash-aware plays over the crash-aware move set $(M, \lambda : M \rightarrow \mathbf{Pol}^\sharp)$.

Note that in the definition of the well-formedness of crash-aware plays, we implicitly already enforce that crashes affect the entire system, as when a crash happens the entire system resets back to the initial P -position. This means that after a crash, the first move for all agents is an O move, no matter what was the last move by that agent in the previous epoch.

We are now ready to define crash-aware games. We will denote by $\sharp$ the unique crash event $\sharp \in M^\sharp$, for any crash-aware move set, which allows us to ignore the actual name of the crash event. While the assumption that there is a single type of crash is not necessary, allowing for several crash events makes several of the definitions more involved. Since for our practical purposes one crash event is enough, we opt for this simpler presentation. The reader will not be surprised, considering our handling of concurrent games, that a crash-aware game is just given by a sequential game. Therefore, the only thing to define is how the set of valid sequential plays generates a set of crash-aware plays.

Definition 19.1.3. Given a sequential game $A = (M_A, P_A)$, we define its set of crash-aware plays by $P_A^\sharp := P_A \cdot (\sharp \cdot P_A)^* \subseteq \mathbb{P}_{M_A}^\sharp$.

Note that any crash-aware game A defines a concurrent game, as concurrent games are also defined by sequential games, and vice-versa. In other words, all of our three models of computation so far Safe^{seq}, Safe, and our (to be defined) model of crash-aware strategies

Crash, all have the same notion of object, changing only in how a game generates plays which are used to define strategies between games. This also means that we inherit all the operations we have over sequential games directly, just like in the concurrent games case, including the $- \otimes -$, $- \multimap -$ and $\dagger -$.

Example 19.1.1. So, for example, a crash-aware play over the game Unit is:

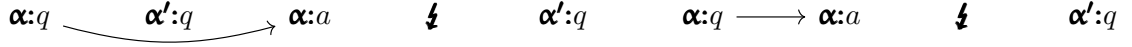


Figure 19.1: Example of a play of Unit with crashes.

Note that the main well-formedness constraint on plays of crash-aware games is that in each epoch they play a well-formed concurrent play, i.e., a locally sequential play.

Definition 19.1.4 (Crash-Aware Strategy). A crash-aware strategy $\sigma :_{\downarrow} A$ over a game A is a non-empty, prefix-closed, subset $\sigma \subseteq P_A^{\downarrow}$, which is moreover $\downarrow$ -receptive in that

$$\forall s \in \sigma. s \cdot \downarrow \in P_A^{\downarrow} \implies s \cdot \downarrow \in \sigma$$

The $\downarrow$ -receptivity property of strategies models the usual assumption that crashes may non-deterministically happen at any point in an execution of a program. Surprisingly, it plays a crucial role in establishing the symmetric monoidal structure of crash-aware games.

Similarly to our handling of starvation tokens, crashes happen in all components of a system at once. This is handled again by our definitions of projections. Given $s \in \mathbb{P}_{M_A+M_B}^{\downarrow}$ we define $s \upharpoonright_{A,-} \in \mathbb{P}_{M_A}^{\downarrow}$ and $s \upharpoonright_{-,B} \in \mathbb{P}_{M_B}^{\downarrow}$ so that only the moves in M_A , resp. M_B , and the $\downarrow$ -tokens are retained. We extend this to plays in $\mathbb{P}_{M_A+M_B+M_C}^{\downarrow}$ just like we did for concurrent games. In the context of games A and B , as opposed to move sets, we write $- \upharpoonright_{A,-}$ and $- \upharpoonright_{-,B}$ for $- \upharpoonright_{M_A,-}$ and $- \upharpoonright_{-,M_B}$ respectively. We will also write $- \upharpoonright_{A,B,-}$ and $- \upharpoonright_{-,B,C}$ for the projections from $\mathbb{P}_{M_A+M_B+M_C}^{\downarrow}$.

It is implicit in this definition that by composing in parallel the two crash-aware plays, the resulting set of traces synchronizes the crash events, merging them into a single crash event and then producing any (locally sequential) parallel composition of the subtraces appearing in each epoch. Consider, for instance, the two plays below on the left, each of

type Unit:

$$\begin{array}{ccccccc}
\alpha:q & \alpha:a & \textcolor{red}{\downarrow} & \alpha:q & \alpha:a & \textcolor{red}{\downarrow} & \alpha':q & \alpha':a \\
& & & \otimes & & & & \\
\alpha:q & \alpha:a & \textcolor{red}{\downarrow} & \alpha':q & \alpha':a & \textcolor{red}{\downarrow} & \alpha':q & \alpha':a
\end{array}
\Rightarrow
\begin{array}{ccc}
\alpha:q & \alpha:a & | \\
\otimes & & \textcolor{red}{\downarrow} \\
\alpha:q & \alpha:a & | \\
& & \otimes \\
& & \alpha':q & \alpha':a & | \\
& & \otimes & & \textcolor{red}{\downarrow} \\
& & \alpha':q & \alpha':a & | \\
& & \otimes & & \alpha':q & \alpha':a
\end{array}$$

The resulting set of traces synchronizes the crashes, as depicted on the right. For example, the following is a valid trace in their horizontal composition:

$$\alpha:q \cdot \alpha:a \cdot \alpha:q \cdot \alpha:a \cdot \textcolor{red}{\downarrow} \cdot \alpha:q \cdot \alpha':q \cdot \alpha:a \cdot \alpha':a \cdot \textcolor{red}{\downarrow} \cdot \alpha':q \cdot \alpha':a \cdot \alpha':q \cdot \alpha':a$$

Consider the following play s of $\text{Unit} \multimap \text{Unit}$ (on the left):

$$\begin{array}{c}
s = \begin{array}{ccccc}
& \alpha:q & & \alpha':q & \\
& \searrow & & \longrightarrow & \\
& \alpha:q & \longrightarrow & \alpha:a & \\
& & & | & \\
& & & \textcolor{red}{\downarrow} & \\
& & & \alpha':q & \searrow \\
& & & & \alpha':q
\end{array}
\end{array}
\quad \left| \quad \begin{array}{l}
s \upharpoonright_{-, \text{Unit}} = \alpha:q \cdot \alpha':q \cdot \textcolor{red}{\downarrow} \cdot \alpha':q \\
\\
s \upharpoonright_{\text{Unit}, -} = \alpha:q \cdot \alpha:a \cdot \textcolor{red}{\downarrow} \cdot \alpha':q
\end{array}
\right.$$

Note that the crash signal synchronizes across the source and target components of the play. This simultaneously models that the crashes are *synchronous* across components (they happen in all components at once) and that they are *instantaneous* (it takes negligible time for the crash to propagate to other components). On the right, above, we see the projections of s to the source and target components. Importantly, the crash event is retained in both projections, so that the crash event $\textcolor{red}{\downarrow}$ of $\text{Unit} \multimap \text{Unit}$ effectively belongs to both components.

Definition 19.1.5. We define the set of interaction plays as

$$A \circledcirc B \circledcirc C := \{s \in \mathbb{P}_{M_A+M_B+M_C}^{\textcolor{red}{\downarrow}} \mid s \upharpoonright_{A,B,-} \in P_{A \multimap B}^{\textcolor{red}{\downarrow}} \wedge s \upharpoonright_{-,B,C} \in P_{B \multimap C}^{\textcolor{red}{\downarrow}}\}$$

Given crash-aware strategies $\sigma :_{\textcolor{red}{\downarrow}} A \multimap B$ and $\tau :_{\textcolor{red}{\downarrow}} B \multimap C$, we define their set of interactions by

$$\sigma \circledcirc \tau := \{s \in A \circledcirc B \circledcirc C \mid s \upharpoonright_{A,B,-} \in \sigma \wedge s \upharpoonright_{-,B,C} \in \tau\}$$

and their composition

$$\sigma ; \tau := \{s \upharpoonright_{A,-,C} \in \mathbb{P}_{A \multimap C}^{\textcolor{red}{\downarrow}} \mid s \in \text{int}(\sigma, \tau)\}$$

Proposition 19.1.1. *Composition of crash-aware strategies is well-defined, associative, and continuous (where joins are given by unions, and order by set inclusion).*

Proof. See Proofs 32.1.1 and 32.1.2. □

These definitions permit us to define:

Definition 19.1.6. We denote by **Crash** the **Inf**-enriched semicategory of crash-aware games, with crash-aware strategies $\sigma :_{\downarrow} A \multimap B$ as morphisms between games A and B , composition given by $- ; -$, and refinement given by $- \supseteq -$.

19.2 The Crashing Copycat and Saturation

Just like **Safe**^{seq} and **Safe**, **Crash** does not assemble into a category for lack of a neutral element for composition. We take the approach described in Ch. 14 and define a copycat with crashes to then compute a corresponding section of the Karoubi envelope for **Crash**. This is the setup necessary for us to then develop a notion of *crash-aware* linearizability.

We define the crash-aware copycat $\text{crashcopy}_A :_{\downarrow} A \multimap A$ as the crash-aware strategy which just plays copy_A within each epoch:

$$\text{crashcopy}_A := (\text{copy}_A \cdot \downarrow)^* \cdot \text{copy}_A$$

It will not be surprising that crashcopy_A defines an idempotent for every A , considering the regularity of the definition, and the fact that copy_A is already an idempotent.

For this, we find it useful to define a semifunctor

$$-\downarrow : \underline{\mathbf{Safe}} \rightarrow \underline{\mathbf{Crash}}$$

defined by the identity on games, and, given a strategy $\sigma : A \multimap B \in \underline{\mathbf{Safe}}$, we define the strategy $\sigma^\downarrow :_{\downarrow} A \multimap B \in \underline{\mathbf{Crash}}$ as

$$\sigma^\downarrow := (\sigma \cdot \downarrow)^* \cdot \sigma$$

The fact that $(-)^{\sharp}$ defines a semifunctor immediately gives that:

Proposition 19.2.1. *crashcopy_A is idempotent for every A.*

Proof. See Proof 32.2.2.

□

Chapter 20

Crash-Aware Linearizability

Following the compositional linearizability construction, we define the category of strategies saturated under `crashcopy` as (see Ch. 14):

Definition 20.0.1. We define the category **Crash** as the section $K_{\text{crashcopy}} : \underline{\mathbf{Crash}} \rightarrow \mathbf{Semi\ Crash}$.

Note that as per the discussion in Ch. 14, the construction comes with a semifunctor

$$K_{\sharp} - := K_{\text{crashcopy}} : \underline{\mathbf{Crash}} \rightarrow \mathbf{Semi\ Crash}$$

defined by:

$$K_{\sharp} (\tau :_{\sharp} A \multimap B) := \text{crashcopy}_A ; \tau ; \text{crashcopy}_B$$

Intuitively, this operation assigns to τ the smallest saturated strategy containing τ . $K_{\sharp} -$ has the important property that it is an oplax semifunctor (lax with respect to the refinement order), i.e.

Proposition 20.0.1. *For all $\sigma :_{\sharp} A \multimap B$ and $\tau :_{\sharp} B \multimap C$ in $\underline{\mathbf{Crash}}$, $K_{\sharp} (\sigma ; \tau) \subseteq K_{\sharp} \sigma ; K_{\sharp} \tau$.*

Proof. See Proof 32.5.1. □

In this chapter, we give a concrete characterization of the notion of linearizability associated to $\mathcal{L}(\mathbf{Crash})$, which is closely related to strict linearizability [Aguilera and Frølund,

2003]. Then, we present the equivalence with observational refinement and the locality property for it.

20.1 Crash-Aware Linearizability

The framework of compositional linearizability proposes that the native notion of linearizability for crash-aware objects should be equivalent to the refinement $\nu' \subseteq K_{\sharp} \nu$. So for the remainder of this section, our goal is to concretely characterize this refinement in terms of plays. For this, we find it useful to recall the concrete formulation of compositional linearizability. This notion of linearizability is a slight generalization of interval-sequential linearizability, and in particular does not require that the linearized specification be atomic nor that all pending operations be removed.

We use linearizability in $\mathcal{L}(\mathbf{Safe})$ to define crash-aware linearizability. It is straightforward: s crash-aware linearizes to t when their corresponding epochs linearize to each other.

Definition 20.1.1. For $A \in \mathbf{Crash}$, we say a play $s \in P_A^{\sharp}$ crash-aware linearizes to a play $t \in P_A^{\sharp}$ when

$$\|s\| = \|t\| \quad \text{and} \quad \forall i \leq \|s\|. \text{epo}_i(s) \rightsquigarrow \text{epo}_i(t)$$

and write $s \rightsquigarrow^{\sharp} t$ when this holds, and extend the notation as we did for linearizability (see Def. 13.1.1).

Observe that crash-aware linearizability relates crash-aware specifications to crash-aware specifications. This is unusual in the literature on linearizability under crashes, as the other criteria relate a crash-aware specification to a crash-less specification. We discuss the reasons for this later when we have defined two other linearizability criteria and can better compare them.

The main result of this section is the following characterization of $K_{\sharp}$ –.

Proposition 20.1.1.

$$K_{\frac{1}{2}} \tau = \{s \in P_{A \multimap B}^{\frac{1}{2}} \mid s \text{ is crash-aware linearizable with respect to } \tau\}$$

Proof. See Proof 32.5.2. □

Corollary 20.1.1. $\nu' :_{\frac{1}{2}} A$ is crash-aware linearizable with respect to $\nu :_{\frac{1}{2}} A$ if and only if $\nu' \subseteq K_{\frac{1}{2}} \nu$.

Proof. Follows from Prop. 32.5.2. □

We use the general result in Ch. 14 to obtain locality and observational refinement. The requirements to obtain these properties are the following.

Proposition 20.1.2.

- For any $\sigma :_{\frac{1}{2}} \mathbf{1} \multimap A \in \mathbf{Crash}$ it holds that $\text{crashcopy}_1; \sigma = \sigma$.
- For $\sigma, \tau :_{\frac{1}{2}} A \multimap B$ and $\sigma', \tau' :_{\frac{1}{2}} A' \multimap B'$ in $\mathbf{Crash}$, we have $\sigma \otimes \sigma' \subseteq \tau \otimes \tau' \implies \sigma \subseteq \tau \wedge \sigma' \subseteq \tau'$

Proof. See Proof 32.5.1. □

This gives as corollaries locality and observational refinement, which we write explicitly now. The proof of these results, under the conditions of our construction and Prop. 20.1.2, follows from the theory of compositional linearizability established in Ch. 14.

Corollary 20.1.2 (Observational Refinement). $\nu'_A :_{\frac{1}{2}} A \in \mathbf{Crash}$ is crash-aware linearizable w.r.t. $\nu_A :_{\frac{1}{2}} A \in \mathbf{Crash}$ if and only if for all $\sigma :_{\frac{1}{2}} A \multimap B \in \mathbf{Crash}$, $\nu'_A; \sigma \subseteq \nu_A; \sigma$

Proof. See Proof 32.5.4. □

Corollary 20.1.3 (Locality). For $\nu'_A :_{\frac{1}{2}} A, \nu'_B :_{\frac{1}{2}} B \in \mathbf{Crash}$ and $\nu_A :_{\frac{1}{2}} A, \nu_B :_{\frac{1}{2}} B \in \mathbf{Crash}$:

$$\nu'_A \xrightarrow{\frac{1}{2}} \nu_A \text{ and } \nu'_B \xrightarrow{\frac{1}{2}} \nu_B \text{ if and only if } \nu'_A \otimes \nu'_B \xrightarrow{\frac{1}{2}} \nu_A \otimes \nu_B$$

Proof. See Proof 32.5.4. □

Of course, as noted in Ch. 14, we really just have a compositional linearizability calculus for crash-aware components, as given by the construction $\mathcal{L}(\mathbf{Crash})$.

20.2 Specifying a Buffered Memory Cell

In §18, we mentioned that we use crash-aware linearizability to specify a buffered memory cell with signature BCell. As an example, we define here what the linearized specification for a buffered memory cell implementation would be under crash-aware linearizability.

An example of a trace of a concrete buffered memory cell ν'_{BCell} is:

$$\alpha_1:\text{wrt}(1) \cdot \alpha_2:\text{rd}() \cdot \alpha_1:\text{ok} \cdot \alpha_2:0 \cdot \alpha_2:\text{flush}() \cdot \alpha_1:\text{wrt}(2) \cdot \alpha_1:\text{ok} \cdot \text{!} \cdot \alpha_3:\text{rd}() \cdot \alpha_3:1$$

The trace above is crash-aware linearizable to the following trace, among others:

$$\alpha_2:\text{rd}() \cdot \alpha_2:0 \cdot \alpha_1:\text{wrt}(1) \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{flush}() \cdot \alpha_2:\text{ok} \cdot \alpha_1:\text{wrt}(2) \cdot \alpha_1:\text{ok} \cdot \text{!} \cdot \alpha_3:\text{rd}() \cdot \alpha_3:1$$

We specify the semantics of the buffered memory cell by a set of traces ν'_{BCell} with only events that are allowed by the signature BCell. Crashes can happen at any point. To specify the correctness of ν'_{BCell} we require it to be crash-aware linearizable to the atomic linearized specification ν_{BCell} . Because we show observational refinement, we are able to leave ν'_{BCell} unspecified for the sake of verifying the FLiT implementation, as only the linearized specification will be necessary. The linearized specification ν_{BCell} is then defined by:

$$s \in \nu_{\text{BCell}} \iff s \text{ is atomic} \wedge (\forall s_1, s_2. \forall v. s = s_1 \cdot \alpha:\text{rd}() \cdot \alpha:v \cdot s_2 \implies v \in \text{snd}(\text{mstate}(s_1)))$$

where $\text{mstate}(s)$ assigns to an atomic complete trace s a set of pairs $\text{mstate}(s) \subseteq \text{Val} \times \text{Val}$. A pair $(v_p, v_b) \in \text{mstate}(s)$ consists of a possibility for a value v_p that has persisted and a value v_b currently in the buffer. $\text{mstate}(s)$ is then the function inductively defined below ($v_0 \in \text{Val}$ is an identified initial value for the memory cell):

$$\begin{aligned} \text{mstate}(\epsilon) &:= \{(v_0, v_0)\} & \text{mstate}(s \cdot \text{!}) &:= \{(v, v) \mid \exists v'. (v, v') \in \text{mstate}(s)\} \\ \text{mstate}(s \cdot \alpha:\text{wrt}(v) \cdot \alpha:\text{ok}) &:= \{(v', v) \mid \exists v''. (v', v'') \in \text{mstate}(s)\} \cup \{(v, v)\} \\ \text{mstate}(s \cdot \alpha:\text{rd}() \cdot \alpha:v) &:= \{(v', v) \mid (v', v) \in \text{mstate}(s)\} \\ \text{mstate}(s \cdot \alpha:\text{flush}() \cdot \alpha:\text{ok}) &:= \{(v, v) \mid (v', v) \in \text{mstate}(s)\} \end{aligned}$$

The function $\text{snd}(p)$ projects into the second component v_b of the pair $p = (v_p, v_b)$, so that

$$\text{snd}(\text{mstate}(s)) = \{v_b \mid \exists v_p. (v_p, v_b) \in \text{mstate}(s)\}.$$

Note that we could have specified it instead using a labeled state transition system (LTS), in which case ν_{BCell} is the set of traces that start from the initial state of the LTS. Either way of defining ν_{BCell} defines the same set of traces.

20.3 Compositional Verification of a File System Fragment

To showcase the benefits of compositionality and to show that crash-aware linearizability provides a flexible criterion for mixing objects with different, and complicated, crash behaviors, we verify against a crash-aware linearizable specification a fragment of a file API. Instead of providing a detailed description, we emphasize the salient aspects to our point (a detailed description is available in [Oliveira Vale et al., 2024b]).

The file system fragment involves four main objects: the file interface **File**, a disk interface **Disk** implemented using a disk array $\text{Disk}[N]$ with a finite number N of disks each with $S+1$ blocks, and a write-ahead log **Log**. The signatures for **File** and **Disk** are given below:

$$\text{File} := \left\{ \begin{array}{l} \text{write} : \text{block\_id} \times \text{file\_id} \times \text{block} \rightarrow 1, \\ \text{read} : \text{block\_id} \times \text{file\_id} \rightarrow \text{block}, \\ \text{swap} : \text{block\_id} \times \text{block\_id} \times \text{file\_id} \times \text{file\_id} \rightarrow 1 \end{array} \right\} \quad \text{Disk} := \left\{ \begin{array}{l} \text{write} : \text{block\_id} \times \text{block} \rightarrow 1, \\ \text{read} : \text{block\_id} \rightarrow \text{block} \end{array} \right\}$$

The file interface exposes a two-level structure. At the first level lies a set of folders, each occupying a single disk block as its inode. For simplicity, the API uses block ids instead of strings to uniquely identify folders. Each folder contains a set of files identified by their file id. The **swap** operation swaps the pointers in the respective folders' inodes, which provides a symmetric move operation as seen in actual file systems. The **write** and **read** operations are as usual. The file interface is implemented on top of a disk, providing **write** and **read** operations to read and write to a block.

All the objects involved are specified using crash-aware linearizability. For instance, a single disk is specified as the horizontal composition of its blocks, using locality, guaranteeing that its concrete specification $\nu'_{\text{Disk}} : \dagger\text{Disk}$ is crash-aware linearizable to a specification $\nu_{\text{Disk}} : \dagger\text{Disk}$ which guarantees read and writes are persistent and atomic. The disk array specification $\nu'_{\text{Disk}[N]}$ is required to be crash-aware linearizable to the horizontal composition

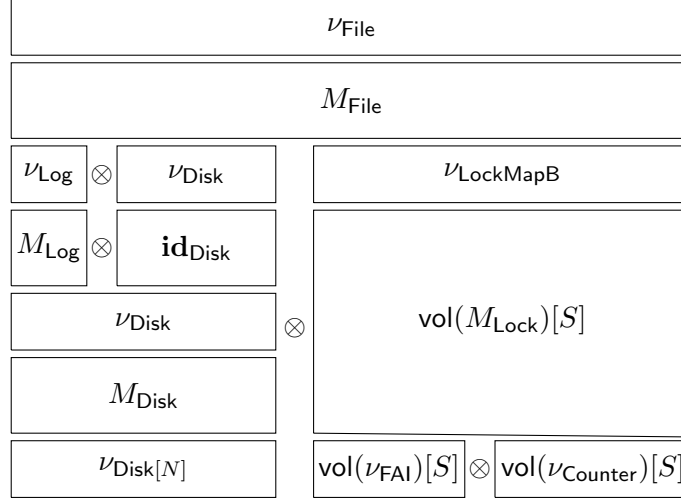


Figure 20.1: The structure of our File example.

of N atomic disk specifications $\nu_{\text{Disk}[N]} := \otimes_{i \in [N]} \nu_{\text{Disk}}$. The concrete object ν'_{File} is required to be crash-aware linearizable to a specification ν_{File} that ensures that writes, reads, and swaps are persistent and seem to happen atomically. All the specifications also enforce that the recovery routines correctly reconstruct any relevant lost state after a crash.

We implement the replicated disk on top of the disk array by replicating writes to all the disks in the array in a specific order. Reads to the disk array non-deterministically choose a disk to read from, mimicking the behavior of a disk array controller. On a crash, a recovery procedure copies the contents of the first disks to all disks.

The File implementation M_{File} for **write** and **read** is mostly straightforward. The **swap** operation requires special treatment for its atomicity. As swap operations need to update two different folders (and thus two different disk blocks), to ensure persistency, we record the operations in a write-ahead log ν'_{Log} so that the recovery routine can finish incomplete operations. The log is itself implemented on top of a single block of the disk together with a volatile array and a volatile lock (omitted from Fig. 20.1). Since the disk is itself equivalent to the parallel composition of individual blocks, we use locality together with our compositionality properties (the symmetric monoidal structure of the model) to separate the part of the disk used for the log from the rest of the disk.

The file system also uses a set of dynamically allocated locks ν'_{LockMapB} to guarantee atomicity when writing to a block. These locks are volatile objects residing in memory that

only last for the duration of a single File operation. Because of this, we use the verified lock from Part II and lift it to a volatile object using Prop. 21.2.1, benefiting from the fact that we have connected our model to their model. The whole structure of the example is depicted in Fig. 20.1.

At this point, it is worth remarking that even this small fragment of a file system features a mix of persistent objects, volatile objects, and objects that fit neither category well. Some of the objects involved are horizontal compositions of these objects, making them have hybrid crash behavior. We model all of these objects using crash-aware linearizability, which proves to be robust enough to verify the whole system compositionally.

Chapter 21

Durable and Strict Linearizability

In the previous chapter, we developed crash-aware linearizability, a native notion of linearizability for our model of concurrent computation with crashes. However, two linearizability criteria predate our work and are widely used in the literature: strict linearizability [Aguilera and Frølund, 2003] and durable linearizability [Izraelevitz et al., 2016]. Both criteria use crash-less linearized specifications, making them well-suited for specifying objects with strong persistency guarantees, but less appropriate for volatile or hybrid objects. In this chapter, we give a compositional account of both criteria in our framework.

We begin in §21.1 by recalling the definitions of strict and durable linearizability, adapted to our setting. In §21.2, we contrast these criteria against crash-aware linearizability, showing how the use of crash-less specifications limits their expressiveness for volatile and hybrid objects. Then, in §21.3, we develop the crash abstraction operation $-^b$, which converts crash-aware strategies into concurrent strategies by removing crashes. This operation, together with the crash re-introduction operation $-^\sharp$, serves as the key building block connecting **Crash** and **Safe**. Using these tools, we formalize strict linearizability (§21.4) and durable linearizability (§21.5) as refinement conditions, and prove observational refinement and locality for each. We finish by proving the FLiT correctness theorem as an application of durable linearizability.

21.1 Background: Linearizability Criteria under Crashes

We start by defining strict linearizability [Aguilera and Frølund, 2003]. Our definition differs from the original one in that it specializes it to full-system crashes (instead of allowing for each agent to crash independently), removes the notion of aborted executions, and generalizes away from atomicity to allow for non-atomic linearized specifications. The first two changes were already considered in Ben-David et al. [2022] and make the criterion appropriate for the settings we are interested in, such as NVM and file systems. The latter change goes along the lines of the way that interval linearizability [Castañeda et al., 2015] and compositional linearizability generalize Herlihy-Wing linearizability [Herlihy and Wing, 1990]. If we restrict our definition so that the linearized trace must be atomic, we obtain the same criterion considered by Ben-David et al. [2022].

Definition 21.1.1. For a crash-aware trace s , we define, whenever well-formed, the crash-less trace

$$\text{ops}(s) := \text{epo}_1(s) \cdot \text{epo}_2(s) \cdot \dots \cdot \text{epo}_{\|s\|}(s)$$

We say a crash-aware trace $s \in \mathbb{P}_M^{\sharp}$ is *strictly linearizable* to a crash-less trace t , written $s \xrightarrow{\text{str}} t$, when there exists a crash-aware trace t' such that $s \xrightarrow{\sharp} t'$ and $\text{ops}(t') = t$.

Note that the definition of $\text{ops}(-)$ here coincides with that in Ch. 12 as it restricts the sequence to only moves with polarities in Pol . Note that our definition of strict linearizability shows a clear factoring of strict linearizability as crash-aware linearizability followed by crash-removal.

The third and final linearizability criterion we consider here is *durable linearizability* [Izraelevitz et al., 2016]. Durable linearizability is more expressive than strict linearizability [Guerraoui and Levy, 2004; Ben-David et al., 2022] in that it considers more objects to be linearizable. This comes at the cost of the extra assumption on the model that new agent names are used in each epoch, which we call the *durability assumption*.

Definition 21.1.2. We say a crash-aware trace $s \in \mathbb{P}_M^{\sharp}$ is *durable* when:

$$\forall i, j \leq \|s\|. i \neq j \implies \Upsilon(\text{epo}_i(s)) \cap \Upsilon(\text{epo}_j(s)) = \emptyset$$

where $\Upsilon(t)$ is the set of agents appearing in a trace t . We denote by $\mathbb{P}_M^{\text{dur}} \subseteq \mathbb{P}_M^{\text{f}}$ the subset of well-formed crash-aware traces that are durable.

When a trace s is durable, $\text{ops}(s)$ is always a well-formed crash-less trace. Durable linearizability is then defined in terms of usual crash-less linearizability. Our definition, similarly to our definitions of the other linearizability criteria we consider, generalizes away from atomicity by allowing linearized traces to be non-atomic and allows for the specification of blocking objects, as it does not require all uncompleted pending invocations to be removed. It is, however, fully equivalent to the original definition of durable linearizability if we require that the linearized trace be atomic.

Definition 21.1.3. We say a durable trace $s \in \mathbb{P}_M^{\text{dur}}$ is *durably linearizable*, written $s \xrightarrow{\text{dur}} t$, to a crash-less trace t when $\text{ops}(s) \rightsquigarrow t$.

Note that durable linearizability corresponds to the inverse factoring to strict linearizability: one first removes crashes and then uses crash-less linearizability. These two factorings play an important technical role in our proofs. Moreover, it is possible to show that both criteria factor (in a different sense) through crash-aware linearizability.

Proposition 21.1.1.

- If $s' \xrightarrow{\text{f}} s$ and $s \xrightarrow{\text{f}} t$ then $s' \xrightarrow{\text{f}} t$
- If $s' \xrightarrow{\text{f}} s$ and $s \xrightarrow{\text{dur}} t$ then $s' \xrightarrow{\text{dur}} t$

Proof. See Proof 32.5.5. □

Because of this fact, in practice, when verifying durably linearizable objects, we find it useful to use a crash-aware specification ν^{mid} satisfying: $\nu' \xrightarrow{\text{f}} \nu^{\text{mid}}$ and $\nu^{\text{mid}} \xrightarrow{\text{dur}} \nu$. This allows us to consider less concurrent traces within the linearized specification for ν' by linearizing as much as possible within each epoch of ν^{mid} first. This allows us to obtain the benefits of both crash-aware and durable linearizability simultaneously: by maintaining both ν^{mid} and ν we can still express durably linearizable specifications, but by manipulating ν^{mid} we achieve the same level of compositionality as crash-aware linearizability. This technique is not necessary for strict linearizability because we can just use crash-aware linearizability directly by always picking ν^{mid} so that $\text{ops}(\nu^{\text{mid}}) = \nu$.

21.2 Contrasting Crash-Aware Linearizability

We now compare crash-aware linearizability against strict and durable linearizability. We will not compare strict and durable linearizability against each other since they are not new to my work, and refer the interested reader to the following references [Guerraoui and Levy, 2004; Izraelevitz et al., 2016; Ben-David et al., 2022]. We do briefly mention a key difference that applies to crash-aware linearizability as well. In strict and crash-aware linearizability, a pending invocation must be linearized *within* the epoch it was issued. Durable linearizability, however, allows for a pending invocation to be linearized in (essentially) a later epoch by allowing those pending invocations to be reordered after events from later epochs. This is what makes it more expressive than strict linearizability, allowing for more complex crash behaviors, such as recovering parts of a data structure only when they are demanded by a client, which could happen several epochs later. As explained in the remark at the end of §21.1, crash-aware linearizability interacts well with durable linearizability.

As we saw, both durable and strict linearizability factor through crash-aware linearizability. The key difference between the two former criteria and the latter is that the former use crash-less linearized specifications, while the latter uses crash-aware linearized specifications. So let's refer to the former as *crash-unaware* criteria.

Crash-unaware criteria reduce the correctness of an object with crashes to that of an object without crashes. This makes them great at specifying objects with very strong persistency guarantees, that is, objects whose whole state (or almost) persists after a crash. But it makes them quite deficient at specifying objects with weaker persistency guarantees such as volatile objects (all of the state is lost on a crash), objects with hybrid persistency (part of the state is volatile and part of the state is persistent), or objects whose persistency features some degree of non-determinism (such as in buffered memory). Some of these issues were already known. For instance, in the original durable linearizability paper [Izraelevitz et al., 2016], it is noted that the criteria do not behave well when used to specify a buffered object, requiring them to define an *ad-hoc* notion of *buffered* durable linearizability which does not satisfy locality, making it not compositional.

Consider the simple problem of specifying the correctness of a volatile object. Given a

crash-less specification ν , we can construct a crash-aware specification $\text{vol}(\nu)$ of a volatile version of that object by the Kleene algebra formula $\text{vol}(\nu) := (\nu \cdot \text{!})^* \cdot \nu^1$.

For example, given the usual atomic counter specification ν_{Counter} , the following trace is allowed by $\text{vol}(\nu_{\text{Counter}})$ (the subscript 1 under the Counter operations will be useful later):

$$s_1 = \alpha_1:\text{inc}_1 \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{get}_1 \cdot \alpha_2:1 \cdot \text{!} \cdot \alpha_3:\text{get}_1 \cdot \alpha_3:0$$

Note that the crash move ! plays a crucial role in the specification, as the counter only resets to 0 after a crash event (such as the last `get` event in s_1), making the linearized specification deterministic.

Under crash-aware linearizability, a concurrent object ν' correctly implements a volatile version of a crash-less object ν when $\nu' \xrightarrow{\text{!}} \text{vol}(\nu)$. With our methods, it is easy to show that

Proposition 21.2.1. *If $\nu' \rightsquigarrow \nu$ then $\text{vol}(\nu') \xrightarrow{\text{!}} \text{vol}(\nu)$.*

Proof. See Proof 32.5.3. □

A consequence of this is that if we have an implementation M that implements a crash-less object linearizable to ν_B on top of a crash-less object ν_A , then if we run M on each epoch on top of $\text{vol}(\nu_A)$ then M implements an object crash-aware linearizable to $\text{vol}(\nu_B)$. Formally,

$$\nu_A; M \rightsquigarrow \nu_B \implies \text{vol}(\nu_A); \text{vol}(M) \xrightarrow{\text{!}} \text{vol}(\nu_B)$$

In other words, crash-aware linearizability is able to appropriately specify and characterize the correctness of volatile objects in a way that is useful to a client.

Now, consider what happens if we try to specify a volatile object using a crash-unaware criterion. A crash-unaware linearized specification will need to include both of the following traces:

$$\text{ops}(s_1) = \alpha_1:\text{inc}_1 \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{get}_1 \cdot \alpha_2:1 \cdot \alpha_3:\text{get}_1 \cdot \alpha_3:0$$

and

$$\alpha_1:\text{inc}_1 \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{get}_1 \cdot \alpha_2:1 \cdot \alpha_3:\text{get}_1 \cdot \alpha_3:1$$

1. Note this is just $\text{vol}(\nu) = \nu^{\text{!}}$, meaning that $\text{vol}(-)$ is in fact a (semi)functor. We choose to rename it to emphasize its use in this different context.

so that the linearized specification under crash-unaware criteria must admit non-deterministically resetting the counter at any point. This can happen at any point, but the point at which it happens is not detectable in the linearized specification, which makes the specification quite weak. This means that even if some observational refinement theorem (*à la* Filipovic et al. [2010]) holds for the crash-unaware criterion, the client to the linearized specification will need to contend with non-determinism, making the contextual refinement, and hence vertical composition, weaker.

This issue is compounded when considering horizontal composition. Both durable and strict linearizability are known to satisfy locality. However, those locality theorems introduce even more non-determinism into the resulting linearized specifications. Consider now a second trace s_2 for a second counter independent of the counter in play s_1 :

$$s_2 = \alpha_1:\text{inc}_2 \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{get}_2 \cdot \alpha_2:1 \cdot \frac{1}{2} \cdot \alpha_3:\text{get}_2 \cdot \alpha_3:0$$

Any trace in their parallel composition $s_1 \otimes s_2$ synchronizes on the crash, so both counters reset their state at the same time. For example, the following crash-aware trace belongs to $s_1 \otimes s_2$:

$$\alpha_1:\text{inc}_1 \cdot \alpha_1:\text{ok} \cdot \alpha_1:\text{inc}_2 \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{get}_1 \cdot \alpha_2:1 \cdot \alpha_2:\text{get}_2 \cdot \alpha_2:1 \cdot \frac{1}{2} \cdot \alpha_3:\text{get}_2 \cdot \alpha_3:0 \cdot \alpha_3:\text{get}_1 \cdot \alpha_3:0$$

Meanwhile, the corresponding linearized specifications under durable or strict linearizability include these traces without the crash event, i.e., $\text{ops}(s_1)$ and $\text{ops}(s_2)$. Hence, the following trace is in their parallel composition $\text{ops}(s_1) \otimes \text{ops}(s_2)$ (the set of their well-formed crash-less interleavings):

$$\alpha_1:\text{inc}_1 \cdot \alpha_1:\text{ok} \cdot \alpha_1:\text{inc}_2 \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{get}_1 \cdot \alpha_2:1 \cdot \alpha_3:\text{get}_1 \cdot \alpha_3:0 \cdot \alpha_2:\text{get}_2 \cdot \alpha_2:1 \cdot \alpha_3:\text{get}_2 \cdot \alpha_3:0$$

There is no trace of the concrete horizontally composed volatile counters that is linearizable to the trace above, as we must at least introduce a crash right before $\alpha_3:\text{get}_1$ to justify its return $\alpha_3:0$:

$$\alpha_1:\text{inc}_1 \cdot \alpha_1:\text{ok} \cdot \alpha_1:\text{inc}_2 \cdot \alpha_1:\text{ok} \cdot \alpha_2:\text{get}_1 \cdot \alpha_2:1 \cdot \frac{1}{2} \cdot \alpha_3:\text{get}_1 \cdot \alpha_3:0 \cdot \alpha_2:\text{get}_2 \cdot \alpha_2:1 \cdot \alpha_3:\text{get}_2 \cdot \alpha_3:0$$

This makes the trace inconsistent with the semantics of the second counter, as the crash should also have reset it, so that $\alpha_2:\text{get}_2$ should not return $\alpha_2:1$. The same kind of argument

shows how crash-unaware criteria fail to accurately handle hybrid and buffered objects (all the traces above are valid for a buffered counter, for example).

21.3 Crash Abstraction

Many specification methodologies for crash-aware objects, including durable linearizability, use specifications without crashes. In our framework, this means that while the concrete crash-aware object ν' lives in **Crash**, the abstract specification ν lives in **Safe**. In this section, we develop conversions between **Crash** and **Safe** that serve as a building block for our compositional treatment of strict and durable linearizability.

The main difficulty in removing crashes from a play s with crashes is that the removal may generate traces not satisfying local sequentiality. This happens when the same agent has a pending invocation in one epoch and also moves in a later epoch. So, in the definition of the operation $-^b$ (read *de-crash*), the projections $\text{ops}(s)$ are required to be well-formed plays.

Definition 21.3.1. Given a strategy $\sigma :_{\downarrow} A \multimap B \in \mathbf{Crash}$ we define $\sigma^b : A \multimap B \in \mathbf{Safe}$ by

$$\sigma^b := \{\text{ops}(s) \in P_{A \multimap B}^\Phi \mid s \in \sigma\}$$

The operation $-^b$ defines an oplax (under the computation order) semifunctor $-^b : \mathbf{Crash} \rightarrow \mathbf{Safe}$, acting as the identity on objects:

Proposition 21.3.1. For all $\sigma :_{\downarrow} A \multimap B$ and $\tau :_{\downarrow} B \multimap C$ in **Crash**, $(\sigma; \tau)^b \subseteq \sigma^b; \tau^b$.

In addition, for all $A \in \mathbf{Crash}$, $\text{crashcopy}_A^b = \text{copy}_A$, and $-^b$ is monotonic and join-preserving.

Proof. See Proofs 32.6.1 and 32.6.2. □

It is also useful to provide a reverse operation $-^\sharp$, read *re-crash*, that lifts, in a persistent way, a strategy $\sigma : A \multimap B \in \mathbf{Safe}$ into a strategy $\sigma :_{\downarrow} A \multimap B \in \mathbf{Crash}$, a situation we depict diagrammatically below.

$$\begin{array}{ccc}
A & \overset{\sigma^\#}{\dashrightarrow} & B \\
\downarrow & & \downarrow \\
A & \xrightarrow{\sigma} & B
\end{array}$$

So, suppose given $\sigma : A \multimap B \in \mathbf{Safe}$. Then, the crash-aware strategy $\sigma^\# :_{\sharp} A \multimap B \in \mathbf{Crash}$ is given by:

$$\sigma^\# := \{s \in \mathbb{P}_A^\sharp \mid \text{ops}(s) \in \sigma\}$$

Re-crash also behaves *like* an enriched semifunctor.

Proposition 21.3.2. *For $\sigma : A \multimap B, \tau : B \multimap C \in \mathbf{Safe}$, $\sigma^\#; \tau^\# \subseteq (\sigma; \tau)^\#$.*

In addition, for all $A \in \mathbf{Crash}$, $\text{copy}_A^\# \subseteq \text{crashcopy}_A$, and $-^\#$ is monotonic and join-preserving.

Proof. See Proofs 32.6.3 and 32.6.4. □

21.4 Strict Linearizability

Strict linearizability [Aguilera and Frølund, 2003], an important linearizability criterion in the presence of crashes often used to specify objects with robust recovery routines, postulates that a pending operation must linearize within the same epoch it was issued. It was originally formulated in a system with individual crashes instead of full-system crashes, so we modify it for our setting. We do not assume atomicity or that all pending operations are removed.

Similarly to how compositional linearizability characterizes linearizability by lifting a non-saturated strategy to a saturated strategy, we formalize strict linearizability by lifting a strategy without crashes into a strategy with crashes. Naively, one might think it is enough to use the lift $\nu^\# :_{\sharp} A \in \mathbf{Crash}$. Unfortunately, this does not make a crash-aware object, mainly because it does not satisfy O -receptivity anymore. Specifically, $\nu^\#$ never has plays such as

$$\alpha : q \cdot \downarrow \cdot \alpha : q \cdot \alpha : a$$

because $\alpha : q \cdot \alpha : q \cdot \alpha : a$ is not well-formed as a concurrent play of A . We can fix this issue by saturating the resulting strategy using $K_{\downarrow} -$.

Definition 21.4.1. Given games $A, B \in \mathbf{Crash}$, we define the strict lift $\text{str}(\sigma) :_{\sharp} A \multimap B$

of a strategy $\sigma : A \multimap B \in \mathbf{Safe}$ as the crash-aware strategy

$$\mathbf{str}(\sigma) := K_{\sharp} \sigma^{\sharp}$$

We now define our variation of strict linearizability, which differs only in that it generalizes the original strict-linearizability by not requiring the linearized specification to be atomic and complete, and specializes it to full-system crashes.

Definition 21.4.2. We say $\nu' :_{\sharp} A \in \mathbf{Crash}$ is strictly linearizable to $\nu : A \in \mathbf{Safe}$ when $\nu' \subseteq \mathbf{str}(\nu)$.

It is not hard to see that when ν is an atomic strategy this is equivalent to strict linearizability. Indeed, the application of $K_{\sharp}$ – within $\mathbf{str}(-)$ corresponds to constructing a strict completion [Ben-David et al., 2022], and $-^{\sharp}$ to removing the crashes. Finally, by first noting that

Proposition 21.4.1. *For all $\sigma : A \multimap B, \tau : B \multimap C \in \mathbf{Safe}$, $\mathbf{str}(\sigma); \mathbf{str}(\tau) \subseteq \mathbf{str}(\sigma; \tau)$.*

Proof. See Proof 32.7.2. □

we are able to prove the following refinement property for strict linearizability.

Proposition 21.4.2. *Suppose $\nu'_A :_{\sharp} A$ is strictly linearizable to $\nu_A : A$ and that $\sigma : A \multimap B$ implements an object linearizable to $\nu_B : B$ using ν_A , i.e.*

$$\nu_A; \sigma \subseteq \nu_B$$

Then, $\mathbf{str}(\sigma)$ implements an object strictly linearizable to $\mathbf{str}(\nu_B)$ using ν'_A , i.e.

$$\nu'_A; \mathbf{str}(\sigma) \subseteq \mathbf{str}(\nu_B)$$

Proof. See Proof 32.7.3. □

The reverse direction, unfortunately, does not hold as $\mathbf{str}(\text{copy}_A) \neq \text{crashcopy}_A$. By similar reasoning as the locality for crash-aware linearizability, we also obtain that

Proposition 21.4.3 (Locality). *For $\nu'_A :_{\downarrow} A, \nu'_B :_{\downarrow} B \in \mathbf{Crash}$ and $\nu_A : A, \nu_B : B \in \mathbf{Safe}$:*

$$\nu'_A \subseteq \text{str}(\nu_A) \text{ and } \nu'_B \subseteq \text{str}(\nu_B) \text{ if and only if } \nu'_A \otimes \nu'_B \subseteq \text{str}(\nu_A \otimes \nu_B)$$

Proof. See Proof 32.7.5. □

21.5 Durable Linearizability

A frequently used linearizability criterion for specifying persistent objects is durable linearizability [Izraelevitz et al., 2016], which appears as an important development in linearizability with crashes [Ben-David et al., 2022]. Durable linearizability makes a core assumption, which we call the *durability assumption*: that the agents appearing in each epoch are disjoint across epochs. This means that there is no agent re-use between epochs. The assumption makes the definition of durable linearizability significantly simpler than previous criteria, including strict linearizability. Moreover, under the durability assumption, persistent atomicity and recoverable linearizability are equivalent. This assumption, however, is quite intrusive and must be enforced throughout, requiring us to define a new model **Dur**. Interestingly, this can be smoothly done by a different choice of copycat for **Crash**. We finish by showing locality and observational refinement for durable linearizability.

21.5.1 Durable Linearizability

We start by formalizing the durability assumption, and from now on assume Υ is countably infinite.

Definition 21.5.1. Given a game A , we write P_A^{dur} for $P_A^{\text{dur}} := P_A \cap \mathbb{P}_A^{\text{dur}}$.

We say a crash-aware strategy is *durable* when all of its plays are durable.

Durable plays s have the important property that $\text{ops}(s) \in P_A$ as

$$\text{ops}(s) = \text{epo}_1(s) \cdot \text{epo}_2(s) \cdot \dots \cdot \text{epo}_n(s) \cdot \dots,$$

so we obtain that when σ is durable $\sigma^\flat = \text{ops}(\sigma)$. An important result is that durable strategies compose:

Proposition 21.5.1. *If $\sigma :_{\downarrow} A \multimap B$ and $\tau :_{\downarrow} B \multimap C$ are durable strategies then $\sigma; \tau$ is a durable strategy.*

Proof. See Proof 32.8.1. □

This means that the restriction of **Crash** to durable strategies defines a semicategory, which we call **Dur**. This motivates defining a durable copycat strategy.

Definition 21.5.2. For $A \in \mathbf{Crash}$, the durable copycat $\text{durcopy}_A :_{\downarrow} A \multimap A$ is the crash-aware strategy:

$$\text{durcopy}_A := \text{crashcopy}_A \cap \mathbb{P}_{A \multimap A}^{\text{dur}}$$

Proposition 21.5.2. *durcopy is idempotent.*

Proof. See Proof 32.8.2. □

This lets us define a model for durable objects and their implementations.

Definition 21.5.3. We define the category **Dur** as the restriction of **Crash** to strategies saturated with respect to durcopy .

The construction, like for **Safe** and **Crash**, comes with its own operation $K_{\text{dur}} : \mathbf{Dur} \rightarrow \mathbf{Dur}$ defined as $K_{\text{dur}} \sigma := \text{durcopy}_A; \sigma; \text{durcopy}_B$, as expected. Its associated notion of linearizability $\nu' \subseteq K_{\text{dur}} \nu$ is the same as crash-aware linearizability restricted to durable strategies. An important fact that we mention in passing is that restriction to durable plays $- \cap \mathbb{P}_{A \multimap B}$ defines a semifunctor from **Crash** to **Dur**. Composing this semifunctor with $-^{\#}$ therefore preserves its functoriality properties. For simplicity, we denote this composition simply as $-^{\#}$, as the context should make it clear when the durable assumption is in place. We are finally ready to define durable linearizability.

Definition 21.5.4. We say a play $s \in P_A^{\text{dur}}$ is durably linearizable to a play $t \in P_A$, written $s \text{ dur} t$, when $\text{ops}(s) \rightsquigarrow t$. We extend the notation to strategies as we did for linearizability (see Def. 13.1.1).

Our definition differs from the traditional one only in that: we do not assume the linearized trace is atomic, and we do not require that all pending invocations be removed in the linearization.

Now, for our refinement-based formulation, we define a durable lift $\text{dur}(-)$, which assigns to a strategy $\nu : A \in \underline{\text{Safe}}$ the crash-aware strategy $\text{dur}(\nu) :_{\downarrow} A \in \mathbf{Dur}$ defined by

$$\text{dur}(\nu) :_{\downarrow} A \in \mathbf{Dur} := (K_{\text{copy}} \nu)^{\sharp} \cap P_A^{\text{dur}}$$

And, indeed, $\text{dur}(-)$ does provide an appropriate lifting operation for durable linearizability.

Proposition 21.5.3. $\nu' :_{\downarrow} A \in \underline{\text{Crash}}$ is durably linearizable to $\nu : A \in \underline{\text{Safe}}$ if and only if $\nu' \subseteq \text{dur}(\nu)$.

Proof. See Proof 32.10.1. □

21.5.2 Observational Refinement and Locality

Observational Refinement

We now show an observational refinement property for durable linearizability. For this, we first note that $\text{dur}(-)$ behaves *like* a lax semifunctor, that is:

Proposition 21.5.4. For all $\sigma : A \multimap B, \tau : B \multimap C \in \text{Conc}$, $\text{dur}(\sigma); \text{dur}(\tau) \subseteq \text{dur}(\sigma; \tau)$.

Proof. See Proof 32.8.5. □

It also satisfies the following property, which $\text{str}(-)$ does not satisfy:

Proposition 21.5.5. For all $A \in \mathbf{Dur}$, $\text{dur}(\text{copy}_A) = \text{durcopy}_A$

Proof. See Proof 32.8.4. □

Each of these two results plays an important role in each of the two directions of the following equivalence with observational refinement.

Proposition 21.5.6. Let $A, B \in \underline{\text{Crash}}$. Then $\nu'_A :_{\downarrow} A$ is durably linearizable to $\nu_A : A$ if and only if whenever $\sigma : A \multimap B \in \underline{\text{Safe}}$ implements a concurrent object linearizable to ν_B using ν_A , then $\text{dur}(\sigma) :_{\downarrow} A \multimap B$ implements an object durably linearizable to ν_B using ν'_A .

Proof. See Proof 32.10.4. □

Locality

For locality, we start by defining a tensor product.

Definition 21.5.5. For crash-aware strategies $\sigma :_{\downarrow} A, \tau :_{\downarrow} B \in \underline{\mathbf{Dur}}$, we define their tensor product $\sigma \boxtimes \tau :_{\downarrow} A \boxtimes B$ as

$$A \boxtimes B := A \otimes B \qquad \sigma \boxtimes \tau := (\sigma \otimes \tau) \cap P_{A \otimes B}^{\text{dur}}$$

Proposition 21.5.7. $(\mathbf{Dur}, - \boxtimes -, 1)$ defines a symmetric monoidal category.

Proof. See Proof 32.9.3. □

With that established, establishing locality follows the same structure as for **Safe** and **Crash**.

Proposition 21.5.8. For any durable $\sigma : A, \tau : B \in \underline{\mathbf{Safe}}$, $\text{dur}(\sigma \otimes \tau) = \text{dur}(\sigma) \boxtimes \text{dur}(\tau)$.

Proof. See Proof 32.9.2. □

Proposition 21.5.9. For $\sigma, \tau :_{\downarrow} A \multimap B \in \underline{\mathbf{Dur}}$ and $\sigma', \tau' :_{\downarrow} A' \multimap B' \in \underline{\mathbf{Dur}}$, we have

$$\sigma \boxtimes \sigma' \subseteq \tau \boxtimes \tau' \implies \sigma \subseteq \tau \wedge \sigma' \subseteq \tau'$$

Proof. See Proof 32.10.3. □

Corollary 21.5.1 (Locality). For $\nu'_A :_{\downarrow} A, \nu'_B :_{\downarrow} B \in \mathbf{Dur}$ and $\nu_A : A, \nu_B : B \in \underline{\mathbf{Safe}}$:

$$\nu'_A \overset{\text{dur}}{\rightsquigarrow} \nu_A \text{ and } \nu'_B \overset{\text{dur}}{\rightsquigarrow} \nu_B \text{ if and only if } \nu'_A \boxtimes \nu'_B \overset{\text{dur}}{\rightsquigarrow} \nu_A \otimes \nu_B$$

Proof. See Proof 32.10.2. □

21.5.3 FLiT Correctness Theorem

For the FLiT correctness theorem, we must assume that ν'_{Cell} is self-linearizable to ν_{FLiT} , in the sense of §13.2. This means that we assume that $\nu_{\text{FLiT}} \subseteq \nu'_{\text{Cell}}$ in addition to $\nu'_{\text{Cell}} \rightsquigarrow \nu_{\text{FLiT}}$. Note as well that the FLiT correctness theorem also assumes that a durably linearizable FLiT implementation is available, that is that an object $\nu'_{\text{FLiT}} :_{\downarrow} \dagger\text{FLiT}$ durably linearizable to ν_{FLiT} has been established. We do this for the implementation displayed in §18.

Proposition 21.5.10 (FLiT Correctness). *For any object signature E , writing $\nu'_{\text{Mem}} := \otimes_{i \in I} \nu'_{\text{Cell}}$, if $\nu'_{\text{Mem}}; M$ is an object linearizable to ν_E then, writing $\nu'_{\text{FLiTMem}} := \boxtimes_{i \in I} \nu'_{\text{FLiT}}$, it follows that $\nu'_{\text{FLiTMem}}; \text{dur}(M)$ is durably linearizable to ν_E .*

Proof. Note that since ν'_{Cell} is linearizable to ν_{FLiT} , from locality for compositional linearizability, it follows that $\nu'_{\text{Mem}} \rightsquigarrow \nu_{\text{Mem}}$, where we write $\nu_{\text{Mem}} := \otimes_{i \in I} \nu_{\text{FLiT}}$. Then, by observational refinement for compositional linearizability and the assumption, we have:

$$\nu_{\text{Mem}}; M = \nu'_{\text{Mem}}; M \subseteq \nu_E$$

Now, by locality for durable linearizability we have that $\nu'_{\text{FLiTMem}} \subseteq \text{dur}(\nu_{\text{Mem}})$. But then, the result follows from observational refinement for durable linearizability. $\square$

Part IV

Liveness

Chapter 22

Introduction

Historically, concurrent objects have been specified by proving both linearizability w.r.t. some specification (usually understood as a safety property) and a progress property (an instance of a liveness property). For instance, the elimination-backoff stack is shown to be linearizable to an atomic stack specification, and shown to be lock-free (always, eventually, some operation returns) [Hendler et al., 2004]. This approach traces back to informal notions of correctness for multiprocessor programs, which were given proper logical meaning by the temporal logic community [Pnueli, 1977; Lamport, 1977], and eventually theoretically justified by the seminal result by Alpern and Schneider [1985] that every trace property decomposes as the conjunction of a safety and a liveness property.

Linearizability is typically used to state safety properties for concurrent objects. The reason for this is that under certain conditions, safety properties are *reflected* by linearizability, in the sense that if a concurrent object ν' is linearizable to a specification ν that satisfies a nice enough safety property ϕ then ν' also satisfies ϕ . It is well-known, however, that linearizability does not reflect liveness properties, as we shall discuss next.

The goal of this part of the dissertation is to extend linearizability into a criterion that reflects both safety and liveness properties, while achieving the same compositionality as compositional linearizability. One of the key advantages of this approach is that liveness properties can be specified in a linearized fashion, similarly to what is already done for safety properties. In our compositional verification setting, this also means that clients of a verified library only have to reason about linearized traces of the library to prove their

<pre> M₁:Counter Import C:Counter Import L:Lock inc () { L.acq(); C.inc(); L.rel(); ret ok } get () { L.acq(); v <- C.get(); L.rel(); ret v } </pre>	<pre> M₂:Counter Import C:Counter Import L:Lock inc () { L.acq(); C.inc(); b <- C.get(); L.rel(); while (b = 1) { C.get() } ret ok } get () { L.acq(); v <- C.get(); L.rel(); ret v } </pre>	<pre> M₃:Counter Import C:Counter Import L:Lock inc () { b <- C.get(); while (b = 0) { C.get() } L.acq(); C.inc(); L.rel(); ret ok } get () { L.acq(); v <- C.get(); L.rel(); ret v } </pre>	<pre> M₄:Counter Import C:Counter Import L:Lock, A:CAS inc () { L.acq(); C.inc(); L.rel(); b <- A.get(); while (b = 1) { b <- A.get() } ret ok } get () { L.acq(); A.cas(0,1); v <- C.get(); A.cas(1,0); L.rel(); ret v } </pre>
(1)	(2)	(3)	(4)

Figure 22.1: Four counter implementations: (1) A typical shared counter implementation using a lock and racy counter; (2) A modification of (1) where the first `inc` call always diverges *after* incrementing the racy counter; (3) A modification of (1) where the first `inc` call always diverges *before* incrementing the racy counter; (4) A modification of (1) where `inc` calls can diverge depending on the scheduling.

own liveness, greatly simplifying liveness verification.

That is, instead of having to state that the elimination-backoff stack is linearizable to an atomic stack and *separately* that the elimination-backoff stack is lock-free (which needs to be verified under arbitrary interleaving), we will simply state that the elimination-backoff stack is safely and lively linearizable to a lock-free atomic stack, and that will immediately reflect lock-freedom to the concrete elimination-backoff stack.

22.1 Linearizability and Progress

To understand why linearizability does not reflect progress properties, we consider several implementations of a shared counter object with signature `Counter` (see Ch. 5). We wish to implement the shared counter so that it is linearizable to the usual atomic counter ν_{Counter} specification (see Ex. 6.1.2).

Focus first on implementation (1) of Fig. 22.1. This is the implementation pattern we discussed in §11.2.1, which turns a racy counter implementation, such as a sequential counter implementation, into a shared counter linearizable w.r.t. the usual counter specification

ν_{Counter} . It imports a lock object with signature $\text{Lock} := \{\text{acq} : \mathbf{1} \rightarrow \mathbf{1}, \text{rel} : \mathbf{1} \rightarrow \mathbf{1}\}$ and implements every method by acquiring the lock, performing the counter operation, and then releasing the lock and returning. This pattern avoids races by only performing counter operations within the mutual exclusion zone. Since the imported racy counter is often wait-free, the progress of the implemented shared counter only depends on the lock. For example, with the starvation-free ticket lock [Mellor-Crummey and Scott, 1991], the resulting counter will also be starvation-free.

Consider now implementation (2) in Fig. 22.1. It works similarly, except that in the implementation of `inc`, before releasing the lock, it retrieves the value of the counter, releases the lock, and if the retrieved value was 1, it spins infinitely while invoking `get`. This means that whoever succeeds in incrementing the counter first diverges. Hence, (2) is not starvation-free. Despite that, it implements an object linearizable w.r.t. ν_{Counter} (assuming calls to `get` do not cause races). Note that a trace such as $\alpha:\text{inc} \cdot \alpha':\text{get} \cdot \alpha':1$ produced by the implementation is linearizable w.r.t. $\alpha:\text{inc} \cdot \alpha:\text{ok} \cdot \alpha':\text{get} \cdot \alpha':1 \in \nu_{\text{Counter}}$, even though in the concrete implementation the first `inc` invocation never returns. In other words, linearizability can add return events that will never happen.

It can also hide diverging invocations, as seen in implementation (3). It works similarly to (2), but differs in that it checks if the counter is 0 before incrementing it, and if it is 0, it diverges. For similar reasons as (2), it is not starvation-free, but it is still linearizable w.r.t. ν_{Counter} . Note that in this case, a trace such as $\alpha:\text{inc} \cdot \alpha':\text{get} \cdot \alpha':0$ produced by the implementation is linearizable w.r.t. $\alpha':\text{get} \cdot \alpha':0 \in \nu_{\text{Counter}}$, even though the first `inc` invocation diverges in the concrete object.

Implementation (4), in Fig. 22.1, shows an even more complex situation than those of examples (2) and (3). Essentially, the `get` method now uses a compare-and-swap object to signal that there is a `get` executing. If the `inc` method, after incrementing, detects that a `get` is running, it enters a loop. In each iteration of the loop, it checks again if there is a `get` executing, and loops again in the positive case. This has the effect that often `inc` invocations do return, but the resulting object is still not starvation-free because there are executions producing traces such as s (below) in such a way that whenever the variable `b` gets updated in the code for `inc`, some `get` invocation has not written 0 to the CAS object

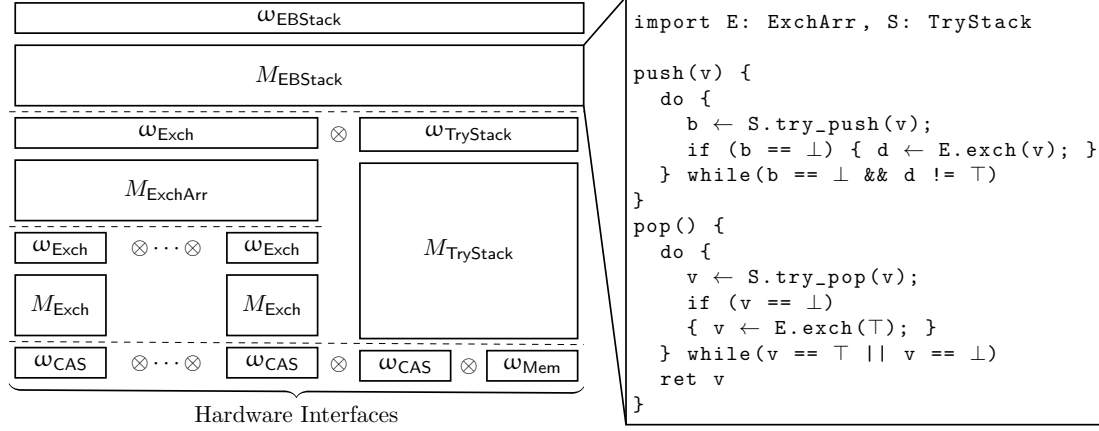


Figure 22.2: The elimination-backoff stack code (right) and the hierarchy of underlying objects (left).

yet. Despite that, we can linearize s to t :

$$s = \alpha:\text{inc} \cdot \alpha':\text{get} \cdot \alpha':1 \cdot \alpha':\text{get} \cdot \alpha':1 \cdot \dots \quad t = \alpha:\text{inc} \cdot \alpha:\text{ok} \cdot \alpha':\text{get} \cdot \alpha':1 \cdot \alpha':\text{get} \cdot \alpha':1 \cdot \dots$$

Although the `inc` may or may not return, linearizability still considers (4) linearizable w.r.t.

ν_{Counter} .

All in all, examples (2), (3), and (4) show different ways in which a `Counter` implementation can fail to make progress while still being identified as linearizable w.r.t. the same specification as (1).

22.2 The Elimination-Backoff Stack

In this section, we provide a brief and informal overview of live linearizability through our main verified example: the elimination-backoff stack [Hendler et al., 2004], whose multi-layered code structure is displayed in Fig. 22.2.

When appropriately decomposed, the elimination-backoff stack may be seen as a protocol to turn a wait-free (but with failures) stack implementation `TryStack`, into a highly efficient lock-free (without failures) stack `EBStack`. The code for the elimination-backoff stack (M_{EBStack}) implements operations by first attempting to modify the `TryStack` directly. In case of failure, it then tries to exchange a `pop`/`push` directly with another thread whose

push/pop also failed. Note that if the values on the stack are of type `Val`, we let the values for the exchanger be `Val + ⊥`. Operation failures return a failure code `⊥`. This is done using an exchanger array `ExchArr`, implemented using several exchangers `Exch` with the code M_{ExchArr} . `ExchArr` exposes to its client a typical exchanger interface, while under-the-hood avoids contention for the same exchanger by instead distributing exchanges across several underlying exchangers. If the thread finds an exchange partner, then the complementary operations exchange the pushed value, and return. Otherwise, they attempt the whole protocol again. Each individual exchanger `Exch` is implemented by M_{Exch} on top of a single CAS (compare-and-swap) object, while the `TryStack` is implemented by M_{TryStack} on top of a fragment of memory `Mem` and another CAS object. In Fig. 22.2, we display only the code for M_{EBStack} .

The stack has the signature:

$$\text{Stack} := \{\text{push} : \text{Val} \rightarrow \mathbf{1}, \text{pop} : \mathbf{1} \rightarrow \text{Val} + \text{emp}\}$$

The typical set of linearized traces of a stack is given by the strategy ν_{Stack} generated by the transition system described by the rules below, from the initial state $(\epsilon, \emptyset)$ (the second component `stv` keeps track of the threads that will no longer be scheduled):

$$\begin{array}{c} \frac{\alpha \notin \text{stv}}{(q, \text{stv}) \xrightarrow{\alpha:\text{push}(v) \cdot \alpha:\text{ok}} (v \cdot q, \text{stv})} \qquad \frac{\alpha \notin \text{stv}}{(v \cdot q, \text{stv}) \xrightarrow{\alpha:\text{pop}() \cdot \alpha:v} (q, \text{stv})} \\[10pt] \frac{\alpha \notin \text{stv}}{(\epsilon, \text{stv}) \xrightarrow{\alpha:\text{pop}() \cdot \alpha:\text{emp}} (\epsilon, \text{stv})} \qquad \frac{\alpha \notin \text{stv}}{(q, \text{stv}) \xrightarrow{\alpha:\bullet} (q, \text{stv} \cup \alpha)} \end{array}$$

Strategies inherently specify partial traces, as in a trace such as “ $\alpha:\text{push}(v)$ ” it is unclear if this call has diverged, or if its return has not happened *yet*. In other words, such specifications are *safety specifications*. Meanwhile, as Alpern and Schneider [1985] have analyzed in their seminal paper, any concurrent object may be specified as a combination of a safety and a liveness property (typically a progress property). So, to specify how an object progresses, we also consider that an object comes with a liveness specification as described

by a *win condition* ω . While ν specifies the partial behaviors of an object, ω specifies the behaviors of the object which are *fully executed*.

Such executions can be formalized as those traces s such that for every agent α , $\pi_\alpha(s)$ is either empty, ends with a response, a $\circ$ -token, a $\bullet$ -token, or is infinite. These are the traces we called *well-scheduled* in Ch. 12, and wrote $\mathbb{P}_M^{\text{ws}}$ for the set of all well-scheduled traces over M . A liveness specification ω is then a non-empty, prefix-closed (within $\mathbb{P}_M^{\text{ws}}$), set of well-scheduled traces. We consider a pair $\langle \nu, \omega \rangle$ of a safety spec. ν and a liveness spec. ω an object specification.

Example 22.2.1. Consider the four counter objects ν_1, ν_2, ν_3 , and ν_4 implemented by the implementations in Fig. 22.1. Our discussion shows that $\nu_1 \neq \nu_2$, $\nu_1 \neq \nu_3$, as the first inc invocation never returns, so a trace such as $\alpha:\text{inc} \cdot \alpha:\text{ok}$ does not appear in ν_2 or ν_3 . On the other hand, $\nu_1 = \nu_4$.

Now, the four implementations have different behaviors when we let the implementation execute indefinitely. For example, ω_2 and ω_3 should have the property that every infinite trace in them with at least one inc invocation is such that at least one inc invocation has diverged. ω_1 , on the other hand, assuming that the underlying lock is starvation-free, would only contain traces where every invocation receives a response, that is:

$$\omega_1 = \{s \in \mathbb{P}_{\text{Counter}}^{\text{ws}} \mid \downarrow s \subseteq \nu_1 \wedge \forall \alpha \in \Upsilon. \pi_\alpha(s) \in (\text{inc} \cdot \text{ok} \cup (\cup_{n \in \mathbb{N}} \text{get} \cdot n))^*\}$$

Example 22.2.2. Our notion of liveness specification is expressive enough to express all the usual progress properties. For instance, the designers of the elimination-backoff stack state its correctness by saying it is a linearizable stack, and that it is lock-free, which they give on paper arguments for. We can state the lock-freedom specification as the set of well-scheduled traces described by:

$$\text{lf} := \forall \alpha \in \Upsilon. \forall p. \forall t. s = p \cdot t. p \in \text{pend}(\alpha) \implies t \in \text{starved}(\alpha) \vee \exists \alpha' \in \Upsilon. t \in \text{conv}(\alpha')$$

which says that a well-scheduled trace s is lock-free if whenever we have a prefix $p \sqsubseteq s$ such that α has a pending invocation in p ($\text{pend}(\alpha)$) then either α is later starved by the scheduler ($\text{starved}(\alpha)$) or some potentially different thread α' returns from their own

pending operation ($\text{conv}(\alpha')$). A potential object specification for the elimination-backoff stack is then $\langle \nu_{\text{Stack}}, \text{lf} \rangle$. The definitions of all the standard progress properties may be seen in Fig. 23.1.

Typically, the notion of linearizability we discussed in Ch. 13 is used to state the safety of an object, which is why we called it safe linearizability.

Example 22.2.3. The claim that the elimination-backoff stack implementation is “linearizable” is formalized in the framework of compositional linearizability for safety as follows. The library correctness condition, written $M_{\text{EBStack}} : \nu_{\text{Exch}} \otimes \nu_{\text{TryStack}} \Rightarrow \nu_{\text{Stack}}$, is informally understood as saying that “ M_{EBStack} is linearizable w.r.t. ν_{Stack} whenever it runs on top of objects linearizable w.r.t. $\nu_{\text{Exch}} \otimes \nu_{\text{TryStack}}$ ”.

When we are given the code of M_{EBStack} , we can use the operational semantics of Ch. 17 to compute its set of (partial) traces $\llbracket M_{\text{EBStack}} \rrbracket_S$ which includes events from EBStack , but also from ExchArr and TryStack . Then, we can take the linearized safety specifications of the exchanger array ν_{Exch} (since the exchanger is a non-atomic object, the generalized linearizability is essential for this) and the try stack ν_{TryStack} , and compose (using horizontal composition $- \otimes -$) them together into $\nu_{\text{Exch}} \otimes \nu_{\text{TryStack}}$ which is guaranteed to be a valid linearization of the linking of M_{ExchArr} and M_{TryStack} by the generalized *locality* property. This results in an object with simultaneously the operations of both Exch and TryStack . Then, we may use vertical composition $(\nu_{\text{Exch}} \otimes \nu_{\text{TryStack}}) ; \llbracket M_{\text{EBStack}} \rrbracket_S$ to obtain all the (partial) stack executions produced by M_{EBStack} running on top of the exchanger array and the try-stack. The claim that $M_{\text{EBStack}} : \nu_{\text{Exch}} \otimes \nu_{\text{TryStack}} \Rightarrow \nu_{\text{Stack}}$ then is reduced to proving: $(\nu_{\text{Exch}} \otimes \nu_{\text{TryStack}}) ; \llbracket M_{\text{EBStack}} \rrbracket_S \rightsquigarrow \nu_{\text{Stack}}$ by using the observational refinement property.

By verifying each of M_{EBStack} , M_{TryStack} , M_{ExchArr} , and M_{Exch} separately against the corresponding interfaces, we are then able to compose all of them into $M = (((M_{\text{Exch}} \uplus \dots \uplus M_{\text{Exch}}) \oplus M_{\text{ExchArr}}) \uplus M_{\text{TryStack}}) \oplus M_{\text{EBStack}}$ ($\oplus$ is the usual syntactic composition operation, and $\uplus$ is library concatenation). The compositional theory guarantees that the correctness of individual components is enough to derive that $M : (\nu_{\text{CAS}} \otimes \dots \otimes \nu_{\text{CAS}}) \otimes (\nu_{\text{CAS}} \otimes \nu_{\text{Mem}}) \Rightarrow \nu_{\text{Stack}}$ holds.

Chapter 23

The Liveness Game Model

The liveness game model is simultaneously motivated by the idea of winning strategies [Joyal, 1977; Blass, 1992; Abramsky and Jagadeesan, 1992; Melliès, 2005a; Clairambault et al., 2012] appearing in models of logics, and that of divergence tracking in models of non-deterministic programming languages [Harmer and McCusker, 1999; Gowers and Laird, 2018], and essentially subsumes both.

To motivate it, we note that in a sequential model with winning strategies such as Abramsky and Jagadeesan [1992], the key challenge is the following: denotations of proofs should be *total* (in any situation where the Proponent is to move, it has a response for the Opponent). However, total strategies do not compose due to the possibility of infinite stuttering. For example, the strategies:

$$\nu : \dagger \mathbf{Unit} := \downarrow (q \cdot a)^* \qquad \sigma : \dagger \mathbf{Unit} \multimap \mathbf{Unit} := \downarrow q \cdot (q \cdot a)^*$$

are both total, as for any Opponent move, the Proponent has a response. In ν , the Proponent always responds with a , and in σ , the Proponent always responds by asking q . Their interaction, however, results in the Proponent repeatedly asking q along their interface, resulting in the strategy:

$$\nu ; \sigma := \downarrow q$$

where Proponent never responds, since it is performing infinitely many (now) silent moves.

Both of these strategies are therefore total, but their composition is not.

To deal with this issue, they extend a sequential game A with a set of “Winning Plays” W_A which consist of the P-positions as well as a subset of infinite plays of A which are considered “winning” for Proponent. A strategy is “winning” if no matter what Opponent does, the Proponent can take a move that brings it “closer” to a winning play. By consequence, if an interaction between a winning Proponent against an Opponent ever stops at a finite play, it must be by Opponent’s fault, meaning the interaction must have stopped in a Proponent position where Opponent is due to play. Therefore, intrinsic to the notion of being “winning” is the idea that a winning Proponent must terminate. This is why Abramsky describes winning strategies as a terminating program satisfying a liveness specification. In this winning plays approach, the infinite play $q \cdot (q \cdot a)^\infty$ is considered losing for Proponent, and therefore the composition issue is solved by rejecting the strategy σ as a losing strategy.

Now, the concurrent setting adds a significant challenge. Key to much of the theory of winning strategies is that strategies σ are deterministic for the Proponent in the sense that in an Opponent position s , if $s \cdot m, s \cdot n \in \sigma$ then $m = n$. This plays a crucial role in showing that when a strategy and a counter-strategy interact the resulting interaction is a down-closed chain which can be given a limit as a *single* infinite interaction (i.e., it is an ideal in P_A and attains a limit in the ideal completion of P_A , $P_A \cup P_A^\infty$). This is part of what guarantees that winning strategies compose. But the combination of determinism and termination ensures that totality does compose. In the concurrent setting, however, there is intrinsic non-determinism due to the non-deterministic interleavings of the sequential agents involved in the computation.

Similar problems arise when attempting to attain full-abstraction for non-deterministic sequential programming languages. Since the denotations of such programs will be non-deterministic, one must find a way to circumvent the need for determinism. The solution that has been fronted [Gowers and Laird, 2018; Harmer and McCusker, 1999] is to track, together with a strategy σ , its set of “diverging” plays D_σ . In this way, when the strategies $\downarrow(q \cdot (q \cdot a)^\star)$ and $\downarrow(q \cdot a)^\star$ are composed, their composition $\downarrow q$ is paired with the set of divergences $D_\sigma = \{q\}$. This ensures that strategies may be considered *total* in a different sense: in any Opponent position the Proponent either has a reply or diverges. Moreover, it

is reasonable to require that any strategy be “locally deterministic” in the sense that it can be decomposed into a choice between deterministic substrategies: $\sigma = \cup_{i \in I} \sigma_i$ with every σ_i deterministic. These deterministic substrategies are also deterministic in a more general sense: they either have a response or diverge, exclusively. Such a setup has been shown to work [Gowers and Laird, 2018; Harmer and McCusker, 1999], but results in complexity in the composition operation where $D_{\sigma;\tau} = D_\sigma; \tau \cup \sigma; D_\tau$ in the simplest version of the setup.

To that, we compound an extra challenge when one is interested in the theory of concurrent objects: we are not interested in terminating programs only! Many concurrent objects are not terminating. In fact, many objects, such as locks, are not terminating, and even reasonable concurrent data structures such as stacks and queues are often only minimally progressing (i.e., they only guarantee that some thread always terminates, but do not disallow that some unlucky thread have its progress delayed forever). This means that sometimes we want to say a program is terminating, sometimes we want to say it is diverging, sometimes we want to say that it only diverges if some other thread is progressing, etc. We also observe that there is a remarkable similarity between the winning strategy setup and the divergence tracking setup: but they differ in that winning plays (characterizing termination) are attached to the *game* (as in models of logics every proof will be required to be terminating), while divergence tracking is attached to the strategies (as programs may have different divergence behaviors). We therefore combine the two ideas and land on the concept that a *winning strategy* should consist of a strategy σ together with a set of winning plays Σ , which we now formalize.

23.1 Win Conditions

After this protracted introduction, we are ready to define win conditions and their compositional structure. Like all of our game models, we use the exact same notion of game as we have used for sequential, concurrent, and crash-aware strategies. The only change is what makes for a morphism between games.

Because we take the point-of-view of the system, the only positions relevant for liveness are the positions where the system has performed all actions it can perform without further

cooperation from the environment. Syntactically, such traces exactly correspond to well-scheduled traces (Def. 12.0.2). Recall that we defined the set of well-formed well-scheduled plays $\mathbb{P}_M^{\text{ws}}$ as:

$$s \in \mathbb{P}_M^{\text{ws}} \iff \forall \alpha \in \Upsilon. \pi_\alpha(s) \in \mathbb{SP}_M^{\text{PQ}} \uplus \mathbb{SP}_M^{\text{PA}} \uplus \mathbb{SP}_M^{\text{P}\circ} \uplus \mathbb{SP}_M^{\bullet}$$

and that the restriction of the set of valid plays over the game A , $P_A^{\leq \infty}$, restricted to only well-scheduled plays is given by $P_A^{\text{ws}} := P_A^{\leq \infty} \cap \mathbb{P}_M^{\text{ws}}$. The requirement that infinite traces are well-scheduled enforces that the scheduler fairly schedules all threads that have not been starved (i.e., received a $\bullet$ -token). Note that this does not mean we assume our scheduling is fair, but instead that it is easy to identify when a trace may be generated by a fair scheduling, as it is enough to check that no $\bullet$ -token appears in the trace.

Definition 23.1.1. A *win condition* $\Sigma :: A \multimap B$ from a game A to a game B consists of a non-empty prefix-closed set of well-scheduled plays $\Sigma \subseteq P_{A \multimap B}^{\text{ws}}$. As with strategies, we write $\omega :: A$ for a win condition $\omega :: \mathbf{1} \multimap A$.

Example 23.1.1. We improve the notion of exchanger defined in §13.2 by using the following generalized signature instead, where Val is some set of values and $\perp$ an error signal:

$$\text{Exch}[\text{Val}] := \{\text{exch} : \text{Val} \rightarrow \text{Val} + \perp\}$$

The implementation of the elimination-backoff stack makes use of an exchanger object with signature $\text{Exch}[\text{Val}_\top]$, where $\text{Val}_\top := \text{Val} + \top$ adds to the set of base values a special value $\top$.

A linearized safety spec. for an exchanger object is $\nu_{\text{Exch}}^{\text{nl}} : \text{Exch}$ defined by:

$$\begin{aligned} \nu_{\text{Exch}}^{\text{nl}} := & \downarrow (\cup_{\alpha \in \Upsilon} \cup_{v \in \text{Val}_\top} \underbrace{(\cup_{\alpha' \in \Upsilon \setminus \alpha} \cup_{v' \in \text{Val}_\top} \alpha : \text{exch}(v) \cdot \alpha' : \text{exch}(v') \cdot \alpha : v' \cdot \alpha' : v)}_{(1)} \\ & \cup \underbrace{\alpha : \text{exch}(v) \cdot \alpha : \perp}_{(2)} \cup \underbrace{\alpha : \bullet}_{(3)})^* \end{aligned}$$

The specification enforces that there are three cases for a linearized exchange, it either (1)

successfully exchanges, or (2) fails to find a partner, or (3) the agent is starved by the scheduler. If the exchanger was blocking, we would use instead:

$$\nu_{\text{Exch}}^{\text{bl}} := \downarrow (\cup_{\alpha \in \Upsilon} (\cup_{\alpha' \in \Upsilon \setminus \alpha} \alpha : \text{exch}(v) \cdot \alpha' : \text{exch}(v') \cdot \alpha : v' \cdot \alpha' : v) \cup \alpha : \bullet)^*$$

where we note that case (2) is omitted, as in a blocking exchanger threads will wait for another thread to attempt an exchange instead of returning a failure.

Linearizability is often taken as *the* notion of functional correctness, i.e., safety, for concurrent objects. Moreover, our definition of linearizability allows for non-atomic specifications. So, let $\nu'_{\text{Exch}} : E$ be the set of all traces that some exchanger library can produce, which would include traces such as:

$$s = \alpha_1 : \text{exch}(1) \cdot \alpha_2 : \text{exch}(2) \cdot \alpha_3 : \text{exch}(3) \cdot \alpha_2 : 3 \cdot \alpha_4 : \text{exch}(4) \cdot \alpha_1 : \perp$$

To be considered functionally correct it would be required of the library that ν'_{Exch} be safely linearizable w.r.t. $\nu_{\text{Exch}}^{\text{abl}}$ defined in Def. 23.1.1 (i.e., $\nu'_{\text{Exch}} \rightsquigarrow \nu_{\text{Exch}}^{\text{abl}}$). For example, the trace s is safely linearizable w.r.t. the trace $t \in \nu_{\text{Exch}}^{\text{abl}}$ below:

$$t = \alpha_2 : \text{exch}(2) \cdot \alpha_3 : \text{exch}(3) \cdot \alpha_2 : 3 \cdot \alpha_3 : 2 \cdot \alpha_1 : \text{exch}(1) \cdot \alpha_1 : \perp$$

However, note that:

$$\nu_{\text{Exch}}^{\text{bl}} \rightsquigarrow \nu_{\text{Exch}}^{\text{abl}}$$

This means that any implementation of a blocking exchanger $\nu_{\text{Exch}}^{\text{bl} \prime}$ is also linearizable to $\nu_{\text{Exch}}^{\text{abl}}$. Indeed, this just follows from the transitivity of compositional linearizability:

$$\nu_{\text{Exch}}^{\text{bl} \prime} \rightsquigarrow \nu_{\text{Exch}}^{\text{bl}} \rightsquigarrow \nu_{\text{Exch}}^{\text{abl}}$$

What this means is that safe linearizability cannot guarantee that the exchanger is non-blocking! This is yet another instance of the failure of linearizability to reflect liveness properties.

This is why the liveness specification, given by a win condition, is essential. A non-

Auxiliary Predicates:

$$\begin{aligned}
s \in \text{starved}_E(\alpha) &\iff \alpha:\bullet \in s \\
s \in \text{pend}_E(\alpha) &\iff \exists p. \exists m. \pi_\alpha(s) = p \cdot m \wedge (\lambda_E(m) = \text{OQ} \vee \lambda_E(m) = \text{P}\circ) \\
s \in \text{conv}_E(\alpha) &\iff \exists m. \lambda_E(m) = \text{PA} \wedge m \in \pi_\alpha(s)
\end{aligned}$$

Maximal and Minimal Progress:

$$\begin{aligned}
s \in \text{maxPr}_E &\iff \forall \alpha \in \Upsilon. \forall p. \forall t. s = p \cdot t \wedge p \in \text{pend}_E(\alpha) \\
&\implies t \in \text{conv}_E(\alpha) \vee t \in \text{starved}_E(\alpha) \\
s \in \text{minPr}_E &\iff \forall \alpha \in \Upsilon. \forall p. \forall t. s = p \cdot t \wedge p \in \text{pend}_E(\alpha) \\
&\implies t \in \text{starved}_E(\alpha) \vee \exists \alpha' \in \Upsilon. t \notin \text{starved}_E(\alpha') \wedge t \in \text{conv}_E(\alpha')
\end{aligned}$$

Scheduling Properties:

$$\begin{aligned}
s \in \text{fair}_E &\iff \forall \alpha \in \Upsilon. \bullet \notin \pi_\alpha(s) \\
s \in \text{iso}_E &\iff \exists \alpha \in \Upsilon. \alpha:\bullet \notin s \wedge (\forall \alpha' \in \Upsilon. \alpha \neq \alpha' \implies \alpha':\bullet \in s \vee \pi_{\alpha'}(s) = \epsilon)
\end{aligned}$$

Progress Properties:

$$\begin{aligned}
\text{lf}_E &:= \text{minPr}_E \\
\text{wf}_E &:= \text{maxPr}_E \\
\text{of}_E &:= \text{iso}_E \implies \text{maxPr}_E \\
\text{df}_E &:= \text{fair}_E \implies \text{minPr}_E \\
\text{sf}_E &:= \text{fair}_E \implies \text{maxPr}_E
\end{aligned}$$

Figure 23.1: The definitions of: maximal progress (maxPr), minimal progress (minPr); fair scheduling (fair) and isolating scheduling (iso); lock-freedom (lf), wait-freedom (wf), deadlock-freedom (df), starvation-freedom (sf), and obstruction-freedom (of). All predicates are defined over P_E^{wfs} . Our definitions are inspired by Herlihy and Shavit [2011] and feature a correction to obstruction-freedom from Liang et al. [2013].

blocking exchanger admits the following win condition:

$$\begin{aligned}
\omega_{\text{Exch}}^{\text{nbl}} &:= (\cup_{\alpha \in \Upsilon} \cup_{v \in \text{Val}_\top} (\alpha:\text{exch}(v) \cdot \alpha:\perp \\
&\cup (\cup_{\alpha' \in \Upsilon \setminus \alpha} \cup_{v' \in \text{Val}_\top} \alpha:\text{exch}(v) \cdot \alpha':\text{exch}(v') \cdot \alpha:v \cdot \alpha':v') \cup \alpha:\bullet))^\star
\end{aligned}$$

While the blocking exchanger would have the following win condition:

$$\begin{aligned}
\omega_{\text{Exch}}^{\text{bl}} &:= ((\cup_{\alpha \in \Upsilon} \cup_{v \in \text{Val}_\top} (\cup_{\alpha' \in \Upsilon \setminus \alpha} \cup_{v' \in \text{Val}_\top} \alpha:\text{exch}(v) \cdot \alpha':\text{exch}(v') \cdot \alpha:v \cdot \alpha':v' \cup \alpha:\bullet))^\star \\
&\cdot ((\cup_{\alpha \in \Upsilon} \cup_{v \in \text{Val}_\top} \alpha:\text{exch}(v) \cdot \alpha:\circ) \cup \epsilon)
\end{aligned}$$

They differ from each other only in the case an exchange does not find a partner. In $\omega_{\text{Exch}}^{\text{bl}}$ this always results in a divergence, while in $\omega_{\text{Exch}}^{\text{nbl}}$ it results in a failure signal ($\perp$). We will see that since our eventual notion of live linearizability reflects progress properties, being lively linearizable w.r.t. either specification makes an object not lively linearizable with respect to the other one.

Example 23.1.2. Fig. 23.1 shows how to define the usual progress properties for concurrent

objects, showcasing how these can be specified as win conditions.

23.2 Refinement and Composition for Win Conditions

Refinement $- \sqsupseteq_L -$ for win conditions makes sure that only the presence of a divergence token matters, but not the position in which it appears. Something critical to realize about well-scheduled plays is that they cannot end in an O -move by definition. For example, in the game **Unit** a sequential well-scheduled play starting with q necessarily has to be one of

$$q \cdot a \qquad q \cdot \circlearrowleft \qquad q \cdot \bullet$$

It is because of this fact that it is enough to remove the divergences and then compare the traces to obtain a notion of refinement that behaves like a version of $- \subseteq -$ that is sensitive to the presence but not the position of the $\circlearrowleft$ -tokens.

Definition 23.2.1. Define the operation $U_{\circlearrowleft} s$ on a play $s \in \mathbb{P}_{A \multimap B}^{\leq \infty}$ as the largest subsequence of s with no $\circlearrowleft$ -tokens. We extend it to sets of plays in the usual way.

We also define a reverse operation:

$$F_{\circlearrowleft} S := \{s' \in P_{A \multimap B}^{ws} \mid U_{\circlearrowleft} s' \in S\}$$

We say a win condition $\Sigma' :: A \multimap B$ refines a win condition $\Sigma :: A \multimap B$, written $\Sigma' \sqsupseteq_L \Sigma$, when $U_{\circlearrowleft} \Sigma' \subseteq U_{\circlearrowleft} \Sigma$.

For win conditions $\Sigma, T :: A \multimap B$, we write $\Sigma \equiv_{\circlearrowleft} T$ when $\Sigma \sqsupseteq_L T$ and $T \sqsupseteq_L \Sigma$, or, equivalently, when $U_{\circlearrowleft} \Sigma = U_{\circlearrowleft} T$.

Note that since for strategies we use $- \sqsupseteq_S - := - \subseteq -$ the corresponding equivalence would just be equality.

One of the most complex aspects of win conditions is defining composition. We have found that the design space for this is quite surprisingly large, with different end results with respect to equality ($=$) but roughly equivalent with respect to equivalence under refinement. The following approach ensures that we have a sensible definition of composition for win

conditions under $- \equiv_{\circ}$, while keeping the theory tractable.

Definition 23.2.2. Given sets of plays $S \subseteq P_{A \multimap B}^{\text{ws}}$ and $T \subseteq P_{B \multimap C}^{\text{ws}}$, we define $S \circ T \subseteq P_{A \multimap C}^{\text{ws}}$:

$$S \circ T := F_{\circ} (U_{\circ} S ; U_{\circ} T)$$

This definition again exploits the fact that well-scheduled plays cannot end with a pending O-move. So, for instance, in our infinite stuttering example in Ch. 23, we would use the win conditions ω and Σ , for ν and σ , defined by:

$$\omega :: \dagger \text{Unit} := \downarrow (q \cdot a)^{\star} \qquad \Sigma :: \dagger \text{Unit} \multimap \text{Unit} := \downarrow q \cdot (q \cdot a)^{\star}$$

Then we would have:

$$\omega \circ \Sigma = F_{\circ} (U_{\circ} \omega ; U_{\circ} \sigma) = F_{\circ} (\omega ; \sigma) = F_{\circ} \downarrow q = \{\epsilon, q \cdot \circ\}$$

With this subtle but rather algebraic definition of composition, we are able to exploit the relationship of the operations $F_{\circ} -$, $U_{\circ} -$ and $- ; -$ to show that:

Proposition 23.2.1. $- \circ -$ is associative.

Proof. See Proof 33.1.1. □

Note that this is a reasonable composition operation under $\equiv_{\circ}$, as shown by:

Proposition 23.2.2. If $S \equiv_{\circ} S' \subseteq P_{A \multimap B}^{\text{ws}}$ and $T \equiv_{\circ} T' \subseteq P_{B \multimap C}^{\text{ws}}$ then $S \circ T \equiv_{\circ} S' \circ T'$.

Proof. See Proof 33.1.2. □

We obtain, in this way, a semicategory of win conditions:

Proposition 23.2.3. Let Live be the structure described by the following data:

Objects: its objects are games A, B, C ;

Homs: as morphisms Live(A, B) the complete pre-lattice of strategies $\Sigma : A \multimap B$, $\text{T} :$

$B \multimap C$, ordered by refinement $- \sqsubseteq_L -$, with pre-sup's given by arbitrary union;

Morphism Composition: *with the categorical composition of morphisms given by $- \circ - :$*

$$\mathbf{Live}(A, B) \times \mathbf{Live}(B, C) \rightarrow \mathbf{Live}(A, C);$$

*Then, this structure defines an **Inf**-enriched semicategory.*

Proof. Associativity and congruence under $\equiv_{\circ}$ follow from Propositions 23.2.1 and 23.2.2. meet-preservation follows from the definition via $F_{\circ} \circ (- ; -) \circ U_{\circ}$ and the meet-preservation of $- ; -$ (see the proof compendium). $\square$

Finally, the monoidal structure of win conditions will be inherited from the same operation we used in the safety model (Def. 6.4.1).

Definition 23.2.3. The tensor of win conditions $\Sigma :: A \multimap A'$ and $T : B \multimap B'$ is given by $\Sigma \otimes T :: A \otimes B \multimap A' \otimes B'$.

23.3 Sidenote: A Unified Model

An alternative way to understand the $- \circ -$ composition operation is by defining an operation directly on well-scheduled plays, which, while more operational, is significantly more complex. We define it over all plays (as opposed to only well-scheduled ones) so it may land us on a unified model. The key idea is that while $- ; -$ will hide the interacting component, $- \circ -$ will not hide the presence of a divergence, either in the form of a $\circ$ -token or an infinite play in the interacting component. In case the remaining components are not infinite or have already a $\circ$ -token after hiding, $- \circ -$ will make sure to add a $\circ$ -token.

Definition 23.3.1. For $X \in \{A, B\}$ and $v \subseteq \Upsilon$.

$$b_X^v S := \{s \in P_{A \multimap B}^{\leq \infty} \mid \forall \alpha \in \Upsilon. (\circ_X \in \pi_{\alpha}(s) \vee |\pi_{\alpha}(s)| = \infty) \iff \alpha \in v\}$$

$$\#_X^v S := \{s' \in P_{A \multimap B}^{\leq \infty} \mid \exists s \in S. \forall \alpha \in \Upsilon. (\alpha \in v \wedge \pi_{\alpha}(s) \cdot \circ_X \in P_{A \multimap B}^{\leq \infty} \implies \pi_{\alpha}(s') = \pi_{\alpha}(s) \cdot \circ_X$$

$$\wedge (\alpha \notin v \implies \pi_{\alpha}(s') = \pi_{\alpha}(s)) \wedge s' \setminus (\cup_{\alpha \in v \cap \text{div}_X(s)} \alpha \circ_X) = s\}$$

$$\text{where } \text{div}_X(s) := \{\alpha \in \Upsilon \mid \alpha \circ_X \in s\}$$

Then, the divergence-aware composition $S \wp T$ of $S \subseteq P_{A \multimap B}^{\leq \infty}$ and $T \subseteq P_{B \multimap C}^{\leq \infty}$ is defined by:

$$S \wp T = \bigcup_{v \in \Upsilon} \sharp_C^v(\flat_B^v S; \flat_B^v T)$$

The key idea is that $\flat_B^v -$ selects those plays such that exactly the agents $\alpha \in \Upsilon$ have diverged in B , either because they have a $\circlearrowleft$ -token or because they make infinitely many moves in B . Then, after computing the composition with the plays of the other component that have also diverged in B , we use $\sharp_C^v -$ to ensure a divergence in C appears in the composition. Since there is no obvious place to add a $\circlearrowleft_C$ token when one is not present, we just add it wherever it can be added. In the case of win conditions, which only contain well-scheduled plays, it behaves slightly better in that $- \wp -$ only arbitrarily adds a divergence token when there is infinite stuttering in B , as in the other cases either the play is infinite in A or C , or it already has a divergence in C , which is kept in the position it appears in. This makes it more precise, but it is not associative in general. The best associativity result we were able to prove is the following.

Definition 23.3.2. We say $S \subseteq P_{A \multimap B}^{\leq \infty}$ satisfies total divergence when whenever $s \cdot \alpha \cdot \circlearrowleft_A \cdot t \in S$ it follows that $s \cdot \alpha \cdot \circlearrowleft_A \cdot \alpha \cdot \circlearrowleft_B \cdot t \in S$.

We say S satisfies $\text{OOP}\circlearrowleft$ -receptivity locally if whenever

$$s \cdot t \in S \wedge s \cdot \alpha \cdot \circlearrowleft_A \cdot \alpha \cdot \circlearrowleft_B \cdot t \in P_{A \multimap B}^{\leq \infty} \implies s \cdot \alpha \cdot \circlearrowleft_A \cdot \alpha \cdot \circlearrowleft_B \cdot t \in S$$

Let $S \subseteq P_{A \multimap B}^{\leq \infty}$. We say S satisfies rooted divergence if for any $\alpha \in \Upsilon$:

$$s \in S \wedge \exists p. \pi_\alpha(s) = p \cdot \circlearrowleft_A \implies s \setminus \alpha \cdot \circlearrowleft_A \in S$$

and

$$s \in S \wedge \exists p. \pi_\alpha(s) = p \cdot \circlearrowleft_A \cdot \circlearrowleft_B \implies s \setminus \alpha \cdot \circlearrowleft_B \setminus \alpha \cdot \circlearrowleft_A \in S$$

We are then able to show the following associativity result:

Proposition 23.3.1. For sets of plays $S \subseteq P_{A \multimap B}^{\leq \infty}$, $T \subseteq P_{B \multimap C}^{\leq \infty}$, $R \subseteq P_{C \multimap D}^{\leq \infty}$ with R

satisfying $\text{O}\odot\text{P}\odot$ -receptivity, total divergence, and rooted divergence:

$$(S \wp T) \wp R = S \wp (T \wp R)$$

Proof. See Proof 33.2.3. □

Luckily, for interpretations of implementations $\llbracket M \rrbracket_L \wp \llbracket N \rrbracket_L = \llbracket M \rrbracket_L \wp \llbracket N \rrbracket_L$. Moreover, even in the verification condition, we have $\omega \wp \llbracket M \rrbracket_L \equiv_{\odot} \omega \wp \llbracket M \rrbracket_L$. Hence, the choice of which composition to use is mostly superficial. The key drawback of $-\wp-$ is that even when composing win conditions $\Sigma \wp T$ will saturate itself with divergences. But in our refinement calculus we can always refine such a composition to a more precise specification if we would like using either $\sqsubseteq_L$ or $\leadsto$.

Now, an interesting aspect of Prop. 23.3.1 is that it provides a composition operation over $P_{A \multimap B}^{\leq \infty}$ which in some sense is consistent with both $-\wp-$ and $-\odot-$. If we are willing to tolerate the associated complexity, we could instead consider an alternative model that has aspects of safety and liveness simultaneously at the expense of much added complexity. The key idea is to define a strategy simply as

Definition 23.3.3. A unified strategy $\sigma : A \multimap B$ is a non-empty, prefix-closed set of plays $\sigma \subseteq P_{A \multimap B}^{\leq \infty}$ which is moreover totally diverging, $\text{O}\odot\text{P}\odot$ -receptive, and satisfies rooted divergence.

One would then use the $-\wp-$ operation as the composition operation. The issue with this is that we cannot ignore the $\odot$ -moves anymore, as a “unified strategy” mixes up the meaning of a partial play (a play of $P_{A \multimap B}^{\bullet}$) with that of a total play (a play of $P_{A \multimap B}^{\text{ws}}$). In particular, a “unified strategy” is allowed to contain both $q \in \sigma$, $q \cdot a$ and $q \cdot \odot \in \sigma$. While $q \cdot a$ and $q \cdot \odot$ may be interpreted as non-determinism, how should we interpret just q ? These issues make such specifications more complex to work with, especially when it comes to linearizability. We therefore opt for the separated approach we have put forth in §23.2.

Chapter 24

Live Linearizability

Toward a compatible account of saturation and linearizability in **Live**, we start by defining an appropriate idempotent element eqcopy_A , given by:

$$\text{eqcopy}_A := \{s \in P_{A \multimap A}^{\leq \infty} \mid \Downarrow s \subseteq \text{copy}_A \wedge \forall \alpha \in \Upsilon. \pi_\alpha(s) \upharpoonright_{A,-} = \pi_\alpha(s) \upharpoonright_{-,A} \vee \bullet \in \pi_\alpha(s) \}$$

In §26, we will establish that:

$$\text{eqcopy}_A \circ \text{eqcopy}_A = \text{eqcopy}_A$$

This means we have all the means to develop an account of linearizability in the liveness setting, which we will call live linearizability, following the compositional linearizability framework (Ch. 14). So we start by defining the enveloping operation associated to eqcopy .

Definition 24.0.1. For a win condition $\Sigma :: A \multimap B$, we define the win condition $K^\circ \Sigma :: A \multimap B := K_{\text{eqcopy}} \Sigma$. Concretely, $K^\circ -$ is given by the formula

$$K^\circ \Sigma := \text{eqcopy}_A \circ \Sigma \circ \text{eqcopy}_B$$

We call a win condition *equisaturated* when $K^\circ \Sigma = \Sigma$.

We say a win condition $\omega' :: A$ is *lively linearizable* w.r.t. $\omega :: A$, written $\omega' \rightsquigarrow_A \omega$,

when $\omega' \sqsupseteq_L K^\omega \omega$.

A straightforward consequence of Prop. 25.3.1 and Prop. 13.1.1 is the following concrete, although descriptive, characterization of live linearizability.

Definition 24.0.2. We say a play s over $A \multimap B$ is lively linearizable w.r.t. a play t , written $s \rightsquigarrow_{A \multimap B} t$ when $\Downarrow s \rightsquigarrow_{A \multimap B} \Downarrow t$ and, moreover, for all $\alpha \in \Upsilon$:

$$\alpha \bullet \in s \iff \alpha \bullet \in t \quad \alpha \bullet \notin s \implies \pi_\alpha(s \upharpoonright_{A,-}) = \pi_\alpha(t \upharpoonright_{A,-}) \wedge \pi_\alpha(s \upharpoonright_{-,B}) = \pi_\alpha(t \upharpoonright_{-,B})$$

Again, this definition is only relevant if it is compatible with the refinement-based definition of linearizability, and indeed:

Proposition 24.0.1.

$$K^\omega \mathsf{T} = \{s \in P_{A \multimap B}^{\text{ws}} \mid \exists t \in \mathsf{T}. s \rightsquigarrow_{A \multimap B} t\}$$

Proof. See Proof 33.4.2. □

Corollary 24.0.1.

$$s \in \text{eqcopy}_A \iff s \upharpoonright_{-,A} \rightsquigarrow_{A \multimap A} s \upharpoonright_{A,-}$$

Proof. Simply use Prop. 24.0.1 and idempotency. □

Corollary 24.0.2. $\omega' \rightsquigarrow_A \omega \iff \forall s \in \omega'. \exists t \in \omega. s \rightsquigarrow t$

Proof. Immediate from Proposition 24.0.1 and the definition of $- \rightsquigarrow -$. □

This means that we may define **Live** := Live_{eqcopy} and then we obtain a compositional calculus of live linearizability $\mathcal{L}(\mathbf{Live})$. In the remainder of this chapter, we discuss some of the key properties of live linearizability.

24.1 Progress Reflection

This characterization is enough for us to show that live linearizability is progress reflecting. Indeed, live linearizability would only be useful if we can show that when the linearized

specification satisfies a liveness property of interest, the concrete implementation also satisfies it. Fortunately, for all of the traditional progress properties, this is the case, as we now show.

The key idea of the proof lies in the observation that if ϕ is a property *reflected* by live linearizability, in the sense that:

$$s \rightsquigarrow t \wedge t \in \phi \implies s \in \phi$$

and if ψ satisfies invariant upon live linearizability:

$$s \rightsquigarrow t \implies (s \in \psi \iff t \in \psi)$$

then the property $\phi \implies \psi$ is also reflected by linearizability. So below, we show the basic progress properties are reflected by linearizability, while the scheduling assumptions satisfy the equivalence described above.

Proposition 24.1.1 (Max./Min. Progress Reflection). *Let $\omega', \omega :: A$ and assume $\omega' \rightsquigarrow \omega$. Then, if $\omega \sqsupseteq_L \rho$ where $\rho \in \{\text{maxPr}_A, \text{minPr}_A\}$ then $\omega' \sqsupseteq_L \rho$.*

Proof. Note that since both maxPr_A and minPr_A satisfy the condition in Prop. 33.1.3, it is enough to show the claim w.r.t. $\subseteq$.

Maximal Progress: Suppose $s' \in \omega'$. By assumption, there is $s \in \omega$ such that $s' \rightsquigarrow s$.

By definition of $-\rightsquigarrow-$, we have that $\forall \alpha \in \Upsilon$ that is not starved, $\pi_\alpha(s') = \pi_\alpha(s)$ (note that for starved agents maximal progress is trivial). So let m be an O-move, α a non-starved agent, and $p' \cdot \alpha:m \cdot t' = s'$. Since

$$\pi_\alpha(p' \cdot \alpha:m) \sqsubseteq \pi_\alpha(s') = \pi_\alpha(s)$$

It follows that $\alpha:m$ appears in s , so that there are p and t such that $p \cdot \alpha:m \cdot t = s$. Since $s \in \omega \sqsupseteq_L \text{maxPr}_A$, there exists a response $\alpha:n$ to m , that is: $s = p \cdot \alpha:m \cdot t_1 \cdot \alpha:n \sqsubseteq s$. But then,

$$\pi_\alpha(p \cdot \alpha:m \cdot t_1 \cdot \alpha:n) \sqsubseteq \pi_\alpha(s) = \pi_\alpha(s')$$

so that $\alpha:m$ also has a response in s' .

Minimal Progress: Suppose $s' \in \omega'$. By assumption, there is $s \in \omega$ such that $s' \leadsto s$. In particular $\forall \alpha \in \Upsilon$ that is not starved in s , $\pi_\alpha(s') = \pi_\alpha(s)$ (note that minimal progress is trivial for starved agents). So let m be an O-move, $\alpha \in \Upsilon$ a non-starved agent, and let p' and t' be such that $p' \cdot \alpha:m \cdot t' = s'$. Since

$$\pi_\alpha(p' \cdot \alpha:m) \sqsubseteq \pi_\alpha(s') = \pi_\alpha(s)$$

It follows that $\alpha:m$ appears in some prefix $p \cdot \alpha:m \sqsubseteq s$. Since s makes minimal progress, there is an extension $p \cdot \alpha:m \cdot t \cdot \alpha':n \sqsubseteq s$ with n a P-move and α' a non-starved agent. Now, we have:

$$\pi_\alpha(p \cdot \alpha:m \cdot t \cdot \alpha':n) \sqsubseteq \pi_\alpha(s) = \pi_\alpha(s')$$

$$\pi_{\alpha'}(p \cdot \alpha:m \cdot t \cdot \alpha':n) \sqsubseteq \pi_{\alpha'}(s) = \pi_{\alpha'}(s')$$

Therefore, $\alpha':n$ appears in s' . Since no rewrite rule allows for the re-ordering $\alpha':n \cdot \alpha:m \leadsto \alpha:m \cdot \alpha':n$, it follows that $\alpha':n$ appears after $\alpha:m$ in s' , as desired.

□

Proposition 24.1.2 (Scheduling Invariance). *Let $s, t \in \mathbb{P}_A^{\leq \infty}$ be plays such that $s \leadsto t$, then,*

$$s \in \text{fair} \iff t \in \text{fair} \quad \text{and} \quad s \in \text{iso} \iff t \in \text{iso}$$

Proof. Note that $\alpha:\bullet \in s \iff \alpha:\bullet \in t$ by definition of live linearizability. Hence, $s \in \text{fair} \iff t \in \text{fair}$.

Now, note that by the above, if either s or t has an isolated agent, then the other must also have the same isolated agent. Indeed, the only missing case to consider is when for some α , $\pi_\alpha(s) = \epsilon$ (or $\pi_\alpha(t) = \epsilon$). But in that case, α is not starved, so $\pi_\alpha(t) = \epsilon$ (resp. $\pi_\alpha(s) = \epsilon$). Hence, when $\alpha:\bullet \notin s$,

$$\pi_\alpha(s) = \epsilon \iff \pi_\alpha(t) = \epsilon$$

So,

$$s \in \text{iso} \iff t \in \text{iso}$$

□

Corollary 24.1.1. *Any property ρ of the form $\text{sched} \implies \text{prog}$ where $\text{sched} \in \{\top_A, \text{fair}_A, \text{iso}_A\}$, and $\text{prog} \in \{\text{maxPr}_A, \text{minPr}_A\}$ is reflected by live linearizability.*

Proof. Let $\omega' \rightsquigarrow \omega$ and $\omega \sqsubseteq_L \rho$. Note that $\text{sched} \implies \text{prog} \equiv \neg \text{sched} \cup \text{prog}$.

First, we note that since $F_\cup U_\cup -$ is a closure operator, and $\neg \text{sched}$ and prog are closed under it, it follows from Prop. 33.1.3 that we may show reflection w.r.t. $\sqsubseteq$.

By Prop. 24.0.1 and join-preservation of $- \circ -$, we have

$$\omega' \sqsubseteq_L K^\omega \omega \sqsubseteq_L K^\omega \rho = K^\omega \neg \text{sched} \cup K^\omega \text{prog}$$

Now, let $s \in \omega'$. By assumption, then again by Prop. 24.0.1, there is t in either $\neg \text{sched}$ or prog such that $s \rightsquigarrow t$. In the first case, by Prop. 24.1.2 it follows that $t \in \neg \text{sched}$ (note that $s \in \top \iff t \in \top$ is trivial), and in the second case, by Prop. 24.1.1 we have $s \in \text{prog}$. Either way, $s \in \rho$. □

Corollary 24.1.2 (Progress Reflection). *Every property in Fig. 23.1 is reflected by live linearizability.*

Proof. All such properties are of the form $\text{sched} \implies \text{prog}$. □

24.2 Honest Linearizability

While Def. 24.0.2 is simple, it is more descriptive than constructive. Here, we provide a definition which is more constructive and amenable to mechanical reasoning. This is, in fact, the definition we use when proving soundness of the program logic we develop for live linearizability.

Definition 24.2.1. Given plays $s, t \in \mathbb{P}_A^{\leq \infty}$, we say s *honestly linearizes* to t , written $s \rightsquigarrow t$, when there exists a relation $R \subseteq \downarrow s \times \downarrow t$ satisfying:

Simulation:

- For every $s' \sqsubseteq s$, there exists t' with $(s', t') \in R$.
- For every $p \sqsubseteq t$, there exists $(s', t') \in R$ with $p \sqsubseteq t'$.

Invariant: For every $(s', t') \in R$:

Linearizability: $U_{\circlearrowleft} s' \rightsquigarrow U_{\circlearrowleft} t'$

Honesty: For every $\alpha \in \Upsilon$ s.t. $\alpha \bullet \notin s$,

$$\pi_{\alpha}(s') \cdot \pi_{\alpha}(t' \setminus s') \sqsubseteq \pi_{\alpha}(s) \qquad \pi_{\alpha}(t') \cdot \pi_{\alpha}(s' \setminus t') \sqsubseteq \pi_{\alpha}(t)$$

For sets of plays $S, T \subseteq \mathbb{P}_M^{\leq \infty}$ we write $S \rightsquigarrow T$ when $\forall s \in S. \exists t \in T. s \rightsquigarrow t$.

The definition is more complex, but it is significantly more amenable to mechanical reasoning as it permits reasoning about the linearizability of potentially infinite traces by instead inducting or co-inducting over them while maintaining linearizability, honesty, and progress. It is in this fashion that we prove the soundness of our program logic. Of course, this is only justified if honest linearizability implies live linearizability, which we now prove.

Proposition 24.2.1. *For win conditions $\omega', \omega :: A$, $\omega' \rightsquigarrow \omega \iff \omega' \rightsquigarrow \omega$.*

Proof. See Proof 33.5.2. □

Chapter 25

Relating Safety to Liveness

At the core of our development is the idea that we want to characterize a computational unit in terms of both its safety and liveness. According to the seminal work of [Alpern and Schneider, 1985], this is enough to characterize any set of behaviors, as discussed in Ch. 4 (well, in the current game semantics setting, any behavior for the Proponent, as we characterize strategies and win conditions from the point-of-view of the Proponent). The Alpern-Schneider characterization of liveness properties has been criticized, however, even at its inception. This has led to refinements of the idea. A possible refinement is that it should be a G_δ dense set [Kwiatkowska, 1991]. For our purposes, we use strategies as safety properties, and win conditions as liveness properties. Now, we note that in our context, where strategies and win conditions specify properties of the system (Proponent), density may be formulated as a notion of *completeness* we will soon define.

That said, inspired by the Alpern and Schneider [1985] result, one first attempt at characterizing safety and liveness is to simply consider the structure **Safe** $\times$ **Live**, where $\times$ is the product of the two (enriched) semicategories. That is, a computational component is specified by a pair $\langle \sigma, \Sigma \rangle$ where the structure of the two sides is treated component-wise. We call such a pair an object. From what we have seen so far, this would be enough to obtain a semicategory with safety and liveness, which we could churn through the compositional linearizability framework. While we will follow this approach generally, we will also develop refined models where σ and Σ will be related to each other more deeply, for which we start developing some notions now.

25.1 Object Specifications

Alpern and Schneider [1985]’s theorem suggests that we may define an object specification over the type E as a pair $\langle \nu, \omega \rangle$ of a strategy $\nu : E$ and a win condition $\omega :: E$. Essentially, our base model for objects is given by the product **Safe** $\times$ **Live**.

Definition 25.1.1. We call a pair $\langle \sigma, \Sigma \rangle$ of a strategy $\sigma : A \multimap B$ and a win condition $\Sigma :: A \multimap B$ an *object*.

Given objects $O = \langle \sigma, \Sigma \rangle : A \multimap B$ and $O' = \langle \tau, T \rangle : B \multimap C$ we define their composition $O ; ; O' := \langle \sigma ; \tau, \Sigma \circ T \rangle$.

We say $O' = \langle \sigma', \Sigma' \rangle : A \multimap B$ is a refinement of $O = \langle \sigma, \Sigma \rangle : A \multimap B$, written $O' \sqsupseteq_O O$, when $\sigma' \sqsupseteq_S \sigma$ and $\Sigma \sqsupseteq_L \Sigma'$.

It is not surprising then that $\mathcal{L}(\mathbf{Safe} \times \mathbf{Live}) = \mathcal{L}(\mathbf{Safe}) \times \mathcal{L}(\mathbf{Live})$. In particular, the corresponding notion of linearizability is just:

Definition 25.1.2. An object spec. $O' = \langle \nu', \omega' \rangle$ is said to be safely and lively linearizable w.r.t. $O = \langle \nu, \omega \rangle$, written $O' \approx O$, when $\nu' \rightsquigarrow \nu$ and $\omega' \rightsquigarrow \omega$.

Combining Prop. 13.1.1 and 24.0.2, and defining $\text{objcopy} := \langle \text{copy}, \text{eqcopy} \rangle$ we obtain:

Corollary 25.1.1. For object specs. $\langle \nu', \omega' \rangle, \langle \nu, \omega \rangle : A$,

$$\langle \nu', \omega' \rangle \approx \langle \nu, \omega \rangle \iff \langle \nu', \omega' \rangle \sqsupseteq_O \langle \nu, \omega \rangle ; ; \text{objcopy}_A$$

While this provides a simple and flexible treatment of object specs, it allows for object specs. that are not intuitive. For example, it could be that no non-empty trace satisfies the object spec. $\langle \nu, \omega \rangle$ even when both ν and ω are non-trivial (e.g. $\langle \{\epsilon, q, q \cdot a\}, \{\epsilon, q \cdot \circ\} \rangle$). One of the reasons for this is that so far we have not made any density requirements (in the topological sense) out of our liveness specs ω . We will soon consider refinements of the notion of an object spec. arriving at two dual conditions *completeness* and *tightness*, and call an object specification satisfying both *precise*. Since the subject is technically dense, for now, it suffices to say that precise specs. allow us to unify safety and liveness reasoning into just liveness reasoning:

Proposition 25.1.1. *For complete $\langle \nu', \omega' \rangle$ and tight $\langle \nu, \omega \rangle$, $\langle \nu', \omega' \rangle \approx \langle \nu, \omega \rangle \iff \omega' \rightsquigarrow \omega$.*

As a result, if we assume that linearized object specs. must be precise, it is enough to verify live linearizability. This is the approach we choose to take in our verified examples.

Example 25.1.1. We briefly discuss different ways to specify an object using different object specs, noting whether they are precise to develop an intuitive understanding of precision. For this, we consider a stack with the signature **Stack** from §5.1.1, and the typical linearized safety spec. for a stack ν_{Stack} in Ex. 22.2. Let $O'_{\text{Stack}} = \langle \nu'_{\text{Stack}}, \omega'_{\text{Stack}} \rangle$ be the object spec. generated (see §14) by some concrete concurrent stack implementation. Our correctness criterion for the object implementation is that the object spec. O'_{Stack} be safely and lively linearizable w.r.t. its linearized object spec.

If we only wish to verify partial correctness, we would use the linearized specification $\langle \nu_{\text{Stack}}, \top_{\text{Stack}} \rangle$ where $\top_{\text{Stack}} := P_{\text{Stack}}^{\text{ws}}$. Since any liveness spec. is lively linearizable w.r.t. $\top_{\text{Stack}}$, this is equivalent to requiring just $\nu'_{\text{Stack}} \rightsquigarrow \nu_{\text{Stack}}$, retrieving the safe linearizability requirement. This specification is not precise both intuitively and in our formal sense.

Efficient stack implementations like the Treiber stack [Treiber et al., 1986], or the elimination-backoff stack [Hendler et al., 2004] are linearizable lock-free stacks. This is equivalent to requiring that O'_{Stack} be safely and lively linearizable w.r.t. $\langle \nu_{\text{Stack}}, \text{lf}_{\text{Stack}} \rangle$ (see Fig. 23.1). This guarantees that the implementation is lock-free, due to Prop. 24.1.2. This is an over-approximation in that there are traces in lf_{Stack} that are not functionally correct, as it merely states the progress of the object, so it is not a precise object spec.

We could precisely describe the (linearized) scenarios in which divergence can happen in either of these implementations. Alternatively, we could use the liveness spec. $\omega_{\text{Stack}}^{\text{lf}} := \{s \in \text{lf}_{\text{Stack}} \mid \downarrow \text{complete}(s) \subseteq \nu_{\text{Stack}}\}$ where **complete** removes any operations from s that are marked with a divergence. $\omega_{\text{Stack}}^{\text{lf}}$ therefore enforces both lock-freedom and functional correctness. Note $\omega_{\text{Stack}}^{\text{lf}}$ only allows a divergence to appear in infinite traces, so the infinite behaviors are essential. This makes for a precise spec. for a lock-free stack. Because this spec. is precise, Prop. 25.1.1 ensures that $\omega'_{\text{Stack}} \rightsquigarrow \omega_{\text{Stack}}^{\text{lf}} \implies \nu'_{\text{Stack}} \rightsquigarrow \nu_{\text{Stack}}$.

Example 25.1.2. $O_{\text{exch}}^{\text{nbl}} := \langle \nu_{\text{Exch}}^{\text{nbl}}, \omega_{\text{Exch}}^{\text{nbl}} \rangle$ (see Ex. 23.1.1 and Ex. 23.1.1) is a precise

specification of a non-blocking exchanger.

Example 25.1.3. Unless provided very strong primitives, wait-free stack implementations are impossible. Because of this, wait-free stacks often allow operations to fail. We could specify this by allowing stack operations to non-deterministically fail, but non-deterministic specifications can lead to complicated reasoning for their clients. An alternative, which plays a role in our main example, is to instead note that such failures are always caused by interference from another agent. So we can raise the degree of concurrency of the specification to a set-sequential specification [Neiger, 1994; Castañeda et al., 2015] which is deterministic. This is quite useful for progress verification as it guarantees that when operations fail, some other operation must succeed concurrently with it.

For this, let $\text{TryStack} := \{\text{try\_push} : \text{Val} \rightarrow \mathbf{1} + \perp, \text{try\_pop} : \mathbf{1} \rightarrow \text{Val} + \text{emp} + \perp\}$. Then, a single transaction of the try-stack is given by trace fragments that may be generated by the grammar:

$$\begin{aligned} T := & \cup_{\alpha \in \Upsilon} \alpha : \text{try\_push}(v) \cdot \alpha : \text{ok} \mid \cup_{\alpha \in \Upsilon} \alpha : \text{try\_pop}() \cdot \alpha : v \\ & \mid \cup_{\alpha \in \Upsilon} \alpha : \text{try\_push}(v) \cdot T \cdot \alpha : \perp \mid \cup_{\alpha \in \Upsilon} \alpha : \text{try\_pop}() \cdot T \cdot \alpha : \perp \end{aligned}$$

That is, every transaction consists of a single successful operation, surrounded by failed operations. Then, the safety specification ν_{TryStack} of the try-stack is: $\nu_{\text{TryStack}} := \{s \in P_{\text{TryStack}} \mid s \in T \wedge \text{succ}(s) \in \nu_{\text{Stack}}\}$ where $\text{succ}(s)$ is the subsequence of s with only successful operations. In our example, this stack is given the liveness spec.: $\omega_{\text{TryStack}} := \{s \in \text{wf}_{\text{Stack}} \mid \Downarrow s \subseteq \nu_{\text{TryStack}}\}$, which makes $\langle \nu_{\text{TryStack}}, \omega_{\text{TryStack}} \rangle$ a precise object spec.

The compositional linearizability framework then implies a quite strong observational refinement property in Prop. 25.1.2.

Proposition 25.1.2 (Safe and Live Observational Refinement). *For object specs. $\langle \nu', \omega' \rangle, \langle \nu, \omega \rangle : E$, $\langle \nu', \omega' \rangle \approx \langle \nu, \omega \rangle$ if and only if for any object $\langle \sigma, \Sigma \rangle : E \rightarrow F$, $\langle \nu', \omega' \rangle ; ; \langle \sigma, \Sigma \rangle \sqsupseteq_O \langle \nu, \omega \rangle ; ; \langle \sigma, \Sigma \rangle$.*

We also introduce a horizontal composition operation into this compositional picture.

Definition 25.1.3. We extend the tensor to object specifications in the obvious way:

$$\langle \nu, \omega \rangle \otimes \langle \nu', \omega' \rangle := \langle \nu \otimes \nu', \omega \otimes \omega' \rangle.$$

This will then satisfy locality, observational refinement, and more as discussed in Ch. 14.

Observational refinement, together with Prop. 24.1.2, implies that:

Corollary 25.1.2. *If $\Sigma : \langle \nu_A, \omega_A \rangle \Rightarrow \langle \nu_B, \omega_B \rangle$ is verified and $\omega_B \sqsupseteq_O \rho$ where ρ is one of the progress properties in Fig. 23.1, then for any object specification $\langle \nu'_A, \omega'_A \rangle : E$ such that $\langle \nu'_A, \omega'_A \rangle \approx \langle \nu_A, \omega_A \rangle$, $\omega'_A \circ \Sigma \sqsupseteq_L \rho$.*

In other words, if $\Sigma : \langle \nu_A, \omega_A \rangle \Rightarrow \langle \nu_B, \omega_B \rangle$ and ω_B satisfies some progress property, then so does the object that Σ implements when running on top of any object safely and lively linearizable w.r.t. $\langle \nu_A, \omega_A \rangle$. This is crucial to justifying our compositional verification approach for progress properties. It is worth reminding this is just part of the structure of $\mathcal{L}(\mathbf{Safe} \times \mathbf{Live})$. In particular, we also obtain the calculus described in Ch. 14.

25.2 Totality, Completeness and Tightness

Given a pair $\langle \sigma, \Sigma \rangle$, a play s should be consistent with it, say $s \models \langle \sigma, \Sigma \rangle$, when $\downarrow U_\circ s \in \sigma$ (it satisfies the safety specification) and when there is a way to extend s into $s \cdot t$ where t only has moves by the Proponent such that $s \cdot t \in \Sigma$. In that case, s is a behavior observable from a computational component satisfying $\langle \sigma, \Sigma \rangle$. We can formalize some of these notions in the following.

Definition 25.2.1. For a well-scheduled play s define

$$\Downarrow s := \downarrow U_\circ s$$

We extend this to a win condition $\Sigma :: A \multimap B$, by the obvious formula:

$$\Downarrow \Sigma := \downarrow U_\circ \Sigma$$

For a strategy $\sigma : A \multimap B$, a play s is said **α :P-maximal** in σ if it is finite and for any

$m \in M_{A \multimap B}^\bullet$ if $s \cdot \alpha : m \in \sigma$ then m is not a P-move. We write $\alpha : \text{P-max}(\sigma)$ for the set of $\alpha : \text{P-maximal}$ plays in σ .

We say a play is P-maximal if it is P-maximal for every $\alpha \in \Upsilon$, and write

$$\text{P-max}(\sigma) := \bigcap_{\alpha \in \Upsilon} \alpha : \text{P-max}(\sigma)$$

for the set of P-maximal plays of σ and

$$\alpha : \text{P-max}_{\circlearrowleft}(\sigma) := F_{\circlearrowleft} \alpha : \text{P-max}(\sigma)$$

for the set of $\alpha : \text{P-maximal}$ plays in σ after completing any pending invocations with a divergence token, and similarly for $\text{P-max}_{\circlearrowleft}(\sigma) := F_{\circlearrowleft} \text{P-max}(\sigma)$.

Finally, we define

$$F_L \sigma := \{s \in P_{A \multimap B}^{ws} \mid \Downarrow s \subseteq \sigma\}$$

It is easy to see that for any strategy $\sigma : A \multimap B$, both $\text{P-max}_{\circlearrowleft}(\sigma), F_L \sigma :: A \multimap B$ define a win condition, and for any win condition $\Sigma :: A \multimap B$, $\Downarrow \Sigma : A \multimap B$ defines a strategy. Now, we are ready to formalize our notion of density.

Definition 25.2.2. We say a win condition $\Sigma :: A \multimap B$ is *total* if for any $p \in \Downarrow \Sigma$, there exists t containing only (potentially none) P-moves such that $p \cdot t \in \Sigma$.

We avoid formalizing how this corresponds to density in some appropriate topology as it isn't directly relevant for our results, so Ch. 4 will suffice. We give some informal intuition instead. As we saw, $\Downarrow \Sigma$ is the safety property (strategy) implied by the win condition Σ . Because of this, if σ is a safety property, then

$$\sigma \cap U_{\circlearrowleft} \Sigma \sqsupseteq_S \Downarrow \Sigma$$

Hence, when considering the conjunction of a safety property against Σ , we need only consider $\Downarrow \Sigma$. Now, if p is a finite play that is not already in Σ , then density will have it that it can be extended into a play that is in Σ . Since we only characterize safety from the point-of-view of the system, we require instead that it can be given a P-extension instead.

This is exactly what totality enforces. We call it *totality*, instead of density, as it is not the usual notion of density in that it is conditional on taking a sub-topology implied by Σ . Moreover, the requirement we make is equivalent to the usual requirement that the strategy $\Downarrow \Sigma$ be total in the game semantics sense (where $\circlearrowleft$ -tokens are considered as valid P-responses). Hence, we find the term *total* appropriate. Note that in a total win condition Σ , a partial execution $s \in \sigma$ can always be validated by some P-extension in Σ , which enforces the usual informal understanding of density that “a liveness property is one that cannot be falsified by any partial execution”.

Thinking of a strategy $\nu : A$ as a safety property, $\text{P-max}_{\circlearrowleft}(-)$ and F_L – provide two solutions to a liveness property consistent with ν , in the sense that both satisfy:

Proposition 25.2.1. *For any strategy $\sigma : A \multimap B$, $\text{P-max}_{\circlearrowleft}(\sigma)$ and $F_L \sigma$ are both total. Moreover,*

$$\Downarrow \text{P-max}_{\circlearrowleft}(\sigma) = \sigma \qquad \Downarrow F_L \sigma = \sigma$$

and for any total win condition $\Sigma :: A \multimap B$ such that $\Downarrow \Sigma = \sigma$ it follows that

$$\text{P-max}_{\circlearrowleft}(\sigma) \sqsupseteq_L \Sigma \sqsupseteq_L F_L \sigma$$

We give some intuition to what these two properties are. Consider the safety specification $S = \downarrow(q \cdot a)$. In this case we have:

$$\text{P-max}_{\circlearrowleft}(S) = \epsilon \cup q \cdot a \qquad F_L S = \epsilon \cup q \cdot a \cup q \cdot \circlearrowleft$$

We see therefore that $\text{P-max}_{\circlearrowleft}(-)$ results in the liveness property stating the termination of q while F_L – results in the liveness property stating that q might non-deterministically terminate or not. The two also differ in what they say about infinite plays. If we let $S' = \downarrow(q \cdot a)^*$ then:

$$\text{P-max}_{\circlearrowleft}(S') = (q \cdot a)^* \qquad F_L S' = (q \cdot a)^* \cdot (q \cdot \circlearrowleft \cup \epsilon)$$

where we see that $\text{P-max}_{\odot}(S')$ includes no infinite plays while $F_L S'$ does. We have to be careful around the infinite plays that are included in a liveness property Σ , something we discuss soon. For now, we note that:

Proposition 25.2.2.

$$\begin{array}{ccc} & \xrightarrow{F_L -} & \\ \underline{\text{Safe}}(A, B) & \top & \underline{\text{Live}}(A, B) \\ & \xleftarrow{\Downarrow -} & \end{array}$$

Defines a Galois insertion.

Note that $F_L -$ is not join-preserving due to its infinite plays. Indeed, consider, for instance:

$$F_L \cup_{n \in \mathbb{N}} \downarrow(q \cdot a)^n = (q \cdot a)^* \cdot (q \cdot \odot + \epsilon) \neq \cup_{n \in \mathbb{N}} (q \cdot a)^n \cdot (q \cdot \odot + \epsilon) = \cup_{n \in \mathbb{N}} F_L \downarrow(q \cdot a)^n$$

Meanwhile, $\text{P-max}_{\odot}(-)$ should result in a Galois insertion $\text{P-max}_{\odot}(-) \dashv \Downarrow -$ where $\text{P-max}_{\odot}(-)$ is the left adjoint instead. However, we face a key challenge: $\text{P-max}_{\odot}(-)$ is not even monotonic. Indeed:

$$\text{P-max}_{\odot}(\downarrow q) = \epsilon \cup q \cdot \odot \not\sqsubseteq_L \epsilon \cup q \cdot a = \text{P-max}_{\odot}(\downarrow(q \cdot a))$$

To avoid this issue, it would be necessary to annotate P-maximal plays in strategies with $\odot$ -tokens before comparing strategies so that behaviors such as q are not comparable to $q \cdot a$. We use the simple refinement we proposed instead which will be enough for our purposes. We note, however, that $\text{P-max}_{\odot}(-)$ satisfies many of the properties that would hold of a Galois connection, such as:

Lemma 25.2.1. *For any total win condition $\Sigma :: A \multimap B$, $\text{P-max}_{\odot}(\Downarrow \Sigma) \sqsupseteq_L \Sigma$.*

We defer the discussion connecting $\text{P-max}_{\odot}(-)$ to eqcopy_A until §25.3, after we have introduced the necessary definitions. For the following few sections it is enough to accept that $\text{objcopy}_A = \langle \text{copy}_A, \text{eqcopy}_A \rangle$ defines a tight complete object, which we will justify in §25.3.

25.3 Completeness

Now, part of the goal of discussing $\mathbf{P}\text{-max}_{\circlearrowleft}(-)$ and $\Downarrow-$ is that objects $\langle\sigma, \Sigma\rangle$ such that σ and Σ are related to each other by $\mathbf{P}\text{-max}_{\circlearrowleft}(-)$ and $\Downarrow-$ are more well-behaved in our verification calculus. However, the challenge at this point is that such objects are harder to compose. A preliminary definition, which we will progressively refine, is that such an object $\langle\sigma, \Sigma\rangle$ is *tight* when $\Downarrow\Sigma \sqsupseteq_S \sigma$ and is *winning* (alternatively, *total*) when $\mathbf{P}\text{-max}_{\circlearrowleft}(\sigma) \sqsupseteq_L \Sigma$. The key challenge starts with the fact that being merely total is not enough to compose due to the possibility of infinite chattering, a notion of completeness is also necessary, which we now develop.

Definition 25.3.1. We call a play $s \in P_{A \multimap B}^{\text{ws}}$ a limit of σ if for every prefix $p \sqsubseteq s$ and every agent α there exists an extension t such that $p \cdot t \sqsubseteq s$ and $p \cdot t \in \alpha\text{-P-max}(\sigma)$. We write $\uparrow\sigma$ for the set of infinite limits of σ and

$$\mathbf{P}\text{-max}_{\circlearrowleft}^{\infty}(\sigma) := \mathbf{P}\text{-max}_{\circlearrowleft}(\sigma) \cup F_{\circlearrowleft} \uparrow\sigma$$

We say an object $\langle\sigma, \Sigma\rangle$ is *complete* when it is total and Σ contains all of σ 's limits, i.e. $F_{\circlearrowleft} \uparrow\sigma \sqsupseteq_L \Sigma$.

The reason why completeness must be a strengthening of totality follows from the simple fact that:

Lemma 25.3.1. *If $s \in \mathbf{P}\text{-max}(-)$ then s is a limit of σ .*

Note that for any $\sigma : A \multimap B$, $\langle\sigma, \mathbf{P}\text{-max}_{\circlearrowleft}^{\infty}(\sigma)\rangle$ is a complete object. Note that $\mathbf{P}\text{-max}_{\circlearrowleft}^{\infty}(\sigma)$ is also trivially total, as $\mathbf{P}\text{-max}_{\circlearrowleft}(\sigma) \subseteq \mathbf{P}\text{-max}_{\circlearrowleft}^{\infty}(\sigma)$ (by Lemma 25.3.1). By Prop. 25.2.1 we also have $\mathbf{P}\text{-max}_{\circlearrowleft}^{\infty}(\sigma) \sqsupseteq_L F_L \sigma$. Hence, it is easy to check that a corollary of Prop. 25.2.1 is:

Corollary 25.3.1. *For any strategy $\sigma : A \multimap B$, $\mathbf{P}\text{-max}_{\circlearrowleft}^{\infty}(\sigma)$ and $F_L \sigma$ are both complete. Moreover,*

$$\Downarrow\mathbf{P}\text{-max}_{\circlearrowleft}^{\infty}(\sigma) = \sigma \qquad \Downarrow F_L \sigma = \sigma$$

and for any complete win condition $\Sigma :: A \multimap B$ such that $\Downarrow \Sigma = \sigma$ and $\langle \sigma, \Sigma \rangle$ is complete it follows that

$$\mathbf{P}\text{-max}_{\odot}^{\infty}(\sigma) \sqsubseteq_L \Sigma \sqsubseteq_L F_L \sigma$$

In particular, an object $\langle \sigma, \Sigma \rangle$ is complete if and only if

$$\mathbf{P}\text{-max}_{\odot}^{\infty}(\sigma) \sqsubseteq_L \Sigma$$

With the definition of $\mathbf{P}\text{-max}_{\odot}^{\infty}(-)$ in hand, we can now observe that:

Proposition 25.3.1.

$$s \in \mathbf{P}\text{-max}_{\odot}^{\infty}(\text{copy}_A) \iff \Downarrow s \subseteq \text{copy}_A \wedge \forall \alpha \in \Upsilon. \pi_{\alpha}(s) \upharpoonright_{A,-} = \pi_{\alpha}(s) \upharpoonright_{-,A} \vee \bullet \in \pi_{\alpha}(s)$$

So that in reality, we defined eqcopy_A so that the following holds:

Corollary 25.3.2. *For any A , $\text{eqcopy}_A = \mathbf{P}\text{-max}_{\odot}^{\infty}(\text{copy}_A)$.*

In §26, we will formally equip copy_A with the win condition $\text{eqcopy}_A = \mathbf{P}\text{-max}_{\odot}^{\infty}(\text{copy}_A)$, which makes it into a tight and complete object $\text{objcopy}_A = \langle \text{copy}_A, \text{eqcopy}_A \rangle$. By Prop. 25.2.1, we observe that:

$$\Downarrow \text{eqcopy}_A = \Downarrow \mathbf{P}\text{-max}_{\odot}^{\infty}(\text{copy}_A) = \text{copy}_A$$

So that $\langle \text{copy}_A, \text{eqcopy}_A \rangle$ is indeed a tight complete object.

The discussion in §26 is substantially more comprehensive than just defining objcopy , so we defer it for later.

Chapter 26

Parallel Implementations

At this point, we discuss the combinatorial properties our parallel program denotations will satisfy.

26.1 Sequential Implementations

At the sequential level, we expect the denotations of the programs each agent runs to satisfy that:

Definition 26.1.1. We say an object $M = \langle \sigma, \Sigma \rangle : A \multimap B$ is:

Receptive if $s \in \sigma$ and m is an O-move or $\bullet$ -move such that $s \cdot m \in P_{A \multimap B}$ then $s \cdot m \in \sigma$;

Deterministic if $s \cdot m \in \sigma \cup \Sigma$ and m is a P-move then for any $s \cdot n \in \sigma \cup \Sigma$, $m = n$ or $n = \bullet$;

Complete $\text{P-max}_{\mathfrak{O}}^{\infty}(\sigma) \sqsubseteq_L \Sigma$.

We say M is a sequential *implementation* if it decomposes as $M = \cup_{i \in I} M_i$ where each M_i is O-receptive, P-deterministic, and complete. We call such a decomposition a *deterministic decomposition* of M . We call an implementation *deterministic* if it is itself O-receptive, P-deterministic, and complete.

We note a small convention: for a strategy, win condition, or object S we say S is nearly P , where P is some property, when it decomposes as $S = \cup_{i \in I} S_i$ for some S_i and every S_i

is P . In this way, an implementation is a nearly O -receptive, P -deterministic, and complete object.

Now, observe that completeness subsumes many usual requirements on strategies. For example, it is common to require strategies to be total (every O -move has a response), or to be winning (every “complete” play is even-length), or in the context of divergence to satisfy things like “diverge-or-reply” [Gowers and Laird, 2018; Harmer and McCusker, 1999]. Being winning often involves the requirement that the infinite plays induced by the strategy are all winning. All such properties follow from our notion of completeness. For example, if we have a play $s \cdot m$ where m is an O -move, then either it is P -maximal, in which case completeness ensures $s \cdot m \cdot \circlearrowleft \in \Sigma$, or it has a response $s \cdot m \cdot n \in \sigma$. Note too that completeness ensures that if the Proponent asks a question to the Opponent, e.g. $q \cdot q \in \sigma$, then $q \cdot q \cdot \circlearrowleft \cdot \circlearrowleft \in \Sigma$. In other words, it ensures that the object is O -receptive with respect to divergences as well.

We make a few preliminary observations.

Lemma 26.1.1. *If $M = \langle \sigma, \Sigma \rangle$ is receptive, P -deterministic, and complete then $P\text{-max}_\circ^\infty(\sigma) = \Sigma$.*

Proof. That $P\text{-max}_\circ^\infty(\sigma) \supseteq_L \Sigma$ is immediate from the assumption that it is complete. Now, let $t \in \Sigma$. Let $p \sqsubseteq t$ be the maximal prefix of t such that $p \in P\text{-max}_\circ^\infty(\sigma)$. If $p = t$ then $t \in P\text{-max}_\circ^\infty(\sigma)$ and we are done. Otherwise, $p \cdot m \sqsubseteq t$ for some move m . If m is an O -move or $\bullet$ -move, then by receptivity $p \cdot m \in \sigma$, contradicting the maximality of p as a longer prefix of t would belong to $P\text{-max}_\circ^\infty(\sigma)$. So m must be a P -move. But then, we have $p \in \sigma$ so by completeness there is some P -move n such that $p \cdot n \in \Sigma$. By P -determinism it follows that $m = n$, contradicting the maximality of p . Hence, $t \in P\text{-max}_\circ^\infty(\sigma)$. $\square$

Lemma 26.1.2. *A tight object is nearly receptive and complete if and only if it is nearly receptive, P -deterministic, and complete.*

Proof sketch. Our proof is inspired by [Girard, 1987a]. We say $s \succ t$ when two plays first disagree at a P -position, and call a pairwise-coherent subset $C \subseteq \Sigma$ a clique. Maximal cliques of Σ (which exist by Zorn’s lemma) yield deterministic components $\langle \Downarrow C, C \rangle$. Each

component is $\mathbf{O}$ -receptive (by maximality of C and receptivity of σ), $\mathbf{P}$ -deterministic (by definition of coherence), and complete (since $\mathbf{P}$ -maximal plays in $\Downarrow C$ admit extensions within C by maximality). The decomposition covers M because every $s \in \Sigma$ belongs to some maximal clique. See Proof 33.8.1 for details. $\square$

In light of the proof of Lemma 26.1.2 we will say $\Sigma :: A \multimap B$ is deterministic when it is a clique under $- \supset -$. Note that if $\langle \sigma, \Sigma \rangle$ is $\mathbf{P}$ -deterministic then Σ is deterministic.

These two lemmas together suggest that we may see an implementation M as either just a nearly receptive, complete strategy, or that we may assume the decomposition is in fact deterministic. In this latter case, we also get that $M = \cup_{i \in I} \langle \sigma_i, \mathbf{P}\text{-max}_{\circlearrowleft}(\sigma_i) \rangle$ so that the win condition is completely determined by the strategy denotation.

Now, we observe that for strategies $\sigma : A \multimap B$ such that no play of σ has a $\bullet$ -token we have the same notion of sequential and concurrent strategies as Oliveira Vale et al. [2022] and Oliveira Vale et al. [2024a], and our standard composition $-; -$ agrees with their notion of composition. So let's call their categories of sequential $\mathbf{P}$ -deterministic, and $\mathbf{O}$ -receptive strategies $\mathbf{Seq}_{\text{det}}$.

Several technical lemmas underpin the composition results that follow. First, determinism is preserved by $\mathbf{P}\text{-max}_{\circlearrowleft}^\infty(-)$ and $\Downarrow -$, and for complete objects the determinism of $\langle \sigma, \Sigma \rangle$ reduces to that of σ and Σ separately (Lemma 33.8.2). Second, when both strategies are deterministic, minimal interaction witnesses for composed plays are unique (Lemma 33.8.3), and this extends to win conditions under divergence composition (Corollary 33.8.2). These results are stated and proved in the proof compendium.

Proposition 26.1.1. *Suppose σ and τ are deterministic implementations. Then,*

$$\mathbf{P}\text{-max}_{\circlearrowleft}^\infty(\sigma; \tau) = \mathbf{P}\text{-max}_{\circlearrowleft}^\infty(\sigma) \circlearrowleft \mathbf{P}\text{-max}_{\circlearrowleft}^\infty(\tau)$$

Proof sketch. One inclusion follows from Prop. 27.1.1. For the other, given a limit play $s \in \Uparrow(\sigma; \tau)$, we take a chain of $\mathbf{P}$ -maximal prefixes and lift them to unique minimal witnesses (Lemma 33.8.3); by uniqueness these form a chain whose supremum witnesses $s \in \mathbf{P}\text{-max}_{\circlearrowleft}^\infty(\sigma) \circlearrowleft \mathbf{P}\text{-max}_{\circlearrowleft}^\infty(\tau)$. The reverse inclusion holds because witnesses of

$\text{P-max}_{\circlearrowleft}^{\infty}(\sigma) \wp \text{P-max}_{\circlearrowleft}^{\infty}(\tau)$ project to P-maximal plays. See Proof 33.8.3 for details. $\square$

Proposition 26.1.2. *Given $\sigma : A \multimap B \in \mathbf{Seq}_{\text{det}}$, the mapping:*

$$\sigma \mapsto \mathcal{O}(\sigma) := \langle \sigma^{\bullet}, \text{P-max}_{\circlearrowleft}^{\infty}(\sigma^{\bullet}) \rangle$$

is functorial, and, moreover, its image always defines a deterministic implementation.

Proof sketch. The key step is showing $\sigma^{\bullet}; \tau^{\bullet} = (\sigma; \tau)^{\bullet}$: if $\bullet$ appears in a composed play, the witness interaction must end with $\bullet$, so we can strip it and recover a witness in $\sigma; \tau$. The reverse inclusion follows by appending $\bullet$ to any witness. Functoriality of $\text{P-max}_{\circlearrowleft}^{\infty}(-)$ then follows from Prop. 26.1.1. See Proof 33.8.4 for details. $\square$

Proposition 26.1.3. *Sequential implementations compose.*

Proof sketch. By join-preservation, it suffices to show that $\langle \sigma, \Sigma \rangle ; ; \langle \tau, T \rangle$ is a deterministic implementation when both components are. We verify:

Receptivity: Follows from the receptivity of σ and τ together with the switching condition.

Completeness: Immediate from Prop. 26.1.1.

Determinism: Reduces via Prop. 33.8.2 to showing $\Sigma \wp T$ is deterministic. This follows from the uniqueness of minimal interaction witnesses (Lemma 33.8.2) together with the determinism of Σ and T .

See Proof 33.8.1 for details. $\square$

An interesting observation is that deterministic implementations are completely characterized by strategies in $\mathbf{Seq}_{\text{det}}$ (i.e. not involving any care about liveness or starvation). Indeed, by Lemma 26.1.1, every deterministic implementation M satisfies $M = \mathcal{O}(\sigma)$ for some $\sigma \in \mathbf{Seq}_{\text{det}}$ (Proposition 33.8.6 in the proof compendium).

One might be tempted to think that in fact, every sequential implementation is in bijection with a non-deterministic strategy in the sense of Ch. 12. This is unfortunately not the case. The crux of the matter is that an implementation such as:

$$\langle \downarrow q \cdot a^{\bullet}, (\epsilon \uplus q \cdot a)^{\bullet} \rangle$$

would have to correspond to $\downarrow(q \cdot a)$. But so would the implementation:

$$\langle \downarrow q \cdot a^\bullet, (\epsilon \uplus q \cdot \circlearrowleft \uplus q \cdot a)^\bullet \rangle$$

Of course, this issue is solved by considering instead the category of sets $\mathcal{S} \subseteq \mathbf{Seq}_{\text{det}}$ of deterministic strategies $\mathcal{P}(\mathbf{Seq}_{\text{det}})$ (with games as objects) with composition defined pointwise. Then, we would obtain a fully faithful (semi)functor by the operation:

$$\mathcal{S} \mapsto \bigcup_{\sigma \in \mathcal{S}} \mathcal{O}(\sigma)$$

By abuse of notation, we also use $\mathcal{O}(-)$ to name this (semi)functor, as the two functors coincide on singletons. Full faithfulness follows from the uniqueness of deterministic decompositions (Proposition 33.8.5 in the proof compendium). Combined with essential surjectivity, this yields:

Corollary 26.1.1. *$\mathcal{O}(-)$ defines an equivalence between $\mathcal{P}(\mathbf{Seq}_{\text{det}})$ and sequential implementations, which restricts to an equivalence between $\mathbf{Seq}_{\text{det}}$ and deterministic sequential implementations.*

Now, recall that the sequential copycat strategy $\text{seqcopy}_A : A \multimap A \in \mathbf{Seq}_{\text{det}}$, which acts as the neutral element for $\mathcal{O}$ -receptive strategies in $\mathbf{Seq}_{\text{det}}$, is defined by:

$$\text{seqcopy}_A := \{s \in P_{A \multimap A} \mid \forall p \sqsubseteq_{\text{even}} s. s \upharpoonright_{A, -} = s \upharpoonright_{-, A}\}$$

By functoriality of $\mathcal{O}(-)$, the object $\mathcal{O}(\text{seqcopy}_A)$ is the neutral element for sequential implementations (Proposition 33.8.7 in the proof compendium).

For the purposes of our model, it is enough to assume we have a sound interpretation function $\llbracket - \rrbracket$ from the space of sequential modules, to our notion of sequential implementation. This can be achieved, for instance, as we have done in the main text by extracting traces out of an operational semantics. We note that since we actually extract a tight interpretation we benefit from Lemma 26.1.2, so that is sufficient to show completeness. We do not show the proof that our denotation is well-defined as it is straightforward, although

long, and mostly follows from previous work (Ch. 5 and [Koenig and Shao, 2020]).

26.2 Parallel Implementations

Definition 26.2.1. A concurrent implementation consists of an object $M : A \multimap B$ such that $\pi_\alpha(M)$ is a sequential implementation for every $\alpha \in \Upsilon$.

We start by showing that parallel composition acts functorially over composition, in the sense that the composition of parallel modules is equal to composing them locally and then parallel composing them.

Proposition 26.2.1. *For sets of plays*

$$S = \|\alpha \in \Upsilon S[\alpha]\| \subseteq P_{A \multimap B}^{\leq \infty} \qquad T = \|\alpha \in \Upsilon T[\alpha]\| \subseteq P_{B \multimap C}^{\leq \infty}$$

it holds that:

$$S \circ T = \|\alpha \in \Upsilon S[\alpha] \circ T[\alpha]\|$$

Proof sketch. We reduce $S \circ T$ to $F_\circ (U_\circ S ; U_\circ T)$ and show $F_\circ$ and $U_\circ$ each commute with parallel composition. For $U_\circ$: removing $\circ$ -tokens from a parallel play is the same as removing them agent-by-agent, since $\circ$ -tokens are agent-local. For $F_\circ$: inserting $\circ$ -tokens into a parallel play likewise decomposes agent-by-agent, as the underlying plays are $\circ$ -token-free. The middle step (composition commutes with parallel) follows from Prop. 12.1.2. See Proof 33.8.8 for details. $\square$

It is also easy to check that parallel composition is join-preserving in the following sense:

Proposition 26.2.2.

$$\|\alpha \in \Upsilon \cup_{i \in I[\alpha]} S[\alpha]_i\| = \bigcup_{i \in \prod_{\alpha \in \Upsilon} I[\alpha]} \|\alpha \in \Upsilon S[\alpha]_{i(\alpha)}\|$$

An immediate corollary is that:

Corollary 26.2.1. *Given a sequential object $O = \langle \sigma, \Sigma \rangle : A \multimap B$ define the concurrent object $O^\Phi : A \multimap B$ by:*

$$\text{for } S \subseteq P_{A \multimap B}^{\leq \infty} \text{ let } S^\Phi := \parallel_{\alpha \in \Upsilon} \iota_\alpha(S) \quad O^\Phi := \langle \sigma^\Phi, \Sigma^\Phi \rangle$$

where $\iota_\alpha(S)$ labels every move of every play of S with α . Then, $-^\Phi$ is semifunctorial in the sense that for any sequential objects $O_1 : A \multimap B$ and $O_2 : B \multimap C$:

$$(O_1 ;; O_2)^\Phi = O_1^\Phi ;; O_2^\Phi$$

Proof. See Prop. 26.2.1. □

In particular, if O is a sequential implementation then O^Φ is a concurrent implementation, by definition.

Then, the concurrent copycat strategy is given by:

$$\text{objcopy}_A := \langle \text{copy}_A, \text{eqcopy}_A \rangle := O(\text{seqcopy}_A)^\Phi$$

This also shows that, by Prop. 26.1.2 and Cor. 26.2.1:

Corollary 26.2.2. $\langle \text{copy}_A, \text{eqcopy}_A \rangle$ is idempotent, i.e.

$$\text{copy}_A ; \text{copy}_A = \text{copy}_A \quad \text{eqcopy}_A \circ \text{eqcopy}_A = \text{eqcopy}_A$$

Proposition 26.2.3. For any parallel strategy $\sigma = \parallel_{\alpha \in \Upsilon} \sigma[\alpha]$,

$$\text{P-max}_{\circlearrowleft}^\infty(\sigma) = \parallel_{\alpha \in \Upsilon} \text{P-max}_{\circlearrowleft}^\infty(\sigma[\alpha])$$

Proof sketch. By Prop. 26.2.1, it suffices to show that $\text{P-max}^\infty(\sigma) = \parallel_\alpha \text{P-max}^\infty(\sigma[\alpha])$. A play in the right-hand side is P-maximal in σ because any P-extension of agent α would extend $\sigma[\alpha]$ past its P-maximum. Conversely, a P-maximal play in σ projects to P-maximal plays in each $\sigma[\alpha]$, since a P-extension of $\sigma[\alpha]$ would lift to a P-extension of σ . See Proof 33.8.9 for details. □

Corollary 26.2.3. *Any concurrent implementation is nearly complete.*

Proof sketch. Commute the parallel and non-deterministic decompositions via Prop. 26.2.2, then apply Prop. 26.2.3 to each component: $\mathbf{P}\text{-max}_{\odot}^{\infty}(\|_{\alpha} \sigma_{i(\alpha)}) = \|_{\alpha} \mathbf{P}\text{-max}_{\odot}^{\infty}(\sigma_{i(\alpha)}) \sqsupseteq_L \|_{\alpha} \Sigma_{i(\alpha)}$. See Proof 33.8.3 for details. $\square$

Proposition 26.2.4. *Concurrent implementations define a category.*

Proof sketch. Composition: by Prop. 26.2.1, $M ; ; N$ decomposes as $\|_{\alpha} (M[\alpha] ; ; N[\alpha])$, and each local composition is a sequential implementation by Prop. 26.1.3. Identity: $\text{objcopy}_A ; ; M ; ; \text{objcopy}_B = M$ follows from the corresponding sequential identity via the same decomposition. See Proof 33.8.10 for details. $\square$

We will call **UImpl** the category of concurrent implementations.

26.3 Symmetric Monoidal Structure

Now, we extend the structure we have established so far on parallel implementations with a horizontal composition operation $- \otimes -$ which allows us to compose two independent implementations together so they can be run simultaneously across different threads.

A consequence of Corollary 26.1.1 together with Prop. 26.2.1 is that continuous functors and natural transformations lift from **Seq_{det}** to **UImpl** (see Corollary 33.8.1 and Proposition 33.8.2 in the proof compendium). We define every structural morphism in **UImpl** as the image under $\mathcal{O}(-)^{\Phi}$ of the corresponding morphism in **Seq_{det}**; functoriality and naturality are preserved by this lifting. The structural morphisms are inherited from **Seq_{det}** through the interleaving functor $-^{\Phi}$ (Prop. 13.5.1).

Crucially, the symmetric monoidal structure of **Seq_{det}** lifts to **UImpl**. We define the tensor operation by the lifting of the sequential tensor $- \otimes^{\Phi} -$, and write $- \otimes^{\Phi} -$ simply as $- \otimes -$. While this presentation allows us to easily obtain that the symmetric monoidal structure is preserved, it is not convenient to work with objects $O : A$ used for specifications. Moreover, it is also more challenging to show join-preservation using this presentation. Therefore, we recall a direct definition of the $- \otimes -$ (note that the name overload will cause no confusion as we are about to show the two definitions are equivalent).

Definition 26.3.1. Given *sets* of plays $S \subseteq P_A^{\leq \infty}$ and $T \subseteq P_B^{\leq \infty}$, we define their horizontal composition $S \otimes T \subseteq P_{A \otimes B}^{\leq \infty}$ by $S \otimes T := \{s \in P_{A \otimes B}^{\leq \infty} \mid s|_{A,-} \in S \wedge s|_{-,B} \in T\}$. This may be extended to object specifications in the obvious way: $\langle \nu, \omega \rangle \otimes \langle \nu', \omega' \rangle := \langle \nu \otimes \nu', \omega \otimes \omega' \rangle$.

Lemma 26.3.1. *The operation $- \otimes^\Phi -$ (which is only defined over **UImpl**) coincides with the $- \otimes -$ defined in Def. 26.3.1.*

Proof. Let $s \in M \otimes N : (A \otimes C) \multimap (B \otimes D)$. Write $X = A \multimap B$ and $Y = C \multimap D$. By definition, $s|_{X,-} \in M$ and $s|_{-,Y} \in N$. But then, $\pi_\alpha(s)|_{X,-} \in M[\alpha]$ and $\pi_\alpha(s)|_{-,Y} \in N[\alpha]$. Hence, $s \in M \otimes^\Phi N$. The reverse direction is analogous. $\square$

Corollary 26.3.1. *$- \otimes -$ is join-preserving and defines an order-isomorphism.*

Proof. See Oliveira Vale et al. [2024a]. Note that $\circlearrowleft$ -tokens behave just like P-moves and $\bullet$ -tokens are straightforward to handle, as we have done previously. $\square$

Corollary 26.3.2. *$(\mathbf{UImpl}, - \otimes^\Phi -, \mathbf{1})$ defines an enriched symmetric monoidal category.*

Proof. The remarks above guarantee the category is indeed symmetric monoidal. By enriched, we mean enrichment over join-semilattices and join-preserving functions (note that a join-preserving function is always monotonic). We proved that $- \otimes -$ is join-preserving in Cor. 26.3.1. $\square$

Chapter 27

Composing Completeness and Tightness

27.1 Preliminary Results

Definition 27.1.1. We say σ is sufficiently O-receptive when for any P-maximal $s \in \sigma$, and for any O-move m such that $s \cdot m \in P_{A \multimap B}^\bullet$, $s \cdot m \in \sigma$.

Now, we have all the language to state some preliminary lemmas for composition of total/complete objects, which also motivate the discussion of completeness as opposed to just totality.

Proposition 27.1.1.

- If $\sigma : A \multimap B$ is sufficiently O-receptive and $\tau : B \multimap C$ is O-receptive then

$$\text{P-max}_{\circlearrowleft}(\sigma; \tau) \sqsupseteq_L \text{P-max}_{\circlearrowleft}^\infty(\sigma) \circledcirc \text{P-max}_{\circlearrowleft}^\infty(\tau)$$

- Similarly, if $\sigma : A \multimap B$ is O-receptive and $\tau : B \multimap C$ is sufficiently O-receptive then

$$\text{P-max}_{\circlearrowleft}(\sigma; \tau) \sqsupseteq_L \text{P-max}_{\circlearrowleft}^\infty(\sigma) \circledcirc \text{P-max}_{\circlearrowleft}^\infty(\tau)$$

- For any $\Sigma :: A \multimap B$ and $T :: B \multimap C$:

$$\Downarrow(\Sigma \circ T) \sqsubseteq_S \Downarrow\Sigma ; \Downarrow T$$

Proof sketch. We prove the three items:

- For $\text{P-max}_{\circ}(\sigma; \tau) \sqsubseteq_L \text{P-max}_{\circ}^{\infty}(\sigma) \circ \text{P-max}_{\circ}^{\infty}(\tau)$: given a P-maximal interaction $s' \in \sigma \circ \tau$, we construct a chain of extensions $(s'_{i,j,b})$ that alternately extend the $A \multimap B$ and $B \multimap C$ projections to be P-maximal. Sufficient O-receptivity allows transfers of O-moves across the hidden interface B . The supremum of this chain yields a witness in $\text{P-max}_{\circ}^{\infty}(\sigma) \circ \text{P-max}_{\circ}^{\infty}(\tau)$.
- The second item is analogous.
- For $\Downarrow(\Sigma \circ T) \sqsubseteq_S \Downarrow\Sigma ; \Downarrow T$: we take a prefix of a witness interaction and project to obtain elements of $\Downarrow\Sigma$ and $\Downarrow T$.

See Proof 33.9.1 for details. □

We are not ready yet to give our full account of tight and complete objects, so we leave the development at this cliffhanger for now and briefly divert to other crucial aspects of our verification calculus so we may develop some of the necessary notions.

27.2 Composing Completeness

We need to define a notion of determinism in order to be able to compose completeness. Since we are not interested in determinism outside of complete objects, we avoid defining determinism for arbitrary objects. Similarly, we define other notions for objects, such as O-receptivity and $\bullet$ -receptivity, in a way that makes the most sense for complete objects. Specifically to define determinism, we make the following observation. Every play s decomposes as:

$$s = O_1 \cdot P_2 \cdot \dots \cdot O_{2n-1} \cdot P_{2n}$$

where each O_i includes only O-moves or $\bullet$ -tokens, P_i includes only P-moves and each sequence O_i or P_i is the maximal such sequence satisfying these properties. We will call this the *alternating* decomposition of s . It is easy to check that there is a unique alternating decomposition of a play s . We will call each such maximal sequence a concurrency class of the alternating decomposition of s . We call a class X_i with i even a P-class, and with i odd an O-class.

Definition 27.2.1. We say an object $M = \langle \sigma, \Sigma \rangle : A \multimap B$ is:

Complete $\text{P-max}_\circ^\infty(\sigma) \sqsubseteq_L \Sigma$;

Sufficiently O-receptive If $s \in \sigma$ is P-maximal or there is $s' \in \Sigma$ such that $U_\circ s' = s$ and $s \cdot m$ is well-formed with m an O-move, then $s \cdot m \in \sigma$.

$\bullet$ -receptive If $s \in \sigma$ and $s \cdot \alpha \bullet$ is well-formed then $s \cdot \alpha \bullet \in \sigma$.

Deterministic For any $s, t \in U_\circ \Sigma$ with alternating decompositions:

$$s = O_1 \cdot P_2 \cdot \dots \cdot O_{2n-1} \cdot P_{2n}$$

$$t = O'_1 \cdot P'_2 \cdot \dots \cdot O'_{2n-1} \cdot P'_{2n}$$

then if $p \cdot X_i \sqsubseteq s$ and $p \cdot Y_i \sqsubseteq t$ with X_i and Y_i concurrency classes, then if $X_i \neq Y_i$ then $i = j$ and both are odd (i.e. both are opponent classes).

Proposition 27.2.1. *Every nearly complete sufficiently O-receptive $\bullet$ -receptive object is a nearly complete deterministic sufficiently O-receptive $\bullet$ -receptive object.*

Proof sketch. This generalizes Lemma 26.1.2 to the concurrent setting. We define coherence over well-scheduled plays using alternating decompositions: two plays $s \supset t$ are coherent when their P-classes agree up to one being a prefix of the other (with the shorter one having an empty successor class). Maximal cliques of Σ again yield the deterministic components. Completeness of each component $\langle \langle C \rangle, C \rangle$ follows from maximality: when a P-extension is incoherent with some $t' \in C$, the source of incoherence must be a P-class, and we extract a coherent extension from the incoherent witness. See Proof 33.9.2 for details. $\square$

Determinism is not good enough to reproduce the results along completeness available in the sequential setting. The key challenge is that plays such as:

$$\begin{array}{ccc} \alpha:q & \alpha':q & \alpha:q \quad \alpha':q \\ & \text{and} & \\ & \alpha:q & \alpha:q' \end{array}$$

are both allowed in a deterministic strategy, even though the fact that α' called q should not be detectable to α (so it can't react to it). To remedy this, we make a refinement of the notion of determinism that reduces the ability of the Proponent to react to such scheduler decisions.

Definition 27.2.2. First, we define a restriction of $- \rightsquigarrow -$, which restricts it to local rewrite rules $s \cdot m \cdot m' \cdot t \rightsquigarrow s \cdot m' \cdot m \cdot t$ such that m and m' (i.e. m is a move in A iff m' is a move in B). More precisely, let $- \rightsquigarrow' -$ be defined by the rules OO, PP, OP, $\bullet$ P and $O\bullet$ restricted to the two moves being swapped happening in different components when they are not $\bullet$ -tokens ((P) and (O) are not allowed). We write $s \approx t$ when $s \rightsquigarrow' t$ and $t \rightsquigarrow' s$, and $s \lesssim t$ if there is a t' such that $t \approx t'$ and $s \sqsubseteq t'$.

We say an object is weakly deterministic if in the definition of determinism equality is replaced by equivalence under $\rightsquigarrow'$ (that is, $s \equiv_{\rightsquigarrow'} t$ when $s \rightsquigarrow' t$ and $t \rightsquigarrow' s$).

By the same argument as Proposition 27.2.1, combined with $\rightsquigarrow'$ -closure, every nearly complete $\rightsquigarrow'$ -saturated sufficiently O-receptive $\bullet$ -receptive object admits a weakly deterministic decomposition (Lemma 33.9.1 in the proof compendium).

Lemma 27.2.1. *If $\langle \sigma, \Sigma \rangle : A \multimap B$, $\langle \tau, T \rangle : B \multimap C$ are $\rightsquigarrow'$ -saturated weakly deterministic objects then:*

$$\uparrow(\sigma ; \tau) \sqsupseteq_L \text{P-max}_{\mathfrak{O}}^{\infty}(\Sigma) \mathbin{\mathfrak{;}} \text{P-max}_{\mathfrak{O}}^{\infty}(T)$$

Proof sketch. Given $s \in \uparrow(\sigma ; \tau)$ with witnesses $s'_i \in \sigma \mathbin{\mathfrak{;}} \tau$, we rearrange each s'_i into a normal form $\langle s'_i \rangle$ that groups hidden moves in B between successive concurrency classes of $s'_i \upharpoonright_{A, -, C}$. Cross-linearization saturation ensures the rearrangement preserves membership, and weak determinism ensures the normalized sequence forms a chain. The supremum then witnesses $s \in \text{P-max}_{\mathfrak{O}}^{\infty}(\Sigma) \mathbin{\mathfrak{;}} \text{P-max}_{\mathfrak{O}}^{\infty}(T)$. See Proof 33.9.2 for the full argument. $\square$

While we do not develop this in full formality, we note that we may use a more natural decomposition than weak determinism.

Definition 27.2.3. We say an object $\langle \sigma, \tau \rangle$ is sufficiently deterministic if in the definition of determinism equality is replaced by $\equiv_{\rightsquigarrow}$ (i.e. $s \equiv_{\rightsquigarrow} t$ if and only if $s \rightsquigarrow t$ and $t \rightsquigarrow s$).

Oliveira Vale et al. [2024a] have shown that this notion of equivalence is the same as requiring that the strategies be closed under OO and PP swaps.

Corollary 27.2.1. *Every nearly complete $\rightsquigarrow'$ -saturated sufficiently O-receptive $\bullet$ -receptive object is a nearly complete $\rightsquigarrow'$ -saturated $\equiv_{\rightsquigarrow}$ -closed sufficiently deterministic sufficiently O-receptive $\bullet$ -receptive object.*

Proof. Take the $\rightsquigarrow'$ - and $\equiv_{\rightsquigarrow}$ -closure in Lemma 33.9.1. □

Corollary 27.2.2. *If $\langle \sigma, \Sigma \rangle : A \multimap B$, $\langle \tau, T \rangle : B \multimap C$ are $\rightsquigarrow'$ -saturated $\equiv_{\rightsquigarrow}$ -closed and sufficiently deterministic objects then:*

$$\uparrow(\sigma ; \tau) \sqsubseteq_L \text{P-max}_{\mathfrak{O}}^{\infty}(\Sigma) ; \text{P-max}_{\mathfrak{O}}^{\infty}(T)$$

Proof. In the proof of Lemma 27.2.1, one can assume an ordering over the set of agents and then re-order each of the concurrency classes in B so they are monotonic sequences on the agent names, which makes them equal across the elements of the sequence, allowing them to be proven to be chains. □

27.3 Putting it Together

We will call a tight nearly complete $\rightsquigarrow'$ -saturated $\equiv_{\rightsquigarrow}$ -closed sufficiently deterministic sufficiently O-receptive $\bullet$ -receptive object a *precise object*.

Lemma 27.3.1. $\bullet$ *Implementations are O-receptive and precise.*

- $\bullet$ *Any tight nearly complete sufficiently O-receptive $\bullet$ -receptive object that is $\equiv_{\rightsquigarrow}$ -closed of type $\mathbf{1} \multimap A$ is precise.*

Proof. The first claim follows by definition and Cor. 27.2.1, and the second claim follows from Cor. 27.2.1 and the fact that $\mathbf{1} \multimap A$ has no moves in $\mathbf{1}$ so any strategy over it is $\sim'$ -saturated vacuously. $\square$

Proposition 27.3.1. *O-receptive precise objects compose.*

Proof sketch. Decompose M and N into components and compose pairwise. Tightness of each pairwise composition follows from Prop. 27.1.1. Completeness follows from combining Prop. 27.1.1 with Cor. 27.2.2. Saturation properties ($\sim'$ and $\equiv_{\sim}$) follow from closure under the copycat idempotent (Prop. 12.2.1). Determinism follows by reapplying Lemma 27.2.1. See Proof 33.9.3 for details. $\square$

Corollary 27.3.1. *If $O = \langle \nu, \omega \rangle$ is a precise object and M an implementation then $O ; M$ is a precise object.*

Proof. The argument follows along the lines of Prop. 27.3.1 except that in the application of Prop. 27.1.1, we make use of the fact that O is sufficiently O-receptive while M is O-receptive. Moreover, their composition is O-receptive because $\nu; \sigma; \text{copy}_B = \nu; \sigma$ together with Prop. 12.3.1. $\square$

This lands us on the key reason to define the notion of a precise object.

Proposition 27.3.2. *If $\langle \nu', \omega' \rangle$ is nearly complete and $\langle \nu, \omega \rangle$ is nearly tight then $\langle \nu', \omega' \rangle \approx \langle \nu, \omega \rangle \iff \omega' \rightsquigarrow \omega$.*

Proof. The forward direction is trivial.

We prove the backward direction. Let $s \in \nu'$ then there is complete component $\langle \nu'_i, \omega'_i \rangle$ such that $s \in \nu'_i \subseteq \nu'$. By completeness, there is a P-extension $U_{\circlearrowleft} s' \sqsupseteq s$ of s with $s' \in \omega'_i$. Since $\omega' \rightsquigarrow \omega$ it follows that there is $t' \in \omega$ such that $s' \rightsquigarrow t'$. Hence $s \in \Downarrow t' \sqsupseteq_S \Downarrow K = t' \sqsupseteq_S K \Downarrow t'$ (see Prop. 27.1.1) and therefore there is $t \in \Downarrow t'$ such that $s \rightsquigarrow t$. But $\Downarrow t' \sqsupseteq_S \Downarrow \omega \sqsupseteq_S \nu$ by tightness and the claim follows. $\square$

Corollary 27.3.2. *For precise objects, $\langle \nu', \omega' \rangle$ and $\langle \nu, \omega \rangle$, $\langle \nu', \omega' \rangle \approx \langle \nu, \omega \rangle \iff \omega' \rightsquigarrow \omega$.*

27.4 Relaxing Tightness

Our final goal is to show that we can relax the tightness requirement. Right now completeness and tightness are based around refinement. For instance, tightness means that $\Downarrow \Sigma \sqsubseteq_S \sigma$. Part of the structural core of the compositional linearizability technique is to turn linearizability into the notion of refinement of a compositional model of computation. Therefore, we would expect to be able to relax tightness to be able to instead require only that $\Downarrow \Sigma \rightsquigarrow \sigma$. This turns out to be convenient from a practical standpoint. For example, a lock-free specification might include infinite traces where finitely many agents diverge (such as α_1 and α_2 in the trace below):

$$\alpha_1:q \cdot \alpha_1:\odot \cdot \alpha_2:q \cdot \alpha_1:\odot \cdot \alpha_3:q \cdot \alpha_3:\text{ok} \cdot \dots$$

These same objects should admit atomic specifications, where no more than 1 call is ever pending. Tightness, however, requires that we include the trace:

$$\alpha_1:q \cdot \alpha_2:q \cdot \alpha_3:q \cdot \alpha_3:\text{ok}$$

Instead, we would like to be able to not include $\alpha_1:q \cdot \alpha_2:q$ in the safety specification. The relaxed version of tightness allows for this, as linearizability allows for the removal of pending calls.

We will therefore show we can use this relaxed version of tightness for our goals. It is also possible to relax completeness to say that $\text{P-max}_{\odot}^{\infty}(\sigma) \sqsubseteq_L \Sigma$. This turns out to be less useful, however, than relaxing tightness, as $\rightsquigarrow$ is more restrictive than $\rightsquigarrow$ already, and $\sqsubseteq_L$ is liberal on the handling of divergences. The proofs that “relaxed completeness” compose (for precise objects) are also more involved, so we will not develop this in detail, as we will for tightness.

Definition 27.4.1. We say an object $\langle \sigma, \Sigma \rangle$ is relaxedly tight when $\Downarrow \Sigma \rightsquigarrow \sigma$ and is relaxedly complete when $\text{P-max}_{\odot}^{\infty}(\sigma) \rightsquigarrow \Sigma$.

Our first observation is that the core motivation for characterizing tightness and com-

pleteness still holds.

Proposition 27.4.1. *If $\langle \nu', \omega' \rangle$ is nearly relaxedly complete and $\langle \nu, \omega \rangle$ is nearly relaxedly tight then $\langle \nu', \omega' \rangle \approx \langle \nu, \omega \rangle \iff \omega' \approx \omega$.*

Proof. The forward direction is trivial.

We prove the backward direction. Let $s \in \nu'$ then there is complete component $\langle \nu'_i, \omega'_i \rangle$ such that $s \in \nu'_i \subseteq \nu'$. By relaxed completeness, there is a P-extension $U_{\circlearrowleft} s' \sqsupseteq s$ of s with $s' \approx \omega'_i$. Since $\omega' \approx \omega$ it follows that there is $t' \in \omega$ such that $s' \approx t'$. Hence $s \in \Downarrow s' \sqsupseteq_S \Downarrow K = t' \sqsupseteq_S K \Downarrow t'$ (see Prop. 27.1.1) and therefore there is $t \in \Downarrow t'$ such that $s \rightsquigarrow t$. But $t \in \Downarrow t' \sqsupseteq_S \Downarrow \omega \sqsupseteq_S K \nu$ by relaxed tightness. Hence, $s \rightsquigarrow t \rightsquigarrow \nu$. $\square$

The remaining concern is to show that relaxedly tight objects compose.

Proposition 27.4.2. *If $\langle \sigma, \Sigma \rangle : A \multimap B$ and $\langle \tau, T \rangle : B \multimap C$ are relaxedly tight and either σ or τ is saturated then $\langle \sigma, \Sigma \rangle ; ; \langle \tau, T \rangle$ is relaxedly tight.*

Proof sketch. By Prop. 27.1.1, $\Downarrow(\Sigma \circ T) \sqsupseteq_S \Downarrow \Sigma ; \Downarrow T \sqsupseteq_S K \sigma ; K \tau$. The copycat idempotents collapse via $\text{copy}_B ; \text{copy}_B = \text{copy}_B$, giving $K(\sigma ; \tau)$, which is the relaxed tightness condition. See Proof 33.9.4 for the full calculation. $\square$

Below, by relaxedly precise we mean the same as a precise object but with tightness replaced for relaxed tightness. Of course, by the remarks above, we should be able to also replace relaxed completeness. Since we opt to not develop relaxed completeness in detail, we merely state the version with relaxed tightness.

Corollary 27.4.1. *If $O = \langle \nu, \omega \rangle$ is a relaxedly precise object and M an implementation then $O ; ; M$ is a relaxedly precise object.*

Chapter 28

Live Linearizability Rely-Guarantee Logic

In this chapter, we extend the program logic of Ch. 17 to reason about live linearizability instead. Note that considering Prop. 25.1.1 this means that we may reason about safety and liveness in the same program logic. First, we briefly discuss the aspects of the operational semantics for $M : E \rightarrow F$ we had ignored there. Then, we discuss how we may obtain a win condition $\llbracket M \rrbracket_L :: \dagger E \multimap \dagger F$. Then, we discuss the extension of the program logic to handle liveness.

28.1 Operational Semantics

We reconsider now the operational semantics of Ch. 17. Recall that we had highlighted several premises in [blue](#) and deferred their explanation until now. We modify the rules by keeping track of a *schedule counter* $\iota : \Upsilon \rightarrow \mathbb{N}$, which assigns to each $\alpha \in \Upsilon$ an upper bound on how many steps other agents may take before α is given a step by the scheduler. The scheduler counter is managed by the rules $- \xrightarrow{+}_{\alpha} -$ and $- \xrightarrow{-}_{\alpha} -$, defined in Fig. 28.1. If an agent is active (i.e., it is executing some code), whenever it takes a step it makes sure to decrease the scheduler counter for all other agents. If an agent α reaches a budget of 0, then ι can no longer be decreased by other threads, so that the only rule that applies is to take an α step and give it a new budget using $- \xrightarrow{+}_{\alpha} -$. An agent may still be starved

$$\begin{array}{c}
\text{POSITIVE} \frac{\begin{array}{l} \iota'[\alpha] = n > 0 \\ \forall \alpha' \in \text{dom}(\iota). \alpha \neq \alpha' \Rightarrow \iota'[\alpha'] < \iota[\alpha'] \\ \forall \alpha' \in \Upsilon \setminus \text{dom}(\iota). \alpha \neq \alpha' \Rightarrow \alpha' \notin \text{dom}(\iota) \end{array}}{\iota \xrightarrow{+}_{\alpha} \iota'} \\
\\
\text{NEGATIVE} \frac{\begin{array}{l} \alpha \notin \text{dom}(\iota') \\ \forall \alpha' \in \text{dom}(\iota). \alpha \neq \alpha' \Rightarrow \iota'[\alpha'] < \iota[\alpha'] \\ \forall \alpha' \in \Upsilon \setminus \text{dom}(\iota). \alpha \neq \alpha' \Rightarrow \alpha' \notin \text{dom}(\iota) \end{array}}{\iota \xrightarrow{-}_{\alpha} \iota'}
\end{array}$$

Figure 28.1: Rules for managing scheduler counters

by the scheduler nonetheless by taking the STARVE rule, which allows any active agent to be starved by the scheduler and adds the starvation token to the trace. $-\xrightarrow{-}_{\alpha}-$ is the rule used to deactivate an α , when it was starved or because it has finished running an operation (RET). These modifications make no change to the finite traces generated by the operational semantics, while guaranteeing that its infinite traces are well-scheduled.

We then define the win condition $\llbracket M \rrbracket_L :: \dagger E \multimap \dagger F$ by:

$$\begin{aligned}
\llbracket M \rrbracket_L &:= F_{\cup} (\{s \mid \exists c \in \text{Cont}, \Delta \in (\Upsilon \rightarrow \text{Env}), \iota \in (\Upsilon \rightarrow \mathbb{N}). \\
&\quad \langle c_0, \Delta_0, \iota_0, \epsilon \rangle \xrightarrow{s}_M \langle c, \Delta, \iota, s \rangle \wedge \text{ws}(c)\} \cup \llbracket M \rrbracket_{\infty}) \\
&\text{where } \text{ws}(c) \iff \forall \alpha \in \Upsilon. c(\alpha) \in \{\text{idle}, \bullet\} \cup \text{Com} \times \text{Prim} \\
\llbracket M \rrbracket_{\infty} &:= \{s \mid \forall n. n \models \langle c_0, \Delta_0, \iota_0, \epsilon \rangle \xrightarrow{-n}_M s\} \\
&\text{where } n \models \langle c, \Delta, \iota, s \rangle \xrightarrow{s_n}_M s' \iff \exists c'', \Delta'', \iota'', s''. \langle c, \Delta, \iota, s \rangle \xrightarrow{M}^n \langle c'', \Delta'', \iota'', s'' \rangle \wedge s'' \sqsubseteq s'
\end{aligned}$$

This ensures that the interpretation below is a precise object representing the module M :

$$\llbracket M \rrbracket : \dagger E \multimap \dagger F := \langle \llbracket M \rrbracket_S, \llbracket M \rrbracket_L \rangle$$

It is worth, at this point, noting that we assume specifications $\mathcal{V}_E := ((S_E, M_{\dagger E}^{\odot \bullet}, -\xrightarrow{-}_E -), s_0)$ are state transition systems with labels in $M_{\dagger E}^{\odot \bullet}$. We define the liveness interpretation

$\llbracket \mathcal{V}_E \rrbracket_L :: \dagger E$ by:

$$\begin{aligned}\llbracket \mathcal{V}_E \rrbracket_L &:= (\llbracket \mathcal{V}_E \rrbracket_S \cap P_{\dagger E}^{\text{ws}}) \cup \llbracket \mathcal{V}_E \rrbracket_\infty \\ \llbracket \mathcal{V}_E \rrbracket_\infty &:= \{s \in (M_{\dagger E}^{\odot \bullet})^\omega \cap P_{\dagger E}^{\text{ws}} \mid \forall n \in \mathbb{N}. \exists (\text{st}_i)_{i \leq n} \in S_E^{n+1}. \text{st}_0 = s_0 \wedge \forall i < n. \text{st}_i \xrightarrow{s[i]}_E \text{st}_{i+1}\}\end{aligned}$$

This allows for interpreting a specification as an object:

$$\llbracket \mathcal{V}_E \rrbracket : \dagger E := \langle \llbracket \mathcal{V}_E \rrbracket_S, \llbracket \mathcal{V}_E \rrbracket_L \rangle$$

28.2 Program Logic

Like in the safety case, we have a module $M : E \rightarrow F$ and are given a library specification $\mathcal{V}_E$ as well as a module specification $\mathcal{V}_F$, all well-defined in that M satisfies the side-conditions discussed in Ch. 9, and $\llbracket \mathcal{V}_E \rrbracket : \dagger E$ and $\llbracket \mathcal{V}_F \rrbracket : \dagger F$. We also assume given a client policy $\xi \subseteq P_{\dagger E \multimap \dagger F}^{\text{ws}}$. The objective of the program logic is to establish the judgement

$$\xi \vdash M : \langle \nu_E, \omega_E \rangle \Rightarrow \langle \nu_F, \omega_F \rangle$$

where

$$\llbracket \mathcal{V}_E \rrbracket = \langle \nu_E, \omega_E \rangle \qquad \llbracket \mathcal{V}_F \rrbracket = \langle \nu_F, \omega_F \rangle$$

which entails that M uses the underlay $\langle \nu_E, \omega_E \rangle$ to implement an object, which lively linearizes to $\langle \nu_F, \omega_F \rangle$ under restrictions on both client and scheduler given by ξ . Formally:

$$\llbracket \mathcal{V}_E \rrbracket ; (\xi \cap \llbracket M \rrbracket) \approx \llbracket \mathcal{V}_F \rrbracket$$

To achieve this, a few modifications are required to the program logic described in Ch. 17. First, the linearization states that are part of the possibilities now include an extra case:

$$\text{linstate} := \text{idle} \mid \text{inv}(f) \mid \text{lin}(f) \mid \text{lin}(f, r) \mid \text{div}(f)$$

The new case $\text{div}(f)$ indicates that the invocation f is currently marked as a diverged invocation. As a result, we have two extra possibility update rules (in addition to those in Fig. 16.1): P-DIVERGE diverges a thread that has not linearized the invocation by appending

$$\begin{array}{c} \text{P-DIVERGE} \frac{\pi[\alpha] = \text{inv}(q) \quad \text{st} \xrightarrow{\alpha:q}_{\mathcal{V}} \text{st}'}{(\text{st}, s, \pi) \dashrightarrow (\text{st}', s \cdot \alpha:q, \pi[\alpha \mapsto \text{div}(q)])} \\ \\ \text{P-CONVERGE} \frac{\pi[\alpha] = \text{div}(q)}{(\text{st}, s, \pi) \dashrightarrow (\text{st}, s, \pi[\alpha \mapsto \text{lin}(q)])} \end{array}$$

the invocation to the linearized trace p and updates the linearization state. This update can only be performed subject to invariant (I2) – the specification ω_F permits diverging at position $p \cdot \alpha:f$. If ω_F makes maximal progress, no divergence is allowed, so the prover cannot perform P-DIVERGE and must construct a decreasing metric in the ITER rule to ensure termination.

However, $\pi[\alpha] = \text{div}(f)$ does not mean the thread definitely diverges. It is just one possibility that the prover speculates based on the partial trace along a potentially infinite execution. With more steps into the future, the prover may find that the operation turns out to be terminating. In this case, they can perform the P-CONVERGE rule to now mark it as a committed, potentially converging call.

In addition, the set of possibilities we maintain now, in addition to invariant (I1), also must maintain the following three additional invariants:

$$(I2) \quad \forall(p, \pi) \in \rho. \exists w \in \omega_F. p \sqsubseteq U_{\circlearrowleft} w \wedge \forall \alpha. \pi[\alpha] = \text{div}(f) \Rightarrow \pi_{\alpha}(w) = \pi_{\alpha}(p) \cdot \alpha:\circlearrowleft$$

$$(I3) \quad \forall \alpha. \text{last}(\pi_{\alpha}(s)) = \circlearrowleft \Rightarrow \forall(p, \pi) \in \rho. \pi[\alpha] = \text{div}(-)$$

$$(I4) \quad \text{P-max}(s) \Rightarrow \forall(p, \pi) \in \rho. p \in \omega_F$$

These encode the following:

- (I2) Any linearized trace should be a prefix of some winning trace with divergence token forgotten. Moreover, the diverging threads claimed in π must have the divergence as their next move in w .

- (I3) If a thread diverges in the concrete trace (due to underlay divergence), all linearized traces at that moment should diverge for the same thread.
- (I4) If the concrete trace is at some P-maximal position, all linearized traces at that moment should be inside the winning strategy. The only P-maximal position the concrete trace can reach in the program logic is where all threads end with an overlay P move, a starvation token, or a divergence token.

Now, a rule that is usually simply a derived rule becomes critical when reasoning about liveness: the ITER rule below. We use the post-condition Q to record all transitions made by the loop body so that we can pose loop variant conditions over it. The first condition establishes the Hoare triple for the loop body, while the second one ensures the precondition P is a loop invariant that holds after applying the effect of the loop body $Q \circ P$.

In the third and fourth condition, the prover needs to provide a metric $(\mathcal{H}, <_{\mathcal{H}}, \perp_{\mathcal{H}})$ and a metric evaluation function $H \in \Upsilon \rightarrow \text{ModState} \rightarrow \mathbb{P}_{E \rightarrow F} \rightarrow \mathcal{H}$ that extracts the metric value for the current thread α from the scheduler counter ι , and the current execution trace s . The third condition requires the metric to be non-decreasing under the influence of the environment described by $\mathcal{R}$, while the fourth condition requires the metric to be strictly decreasing for each loop iteration.

To ensure that either the current loop terminates or the divergence of the current operation eventually be admitted, we require a metric $(\mathcal{H}, <_{\mathcal{H}}, \perp_{\mathcal{H}})$ and a metric evaluation function H that computes the metric from the module state and a future scheduling trace. The prover needs to show either all possible linearized traces at the beginning of the iteration diverge for the current thread, or the metric decreases for the iteration. All environment steps in $\mathcal{R}$ should also not increase the metric H . To ensure the tightness of the linearization, another metric $(\mathcal{D}, <_{\mathcal{D}}, \perp_{\mathcal{D}})$ is required to show the possibility is eventually increased as long as the loop is running.

$$\begin{array}{c}
\mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} \ C \ \{Q\} \\
Q \circ P \subseteq P \quad \forall ((\Delta, \iota, s, \rho), (\Delta', \iota', s', \rho')) \in \mathcal{R}[\alpha]. (\iota', s') \leq_{\alpha, H} (\iota, s) \\
\frac{\forall ((\Delta, \iota, s, (p, \pi)), (\Delta', \iota', s', \rho')) \in Q. (\iota', s') <_{\alpha, H} (\iota, s) \vee (\pi[\alpha] = \text{div}(-) \wedge \rho < \rho')}{\mathcal{R}, \mathcal{G} \vdash_{\alpha} \{P\} \ C^* \ \{Q\}} \text{ITER}
\end{array}$$

$$\text{where } (\iota', s') <_{\alpha, H} (\iota, s) \iff H(\alpha, \iota', s') <_{\mathcal{H}} H(\alpha, \iota, s)$$

$$\text{and } (p, \pi) < (p', \pi') \iff p \sqsubseteq p' \wedge p \neq p'$$

However, it might be impossible to find this decreasing metric for all objects. For example, the **EBStack** is only minimally progressing, and there is no way to ensure the loop terminates. Therefore, instead of showing the metric is decreasing, we allow the prover to diverge the thread in the possibility $(\pi[\alpha] = \text{div}(-))$ and show the possibility is increasing $(\rho < \rho')$, i.e., some threads have linearized their overlay operations, during the loop iteration. This not only conforms to the informal understanding of deadlock-freedom – the entire threadpool is making progress while some threads may diverge, but it is also essential for showing the soundness. In the soundness proof, we construct an infinite chain of linearized traces to approach the linearization of a diverging execution. The condition $\rho < \rho'$ ensures any trace along this chain will eventually be reached.

The soundness of the program logic is then stated as follows:

Lemma 28.2.1 (Soundness- ν_F). *If $\xi \vdash M : \langle \nu_E, \omega_E \rangle \Rightarrow \langle \nu_F, \omega_F \rangle$ is true, then $\nu'_F \rightsquigarrow \nu_F$.*

Proof. Cf. Proposition 34.1.1. □

Lemma 28.2.2 (Soundness- ω_F). *If $\xi \vdash M : \langle \nu_E, \omega_E \rangle \Rightarrow \langle \nu_F, \omega_F \rangle$ is true, then $\omega'_F \rightsquigarrow \omega_F$.*

Proof. Cf. Lemma 34.2.1. □

Proposition 28.2.1 (Soundness). *If $\xi \vdash M : \langle \nu_E, \omega_E \rangle \Rightarrow \langle \nu_F, \omega_F \rangle$ is provable, then*

$$\llbracket \mathcal{V}_E \rrbracket ; (\xi \cap \llbracket M \rrbracket) \approx \llbracket \mathcal{V}_F \rrbracket$$

Proof. By Lemma 28.2.1 and Lemma 28.2.2. □

Part V

Conclusion

Chapter 29

Conclusion

This dissertation has established new foundations for the correctness and abstraction of concurrent objects rooted in compositionality. In particular, it provides a generic approach to derive a linearizability-like criterion on a reasonable compositional model of concurrency, together with a compositional theory and verification calculus for it. We have shown the construction is sensible by using it to derive a criterion in the context of a traditional model for the safety of concurrent components, a model with full-system crashes, and a model for liveness. We have also shown that it helps explain much of the pre-existing theory of linearizability as well as previously existing linearizability criteria, and developed verification techniques to verify individual components and compose certified components.

While we have mechanized some of these techniques, there is a long road ahead to both improve the reasoning principles available and to engineer a practical implementation of it. To a large extent, this is because we are still learning what the underlying logic of linearizability proofs is. This also precludes the development of effective automated verification techniques. Much work along those lines will be necessary to fully enable the practical verification of large-scale systems.

At the same time, this dissertation mostly covered standard computation models and properties. That is, our models are locally sequential and, other than the model with full-system crashes, we mostly focused on traditional safety and liveness properties. Modern systems are increasingly weakly consistent, distributed, and incorporate probabilistic, cryptographic, and security guarantees. On one hand, while in the past a linearizability-like

criterion had to be handcrafted for every model, the techniques exposed in this dissertation show how to derive such a criterion, so that there is no guesswork anymore. On the other hand, building a compositional model for liveness was itself an open problem, and developing compositional techniques for safety and liveness in such new settings remains open as well.

There is also an opportunity to unify domain theory and coalgebra theory to arrive at a generic treatment of systems. Given the techniques of this dissertation, such a unification could then also accommodate linearizability-like criteria. I expect that, given such a model, compositional linearizability does give an appropriate correctness criterion featuring abstraction along both space and time.

Finally, this dissertation provides another instance where topological ideas are useful in the theory of specifications. While domain theory has taken great interest in this matter, the distributed systems theory community has since moved from point-set topology (or perhaps locale theory) to combinatorial topology. An interesting agenda would be to reformulate domain theory and locale theory as a general theory of specifications; distributed systems theory should then provide fertile ground for applications. In particular, temporal logics have seen a strong connection with such topological techniques, but it is hard to accommodate linearizability's ability to reorder events in such a rigid calculus. An interesting open direction is to analyze the topological content of linearizability as a refinement notion, which through Stone duality could lead to new logics for concurrent and distributed computation. I hope this dissertation provides at least a faint light towards this ambitious agenda.

Part VI

Proof Compendium

Chapter 30

Sequential Games

30.1 Safety and Liveness

Proposition 30.1.1 (cf. Proposition 4.1.3). *The topology $\mathcal{T}$ used by Alpern-Schneider (the Cantor topology) on the set of traces $\text{Traces} := \Sigma^\omega$ is equivalent to the topology $\prod_{i \in \omega} \top$.*

Proof. By Prop. 4.1.2 sets $B = \prod_{i \in \omega} U_i$ where only finitely many $i \in \omega$ satisfy $U_i \neq \Sigma$ define a basis for $\prod_{i \in \omega} \top$. Let k be the $k \in \omega$ such that $\forall j > k. U_j = \Sigma$. Then, let $P = \prod_{i \leq k} U_i$ and note that $B = \cup_{p \in P} p \times (\prod_{i > k} \Sigma)$. Now, by definition every $p \times (\prod_{i > k} \Sigma)$ is also an open set. Hence, sets of the form $B_p = p \times (\prod_{i > |p|} \Sigma)$ where $p \in \Sigma^*$ define a basis for $\prod_{i \in \omega} \top$. But such sets are exactly the sets Alpern-Schneider define as the basic open sets of their topology, so the claim follows. $\square$

Proposition 30.1.2 (cf. Proposition 4.1.4). *Safety properties are exactly the closed sets of $\prod_{i \in \omega} \top$ and liveness properties are exactly the dense sets of $\prod_{i \in \omega} \top$.*

Proof. Let $\overline{X}$ denote the closure of X , i.e. its union with its limit points. Recall that a set is closed if and only if $\overline{X} = X$. So first, assume P is a safety property and let l be a limit point of P . Then, for every open set U s.t. $l \in U$, $\exists s \neq l. s \in P \cap U$. Now, if $l \notin P$, by assumption, there is a $p \sqsubseteq l$ such that $\forall t. p \cdot t \notin P$. Note then that the set $U = \{p \cdot t \mid t \in \Sigma^\omega\}$ is an open set containing l (as $p \sqsubseteq l$), hence $\exists s \neq l \in P \cap U$. But then, $s = p \cdot t'$ for some t' and $p \cdot t' \in P$, contradicting that P is a safety property. Hence, $l \in P$. Conversely, assume P is a closed set and that $s \notin P$. In particular, s is not a limit point of P , as otherwise

$s \in \overline{P} = P$. Therefore, there is a neighborhood U of s such that $U \cap P \subseteq \{s\}$. Now, let $U = \cup_{p \in X} B_p$ be the basis decomposition of U . Then $\cup_{p \in X} B_p \cap P \subseteq \{s\}$ so that for every $p \in X$, $B_p \cap P \subseteq \{s\}$. By assumption, $s \notin P$, so fix $p \in X$ (note that $X \neq \emptyset$ as otherwise, $s \notin U$). Then, by definition of B_p , for every $t \in \Sigma^\omega$, $p \cdot t \notin P$, and hence P is a safety property for any choice of $p \in X$.

Now, recall that a dense set X is a set for which $\overline{X} = \Sigma^\omega$. So let P be a liveness property and let $s \in \Sigma^\omega$. We argue that $s \in \overline{P}$. So let B_p be a basic open set such that $s \in B_p$. By definition of liveness, there exists $t \in \Sigma^\omega$, such that $p \cdot t \in P$. But then, $p \cdot t \in B_p \cap P$. Now, if $s = p \cdot t$, then it is immediate that $s \in P$. Otherwise, $B_p \cap (P \setminus \{s\})$ is non-empty and the result also follows. Now, let P be a dense set, and let $p \in \Sigma^*$. Since P is dense, $B_p \cap P$ is non-empty, so there is $s \in B_p \cap P$. But then, $s = p \cdot t$ and $p \cdot t \models P$. Hence, P is a liveness property. $\square$

Proposition 30.1.3 (cf. Proposition 4.2.3). *A property S is a safety property in $\mathcal{S}(\text{Traces})$ if and only if:*

$$s \not\models S \implies \exists p \in \Sigma^*. p \sqsubseteq s \wedge \forall t. p \cdot t \not\models S$$

A property L is a liveness property in $\mathcal{S}(\text{Traces})$ if and only if:

$$\forall s \in \Sigma^*. \exists t \in \Sigma^* \cup \Sigma^\omega. s \cdot t \models L$$

Proof. Recall that a set is closed if and only if $\overline{X} = X$. So first, assume S is a safety property and let l be a limit point of S . Then, for every open set U s.t. $l \in U$, $\exists s \neq l. s \in S \cap U$. Now, if $l \notin S$, by assumption, there is a $p \sqsubseteq l$ such that $\forall t. p \cdot t \notin S$. Note then that the set $U = \uparrow p$ is a (basic) open set containing l (as $p \sqsubseteq l$), hence $\exists s \neq l \in S \cap U$. But then, $s = p \cdot t'$ for some t' and $p \cdot t' \in S$, contradicting that S is a safety property. Hence, $l \in S$. Conversely, assume S is a closed set and that $s \notin S$. In particular, s is not a limit point of S , as otherwise $s \in \overline{S} = S$. Therefore, there is a neighborhood U of s such that $U \cap S \subseteq \{s\}$. Now, let $U = \cup_{p \in X} \uparrow p$ be the basis decomposition of U . Then $\cup_{p \in X} \uparrow p \cap S \subseteq \{s\}$ so that for every $p \in X$, $\uparrow p \cap S \subseteq \{s\}$. By assumption, $s \notin S$, so fix $p \in X$ (note that $X \neq \emptyset$ as otherwise, $s \notin U$). Then, by definition of $\uparrow p$, for every $t \in \text{Traces}$, $p \cdot t \in \uparrow p$. So

$\{p \cdot t\} \cap S \subseteq \uparrow p \cap S \subseteq \{s\}$. If $p \cdot t \cap S = \emptyset$, then $p \cdot t \notin S$, otherwise $p \cdot t = s \notin S$ by assumption. Hence, S is a safety property.

Now, recall that a dense set X is a set for which $\overline{X} = \text{Traces}$. So let L be a liveness property and let $s \in \text{Traces}$. We argue that $s \in \overline{L}$. So let $\uparrow p$ be a basic open set such that $s \in \uparrow p$. By definition of liveness, there exists $t \in \Sigma^* \cup \Sigma^\omega$, such that $p \cdot t \in L$. But then, $p \cdot t \in \uparrow p \cap L$. Now, if $s = p \cdot t$, then it is immediate that $s \in L$. Otherwise, $\uparrow p \cap (L \setminus \{s\})$ is non-empty and the result also follows. Now, let L be a dense set, and let $p \in \Sigma^*$. Since L is dense, $\uparrow p \cap L$ is non-empty, so there is $s \in \uparrow p \cap L$. But then, $s = p \cdot t$ and $p \cdot t \models L$. Hence, L is a liveness property. $\square$

30.2 The Replay Modality

Proposition 30.2.1 (cf. Proposition 8.2.2). $- \triangleright -$ endows $\mathbf{Safe}^{\text{seq}}$ with a non-symmetric monoidal structure.

Proof. The game **1** is the identity for $\triangleright$, as witnessed by the equivalences

$$\iota_A^*(s) \cdot \epsilon = \iota_A^*(s) \cong s \cong \iota_A^*(s) = \epsilon \cdot \iota_A^*(s)$$

obtaining the isomorphism

$$A \triangleright \mathbf{1} \cong A \cong \mathbf{1} \triangleright A$$

Similarly, for associativity we may use

$$\iota_A^*(s_A) \cdot (\iota_{B \triangleright C}^*(\iota_B^*(s_B)) \cdot \iota_{B \triangleright C}^*(\iota_C^*(s_C))) \cong (\iota_{A \triangleright B}^*(\iota_A^*(s_A)) \cdot \iota_{A \triangleright B}^*(\iota_B^*(s_B))) \cdot \iota_C^*(s_C)$$

as the associator showing

$$A \triangleright (B \triangleright C) \cong (A \triangleright B) \triangleright C$$

Checking $\sigma \triangleright \tau : A \triangleright B \multimap C \triangleright D$ is a strategy is straightforward. The key challenge in proving functoriality is establishing that σ and σ' interacting before τ and τ' interact is the same as having σ followed by τ interact with σ' followed by τ' . This is a straightforward but long argument, which I omit here. $\square$

Proposition 30.2.2 (cf. Proposition 8.2.3). $\dagger A$ in **Game** is the limit of the diagram

$$1 \longleftarrow A \longleftarrow A \triangleright A \longleftarrow A \triangleright A \triangleright A \longleftarrow \dots \longleftarrow A^{\triangleright n} \longleftarrow \dots$$

where each morphism $A^{\triangleright n} \longleftarrow A^{\triangleright n+1}$ is the partial copycat strategy such that Opponent never opens the $n+1$ -th component in $A^{\triangleright n+1}$. The legs $\lambda_n : \dagger A \multimap A^{\triangleright n}$ are given by the strategy

$$\lambda_n := \{s \in P_{\dagger A \multimap A^{\triangleright n}} \mid \forall t \sqsubseteq_{\text{even}} s. t \upharpoonright_{\dagger A} = t \upharpoonright_{A^{\triangleright n}}\}$$

defined up-to some re-indexing.

Proof. We show first that the following diagram commutes

$$\begin{array}{ccc} \dagger A & & \\ \lambda_n \downarrow & \searrow \lambda_{n+1} & \\ A^{\triangleright n} & \longleftarrow & A^{\triangleright n+1} \end{array}$$

So let $s \in \lambda_n$. Then, notice that $\epsilon \in P_A$ and therefore

$$s = s \cdot \epsilon \in A^{\triangleright n+1}$$

But then, the partial copycat strategy defining the morphism $A^{\triangleright n} \longleftarrow A^{\triangleright n+1}$ embeds s in $A^{\triangleright n}$ as s itself. This shows the diagram commutes. From this we obtain that $(\dagger A, \lambda_-)$ forms a cone of the diagram. We must now show it is indeed a limit of the diagram. For that, let (C, μ) be a cone of the diagram and define the morphism $f_C : C \multimap \dagger A$ as

$$f_C := \{s \in P_{C \multimap \dagger A} \mid \exists n. \exists s' \in \mu_n. \forall t \sqsubseteq_{\text{even}} s. t \upharpoonright_{\dagger A} \sqsubseteq_{\text{even}} s' \upharpoonright_{A^{\triangleright n}} \wedge t \upharpoonright_C \sqsubseteq_{\text{even}} s' \upharpoonright_C\}$$

We must show that for any n the diagram below commutes with f_C being the unique map making it commute

$$\begin{array}{ccc} C & & \\ f_C \downarrow & \searrow \mu_n & \\ \dagger A & \xrightarrow{\lambda_n} & A^{\triangleright n} \end{array}$$

Now, the fact that the diagram commutes is almost immediate. To see that, call $e_{n,m}$ the composition of the arrows

$$A^{\triangleright n} \longleftarrow \dots \longleftarrow A^{\triangleright m}$$

Note that $e_{n,m}$ simply embeds the plays in $A^{\triangleright m}$ that play the empty play for any component of $A^{\triangleright m}$ after the n -th one. Now, notice that

$$s|_{C, A^{\triangleright n}} \in f_C \Leftrightarrow \exists m. \exists s' \in \mu_m. s|_{A^{\triangleright n}} = s|_{\dagger A} = s'|_{A^{\triangleright m}} \wedge s|_C = s'|_C \Leftrightarrow s \in e_{n,m} \circ \mu_m = \mu_n$$

where $e_{n,m} \circ \mu_m = \mu_n$ follows from (C, μ) being a cone.

Now, for uniqueness, suppose there was another map g_C . Then, for all n

$$\lambda_n \circ g_C = \mu_n = \lambda_n \circ f_C$$

but notice that

$$s \in g_C \Rightarrow s|_{\dagger A} \in \dagger A \Rightarrow \exists n. \exists s'. s|_{\dagger A} = s'|_{A^{\triangleright n}} \Rightarrow s' \in \lambda_n \circ g_C = \mu_n \Rightarrow s' \in \mu_n \Rightarrow s \in f_C$$

On the other hand,

$$s \in f_C \Rightarrow \exists n. \exists s' \in \mu_n. s|_{\dagger A} = s'|_{A^{\triangleright n}} \wedge s|_C = s'|_C \Rightarrow s' \in \lambda_n \circ g_C$$

But then, that means that there is t such that

$$t|_{C, A^{\triangleright n}} = s' \wedge t|_{\dagger A, A^{\triangleright n}} \in \lambda_n \wedge t|_{C, \dagger A} \in g_C$$

But

$$t|_{C, \dagger A}|_C = s'|_C = s|_C \wedge t|_{C, \dagger A}|_{\dagger A} = s'|_{A^{\triangleright n}} = s|_{\dagger A}$$

and furthermore, n and s' witness that $t|_{C, \dagger A} \in f_C$. But then, from the equations just derived it follows from the determinism of f_C that $t|_{C, \dagger A} = s$ and in particular $s \in f_C$. Therefore, indeed f_C is unique. $\square$

Proposition 30.2.3 (cf. Proposition 8.2.4). $\dagger$ – defines a comonad in **Safe**^{seq}.

Proof. For each n define the map $\mu_n^A : \dagger A \multimap (\dagger A)^{\triangleright n}$ as follows

$$\mu_n^A := \{s \in P_{\dagger A \multimap (\dagger A)^{\triangleright n}} \mid \exists s_1, \dots, s_n. \forall t \sqsubseteq_{\text{even}} s. t|_{\dagger A} \sqsubseteq_{\text{even}} s_1 \dots s_n \wedge t|_{(\dagger A)^{\triangleright n}} \sqsubseteq_{\text{even}} \iota_1^*(\langle s_1 \rangle) \dots \iota_n^*(\langle s_n \rangle)\}$$

where $\langle - \rangle$ does the following rebasing operation

$$\iota_i^*(s_1) \cdot \dots \cdot \iota_j^*(s_n) \mapsto \iota_1^*(s_1) \cdot \dots \cdot \iota_n^*(s_n)$$

It is straightforward to check that $(\dagger A, \mu^A)$ forms a cone over $D_{\dagger A}$, the sequential diagram defining $\dagger\dagger A$. We then define $\epsilon_A : \dagger A \multimap A$ as λ_1^A and $\delta_A : \dagger A \multimap \dagger\dagger A$ as the map induced by the universal property of $\dagger\dagger A$ in relation to the cone $(\dagger A, \mu^A)$. We leave the rest of the argument, which mostly consists of careful diagram chasing, to the reader. $\square$

Chapter 31

Safety

31.1 Strategy Composition

Proposition 31.1.1 (cf. Proposition 12.1.1). *Strategy composition is well-defined and associative.*

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

$$\epsilon \in A \mathbin{\circ} B \mathbin{\circ} C$$

we have that

$$\epsilon \upharpoonright_{A,B} = \epsilon \quad \epsilon \upharpoonright_{B,C} = \epsilon$$

And therefore

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

from which it follows that $\sigma; \tau$ is non-empty.

Now, suppose $s \in \sigma; \tau$ and that $p \sqsubseteq s$. Then, there exists $s' \in \sigma; \tau$ such that $s' \upharpoonright_{A,C} = s$. In particular $p \sqsubseteq s' \upharpoonright_{A,C}$. Hence, there is prefix $p' \sqsubseteq s'$ such that $p' \upharpoonright_{A,C} = p$. Now, consider p' . Since $s' \upharpoonright_{A,B} \in \sigma$ and σ is prefix-closed it follows that because $p' \upharpoonright_{A,B} \sqsubseteq s' \upharpoonright_{A,B} \in \sigma$ we have that $p' \upharpoonright_{A,B} \in \sigma$. Similarly, $p' \upharpoonright_{B,C} \in \tau$. Hence, it follows that $p' \in \sigma; \tau$ and therefore $p \in \sigma; \tau$. Note that the argument above applies uniformly to plays containing $\bullet$ -tokens: since a $\bullet$ -token is a single event in the interaction that

projects to all components, if $p \sqsubseteq s$ then the prefix $p' \sqsubseteq s'$ consistently includes or excludes $\bullet$ -tokens in all component projections simultaneously.

Associative Indeed, suppose $\sigma : A \multimap B$, $\tau : B \multimap C$, and $\nu : C \multimap D$. Let $s \upharpoonright_{A,D} \in (\sigma; \tau); \nu$ where $s \in (\sigma; \tau) \circ \nu$. Then, there is $t \in \sigma \circ \tau$ such that $s \upharpoonright_{A,C} = t \upharpoonright_{A,C} \in \sigma; \tau$.

Then, because $s \upharpoonright_{A,C} = t \upharpoonright_{A,C}$ we can define v a sequence of moves such that $v \upharpoonright_{A,C,D} = s$ and $v \upharpoonright_{A,B,C} = t$ (this construction is well-defined: each regular move belongs to exactly one component and is thus uniquely determined by either s or t , while $\bullet$ -tokens, which project to all components, are consistently positioned in s and t via their shared projection to A, C and therefore map to a single $\bullet$ -token in v). Finally, since $v \upharpoonright_{B,C} = t \upharpoonright_{B,C}$ it follows that $v \upharpoonright_{B,C} \in \tau$. Similarly, $v \upharpoonright_{C,D} = s \upharpoonright_{C,D}$ implies $v \upharpoonright_{C,D} \in \nu$. Hence, $v \upharpoonright_{B,C,D} \in \tau \circ \nu$ and $v \upharpoonright_{B,D} \in \tau; \nu$. Now, $v \upharpoonright_{A,B} = t \upharpoonright_{A,B} \in \sigma$, it follows then that $v \upharpoonright_{A,B,D} \in \sigma \circ (\tau; \nu)$ and hence

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

The other inclusion is symmetric.

□

31.2 Order Enrichment

Lemma 31.2.1. *Let $A = (M_A, P_A)$, $B = (M_B, P_B)$ and $C = (M_C, P_C)$ be concurrent games and $\sigma : A \multimap B$ and $\tau : B \multimap C$ be concurrent strategies. Then, for every $\alpha \in \Upsilon$:*

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

Proof. Suppose $s \upharpoonright_{A,C} \in \sigma; \tau$ where $s \in \sigma \circ \tau$. Then:

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

and similarly:

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

and hence:

$$\pi_\alpha(s) \in \pi_\alpha(\sigma) \circ \pi_\alpha(\tau)$$

so that

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

□

Proposition 31.2.1.

$$-\Phi : \underline{\mathbf{Safe}}^{\text{seq}} \rightarrow \underline{\mathbf{Safe}}$$

defines a semifunctor.

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

It remains to show that $(\sigma; \tau)^\Phi = \sigma^\Phi; \tau^\Phi$. By definition and Lemma 31.2.1

$$\forall \alpha \in \Upsilon. \pi_\alpha(\sigma^\Phi; \tau^\Phi) \subseteq \pi_\alpha(\sigma^\Phi); \pi_\alpha(\tau^\Phi) = \sigma; \tau$$

and hence

$$\sigma^\Phi; \tau^\Phi \subseteq (\sigma; \tau)^\Phi$$

Now suppose $s \in (\sigma; \tau)^\Phi$. This means that:

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

In particular, for all $\alpha \in \Upsilon$ there is a play $s_\alpha \in \sigma \circ \tau$ such that

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

while

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

It is straight-forward to show that one can construct a sequence $s' \in \sigma^\Phi ; \tau^\Phi$ by interleaving the s_α such that

$$s' \upharpoonright_{A,C} = s$$

□

Proposition 31.2.2 (cf. Proposition 12.2.1). *copy_A is idempotent for every A.*

Proof. By Prop. 31.2.1

$$\text{copy}_A ; \text{copy}_A = \text{copy}_A^\Phi ; \text{copy}_A^\Phi = (\text{copy}_A ; \text{copy}_A)^\Phi = \text{copy}_A^\Phi = \text{copy}_A$$

□

31.3 Order-Enriched Semicategory

Proposition 31.3.1 (cf. Proposition 12.1.2). *For strategies*

$$\sigma : A \multimap B \qquad \tau : B \multimap C$$

the following all hold:

1. *If*

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

then

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

2. *Given a family*

$$\sigma_i : A \multimap B$$

it holds that

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

3. Given a family

$$\tau_i : B \multimap C$$

it holds that

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

Proof. 1. Suppose $s \upharpoonright_{A,C} \in \sigma; \tau$. Then:

$$s \upharpoonright_{A,B} \in \sigma \Rightarrow s \upharpoonright_{A,B} \in \sigma'$$

$$s \upharpoonright_{B,C} \in \tau \Rightarrow s \upharpoonright_{B,C} \in \tau'$$

hence

$$s \in \sigma' \circ \tau' \Rightarrow s \upharpoonright_{A,C} \in \sigma'; \tau'$$

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

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

For the other direction, suppose

$$s \upharpoonright_{A,C} \in \left(\bigcup_{i \in I} \sigma_i \right); \tau$$

then

$$s \upharpoonright_{A,B} \in \bigcup_{i \in I} \sigma \quad s \upharpoonright_{B,C} \in \tau$$

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

$$s \upharpoonright_{A,B} \in \sigma_j$$

and therefore

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

so that

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

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

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

then

$$s \upharpoonright_{A,B} \in \sigma \quad s \upharpoonright_{B,C} \in \bigcup_{i \in I} \tau_i$$

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

$$s \upharpoonright_{B,C} \in \tau_j$$

and therefore

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

so that

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

□

31.4 Symmetric Monoidal Structure

In §13.5 we briefly discussed a notion of tensor on **Safe**. We noted there that this notion of tensor lifts to a symmetric monoidal closed structure in **Safe**, what we develop in detail here. Moreover, we gave most of it, but omitted the proof of Prop. 13.5.2.

Proposition 31.4.1 (cf. Proposition 13.5.2).

$$\text{copy}_{A \otimes B} = \text{copy}_A \otimes \text{copy}_B$$

Proof. Observe first that

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

Now, assuming that s is sequentially consistent, observe that

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

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

$$\pi_\alpha(s|_A) \in \text{copy}_A \quad \pi_\alpha(s|_B) \in \text{copy}_B$$

which is the case if and only if

$$s|_A \in \text{copy}_A \quad s|_B \in \text{copy}_B$$

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

$$s \in \text{copy}_A \otimes \text{copy}_B$$

Now, if $s \in \text{copy}_{A \otimes B}$ then s is sequentially consistent, and if $s \in \text{copy}_A \otimes \text{copy}_B$ the same holds. Hence, the assumption is justified. $\square$

Proposition 31.4.2.

$$- \otimes - : \underline{\text{Safe}} \otimes \underline{\text{Safe}} \rightarrow \underline{\text{Safe}}$$

is a bi-semifunctor.

Proof. Let

$$\sigma : A_1 \multimap A_2 \qquad \sigma' : A_2 \multimap A_3$$

$$\tau : B_1 \multimap B_2 \qquad \tau' : B_2 \multimap B_3$$

Suppose first that

$$s \in ((\sigma; \sigma') \parallel (\tau; \tau')) \cap P_{(A_1 \multimap A_3) \otimes (B_1 \multimap B_3)}$$

then

$$s_A = s \upharpoonright_{A_1 \multimap A_3} \in \sigma; \sigma' \qquad s_B = s \upharpoonright_{B_1 \multimap B_3} \in \tau; \tau'$$

Hence, there are $t_A \in \sigma \circ \sigma'$ and $t_B \in \tau \circ \tau'$ such that

$$t_A \upharpoonright_{A_1, A_3} = s_A \qquad t_B \upharpoonright_{B_1, B_3} = s_B$$

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

$$t|_{A_1 \otimes B_1, A_3 \otimes B_3} = s$$

and moreover

$$t|_{A_1, A_2, A_3} = t_A \quad t|_{B_1, B_2, B_3} = t_B$$

so that

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

Now, suppose

$$s = t|_{A_1 \otimes B_1, A_3 \otimes B_3} \in (\sigma \parallel \tau); (\sigma' \parallel \tau')$$

Then,

$$t|_{A_1 \otimes B_1, A_2 \otimes B_2} \in \sigma \parallel \tau \quad t|_{A_2 \otimes B_2, A_3 \otimes B_3} \in \sigma' \parallel \tau'$$

hence,

$$t|_{A_1, A_2} \in \sigma \quad t|_{A_2, A_3} \in \sigma'$$

and

$$t|_{B_1, B_2} \in \tau \quad t|_{B_2, B_3} \in \tau'$$

Hence,

$$t|_{A_1, A_2, A_3} \in \sigma; \sigma' \quad t|_{B_1, B_2, B_3} \in \tau; \tau'$$

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 31.4.3 (cf. Proposition 13.5.1). *(Safe, $\otimes$, $\mathbf{1}$) is symmetric monoidal.*

Proof. We've already proven bi-semifunctoriality in Prop. 31.4.2.

To obtain bifunctionality, note that

$$\text{copy}_A \parallel \text{copy}_A = \text{copy}_A^\Phi \parallel \text{copy}_A^\Phi = \text{copy}_A^\Phi = \text{copy}_A$$

We now move to the monoidal structure.

Lemma 31.4.1. *(Safe, $K_{\text{copy}} \circ - \otimes -, \mathbf{1}$) is a symmetric monoidal closed category.*

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

$$\begin{array}{ccccc} A \otimes (B \otimes C) & \xrightarrow{\alpha_{A,B,C}} & (A \otimes B) \otimes C & \mathbf{1} \otimes A & \xrightarrow{\lambda_A} & A & A \otimes \mathbf{1} & \xrightarrow{\rho_A} & A \\ \sigma_A \otimes (\sigma_B \otimes \sigma_C) \downarrow & \cong & \downarrow (\sigma_A \otimes \sigma_B) \otimes \sigma_C & \mathbf{1} \otimes \sigma \downarrow & \cong & \downarrow \sigma & \sigma \otimes \mathbf{1} \downarrow & \cong & \downarrow \sigma \\ A' \otimes (B' \otimes C') & \xrightarrow{\alpha_{A',B',C'}} & (A' \otimes B') \otimes C' & \mathbf{1} \otimes B & \xrightarrow{\lambda_B} & B & B \otimes \mathbf{1} & \xrightarrow{\rho_B} & B \end{array}$$

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

$$\lambda_A = \lambda_A^\Phi = \text{copy}_A^\Phi = \text{copy}_A$$

$$\rho_A = \rho_A^\Phi = \text{copy}_A^\Phi = \text{copy}_A$$

Meanwhile,

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

Therefore, we easily check that:

$$(\mathbf{1} \otimes \sigma); \lambda_B = \sigma; \text{copy}_B = \sigma = \text{copy}_A; \sigma = \lambda_A; \sigma$$

$$(\sigma \otimes \mathbf{1}); \rho_B = \sigma; \text{copy}_B = \sigma = \text{copy}_A; \sigma = \rho_A; \sigma$$

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 $-\Phi$, the fact that the structural morphisms are defined by lifting the sequential ones through $-\Phi$. Moreover,

$$\sigma^\Phi \otimes \tau^\Phi = (\sigma \otimes \tau)^\Phi$$

as is easily checked.

The same argument shows that the braiding morphism is a natural transformation, that it is invertible and the functoriality of $-\Phi$ implies that the hexagonal diagram commutes. $\square$

Finally, we establish that **Safe** is closed.

Lemma 31.4.2. *The symmetric monoidal category $(\mathbf{Safe}, \otimes, \mathbf{1})$ is closed.*

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

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

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. $\square$

$\square$

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

$$\mathbf{Atom} : \mathbf{Safe}^{\text{seq}} \longrightarrow \mathbf{Atomic}$$

that plays the same role as $-\Phi$ plays in the proof of 31.4.3.

31.5 The K_{copy} Semifunctor

Proposition 31.5.1 (cf. Proposition 12.2.2). *For any $\sigma : A \multimap B$ and $\tau : B \multimap C$:*

$$K_{\text{copy}}(\sigma; \tau) \subseteq K_{\text{copy}}(\sigma); K_{\text{copy}}(\tau)$$

Proof. The argument is quite simple, we just verify the following sequence of equalities and inclusions taking note of the use of Lemma 32.3.1, Proposition 31.3.1, Proposition 12.2.1 and associativity of interaction:

$$\begin{aligned} K_{\text{copy}}(\sigma; \tau) &= \text{copy}_A; \sigma; \tau; \text{copy}_C \\ &\subseteq \text{copy}_A; \sigma; \text{copy}_B; \tau; \text{copy}_C \\ &= \text{copy}_A; \sigma; \text{copy}_B; \text{copy}_B; \tau; \text{copy}_C \\ &= K_{\text{copy}}(\sigma); K_{\text{copy}}(\tau) \end{aligned}$$

□

Proposition 31.5.2. *K_{copy} is monotonic and join-preserving.*

Proof. Suppose $\sigma \subseteq \sigma'$. Then:

$$K_{\text{copy}} \sigma = \text{copy}_A; \sigma; \text{copy}_B \subseteq \text{copy}_A; \sigma'; \text{copy}_B = K_{\text{copy}} \sigma'$$

by Proposition 31.3.1.

Similarly, if we have a collection

$$\{\sigma_i\}_{i \in I}$$

we have

$$K_{\text{copy}} \left(\bigcup_{i \in I} \sigma_i \right) = \text{copy}_A; \left(\bigcup_{i \in I} \sigma_i \right); \text{copy}_B = \bigcup_{i \in I} \text{copy}_A; \sigma_i; \text{copy}_B = \bigcup_{i \in I} K_{\text{copy}} \sigma_i$$

by Proposition 31.3.1. □

Corollary 31.5.1.

$$K_{\text{copy}} : \underline{\text{Safe}} \rightarrow \text{Semi Safe}$$

defines an oplax semifunctor.

Lemma 31.5.1. *If*

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

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

$$e_A \Rightarrow e'_A$$

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

$$L : \tilde{\mathbf{C}}_e \rightarrow \tilde{\mathbf{C}}_{e'} := K' \circ \text{Emb} \qquad R : \tilde{\mathbf{C}}_{e'} \rightarrow \tilde{\mathbf{C}}_e := K \circ \text{Emb}'$$

define an oplax functor and a lax functor respectively.

31.6 Fine-Grained Synchronization

Lemma 31.6.1 (Synchronization Lemma; cf. Proposition 12.3.2). *Let $s = p \cdot \alpha : m \cdot \alpha' : m' \cdot p' \in P_{A \multimap B}^{\leq \infty}$. Then, s' is a witness that $p \cdot \alpha' : m' \cdot \alpha : m \cdot p' \in \text{copy}_A ; s ; \text{copy}_B$ with $s' \upharpoonright_{-,A,B,-} = s$ if and only if $\alpha \neq \alpha'$ and $\alpha' : m' \cdot \alpha : m \rightsquigarrow_A \alpha : m \cdot \alpha' : m'$.*

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 $\text{copy}_A; \sigma; \text{copy}_B$ as $p \cdot m' \cdot m \cdot p'$ is not locally alternating and hence is not in $P_{A \multimap B}$. 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 $\text{copy}_A : A_0 \multimap A_1$ and $\text{copy}_B : B_0 \multimap B_1$. If m is a move in A then it appears in a play of $\text{copy}_A; \sigma; \text{copy}_B$ as a result of the projection to A_0 . It has a corresponding copy in 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 $\lambda_A(m) = O$ then its copy appeared *earlier* in A_1 , while if it was a P move then its copy will appear *later* in A_1 . Meanwhile, if m is a move in B then it appears as a result of the projection to B_1 . Hence, if $\lambda_B(m) = O$ its copy will appear *later* in B_0 , while if it is a P move then its copy has already appeared *earlier* in B_0 .

$$m, m' \in M_B$$

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = O$ Note that in this case we have that

$$\lambda_{B_0 \multimap B_1}(m) = \lambda_{B_0 \multimap B_1}(m') = P$$

so that by the reasoning above their copy appeared earlier in B_0 as O moves, and copy_B allows for both orderings. In particular,

$$p \cdot m' \cdot m \cdot p' \in \text{copy}_A; \sigma; \text{copy}_B$$

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = P$ The reasoning here is analogous to the previous case, except that in this case the corresponding moves appear later in copy_B but both orderings are still allowed.

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = P$ In this case $\lambda_{B_0 \multimap B_1}(m) = P$ and $\lambda_{B_0 \multimap B_1}(m') = O$. Hence, m is the copy of an earlier move in copy_B and m' is copied later in copy_B . But m already occurs before m' so that so will their copies in B_1 . Hence,

the only order possible is m before m' , giving the only negative case. But notice that it does not hold that $m' \cdot m \rightsquigarrow_{A \multimap B} m \cdot m'$ in this case either.

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = O$ In this case, $\lambda_{B_0 \multimap B_1}(m) = O$ and $\lambda_{B_0 \multimap B_1}(m') = P$. So that the copy of m in B_1 appears later while the copy of m' appears earlier. In particular, there is a play of copy_B where the copy of m' appears earlier then the copy of m and therefore

$$p \cdot m' \cdot m \cdot p' \in \text{copy}_A; \sigma; \text{copy}_B$$

$$m, m' \in M_A$$

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = O$ Similarly to before, the polarities are dualized once we consider the move within the game $A_0 \multimap A_1$ so that:

$$\lambda_{A_0 \multimap A_1}(m) = \lambda_{A_0 \multimap A_1}(m') = P$$

and their respective copies in A_1 therefore appear earlier in the copy_A play. Other than that, copy_A does not prescribe any particular ordering between them, so both are allowed.

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = P$ As before the polarities switch so that:

$$\lambda_{A_0 \multimap A_1}(m) = \lambda_{A_0 \multimap A_1}(m') = O$$

and hence their copies in A_0 appear later with no particular order required.

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = P$ In this case:

$$\lambda_{A_0 \multimap A_1}(m) = P \quad \text{and} \quad \lambda_{A_0 \multimap A_1}(m') = O$$

in this case the copy of m in A_0 appears earlier in copy_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 $m' \cdot m \rightsquigarrow_{A \multimap B} m \cdot m'$.

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = O$ In this case we have:

$$\lambda_{A_0 \multimap A_1}(m) = O \quad \text{and} \quad \lambda_{A_0 \multimap A_1}(m') = P$$

So that the copy of m in A_0 appears later than m in copy_A while the copy of m' appears earlier. In particular, both orders are allowed.

$m \in M_B$ **and** $m' \in M_A$

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = O$ In this case

$$\lambda_{B_0 \multimap B_1}(m) = P \quad \text{and} \quad \lambda_{A_0 \multimap A_1}(m') = P$$

but as m occurs in B_0 while m' occurs in A_1 the copy of m in B_1 so that both copies appear earlier in the respective plays of copy_B and copy_A so that both orderings are possible.

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = P$ The situation in this case is analogous to the previous case except that the copies of m and m' appear later.

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = P$ In this case we are in a situation where

$$\lambda_{B_0 \multimap B_1}(m) = P \quad \text{and} \quad \lambda_{A_0 \multimap A_1}(m') = O$$

so that m 's copy appears earlier while m' 's copy appears later. Hence, the ordering must still be m preceded by m' in $\text{copy}_A; \sigma; \text{copy}_B$ so that

$$p \cdot m' \cdot m \cdot p' \notin \text{copy}_A; \sigma; \text{copy}_B$$

but this is the only case where it does not hold $m' \cdot m \rightsquigarrow_{A \multimap B} m \cdot m'$.

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = O$ In this case,

$$\lambda_{B_0 \multimap B_1}(m) = O \quad \text{and} \quad \lambda_{A_0 \multimap A_1}(m') = P$$

that means that the copy of m in B_1 appears later in copy_B as well as the copy

of m' in A_0 , and therefore no order is imposed on them.

$m \in M_A$ **and** $m' \in M_B$

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = O$ We have the polarities:

$$\lambda_{A_0 \multimap A_1}(m) = P \quad \text{and} \quad \lambda_{B_0 \multimap B_1}(m') = P$$

so that the copy of m in A_0 happens earlier as does the copy of m' in B_1 . No order is required between them and therefore both orderings are possible.

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = P$ This case works as before, except that the corresponding copies into A_0 and B_1 happen later, but still no particular order is required.

$\lambda_{A \multimap B}(m) = O$ **and** $\lambda_{A \multimap B}(m') = P$ We have the polarities:

$$\lambda_{A_0 \multimap A_1}(m) = P \quad \text{and} \quad \lambda_{B_0 \multimap B_1}(m') = O$$

which means that the copy of m in A_0 happens earlier while the copy of m' in B_1 happens later. Hence, the only possible order allowed is for m to precede m' .

But this is the only negative case, where $m' \cdot m \rightsquigarrow_{A \multimap B} m \cdot m'$ does not hold.

$\lambda_{A \multimap B}(m) = P$ **and** $\lambda_{A \multimap B}(m') = O$ In this case:

$$\lambda_{A_0 \multimap A_1}(m) = O \quad \text{and} \quad \lambda_{B_0 \multimap B_1}(m') = P$$

so that m 's copy in A_0 happens later while m' 's copy in B_1 happens earlier. No order is required between them.

Under $-;$, $-$, $\circ$ -tokens just behave like either O- or P-moves. Therefore, it is already handled by the cases above, or the ones below. We now consider the rules involving $\bullet$ -moves. Crucially, if a $\bullet$ -move appears at all it appears in all components. Here are the added cases:

$m \in M_B^O$, $m' = \bullet$: In this case, $\lambda_{B_0 \multimap B_1}(m) = P$ so the copy of m in B_1 can only appear earlier in copy_B , and therefore $\alpha:m$ must appear before $\alpha':\bullet$, giving a negative case.

But notice that $\alpha':\bullet \cdot \alpha:m \rightsquigarrow_{A \multimap B} \alpha:m \cdot \alpha':\bullet$ does not hold either.

$m \in M_B^P, m' = \bullet$: In this case, $\lambda_{B_0 \multimap B_1}(m) = O$ so the copy of $\alpha:m$ in B_1 can only appear later in copy_B . In particular, there is a play of copy_B where the copy of $\alpha:m$ appears after $\alpha':\bullet$ in B_1 , and the claim follows as $\alpha':\bullet \cdot \alpha:m \rightsquigarrow_{A \multimap B} \alpha:m \cdot \alpha':\bullet$ does hold.

$m = \bullet, m' \in M_B^O$: In this case, $\lambda_{B_0 \multimap B_1}(m) = P$ so the copy of m in B_1 can only appear earlier in copy_B , and therefore $\alpha:m$ must appear before $\alpha':\bullet$. In particular, there is a play of copy_B where the copy of $\alpha':m'$ appears before $\alpha:\bullet$ in B_1 , and the claim follows as $\alpha':m' \cdot \alpha:\bullet \rightsquigarrow_{A \multimap B} \alpha:\bullet \cdot \alpha':m'$ does hold.

$m = \bullet, m' \in M_B^P$: In this case, $\lambda_{B_0 \multimap B_1}(m) = O$ so the copy of m in B_1 can only appear later in copy_B and therefore $\alpha':m'$ must appear after $\alpha:\bullet$, giving a negative case. But notice that it does not hold that $\alpha':m' \cdot \alpha:\bullet \rightsquigarrow_{A \multimap B} \alpha:\bullet \cdot \alpha':m'$.

$m \in M_A^O, m' = \bullet$: In this case, $\lambda_{A_0 \multimap A_1}(m) = O$ so the copy of m in B_0 can only appear later in copy_A . In particular, there is a play of copy_A where the copy of $\alpha:m$ appears after $\alpha':\bullet$ in A_0 , and the claim follows as $\alpha':\bullet \cdot \alpha:m \rightsquigarrow_{A \multimap B} \alpha:m \cdot \alpha':\bullet$ does hold as $\lambda_{A \multimap B}(m) = P$.

$m \in M_A^P, m' = \bullet$: In this case, $\lambda_{A_0 \multimap A_1}(m) = P$ so the copy of $\alpha:m$ in A_0 can only appear earlier in copy_A , and therefore $\alpha:m$ must appear before $\alpha':\bullet$, giving a negative case. But notice that $\alpha':\bullet \cdot \alpha:m \rightsquigarrow_{A \multimap B} \alpha:m \cdot \alpha':\bullet$ does not hold either, as $\lambda_{A \multimap B}(m) = O$.

$m = \bullet, m' \in M_A^O$: In this case, $\lambda_{A_0 \multimap A_1}(m) = O$ so the copy of m in A_0 can only appear later in copy_A and therefore $\alpha:m'$ must appear after $\alpha:\bullet$, giving a negative case. But notice that $\alpha':m' \cdot \alpha:\bullet \rightsquigarrow_{A \multimap B} \alpha:\bullet \cdot \alpha':m'$ does not hold either, as $\lambda_{A \multimap B}(m') = P$.

$m = \bullet, m' \in M_A^P$: In this case, $\lambda_{A_0 \multimap A_1}(m) = P$ so the copy of $\alpha':m'$ in A_0 must appear before $\alpha':m'$. In particular, there is a play of copy_B where the copy of $\alpha':m'$ appears before $\alpha:\bullet$ in A_0 , and the claim follows as $\alpha':m' \cdot \alpha:\bullet \rightsquigarrow_{A \multimap B} \alpha:\bullet \cdot \alpha':m'$ does hold as $\lambda_{A \multimap B}(m') = O$.

$m = \bullet, m' = \bullet$: In this case, they must appear in the same position as there will be only a single copy of m and m' in the resulting interaction sequence, giving a negative case.

But note that no rewrite rule allows to swap two $\bullet$ -moves.

□

Lemma 31.6.2. *For $s' \in P_{A \multimap B}^\bullet$, $s' \in K \tau$, if and only if there is $s \in \tau$ s.t. $s' \rightsquigarrow s$.*

Proof. Note that $K \tau = \text{copy}_A; \tau; \text{copy}_B$. It is straight-forward to show that $\tau \subseteq K \tau$ (Prop. 14.4.1). So for the backward direction, note that we may induct on the length of the derivation $s \rightsquigarrow t$. The reflexive case follows from our initial claim that $\tau \subseteq K \tau$. All cases where the $- \rightsquigarrow -$ rule reduces to a $- \rightsquigarrow -$ rule follow from the Synchronization lemma. This covers all the rules except (P) and (O). So, similarly to the setup of the Synchronization lemma proof, let s'' witness that $s' \in K s$ for some $s \in \tau$.

(P) Note that we must have $\lambda_{A \multimap B}(m) = \text{P}$ and:

$$p \cdot \epsilon \cdot p' \rightsquigarrow_A p \cdot \alpha : m \cdot p'$$

Either $\lambda_A(m) = \text{O}$ or $\lambda_B(m) = \text{P}$. In either case, we may pick the interaction sequence s'' of copy_A and copy_B witnessing $s \in K s$ and then observe that $s'' = s_1 \cdot m \cdot s_2$ where m is in A_0 or B_1 . But then, $s_1 \cdot s_2$ is also a valid interaction sequence, and therefore $s' \in K s$.

(O) Note that we must have $\lambda_{A \multimap B}(m) = \text{O}$ and:

$$p \cdot \alpha : m \cdot p' \rightsquigarrow_A p \cdot \epsilon \cdot p'$$

Either $\lambda_A(m) = \text{P}$ or $\lambda_B(m) = \text{O}$. In either case, we may pick the interaction sequence s'' of copy_A and copy_B witnessing $s \in K s$ and then observe that we may find a decomposition $s'' = s_1 \cdot s_2$ such that the copies of the events of p in A_0 and B_1 are all in s_1 and the copies of the events in p' are all in s' . Then, $s'' = s_1 \cdot m \cdot s_2$ is a valid interaction sequence (as can be checked by considering well-formedness).

For the reverse direction, we note that the only rules we have not proved that are an if and only if are (P) and (O) (in combination with the Synchronization Lemma). So, assume

$s' \in K s$ with witness s'' . let s_O be $s_O = s' \setminus s$ and $s_P = s \setminus s'$. We argue s_O contains only O moves and s_P contains only P moves. Indeed, assume $m \in s_O$. Then, $s'' = s_1 \cdot m \cdot s_2$ and m is a move in A_0 or B_1 . But then, if it was a P-move, then either its copy in B_0 or its copy in A_1 must have already occurred by the definition of **copy**, and if it is a $\bullet$ -move, then it occurs in all components, and both cases are a contradiction with the definition of s_O . Therefore, m is an O-move. The argument is dual for s_P . Now, by following the same kind of argument found in §12.3 and the cases of the swap rule, we can show $s' \setminus s_O \rightsquigarrow_{A \rightarrow B} s \setminus s_P$. Finally, a finite chain of applications of (P) and (O) derives from $s' \setminus s_O \rightsquigarrow_{A \rightarrow B} s \setminus s_P$, that $s' \rightsquigarrow s$. $\square$

31.7 Computational Interpretation

Lemma 31.7.1. *Let $s \in P_A$. Then, there exists t an alternating play and s_O a sequence of Opponent moves such that*

$$s \rightsquigarrow_A t \cdot s_O$$

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

$$p \rightsquigarrow_A p' \cdot p_O$$

Hence,

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

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, it 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.

$$s = p \cdot m \rightsquigarrow_A p' \cdot p_O \cdot m = p' \cdot p_1 \cdot m' \cdot p_2 \cdot m \rightsquigarrow_A p' \cdot m' \cdot p_1 \cdot p_2 \cdot m \rightsquigarrow_A p' \cdot m' \cdot m \cdot p_1 \cdot p_2 = t \cdot s_O$$

□

Lemma 31.7.2. *Let $s \in P_{A \multimap B}$. Then, for any $s'_A \in P_A$ and $s'_B \in P_B$ such that*

$$s'_B \rightsquigarrow_B s \restriction_B \quad s \restriction_A \rightsquigarrow_A s'_A$$

then there exists an

$$s' \in P_{A \multimap B}$$

such that

$$s' \rightsquigarrow_{A \multimap B} s \quad s' \restriction_A = s'_A \quad s' \restriction_B = s'_B$$

Proof. We let $s_A = s \restriction_A$ and $s_B = s \restriction_B$.

Suppose first that $s_A \rightsquigarrow_A s'_A$. We construct by induction on the length of the derivation $s_A \rightsquigarrow_A s'_A$ an s' such that $s' \restriction_A = s'_A$ and $s' \restriction_B = s_B$. If the length of the derivation is 0 then $s_A = s'_A$ and the result is immediate by taking $s' = s$. Now, Suppose

$$s_A \rightsquigarrow_A s_1 \cdot m \cdot n \cdot s_2 \rightsquigarrow_A s_1 \cdot n \cdot m \cdot s_2 = s'_A$$

By induction we have $s' \rightsquigarrow_{A \multimap B} s$ such that $s' \restriction_A = s_1 \cdot m \cdot n \cdot s_2$ and $s' \restriction_B = s_B$. Then, we have that

$$s' = t_1 \cdot m \cdot t_2 \cdot n \cdot t_3$$

where t_2 only has moves in B and $t_1 \restriction_A = s_1$ and $t_3 \restriction_A = s_2$. We split into cases depending on the polarity of m, n in A . Note that since we can swap m and n in A it follows that

either $\lambda_A(m) = \lambda_A(n)$ or $\lambda_A(m) = O$ and $\lambda_A(n) = P$.

m is O and n is P Then, in $A \multimap B$ 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 A . Hence, there is no move by the same agent as m in t_2 . Therefore:

$$s'' = t_1 \cdot t_2 \cdot n \cdot m \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot t_2 \cdot m \cdot n \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot m \cdot t_2 \cdot n \cdot t_3 = s'$$

m is O and n is O Then, in $A \multimap B$ 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

$$s'' = t_1 \cdot t_2 \cdot n \cdot m \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot t_2 \cdot m \cdot n \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot m \cdot t_2 \cdot n \cdot t_3 = s'$$

m is P and n is P Then, in $A \multimap B$ 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 A move between m and n so must be that there is no move in t_2 by the same agent as n . Then:

$$s'' = t_1 \cdot n \cdot m \cdot t_2 \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot m \cdot n \cdot t_2 \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot m \cdot t_2 \cdot n \cdot t_3 = s'$$

in all cases, since $s'' \rightsquigarrow_{A \multimap B} s' \rightsquigarrow s$. Furthermore, in all cases

$$s'' \upharpoonright_A = t_1 \upharpoonright_A \cdot n \cdot m \cdot t_3 \upharpoonright_A = s_1 \cdot n \cdot m \cdot s_2 = s'_A \quad s'' \upharpoonright_B = s' \upharpoonright_B = s_B$$

as desired.

Now, suppose $s'_B \rightsquigarrow_B s_B$. We construct by induction on the length of the derivation $s'_B \rightsquigarrow_B s_B$ an $s' \rightsquigarrow_{A \multimap B} s$ such that $s' \upharpoonright_A = s_A$ and $s' \upharpoonright_B = s'_B$. If the length of the derivation is 0 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 $s' \rightsquigarrow_{A \multimap B} s$ such that $s' \upharpoonright_A = s_A$ and $s' \upharpoonright_B = s_1 \cdot n \cdot m \cdot s_2$. 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 $t_1|_B = s_1$ and $t_3|_B = s_2$. 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 $\lambda_B(m) = \lambda_B(n)$ or $\lambda_B(m) = O$ and $\lambda_B(n) = P$. In all cases the polarity is preserved as B is positive in $A \multimap B$.

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:

$$s'' = t_1 \cdot m \cdot n \cdot t_2 \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot n \cdot m \cdot t_2 \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot n \cdot t_2 \cdot m \cdot t_3 = s'$$

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:

$$s'' = t_1 \cdot m \cdot n \cdot t_2 \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot n \cdot m \cdot t_2 \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot n \cdot t_2 \cdot m \cdot t_3 = s'$$

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

$$s'' = t_1 \cdot t_2 \cdot m \cdot n \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot t_2 \cdot n \cdot m \cdot t_3 \rightsquigarrow_{A \multimap B} t_1 \cdot n \cdot t_2 \cdot m \cdot t_3 = s'$$

In all cases, since $s' \rightsquigarrow_{A \multimap B} s$ it follows that $s'' \rightsquigarrow_{A \multimap B} s' \rightsquigarrow_{A \multimap B} s$. Furthermore, in all cases

$$s''|_A = s'|_A = s_A \quad s''|_B = t_1|_B \cdot m \cdot n \cdot t_3|_B = s_1 \cdot m \cdot n \cdot s_2 = s'_B$$

as desired.

The claim follows from applying the two arguments above in sequence. $\square$

Lemma 31.7.3. *If*

$$s \rightsquigarrow_{A \multimap B} t$$

then

$$s \downarrow_B \rightsquigarrow_B t \downarrow_B \quad t \downarrow_A \rightsquigarrow_A s \downarrow_A$$

Proof. We prove the result by induction on the length of the derivation

$$s \rightsquigarrow_{A \multimap B} t$$

If the derivation has length 0 then $s = t$ and hence

$$s \downarrow_A = t \downarrow_A \quad s \downarrow_B = t \downarrow_B$$

and in particular

$$s \downarrow_B \rightsquigarrow_B t \downarrow_B \quad t \downarrow_A \rightsquigarrow_A s \downarrow_A$$

Otherwise, suppose

$$s = s_1 \cdot m \cdot n \cdot s_2 \rightsquigarrow_{A \multimap B}^1 s_1 \cdot n \cdot m \cdot s_2 \rightsquigarrow_{A \multimap B} t$$

By induction there are derivations

$$(s_1 \cdot n \cdot m \cdot s_2) \downarrow_B \rightsquigarrow_B t \downarrow_B \quad t \downarrow_A \rightsquigarrow_A (s_1 \cdot n \cdot m \cdot s_2) \downarrow_A$$

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,

$$s \downarrow_B = s_1 \downarrow_B \cdot m \cdot n \cdot s_2 \downarrow_B \rightsquigarrow_B s_1 \downarrow_B \cdot n \cdot m \cdot s_2 \downarrow_B = s_1 \cdot n \cdot m \cdot s_2 \downarrow_B \rightsquigarrow_B t \downarrow_B$$

and

$$t \downarrow_A \rightsquigarrow_A (s_1 \cdot n \cdot m \cdot s_2) \downarrow_A = (s_1 \cdot s_2) \downarrow_A = s \downarrow_A$$

m is a move in B and n is a move in A Note that in this case:

$$s \downarrow_B = s_1 \downarrow_B \cdot m \cdot s_2 \downarrow_B = (s_1 \cdot n \cdot m \cdot s_2) \downarrow_B \rightsquigarrow_B t \downarrow_B$$

$$t \upharpoonright_A \rightsquigarrow_A (s_1 \cdot n \cdot m \cdot s_2) \upharpoonright_A = s_1 \upharpoonright_A \cdot n \cdot s_2 \upharpoonright_A = s \upharpoonright_A$$

m is a move in A and n is a move in B

$$s \upharpoonright_B = s_1 \upharpoonright_B \cdot n \cdot s_2 \upharpoonright_B = (s_1 \cdot n \cdot m \cdot s_2) \upharpoonright_B \rightsquigarrow_B t \upharpoonright_B$$

$$t \upharpoonright_A \rightsquigarrow_A (s_1 \cdot n \cdot m \cdot s_2) \upharpoonright_A = s_1 \upharpoonright_A \cdot m \cdot s_2 \upharpoonright_A = s \upharpoonright_A$$

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 $A \multimap B$ than they have in A , so that

$$n \cdot m \rightsquigarrow_A m \cdot n$$

This justifies that:

$$s \upharpoonright_B = s_1 \upharpoonright_B \cdot s_2 \upharpoonright_B = s_1 \cdot n \cdot m \cdot s_2 \upharpoonright_B \rightsquigarrow_B t \upharpoonright_B$$

and

$$t \upharpoonright_A \rightsquigarrow_A (s_1 \cdot n \cdot m \cdot s_2) \upharpoonright_A = s_1 \upharpoonright_A \cdot n \cdot m \cdot s_2 \upharpoonright_A \rightsquigarrow_A s_1 \upharpoonright_A \cdot m \cdot n \cdot s_2 \upharpoonright_A (s_1 \cdot s_2) \upharpoonright_A = (s_1 \cdot m \cdot n \cdot s_2) \upharpoonright_A = s \upharpoonright_A$$

In all cases we obtain derivations

$$s \upharpoonright_B \rightsquigarrow_B t \upharpoonright_B \quad t \upharpoonright_A \rightsquigarrow_A s \upharpoonright_A$$

□

Lemma 31.7.4. *For plays $s_0, s_1 \in P_A$ such that*

$$\forall \alpha \in \Upsilon. \pi_\alpha(s_0) = \pi_\alpha(s_1)$$

there is a derivation

$$s_1 \rightsquigarrow_A s_0$$

if and only if there is a play $s \in \text{copy}_A$ such that

$$s \upharpoonright_{A_1} = s_1 \quad s \upharpoonright_{A_0} = s_0$$

Proof. For the forward direction, note that by the definition of copy_A there is at least one play $s' \in \text{copy}_A$ such that

$$s' \upharpoonright_{A_0} = s_0 \quad s' \upharpoonright_{A_1} = s_1$$

By Lemma 31.7.2 it follows that there is a play s such that

$$s \upharpoonright_{A_0} = s_0 \quad s \upharpoonright_{A_1} = s_1 \quad s \rightsquigarrow_{A \multimap A} s'$$

and then, by Proposition 12.3.1 it follows that

$$s \in \text{copy}_A$$

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

$$p \upharpoonright_{A_0} = p \upharpoonright_{A_1}$$

If $p = s$, then in particular

$$s_0 = s \upharpoonright_{A_0} = s \upharpoonright_{A_1} = s_1$$

Otherwise,

For the reverse direction, first note that if a play $t \in \text{copy}_A$ is alternating then

$$t \upharpoonright_{A_0} = t \upharpoonright_{A_1}$$

Indeed, by Lemma 31.8.5, it follows that if

$$t = t_1 \cdot m_1 \cdot m_2 \cdot t_2$$

and $\lambda_{A \multimap A}(m_1) = O$ then $\alpha(m_1) = \alpha(m_2)$. But as every agent plays according to copy_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

$$t \upharpoonright_{A_0} = t \upharpoonright_{A_1}$$

Now, note that by Lemma 31.7.1 there exists t an alternating play and s_O a sequence of Opponent moves such that

$$s \rightsquigarrow_{A \multimap A} t \cdot s_O$$

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

$$\pi_\alpha(t \cdot s_O) = \pi_\alpha(s) \in \text{copy}_A$$

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

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

In particular, by Lemma 31.7.3,

$$s \upharpoonright_{A_1} \rightsquigarrow_A t \upharpoonright_{A_1} \qquad t \upharpoonright_{A_0} \rightsquigarrow_A s \upharpoonright_{A_0}$$

but then:

$$s_1 = s \upharpoonright_{A_1} \rightsquigarrow_A t \upharpoonright_{A_1} = t \upharpoonright_{A_0} \rightsquigarrow_A s \upharpoonright_{A_0} = s_0$$

as desired. $\square$

Proposition 31.7.1 (cf. Proposition 13.3.1). *s_1 linearizes to s_0 if and only if there exists a play $s \in \text{copy}_A$ such that*

$$s \upharpoonright_{A_0} = s_0 \quad s \upharpoonright_{A_1} = s_1$$

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

$$s_1 \cdot s_P \rightsquigarrow_A s_0 \cdot s_O$$

But then, note that

$$s_1 / s_O \cdot s_P \rightsquigarrow_A s_0$$

by Lemma 31.9.2. Hence, there is a play s of copy_A such that

$$s \upharpoonright_{A_0} = s_1 / s_O \cdot s_P \quad s \upharpoonright_{A_1} = s_0$$

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,

$$s / s_P \cdot s_P \rightsquigarrow_{A \multimap A} s$$

so that s / s_P (the subsequence of s where the moves in s_P have been removed) is in copy_A by Prop. 12.3.1 and prefix-closure. Note that

$$(s / s_P) \upharpoonright_{A_1} = s_1 / s_O \quad (s / s_P) \upharpoonright_{A_0} = s_0$$

Now, let m be a move in s_O . Because,

$$s_1 \cdot s_P \rightsquigarrow_A s_0 \cdot s_O$$

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 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

$$s' \rightsquigarrow_{A \multimap A} s / s_P \cdot s_O$$

and moreover

$$s' \upharpoonright_{A_0} = s_1$$

constructed by reversing the swaps involving s_O up to the swaps with A_0 moves and the removal of the swaps that involve s_P , which is possible by the remarks above. By Prop. 12.3.1 $s' \in \text{copy}_A$. Moreover, as the derivation above does not involve swaps between two moves of A_0 it follows that

$$s' \upharpoonright_{A_0} = (s / s_P) \upharpoonright_{A_0} = s_0$$

And therefore s' is the desired play.

Conversely, suppose there exists such a play $s \in \text{copy}_A$. Then, note that for any $\alpha \in \Upsilon$,

$$\pi_\alpha(s) \in \text{copy}_A$$

so that in particular there is a sequence of at most one Opponent move s_α such that either

$$\pi_\alpha(s) \upharpoonright_{A_0} \cdot s_\alpha = \pi_\alpha(s) \upharpoonright_{A_1}$$

or

$$\pi_\alpha(s) \upharpoonright_{A_0} = \pi_\alpha(s) \upharpoonright_{A_1} \cdot s_\alpha$$

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:

$$\forall \alpha \in \Upsilon. \pi_\alpha((s/s_O) \upharpoonright_{A_0}) = \pi_\alpha((s/s'_O) \upharpoonright_{A_1})$$

so let $p = s/s'_O$ and note that by Lemma 31.7.4 it follows that

$$p \upharpoonright_{A_1} \rightsquigarrow_B p \upharpoonright_{A_0}$$

Now, note that $s'_O \upharpoonright_{A_0}$ is a sequence of P moves in A while $s'_O \upharpoonright_{A_1}$ is a sequence of O moves in A . We claim that:

$$(p \cdot s'_O) \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_0} \rightsquigarrow_A (p \cdot s'_O) \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_1}$$

Indeed, note that

$$(p \cdot s'_O) \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_0} = p \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_0} \rightsquigarrow_A p \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_1} = p \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_1} = (p \cdot s'_O) \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_1}$$

is valid as long as

$$s'_O \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_0} \rightsquigarrow_A s'_O \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_1}$$

As $s'_O \upharpoonright_{A_1}$ only contains O moves and $s'_O \upharpoonright_{A_0}$ only contains P moves the reduction is valid as long as no agent that appears in $s'_O \upharpoonright_{A_1}$, appears in $s'_O \upharpoonright_{A_0}$. 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

$$s_1 = s \upharpoonright_{A_1} \rightsquigarrow_A (s/s'_O) \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_1} = (s/s'_O \cdot s'_O) \upharpoonright_{A_1} = (p \cdot s'_O) \upharpoonright_{A_1}$$

and that

$$(p \cdot s'_O) \upharpoonright_{A_0} = (s/s'_O \cdot s'_O) \upharpoonright_{A_0} = (s/s'_O) \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_0} \rightsquigarrow_A s \upharpoonright_{A_0} = s_0$$

Hence, by finally taking

$$s_P = s'_O \upharpoonright_{A_0} \quad s_O = s'_O \upharpoonright_{A_1}$$

$$s_1 = s \upharpoonright_{A_1} \cdot s_P \rightsquigarrow_A (p \cdot s'_O) \upharpoonright_{A_1} \cdot s'_O \upharpoonright_{A_0} \rightsquigarrow_A (p \cdot s'_O) \upharpoonright_{A_0} \cdot s'_O \upharpoonright_{A_1} = s \upharpoonright_{A_0} \cdot s_O \rightsquigarrow_A s_0 \cdot s_O$$

so that s_1 linearizes to s_0 . $\square$

31.8 Atomicity

Proposition 31.8.1. *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 $s \upharpoonright_{A,C} \in \sigma \circ \tau$. First, let $s \upharpoonright_C = p_1 \cdot m \cdot m' \cdot p_2$ where $\alpha(m) = \alpha(m')$ and $\lambda_C(m) = O$. Then, since τ is atomic, $s \upharpoonright_{B,C} = p'_1 \cdot m \cdot s' \cdot m' \cdot p'_2$ 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 $s \upharpoonright_{A,C} = p''_1 \cdot m \cdot s'' \cdot m' \cdot p''_2$ then every move in s'' is by $\alpha(m)$. The argument is analogous for $s \upharpoonright_C = p \cdot m$ with $\lambda_C(m) = O$. $\square$

Lemma 31.8.1. *Let $s \in P_{A \multimap B}$. Then if $s \upharpoonright_B \in P_{!B}$ then $s \in P_{!A \multimap !B}$.*

Proof. We argue by induction by keeping track of a prefix $p \sqsubseteq_{\text{even}} s$ such that $p \in P_{!A \multimap !B}$ and such that every agents last move was in 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 B). Otherwise, let $p \sqsubseteq_{\text{even}} s$ be such that $p \in P_{!A \multimap !B}$. If $p = s$ we are also done. Otherwise, there is m a move in 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 A . As at p every agent had its last move in B at $p \cdot m$ only $\alpha(m)$ can move in 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 A and a move m' in B such that

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

By alternation in 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. $\square$

Lemma 31.8.2. *For sets of plays*

$$S \subseteq P_{A \multimap B} \quad T \subseteq P_{B \multimap C}$$

and

$$(S \cap P_{!A \multimap !B}); (T \cap P_{!B \multimap !C}) = (S; T) \cap P_{!A \multimap !C}$$

Proof. For simplicity we will use the following notation:

$$U = (- \cap P_{! \multimap ! \multimap !})$$

First, suppose $s \in U(S); U(T)$. Then, there is

$$t \in U \ S \ ; \ U \ T$$

such that

$$t \upharpoonright_{!A, !B} \in U(S) \quad t \upharpoonright_{!B, !C} \in U(T) \quad t \upharpoonright_{!A, !C} = s$$

in particular, seen as a play of $S \ ; \ T$, we have:

$$t \upharpoonright_{A, B} \in S \quad t \upharpoonright_{B, C} \in T \quad t \upharpoonright_{A, C} = s$$

and hence

$$s \in S; T$$

so that

$$U(S); U(T) \subseteq U(S; T)$$

Now, for the reverse inclusion let $s \upharpoonright_{!A, !C} \in U(S; T)$. Then, $s \upharpoonright_C \in P_{!C}$ so that by Lemma 31.8.1 it follows that $s \upharpoonright_{B, C} \in P_{!B \multimap !C}$ and therefore $s \upharpoonright_{B, C} \in U(T)$. In addition, we now have that $s \upharpoonright_B \in P_{!B}$. Another application of Lemma 31.8.1 then gives that $s \upharpoonright_{A, B} \in P_{!A \multimap !B}$ and

hence $s \upharpoonright_{A,B} \in \mathbf{Emb}_{\mathbf{Atom}}(S)$. Hence, $s \upharpoonright_{A,C} \in U(S); U(T)$. Therefore

$$U(S;T) = U(S);U(T)$$

□

For the remainder of this section, let $\mathbf{O-recep}(-)$ be the receptive closure operation in **Atomic**.

Lemma 31.8.3.

$$\mathbf{O-recep}(-) : \underline{\mathbf{Atomic}} \rightarrow \underline{\mathbf{Safe}}$$

defines a semifunctor.

Proof. Let $\tau : !A \multimap !B$ and $\sigma : !B \multimap !C$ be atomic strategies. It is easy to see that:

$$\mathbf{O-recep}(\tau; \sigma) \subseteq \mathbf{O-recep}(\mathbf{O-recep}(\tau); \mathbf{O-recep}(\sigma)) = \mathbf{O-recep}(\tau); \mathbf{O-recep}(\sigma)$$

For the reverse direction, let $t \cdot o \in \mathbf{O-recep}(\tau)$ and $s \cdot o' \in \mathbf{O-recep}(\sigma)$ where o, o' are either ϵ or an Opponent move, such that moreover they are composable. Notice that by the switching condition, either

- $o = m$ is a move in $!A$ and $o' = \epsilon$,
- $o = \epsilon$ and $o' = m$ is a move in $!C$,
- $o = m$ is a P move in $!B$ and $o' = \epsilon$,
- $o' = m$ is an O move in $!B$ and $o = \epsilon$,

The first two cases are easily handled as in both cases the added O -move must appear at the end of the interaction sequence (by the switching condition) so that $(t; s) \cdot m \in \mathbf{O-recep}(\tau; \sigma)$. For the third case, observe that if the matching move to m appears in the middle of t , then the following move must be in $!B$ as well. But then, that move must also appear in s , a contradiction. Therefore, m is the last move of t so that $t = t' \cdot m$ and hence $t'; (s \cdot m) = t'; s \in \tau; \sigma \subseteq \mathbf{O-recep}(\tau; \sigma)$ already. The last case is handled similarly. □

Lemma 31.8.4. *For any $\sigma : A \multimap B$*

$$K_{\text{Atom}} \sigma = \text{O-recep}(\sigma \cap P_{!A \multimap !B})$$

Proof. It is straight-forward to see that by 31.8.1

$$K_{\text{Atom}} \sigma = \text{atocopy}_A; \sigma; \text{atocopy}_B = \text{atocopy}_A; (\sigma \cap P_{!A \multimap !B}); \text{atocopy}_B$$

But $(\sigma \cap P_{!A \multimap !B})$ is just an atomic strategy, and atocopy is just the sequential copycat, so

$$\text{atocopy}_A; (\sigma \cap P_{!A \multimap !B}); \text{atocopy}_B = \text{O-recep}(\sigma \cap P_{!A \multimap !B})$$

□

Proposition 31.8.2.

$$K_{\text{Atom}} : \underline{\text{Safe}} \longrightarrow \mathbf{K}_{\text{Atom}}$$

defines an enriched semifunctor.

Proof. By monotonicity of composition, Lemmas 31.8.4 and 31.8.2 and Proposition 31.8.3

$$\begin{aligned} K_{\text{Atom}}(\sigma); K_{\text{Atom}}(\tau) &= \text{O-recep}(\sigma \cap P_{!A \multimap !B}); \text{O-recep}(\tau \cap P_{!B \multimap !C}) \\ &= \text{O-recep}(\sigma \cap P_{!A \multimap !B}); (\tau \cap P_{!B \multimap !C}) \\ &= \text{O-recep}((\sigma; \tau) \cap P_{!A \multimap !B}) \\ &= \text{Lin}_{\text{Atom}}(\sigma; \tau) \end{aligned}$$

Now, suppose $\sigma \subseteq \sigma'$, then:

$$K_{\text{Atom}}(\sigma) = \text{atocopy}_A; \sigma; \text{atocopy}_B \subseteq \text{atocopy}_A; \sigma'; \text{atocopy}_B = K_{\text{Atom}}(\sigma')$$

by monotonicity of composition.

Similarly,

$$K_{\text{Atom}}(\cup_{i \in I} \sigma_i) = \text{atocopy}_A; \cup_{i \in I} \sigma_i; \text{atocopy}_B = \cup_{i \in I} \text{atocopy}_A; \sigma_i; \text{atocopy}_B = \cup_{i \in I} K_{\text{Atom}}(\sigma_i)$$

□

Proposition 31.8.3 (cf. Proposition 15.1.2). *There is an equivalence of categories:*

$$\mathbf{Atomic} \cong \mathbf{K}_{\text{Atom}}$$

Proof. The fact that both E and E^{-1} are semifunctors is immediate, both are essentially the identity functor.

It is immediate to see that

$$E E^{-1} A = A \quad E^{-1} E !A = !A$$

moreover, let

$$\tau : !A \multimap !B$$

be an atomic strategy. Then,

$$E^{-1} E \tau = E^{-1} \tau = \tau \cap P_{!A \multimap !B} = \tau$$

that

$$E^{-1} E \sigma = \sigma$$

follows similarly. □

Lemma 31.8.5. *If $s \in P_A$ is alternating then if $s = p \cdot m \cdot m' \cdot p'$ is such that $\lambda_A(m) = O$ and $\lambda_A(m') = P$ then $\alpha(m) = \alpha(m')$.*

Proof. We prove the result by induction over the size of the play s , where we also maintain that if $p \sqsubseteq_{\text{even}} s$ then for every $\alpha \in \Upsilon$, $\pi_\alpha(p)$ is even-length. If $s = \epsilon$ the result is vacuously true. So suppose $p \sqsubseteq_{\text{even}} s$ satisfies the lemma. If $p = s$ we are done. Otherwise there is

at least one move m such that $p \cdot m \sqsubseteq s$. Since s is alternating and p is even it follows that $\lambda_A(m) = O$. If $p \cdot m = s$ we are done, as the move preceding m is by Proponent (so that the claim does not apply) and p already satisfies it. Otherwise, there is another move m' such that $p \cdot m \cdot m' \sqsubseteq s$. Again, by alternation, $\lambda_A(m') = P$. So we must show that $\alpha(m) = \alpha(m')$. But notice that for every $\alpha \in \Upsilon$, $\pi_\alpha(p)$ is even-length, by induction. Therefore, by local alternation it follows that in every α Opponent is to move. Hence, it must be that $\alpha(m') = \alpha(m)$. In particular, it is still the case that every agent's play is even-length. $\square$

Proposition 31.8.4 (cf. Proposition 15.1.3). *The irreducibles of $\rightsquigarrow_A$ are precisely the alternating plays of P_A .*

Proof. Let $s \in P_A$. Note that by definition the projection $\pi_\alpha(s)$ is alternating for every $\alpha \in \Upsilon$. So suppose

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

and $m \neq m'$. Then, note that if $\lambda_A(m) = \lambda_A(m')$ then the play is not alternating. Otherwise, the only rule that applies is when $\alpha(m) \neq \alpha(m')$ and $\lambda_A(m) = O$ and $\lambda_A(m') = P$. By lemma 31.8.5 it follows that s is not alternating.

This shows that every alternating play is irreducible. We now argue that every irreducible s is alternating. Indeed, suppose that no rule can be applied. It follows then that if:

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

then either $\alpha(m) = \alpha(m')$ or $\lambda_A(m) = P$ and $\lambda_A(m') = O$. We argue by induction that s is alternating. ϵ is trivially alternating, so let $p \sqsubseteq_{\text{even}} s$ be such that p is alternating. If $p = s$ we are done. Otherwise, there is some m such that $p \cdot m \sqsubseteq s$. Since p is alternating and s is locally alternating by lemma 31.8.5 it follows that $\lambda(m) = O$ (otherwise it breaks local alternation for $\alpha(m)$) and in particular $p \cdot m$ is alternating. If $p \cdot m = s$ we are done, otherwise there is some m' such that $p \cdot m \cdot m' \sqsubseteq s$. As no rule applies either $\alpha(m) = \alpha(m')$ or $\lambda_A(m) = P$ and $\lambda_A(m') = O$. In the first case, again, by local alternation, $\lambda_A(m') = P$ and alternation follows for $p \cdot m \cdot m'$. The second case can't apply as we already argued that

$$\lambda_A(m) = O.$$

□

Proposition 31.8.5 (cf. Proposition 15.1.4). *For any saturated $\sigma : A \multimap B$:*

$$U_{\text{Atom}} \sigma = \{s \in \sigma \mid s \upharpoonright_A \in \Downarrow (\sigma \upharpoonright_A)\}$$

Proof. After unrolling the definition of $\Downarrow (\sigma \upharpoonright_A)$ and U_{Atom} this follows from Lemma 31.8.1 together with Proposition 15.1.3. □

31.9 Atomic Linearizability

Proposition 31.9.1 (cf. Proposition 15.1.5). *If $s, t \in P_A$ then $s \equiv_A t$ if and only if s and t are compatible and $\prec_s = \prec_t$.*

Proof. ($\Rightarrow$) Since all the swaps allowed by $\rightsquigarrow_A$ 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 $s \not\equiv_A t$. 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 $\prec_t = \prec_s$. $\square$

Proposition 31.9.2 (cf. Proposition 15.1.6). *For plays $s, t \in P_A$, there is a derivation*

$$s \rightsquigarrow_A t$$

if and only if s is compatible with t and

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

Proof. ($\Rightarrow$) Note that if

$$s \rightsquigarrow_A^1 t$$

then either $\prec_s = \prec_t$ by Prop. 15.1.5 or the derivation is a $OP \rightsquigarrow PO$ swap. We argue that

$$\prec_s \subseteq \prec_t$$

in that case. Indeed, suppose

$$s = s_1 \cdot m \cdot n \cdot s_2 \rightsquigarrow_A^1 s_1 \cdot n \cdot m \cdot s_2 = t$$

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

$$o_1 = (p, q) \quad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p + 1, q) \quad \rho(o_2) = (p', q')$$

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

$$o_1 = (p, q) \quad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p, q - 1) \quad \rho(o_2) = (p', q')$$

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

$$o_1 = (p, q) \quad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p, q) \quad \rho(o_2) = (p' + 1, q')$$

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

$$o_1 = (p, q) \quad o_2 = (p', q')$$

moreover

$$\rho(o_1) = (p, q) \quad \rho(o_2) = (p', q' - 1)$$

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. 15.1.5, 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 $s \rightsquigarrow_A s'$. 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

$$o = (p, q) \quad o' = (p', q')$$

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

$$o_1 = (p_1, q_1) \quad o_2 = (p_2, q_2)$$

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,

$$q_1 < p \quad \text{and} \quad q' < p_2$$

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:

$$s = s_1 \cdot m \cdot s_O \cdot s_P \cdot n \cdot s_3 \equiv_A s_1 \cdot s_O \cdot m \cdot s_P \cdot n \cdot s_3 \equiv_A s_1 \cdot s_O \cdot m \cdot n \cdot s_P \cdot s_3 \rightsquigarrow_A s_O \cdot n \cdot m \cdot s_P \cdot s_3$$

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 $\text{op}(s)$. $\square$

The following couple of lemmas are straight-forward.

Lemma 31.9.1. *If*

$$s \cdot m \cdot t \rightsquigarrow_A s' \cdot m$$

then

$$s \cdot t \rightsquigarrow_A s'$$

Lemma 31.9.2. *Let $s, s' \in P_A$, s_P a sequence of Proponent moves and s_O a sequence of Opponent moves. If*

$$s \cdot s_P \rightsquigarrow_A s' \cdot s_O$$

then let $(s \setminus s_O) \in P_A$ be the subsequence of s obtained by removing the pending Opponent moves that appear in s_O , then

$$s \cdot s_P \rightsquigarrow_A (s \setminus s_O) \cdot s_P \cdot s_O \rightsquigarrow_A s' \cdot s_O$$

Proposition 31.9.3 (cf. Corollary 15.1.1). *A play $s \in P_A$ 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

$$s \cdot s_P \rightsquigarrow_A t \cdot s_O$$

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 31.9.2

$$s \cdot s_P \rightsquigarrow_A \text{complete}(s \cdot s_P) \cdot s_O \rightsquigarrow t \cdot s_O$$

and then by Lemma 31.9.1 we have that

$$\text{complete}(s \cdot s_P) \rightsquigarrow t$$

so that by Prop. 15.1.6 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 $\text{complete}(s \cdot s_P) \rightsquigarrow t'$. By Lemma 31.9.2 we have that there is s' such that

$$s \cdot s_P \rightsquigarrow_A s' \cdot s_P \cdot s_O \rightsquigarrow_A t \cdot s_O$$

but then, by the reasoning above, there is at most one pending Opponent move in s' so that

$$s' \cdot s_O \cdot s_P \rightsquigarrow_A s' \cdot s_P \cdot s_O \rightsquigarrow_A t \cdot s_O = t' \cdot (o \cdot s_O)$$

implies by 31.9.2 that there is $s'' \in P_A$ such that

$$s' \cdot s_O \cdot s_P \rightsquigarrow_A s'' \cdot s_P \cdot (o \cdot s_O) \rightsquigarrow_A t' \cdot (o \cdot s_O)$$

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, $s'' \cdot s_P = \text{complete}(s \cdot s_P)$. By the previous reasoning, the result follows.

($\Leftarrow$) By Proposition 15.1.6 it follows that there is a reduction $\text{complete}(s \cdot s_P) \rightsquigarrow_A t$. Now, let s_O be a sequence containing all the Opponent moves removed by $\text{complete}(-)$. 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:

$$s \cdot s_P \rightsquigarrow_A \text{complete}(s \cdot s_P) \cdot s_O \rightsquigarrow_A t \cdot s_O$$

proving that s is linearizable to t . □

31.10 Linearization Points

Proposition 31.10.1. *Let*

$$\nu_A : \mathbf{1} \multimap A \qquad \nu_B : \mathbf{1} \multimap B$$

and

$$\sigma : A \multimap B$$

Then, if there exists a punctual extension σ_{lp} of σ such that

$$((\nu_A \otimes P_B); \sigma_{\text{lp}}) \upharpoonright_{B_0} \subseteq \nu_B$$

then

$$\nu_A; \sigma \subseteq K_{\text{copy}} \nu_B$$

Proof. Suppose that

$$((\nu_A \otimes P_B); \sigma_{\text{lp}}) \upharpoonright_{B_0} \subseteq \nu_B$$

and let $s \upharpoonright_{\mathbf{1}, B} \in \nu_A; \sigma$. Now, as

$$\sigma_{\text{lp}} \upharpoonright_{A, B_1} = \sigma$$

it follows that for any play $s'' \in \sigma$ there is a play $s' \in \sigma_{\text{lp}}$ such that

$$s'' \upharpoonright_{A, B_1} = s'$$

So let s' be the play in σ_{lp} corresponding to $s \upharpoonright_{A, B}$, in particular

$$s' \upharpoonright_{A, B_1} = s \qquad s' \upharpoonright_A = s \upharpoonright_A \in \nu_A$$

By assumption,

$$s' \upharpoonright_{B_0} \in \nu_B$$

Moreover, because σ_{lp} is punctual,

$$\sigma_{\text{lp}} \upharpoonright_{B_0, B_1} \subseteq \text{copy}_B$$

so that

$$s' \upharpoonright_{B_0, B_1} \in \text{copy}_B$$

by Prop. 13.3.1 it follows that $s' \upharpoonright_{B_1} = s \upharpoonright_{1, B}$ is linearizable to $s' \upharpoonright_{B_0}$. Hence, by Proposition 13.1.1,

$$s \upharpoonright_{1, B} \in K_{\text{copy}} \nu_B$$

□

Proposition 31.10.2 (cf. Proposition 16.3.1). *Let*

$$\nu_A : \mathbf{1} \multimap (A, \text{atocopy}_A) \qquad \nu_B : \mathbf{1} \multimap (B, \text{atocopy}_B)$$

and

$$\sigma : (A, \text{atocopy}_A) \multimap (B, \text{copy}_B)$$

Then,

$$\nu_A; \sigma \subseteq \text{Lin}_{\text{Atom}} \nu_B$$

if and only if there exists a punctual extension σ_{lp} of σ such that

$$((\nu_A \otimes P_B^{\text{Atom}}); \sigma_{\text{lp}}) \upharpoonright_{B_0} \subseteq \nu_B$$

Proof. The reverse direction is immediate from Prop. 16.3.1. Hence, we only prove the forward direction.

Suppose that

$$\nu_A; \sigma \subseteq \text{Lin}_{\text{Atom}} \nu_B$$

we will construct a punctual extension σ_{lp} of σ such that

$$((\nu_A \otimes P_B); \sigma_{\text{lp}}) \upharpoonright_{B_0} \subseteq \nu_B$$

We will do this by assigning a play $\text{lp}(s) \in P_{A \otimes B \multimap B}$ for every play $s \in \sigma$ such that

$$\text{lp}(s) \upharpoonright_{A, B_1} = s \quad \text{lp}(s) \upharpoonright_{B_0, B_1} \in \text{copy}_B$$

and moreover, if $s \upharpoonright_A \in \nu_A$ then

$$\text{lp}(s) \upharpoonright_{B_0} \in \nu_B$$

and then we will finish by defining

$$\sigma_{\text{lp}} = \text{strat}(\{\text{lp}(s) \mid s \in \sigma\})$$

First, note that if $s \in \sigma$ and $s \upharpoonright_A \notin \nu_A$ then there is no constraint on how we may construct the corresponding play of σ_{lp} other than that the projection to the B components must play as copy_B . Now, by definition of copy_B there is a play $\text{copy}_B(s)$ of copy_B such that

$$\text{copy}_B(s) \upharpoonright_{B_1} = s \upharpoonright_B$$

Now, if $s \upharpoonright_A \in \nu_A$ we must ensure that $\text{lp}(s) \upharpoonright_{B_0} \in \nu_B$. By assumption,

$$s \upharpoonright_B \in \text{Lin}_{\text{Atom}} \nu_B$$

therefore, there is $t \in \nu_B$ such that s is linearizable to ν_B by Prop. 13.1.1. By Prop. 13.3.1 it follows that there is a play $\text{copy}_B(s)$ such that

$$\text{copy}_B(s) \upharpoonright_{B_1} = s \upharpoonright_B \quad \text{copy}_B \upharpoonright_{B_0} = t$$

Either way, we proceed by constructing $\text{lp}(s)$ by using s and $\text{copy}_B(s)$. We do so inductively

and keep track of suffixes s_A and s_B of s and $\text{copy}_B(s)$ with moreover the invariant that

$$s_A \upharpoonright_B = s_B \upharpoonright_{B_1}$$

and that the first move in s_A is in B , and the first move in s_B is the same move in B_1 . Initially we let $s_A = s$ and $s_B = \text{copy}_B(s)$, and at any point we have $s = p_A \cdot s_A$ and $\text{copy}_B(s) = p_B \cdot s_B$. This justifies the last invariant in that we keep track of a play $\text{lp}(p_A)$ satisfying:

$$\text{lp}(p_A) \upharpoonright_{A, B_1} = p_A \quad \text{lp}(p_A) \upharpoonright_{B_0, B_1} = p_B \quad \text{lp}(p_A) \upharpoonright_{A, B_0} \text{ is atomic}$$

Moreover, we will maintain that for every $\alpha \in \Upsilon$, the last move by α in p_B is a P move in B_1 , if it exists, and that no agent's next move in $s_B \upharpoonright_{B_0}$ is a Proponent move. If $s_A = \epsilon$ or $s_B = \epsilon$ then in fact $s_A = s_B = \epsilon$ as

$$s_A \upharpoonright_B = s_B \upharpoonright_{B_1} = \epsilon$$

and the first move in both is in B and B_1 respectively. In this case we let

$$\text{lp}(s) = \text{lp}(s \cdot s_A) = \text{lp}(p_A) \cdot \epsilon$$

which serves as our base case. Otherwise,

$$s_A = m \cdot s'_A \quad s_B = m \cdot s'_B$$

Suppose first that there are no more moves in B in s'_A . Then, there are also no more moves in B_1 in s'_A . Hence, all the moves in s'_A are moves in A and all the moves in s'_B are in B_0 . Then, we let

$$\text{lp}(p_A \cdot s_A) = \text{lp}(p_A) \cdot m \cdot s'_A$$

Note that at this is a terminal case for the induction as we have:

$$\text{lp}(s) \upharpoonright_{A,B_1} = \text{lp}(p_A \cdot s_A) \upharpoonright_{A,B_1} = \text{lp}(p_A) \cdot m \cdot s'_A \upharpoonright_{A,B_1} = \text{lp}(p_A) \upharpoonright_{A,B_1} \cdot m \cdot s'_A \upharpoonright_{A,B_1} = p_A \cdot m \cdot s'_A = s$$

$$\begin{aligned} \text{lp}(s) \upharpoonright_{B_0,B_1} &= \text{lp}(p_A \cdot s_A) \upharpoonright_{B_0,B_1} = \text{lp}(p_A) \cdot m \cdot s'_A \upharpoonright_{B_0,B_1} = \text{lp}(p_A) \upharpoonright_{B_0,B_1} \cdot m \cdot s'_A \upharpoonright_{B_0,B_1} \\ &= p_B \cdot m \sqsubseteq p_B \cdot s_B = \text{copy}_B(s) \end{aligned}$$

In particular, by prefix-closure,

$$\text{lp}(s) \upharpoonright_{B_0,B_1} \in \text{copy}_B$$

and in the case where $s \upharpoonright_A \in \nu_A$

$$\text{lp}(s) \upharpoonright_{B_0,B_1} \sqsubseteq \text{copy}_B(s) \in \text{copy}_B$$

so that by prefix-closure again

$$\text{lp}(s) \upharpoonright_{B_0} \sqsubseteq t \in \nu_B$$

Now, suppose there is some move n in component B in s'_A , and let moreover n be the first such move. Then:

$$s_A = m \cdot s_{A,1} \cdot n \cdot s_{A,2} \quad s_B = m \cdot s_{B,1} \cdot n \cdot s_{B,2}$$

If there is a pending Opponent move in $s_{B,1} \upharpoonright_{B_0} = s_{B,1}$, it can't be by the same agent as n . Hence, in case there is such an Opponent move, so that

$$s_{B,1} = s'_{B,1} \cdot m_O$$

Then, either $\alpha(m_O)$ has no further moves in $n \cdot s_{B,2}$, in which case we modify s_B to no harm as

$$s'_B = m \cdot s'_{B,1} \cdot n \cdot s_{B,2} \cdot m_O$$

Or there is a response m_P (and by atomicity that is at most one such pending Opponent move with a later response) so that

$$s_{B,2} = s'_{B,2} \cdot m_P \cdot s''_{B,2}$$

we modify s_B to no harm as

$$s'_B = m \cdot s'_{B,1} \cdot m_O \cdot m_P \cdot n \cdot s'_{B,2} \cdot s''_{B,2}$$

these changes cause no trouble as all the invariants are still satisfied with this modified s_B , and the modifications are essential to maintain that for every agent the next move is Opponent in B_0 for our next suffix. We therefore assume from now on that there are no pending Opponent moves in $s_{B,1}$. Moreover, $s_{A,1} \upharpoonright_A$ has at most one pending O move. If there is no such move, we let

$$\text{lp}(p_A \cdot m \cdot s_{A,1} \cdot n) = \text{lp}(p_A) \cdot m \cdot s_{A,1} \cdot s_{B,1}$$

which is a valid play by the modifications to s_B made above and from the fact that $s_{B,1}$ is atomic. Otherwise,

$$s_{A,1} = s_{A,1'} \cdot m_O$$

for some move m_O , Opponent in A . In that case, we let:

$$\text{lp}(p_A \cdot m \cdot s_{A,1}) = \text{lp}(p_A) \cdot m \cdot s'_{A,1} \cdot s_{B,1} \cdot m_O$$

We now show the invariants are maintained. In the first case, we have:

$$\text{lp}(p_A \cdot m \cdot s_{A,1}) \upharpoonright_{A,B_1} = \text{lp}(p_A) \cdot m \cdot s_{A,1} \cdot s_{B,1} \upharpoonright_{A,B_1} = \text{lp}(p_A) \upharpoonright_{A,B_1} \cdot m \cdot s_{A,1} = p_A \cdot m \cdot s_{A,1}$$

$$\text{lp}(p_A \cdot m \cdot s_{A,1}) \upharpoonright_{B_0,B_1} = \text{lp}(p_A) \cdot m \cdot s_{A,1} \cdot s_{B,1} \upharpoonright_{B_0,B_1} = \text{lp}(p_A) \upharpoonright_{B_0,B_1} \cdot m \cdot s_{B,1} = p_B \cdot m \cdot s_{B,1}$$

while in the second case we have:

$$\text{lp}(p_A \cdot m \cdot s_{A,1}) \upharpoonright_{A,B_1} = \text{lp}(p_A) \cdot m \cdot s'_{A,1} \cdot s_{B,1} \cdot m_O \cdot n \upharpoonright_{A,B_1} = \text{lp}(p_A) \upharpoonright_{A,B_1} \cdot m \cdot s'_{A,1} \cdot m_O = p_A \cdot m \cdot s_{A,1}$$

$$\text{lp}(p_A \cdot m \cdot s_{A,1}) \upharpoonright_{B_0,B_1} = \text{lp}(p_A) \cdot m \cdot s'_{A,1} \cdot s_{B,1} \cdot m_O \upharpoonright_{B_0,B_1} = \text{lp}(p_A) \upharpoonright_{B_0,B_1} \cdot m \cdot s_{B,1} = p_B \cdot m \cdot s_{B,1}$$

At this point, it is justified to define the new instances p'_A, p'_B, s'_A, s'_B of p_A, p_B, s_A, s_B as

$$p'_A = p_A \cdot m \cdot s_{A,1} \qquad s'_A = n \cdot s_{A,2}$$

$$p'_B = p_B \cdot m \cdot s_{B,1} \qquad s'_B = n \cdot s_{B,2}$$

the remaining invariants follow readily from this definitions and the remarks above.

Finally, at the end of this inductive procedure we note that we obtain $\text{lp}(s)$ satisfying all the desired claims. Finally, we let

$$\sigma_{\text{lp}} = \text{strat}(\{\text{lp}(s) \mid s \in \sigma\})$$

which is our desired punctual extension. □

Chapter 32

Crash-Safety

32.1 Basic Semicategorical Structure

In the following we assume that the operation $-; -$ is defined on arbitrary sets of plays, instead of just on strategies. The definition is exactly the same. Given $s \in P_{A \multimap B}$ and $t \in P_{B \multimap C}$ we write $s; t$ for $\{s\}; \{t\}$. We also take the convention of writing, for $s \in P_A$ with $A \in \mathbf{Crash}$,

$$s = s_1 \cdot \text{!} \cdot s_2 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

where for each i , $s_i = \text{epo}_i(s)$.

The following important lemma characterizes composition in $\mathbf{Crash}$ using composition in $\mathbf{Safe}$.

Lemma 32.1.1. *Given strategies $\sigma : A \multimap B$, $\tau : B \multimap C \in \mathbf{Crash}$, for any play $u \in \sigma; \tau$, there exists $s \in \sigma, t \in \tau$ such that*

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1} \quad t = t_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot t_{n+1}$$

and, for all i ,

$$s_i \in P_{A \multimap B} \quad \text{and} \quad t_i \in P_{B \multimap C}$$

and such that u decomposes as

$$u = u_1 \cdot \text{⚡} \cdot u_2 \cdot \text{⚡} \cdot \dots \cdot \text{⚡} \cdot u_{n+1}$$

with $u_i \in s_i; t_i$ for all i .

Proof. Suppose $u \in \sigma; \tau$. By definition there exists $u' \in \sigma \circ \tau$ such that $u' \upharpoonright_{A,B,-} \in \sigma$, $u' \upharpoonright_{-,B,C} \in \tau$ and $u' \upharpoonright_{A,-,C} = u$, we claim that assigning $s := u' \upharpoonright_{A,B,-}$ and $t := u' \upharpoonright_{-,B,C}$ the claim is proven.

Since all three of u' , s , and t are well-formed it follows that they can be written as (it will soon be clear why all plays feature the same number of epochs):

$$s = s_1 \cdot \text{⚡} \cdot \dots \cdot \text{⚡} \cdot s_{n+1} \quad t = t_1 \cdot \text{⚡} \cdot \dots \cdot \text{⚡} \cdot t_{n+1} \quad u' = u'_1 \cdot \text{⚡} \cdot \dots \cdot \text{⚡} \cdot u'_{n+1}$$

So let $u_i = u'_i \upharpoonright_{A,-,C}$. By definition we know $u' \upharpoonright_{-,B,-} = s \upharpoonright_{-,B} = t \upharpoonright_{B,-}$, so it follows that the i -th crash signal in s matches t with the i -th crash signal in u' , and that they have the same number of crash signals. Furthermore, since $s = u' \upharpoonright_{A,B,-}$, the i -th epoch of u' , i.e. u'_i , projects to the i -th epoch of s , i.e. $u'_i \upharpoonright_{A,B,-} = s_i$. Similarly, $u'_i \upharpoonright_{-,B,C} = t_i$. Hence, it follows that $u'_i \upharpoonright_{A,-,C} = u_i \in s_i; t_i$. $\square$

Proposition 32.1.1 (cf. Proposition 19.1.1). *strategy composition is well-defined and associative.*

Proof. **Well-defined** Suppose $\sigma : A \multimap B$ and $\tau : B \multimap C$, then we have $\epsilon \in \sigma$ and $\epsilon \in \tau$.

Then taking

$$\epsilon \in A \circ B \circ C$$

we have $\epsilon \upharpoonright_{A,-,C} = \epsilon \in P_{A \multimap C}^{\text{⚡}}$ which implies $\epsilon \in \sigma; \tau$. So $\sigma; \tau$ is non-empty

Now suppose $s \in \sigma; \tau$ and that $p \sqsubseteq s$, then there exists $s' \in \sigma \circ \tau$ such that $s' \upharpoonright_{A,-,C} = s$.

In particular $p \sqsubseteq s' \upharpoonright_{A,-,C}$. Hence there exists $p' \sqsubseteq s'$ such that $p' \upharpoonright_{A,-,C} = p$. Since $s' \upharpoonright_{A,B,-} \in \sigma$ and $s' \upharpoonright_{-,B,C} \in \tau$ and both σ, τ are prefix-closed, so $p' \upharpoonright_{A,B,-} \in \sigma$ and $p' \upharpoonright_{-,B,C} \in \tau$. So $p' \in \sigma \circ \tau$. Since $P_{A \multimap C}^{\text{⚡}}$ is prefix-closed, $p' \upharpoonright_{A,-,C} \sqsubseteq s' \upharpoonright_{A,-,C} \in P_{A \multimap C}^{\text{⚡}}$ so $p' \upharpoonright_{A,-,C} \in P_{A \multimap C}^{\text{⚡}}$. Hence $p \in \sigma; \tau$.

Lastly, we show $\sigma; \tau$ is $\downarrow$ -receptive. Suppose $s \in \sigma; \tau$. So there exists $s' \in \sigma \circ \tau$ such that $s' \upharpoonright_{A,-,C} = s$. Note then that we have $s' \cdot \downarrow \in A \circ B \circ C$. By $\downarrow$ -receptivity, it holds that $s' \cdot \downarrow \upharpoonright_{A,B,-} \in \sigma$ and $s' \cdot \downarrow \upharpoonright_{-,B,C} \in \tau$. So it follows that $s' \cdot \downarrow \upharpoonright_{A,-,C} = s \cdot \downarrow \in \sigma; \tau$.

Associative Suppose $\sigma : A \multimap B, \tau : B \multimap C$ and $\rho : C \multimap D$, fix $s \upharpoonright_{A,-,D} \in (\sigma; \tau); \rho$ where $u \in (\sigma; \tau) \circ \rho$. Applying 32.1.1 between the composition of $\sigma; \tau$ and ρ , and then applying the lemma again to decompose its projection to $\sigma; \tau$, we obtain that $u \upharpoonright_{A,-,D}$ can be written as

$$u_1 \cdot \downarrow \cdot \dots \cdot \downarrow \cdot u_{n+1}$$

and $u'_i \in (s_i; t_i); r_i$ where $s_i \in P_{A \multimap B}, t_i \in P_{B \multimap C}$ and $r_i \in P_{C \multimap D}$, with, furthermore

$$s := s_1 \cdot \downarrow \cdot \dots \cdot \downarrow \cdot s_{n+1} \in \sigma$$

$$t := t_1 \cdot \downarrow \cdot \dots \cdot \downarrow \cdot t_{n+1} \in \tau$$

$$r := r_1 \cdot \downarrow \cdot \dots \cdot \downarrow \cdot r_{n+1} \in \rho$$

From concurrent games (Prop. 12.1.1) we already know that $(s_i; t_i); r_i = s_i; (t_i; r_i)$. So there exists $u'_i \in s_i \circ t_i; r_i$ such that $u'_i \upharpoonright_{A,-,D} = u_i$, $u'_i \upharpoonright_{A,B,-} = s_i$ and $u'_i \upharpoonright_{-,B,D} \in t_i; r_i$. Now let's define

$$u'' := u'_1 \cdot \downarrow \cdot \dots \cdot \downarrow \cdot u'_{n+1}$$

Now we have $u'' \upharpoonright_{A,B,-} = s_1 \cdot \downarrow \cdot \dots \cdot \downarrow \cdot s_{n+1}$ and $u'' \upharpoonright_{-,B,D} \in t; r \subseteq \tau; \rho$. So $u = u'' \upharpoonright_{A,-,D} \in \sigma; (\tau; \rho)$.

The other direction is analogous. □

We also prove that composition is monotonic and join-preserving, the main requirement to obtain an enriched semicategory.

Proposition 32.1.2 (cf. Proposition 19.1.1). *For strategies*

$$\sigma : A \multimap B, \quad \tau : B \multimap C$$

the following hold:

1. if $\sigma \subseteq \sigma' : A \multimap B$ and $\tau \subseteq \tau' : B \multimap C$ then $\sigma; \tau \subseteq \sigma'; \tau'$
2. Given a family of strategies $(\sigma_i : A \multimap B)_{i \in I}$ it holds that

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

3. Given a family of strategies $(\tau_i : B \multimap C)_{i \in I}$ it holds that

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

Proof. 1. Suppose $s \upharpoonright_{A, -, C} \in \sigma; \tau$ then

$$s \upharpoonright_{A, B, -} \in \sigma \implies s \upharpoonright_{A, B, -} \in \sigma'$$

$$s \upharpoonright_{-, B, C} \in \tau \implies s \upharpoonright_{-, B, C} \in \tau'$$

Also since $s \in A \circ B \circ C$, $s \upharpoonright_{A, -, C} \in P_{A \multimap C}^{\sharp}$ so $s \upharpoonright_{A, -, C} \in \sigma'; \tau'$.

2. For one direction, since we have $\sigma_i \subseteq \bigcup_{i \in I} \sigma_i$ so

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

hence

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

For the other direction, suppose $s \upharpoonright_{A, -, C} \in \left(\bigcup_{i \in I} \sigma_i \right); \tau$ which means $s \upharpoonright_{A, B, -} \in \left(\bigcup_{i \in I} \sigma_i \right)$ and $s \upharpoonright_{-, B, C} \in \tau$.

so there exists i such that $s \upharpoonright_{A, B, -} \in \sigma_i$, and therefore $s \upharpoonright_{A, -, C} \in \sigma_i; \tau \subseteq \bigcup_{i \in I} (\sigma_i; \tau)$ so that

$$s \upharpoonright_{A, -, C} \in \bigcup_{i \in I} (\sigma_i; \tau)$$

3. For one direction we have $\tau_i \subseteq \bigcup_{i \in I} \tau_i$ so

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

hence

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

For the other direction, suppose $s \upharpoonright_{A,-,C} \in \sigma; \left(\bigcup_{i \in I} \tau_i \right)$ which means $s \upharpoonright_{A,B,-} \in \sigma$ and $s \upharpoonright_{-,B,C} \in \bigcup_{i \in I} \tau_i$.

so there exists i such that $s \upharpoonright_{-,B,C} \in \tau_i$, and therefore $s \upharpoonright_{A,-,C} \in \sigma; \tau_i \subseteq \bigcup_{i \in I} (\sigma; \tau_i)$ so that

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

□

32.2 Volatile Lift and Idempotence

Proposition 32.2.1.

$$\text{vol}(-) : \underline{\text{Safe}} \rightarrow \underline{\text{Crash}}$$

is a semi-functor.

Proof. For given

$$\sigma : A \multimap B \in \underline{\text{Safe}} \qquad \sigma' : B \multimap C \in \underline{\text{Safe}}$$

we want to show

$$\text{vol}(\sigma); \text{vol}(\sigma') = \text{vol}(\sigma; \sigma')$$

For one direction, fix $s \in \text{vol}(\sigma); \text{vol}(\sigma')$. By lemma 32.1.1, we have that s can be decomposed as

$$s := s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

where for each i , $s_i \in \sigma; \sigma'$. It then follows immediately by the definition of $\text{vol}(-)$ that $s \in \text{vol}(\sigma; \sigma')$.

For the other direction, fix $s \in \text{vol}(\sigma; \sigma')$. We have that, by well-formedness, s can be decomposed as

$$s := s_1 \cdot \mathbf{\downarrow} \cdot \dots \cdot \mathbf{\downarrow} \cdot s_{n+1}$$

where for each i $s_i \in \sigma; \sigma'$. So there exists s'_i such that $s'_i \upharpoonright_{A,B,-} \in \sigma$ and $s'_i \upharpoonright_{-,B,C} \in \sigma'$.

Let

$$t := s'_1 \upharpoonright_{A,B,-} \cdot \mathbf{\downarrow} \cdot \dots \cdot \mathbf{\downarrow} \cdot s'_{n+1} \upharpoonright_{A,B,-}$$

$$t' := s'_1 \upharpoonright_{-,B,C} \cdot \mathbf{\downarrow} \cdot \dots \cdot \mathbf{\downarrow} \cdot s'_{n+1} \upharpoonright_{-,B,C}$$

It is immediate from the definition of $\text{vol}(-)$ that $t \in \text{vol}(\sigma)$, $t' \in \text{vol}(\sigma')$, and so it follows that $s \in \text{vol}(\sigma); \text{vol}(\sigma')$.

We also need to show $\text{vol}(\sigma)$ satisfies $\mathbf{\downarrow}$ -receptivity. Suppose $s \in \text{vol}(\sigma)$, $\mathbf{\downarrow} \in M_{A \multimap B}^{\mathbf{\downarrow}}$, $s \cdot \mathbf{\downarrow} \in P_{A \multimap B}^{\mathbf{\downarrow}}$, by definition we know s may be decomposed as

$$s = s_1 \cdot \mathbf{\downarrow} \cdot \dots \cdot \mathbf{\downarrow} \cdot s_{n+1}$$

where for each i , $s_i \in \sigma$ and $r_i \in R$. Since $\epsilon \in \sigma$ so we could set

$$s' := s_1 \cdot \mathbf{\downarrow} \cdot \dots \cdot \mathbf{\downarrow} \cdot s_{n+1} \cdot \mathbf{\downarrow} \cdot \epsilon$$

we obtain by definition that $s' \in \text{vol}(\sigma)$. □

Proposition 32.2.2 (cf. Proposition 19.2.1). *The copycat strategy crashcopy_A is idempotent, i.e.*

$$\text{crashcopy}_A; \text{crashcopy}_A = \text{crashcopy}_A$$

Proof. Note first that it is immediate from the definition of crashcopy_A that

$$\text{crashcopy}_A = \text{vol}(\text{copy}_A)$$

Then, observe that

$$\begin{aligned}
\text{crashcopy}_A; \text{crashcopy}_A &= \text{vol}(\text{copy}_A); \text{vol}(\text{copy}_A) && (\text{Def. of crashcopy}) \\
&= \text{vol}(\text{copy}_A; \text{copy}_A) && (\text{Prop. 32.2.1}) \\
&= \text{vol}(\text{copy}_A) && (\text{Prop. ccopy is idempotent}) \\
&= \text{crashcopy}_A && (\text{Def. of crashcopy})
\end{aligned}$$

□

Proposition 32.2.3. *The restriction*

$$\text{vol}(-) : \mathbf{Safe} \rightarrow \mathbf{Crash}$$

of $\text{vol}(-)$ defines a functor.

Proof. We already argued that

$$\text{crashcopy}_A = \text{vol}(\text{copy}_A)$$

in the proof of Prop. 32.2.2. It remains to show that whenever σ is saturated with respect to copy i.e. $\text{copy}_A; \sigma; \text{copy}_B = \sigma$ then $\text{vol}(\sigma)$ is saturated with respect to crashcopy . For that, just note that:

$$\begin{aligned}
\text{crashcopy}_A; \text{vol}(\sigma); \text{crashcopy}_B &= \text{vol}(\text{copy}_A); \text{vol}(\sigma); \text{vol}(\text{copy}_B) \\
&= \text{vol}(\text{copy}_A; \sigma; \text{copy}_B) \\
&= \text{vol}(\sigma)
\end{aligned}$$

□

32.3 Concrete Saturation for Crash-Aware Games

We start by providing the statement of the main result of this section, which concretely characterizes what a saturated strategy in **Crash** looks like.

Proposition 32.3.1. *A strategy $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ is saturated with respect to crashcopy if and only if it is*

O-receptive: $\forall s \in \sigma. \forall \alpha \in \Upsilon. \forall m \in M_{A \multimap B}^{\alpha:O}. \exists 1 \leq i \leq \|s\|. \text{epo}_i(s) \cdot m \in P_{A \multimap B} \implies$

$$\text{epo}_1(s) \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \text{epo}_i(s) \cdot m \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \text{epo}_{\|s\|}(s) \in \sigma$$

$\rightsquigarrow$ -closed: $\forall s \in \sigma. \forall t \in P_{A \multimap B}. t \rightsquigarrow_{A \multimap B} s \implies t \in \sigma$

P-delaying: $\forall s \in \sigma. \forall m \in M_{A \multimap B}. \lambda_{A \multimap B}(m) = \mathbf{P} \wedge s = p \cdot m \cdot \text{!} \cdot t \Rightarrow p \cdot \text{!} \cdot t \in \sigma$

One might think that it is possible to directly use the proofs of concrete saturation for **Safe** from Oliveira Vale et al. [2024a] across each epoch of a crash-aware strategy to obtain the corresponding concrete saturation result for **Crash**. It turns out that those theorems made a key use of prefix-closure of **Safe** strategies at specific points that make the proofs not translate to our setting, as strategies in **Crash** do not have per-epoch prefix-closure. Because of this, we must reprove some of the results appearing in their appendix, including the Synchronization Lemma.

In the following, we refer to *O*-receptive, *P*-delaying closure of a set of plays $S \subseteq P_A$ with $A \in \underline{\mathbf{Safe}}$ by $\text{dr}(S)$. That is, $\text{dr}(S)$ is the smallest set of plays of P_A such that

$$\forall s \in \text{dr}(S). \forall m \in M_A. \lambda_A(m) = \mathbf{O} \wedge s \cdot m \in P_A \implies s \cdot m \in \text{dr}(S)$$

$$\forall m \in M_A. \lambda_A(m) = \mathbf{P} \wedge \forall s \cdot t \in P_A. s \cdot m \cdot t \in \text{dr}(S) \implies s \cdot t \in \text{dr}(S)$$

The following few re-statements of propositions from Oliveira Vale et al. [2024a] admit essentially the same proofs as might be found there.

Proposition 32.3.2 (Synchronization Lemma). *Let $s = p \cdot \alpha : m \cdot \alpha' : m' \cdot p'$ be a play of*

$A \multimap B \in \mathbf{Safe}$. Let $S = \text{dr}(p \cdot m \cdot m' \cdot p')$. Then,

$$p \cdot m' \cdot m \cdot p' \in \text{copy}_A; S; \text{copy}_B \iff m' \cdot m \rightsquigarrow_{A \multimap B} m \cdot m'$$

Corollary 32.3.1. Let $s \in P_{A \multimap B}$ with $A \multimap B \in \mathbf{Safe}$ and that t is a play such that

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

and moreover

$$t \in \text{copy}_A; \text{dr}(s); \text{copy}_B$$

then,

$$t \rightsquigarrow_{A \multimap B} s$$

Lemma 32.3.1. For every set S of plays of $P_{A \multimap B}$:

$$S \subseteq \text{copy}_A; S; \text{copy}_B$$

The first lemma we need that does require a novel proof is the following.

Lemma 32.3.2. For any O -receptive and P -delaying set S of plays of $P_{A \multimap B}$ with $A \multimap B \in \mathbf{Safe}$ it holds that for all $t \in \text{copy}_A; S; \text{copy}_B$ there exists $s \in S$ such that $t \in \text{copy}_A; S; \text{copy}_B$ and $\forall \alpha \in \Upsilon. \pi_\alpha(t) = \pi_\alpha(s)$.

Proof. Fix $t \in \text{copy}_A; S; \text{copy}_B$ there exists t' such that $t' \upharpoonright_{A,A,-,-} \in \text{copy}_A$, $t' \upharpoonright_{-,A,B,-} \in S$ and $t' \upharpoonright_{-,-,B,B} \in \text{copy}_B$.

Now, notice that for any $\alpha \in \Upsilon$ there are four possibilities for the lengths of $t' \upharpoonright_{A,A,-,-}$ and $t' \upharpoonright_{-,-,B,B}$.

Both are even-length: It is immediate from the definition of copy that $\pi_\alpha(t) = \pi_\alpha(t' \upharpoonright_{-,A,B,-})$.

$t' \upharpoonright_{A,A,-,-}$ is even-length and $t' \upharpoonright_{-,-,B,B}$ is odd-length: Then either $\pi_\alpha(t' \upharpoonright_{-,A,B,-})$ differs from $\pi_\alpha(t)$ by having an extra O -move in the end or by missing a P -move. Either

way, we can find a new $t'' \in S$ that either adds the required O -move or removes the missing P -move by O -receptivity or P -delaying respectively.

$t' \upharpoonright_{A,A,-,-}$ **is odd-length** and $t' \upharpoonright_{-,-,B,B}$ **is even-length**: This case is similar to the previous one.

Both lengths are odd-length: This case is impossible by the switching conditions.

□

Proposition 32.3.3. *An O -receptive and P -delaying set S of plays $P_{A \multimap B}$, for $A \multimap B \in \mathbf{Safe}$ is saturated with respect to copy if and only if*

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

Proof. Suppose S is saturated. It follows that if $s \in S = \text{copy}_A; S; \text{copy}_B$ and $t \rightsquigarrow_{A \multimap B} s$ then there is a sequence of single steps:

$$t = t_0 \rightsquigarrow_{A \multimap B} t_1 \rightsquigarrow_{A \multimap B} \dots \rightsquigarrow_{A \multimap B} t_n = s$$

then by applying the Synchronization Lemma (Prop. 32.3.2) starting with

$$t_{n-1} \rightsquigarrow_{A \multimap B} s$$

to conclude that

$$t_{n-1} \in \text{copy}_A; \text{dr}(s); \text{copy}_B \subseteq \sigma$$

in a finite number of applications we obtain that

$$t = t_0 \in \text{copy}_A; \text{dr}(t_1); \text{copy}_B \subseteq \text{copy}_A; \text{dr}(s); \text{copy}_B \subseteq S$$

as desired.

Note that for every set of plays S that satisfies O -receptivity and P -delaying it holds

that:

$$S = \bigcup_{s \in S} \text{dr}(s)$$

But

$$\text{copy}_A; S; \text{copy}_B = \bigcup_{s \in S} \text{copy}_A; \text{dr}(s); \text{copy}_B$$

by the fact that composition is join-preserving. Hence,

$$t \in \text{copy}_A; S; \text{copy}_B \iff \exists s \in S. t \in \text{copy}_A; \text{dr}(s); \text{copy}_B$$

moreover, by 32.3.2, s can be chosen so that

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

by corollary to the Synchronization Lemma (Prop. 32.3.2) it follows that

$$t \in \text{copy}_A; \text{dr}(s); \text{copy}_B \iff t \rightsquigarrow_{A \multimap B} s$$

And hence

$$t \in \text{copy}_A; S; \text{copy}_B \iff \exists s \in S. t \rightsquigarrow_{A \multimap B} s$$

So, suppose $t \in \text{copy}_A; S; \text{copy}_B$. Then, there is some $s \in \sigma$ such that $t \rightsquigarrow_{A \multimap B} s$ and hence by assumption $t \in \sigma$. Hence,

$$\text{copy}_A; S; \text{copy}_B \subseteq S$$

the reverse containment is exactly lemma 32.3.1 so that it follows that

$$\text{copy}_A; S; \text{copy}_B = S$$

and hence S is saturated. □

Finally, toward the concrete saturation theorem.

Lemma 32.3.3. *For every strategy $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ we have*

$$\sigma \subseteq \text{crashcopy}_A; \sigma; \text{crashcopy}_B$$

Proof. Suppose $s \in \sigma$ then we know s can be decomposed as

$$s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

where $s_i \in P_{A \multimap B}$. Note then that we have $\text{dr}(s_i) \subseteq \text{copy}_A; \text{dr}(s_i); \text{copy}_B$. So there exists $t_i \upharpoonright_{A, -, -, B} \in P_{A \multimap B}$ such that $t_i \upharpoonright_{A, A, -, -} \in \text{copy}_A$, $t_i \upharpoonright_{-, A, B, -} = s_i$ and $t_i \upharpoonright_{-, -, B, B} \in \text{copy}_B$.

So set

$$\begin{aligned} s' &:= t_1 \upharpoonright_{A, A, -, -} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot t_{n+1} \upharpoonright_{A, A, -, -} \\ s'' &:= t_1 \upharpoonright_{-, A, B, -} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot t_{n+1} \upharpoonright_{-, A, B, -} \\ s''' &:= t_1 \upharpoonright_{-, -, B, B} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot t_{n+1} \upharpoonright_{-, -, B, B} \end{aligned}$$

By definition we have $s' \in \text{crashcopy}_A$, $s'' \in \sigma$ and $s''' \in \text{crashcopy}_B$. Hence, $s \in K_{\text{!}} \sigma$ $\square$

Proposition 32.3.4. *A strategy $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ is saturated with respect to crashcopy if and only if it is*

O-receptive: $\forall s \in \sigma. \forall \alpha \in \Upsilon. \forall m \in M_{A \multimap B}^{\alpha:O}. \exists 1 \leq i \leq \|s\|. \text{epo}_i(s) \cdot m \in P_{A \multimap B} \implies$

$$\text{epo}_1(s) \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \text{epo}_i(s) \cdot m \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \text{epo}_{\|s\|}(s) \in \sigma$$

$\rightsquigarrow$ -closed: $\forall s \in \sigma. \forall t \in P_{A \multimap B}. t \rightsquigarrow_{A \multimap B} s \implies t \in \sigma$

P-delaying: $\forall s \in \sigma. \forall m \in M_{A \multimap B}. \lambda_{A \multimap B}(m) = \mathbf{P} \wedge s = p \cdot m \cdot \text{!} \cdot t \Rightarrow p \cdot \text{!} \cdot t \in \sigma$

Proof.

($\implies$): suppose $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ is saturated i.e. $\text{crashcopy}_A; \sigma; \text{crashcopy}_B = \sigma$. Let's

first show that σ is O -receptive. Fix $s \in \sigma$ by definition s can be decomposed as

$$s = s_1 \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot s_{n+1}$$

Note that for $m \in M_{A \multimap B}^{\alpha:O}$, if $s_i \cdot m = \text{epo}_i(s) \cdot m \in P_{A \multimap B}$ then $s_i \cdot m \in P_{A \multimap B}$. Thanks to Oliveira Vale et al. [2024a] we know there exists u_j for each j such that $u_j \upharpoonright_{A,-,-,B} = s_j \cdot m$ if $j = i$, and otherwise $u_j \upharpoonright_{A,-,-,B} = s_j$, and $u_j \upharpoonright_{A,A,-,-} \in \text{copy}_A$, $u_j \upharpoonright_{-,-,B,B} \in \text{copy}_B$ and $u_i \upharpoonright_{-,A,B,-} \in \text{dr}(s_i)$.

Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

Then, note that $u \in \text{crashcopy}_A$, $u' = s$, $u'' \in \text{crashcopy}_B$ and

$$s_1 \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot s_i \cdot m \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot s_{n+1} \in u; u'; u''$$

so that

$$s_1 \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot s_i \cdot m \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot s_{n+1} \in \text{crashcopy}_A; \sigma; \text{crashcopy}_B = \sigma$$

as σ is saturated.

Now let's show σ is $\rightsquigarrow$ -closed. Fix $s \in \sigma$, $t \in P_{A \multimap B}$ and suppose $t \rightsquigarrow_{A \multimap B} s$. More precisely we can decompose s and t as:

$$s = s_1 \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot s_{n+1} \quad t = t_1 \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot t_{n+1}$$

And we have for all i , $t_i \rightsquigarrow_{A \multimap B} s_i$. Now we want to show $t \in \sigma$. Since for all i we have $t_i \rightsquigarrow_{A \multimap B} s_i$, it follows that $t_i \in \text{copy}_A; \text{dr}(s_i); \text{copy}_B$ which means there exists u_i such

that $u_i \upharpoonright_{A,A,-,-} \in \text{copy}_A$, $u_i \upharpoonright_{-,A,B,-} = s_i$, $u_i \upharpoonright_{-,-,B,B} \in \text{copy}_B$ and $u_i \upharpoonright_{A,-,-,B} = t_i$. Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

we have $t \in u; u'; u''$ and $u \in \text{crashcopy}_A$, $u' \in \text{strat}(s)$ and $u'' \in \text{crashcopy}_B$. So $t \in \text{crashcopy}_A; \text{strat}(s); \text{crashcopy}_B \subseteq \text{crashcopy}_A; \sigma; \text{crashcopy}_B = \sigma$.

Finally, we want to show σ is P -delaying. Fix $s \in \sigma$ such that moreover there is an epoch i and a P -move $m \in M^{\alpha:P}$ such that $s_i = p \cdot m$. By definition s can be decomposed as

$$s = s_1 \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot s_{n+1}$$

From 32.3.3 we know there exists u_j for each j such that $u_j \upharpoonright_{A,-,-,B} = p$ if $i = j$ otherwise $u_j \upharpoonright_{A,-,-,B} = s_j$, and $u_j \upharpoonright_{A,A,-,-} \in \text{copy}_A$, $u_j \upharpoonright_{-,-,B,B} \in \text{copy}_B$ and $u_j \upharpoonright_{-,A,B,-} \in \text{dr}(s_j)$

Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

Then, note that $u \in \text{crashcopy}_A$, $u' = s$, $u'' \in \text{crashcopy}_B$ and

$$s_1 \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot p \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot s_{n+1} \in u; u'; u''$$

so that

$$s_1 \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot p \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot \dots \cdot \text{\textcolor{blue}{\(\hookrightarrow\)}} \cdot s_{n+1} \in K_{\text{\textcolor{blue}{\(\hookrightarrow\)}}} \sigma = \sigma$$

as σ is saturated.

($\Leftarrow$): Suppose $\sigma : A \multimap B$ satisfies the O -receptive, P -delaying and $\rightsquigarrow$ -closed conditions.

We want to show $\text{crashcopy}_A; \sigma; \text{crashcopy}_B = \sigma$.

By lemma 32.1.1 we know that for any $u \in \text{crashcopy}_A; \sigma; \text{crashcopy}_B$, there is

$$t = t_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot t_{n+1}$$

such that u can be decomposed as

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

where all $s_i \in \text{copy}_A; \text{dr}(t_i); \text{copy}_B$. Since s satisfies the O -receptive, P -delaying and $\rightsquigarrow$ -closed conditions, which means there exists S_i for each i such that $s_i \in S_i$ for all i , S_i satisfy the O -receptive, P -delaying and $\rightsquigarrow$ -closed. Thanks to 32.3.3 we obtain that $\text{copy}_A; \text{dr}(t_i); \text{copy}_B \subseteq S_i$. So we have $s \in \sigma$ so $\text{crashcopy}_A; \sigma; \text{crashcopy}_B \subseteq \sigma$

The other direction follows from lemma 32.3.3.

□

32.4 Symmetric Monoidal Structure of Crash

Proposition 32.4.1. *For any $\sigma : A \multimap B \in \mathbf{Safe}$ and $\sigma' : A' \multimap B' \in \mathbf{Safe}$ it holds that*

$$\text{vol}(\sigma \otimes \sigma') = \text{vol}(\sigma) \otimes \text{vol}(\sigma')$$

Proof. For one direction, fix $s \in \text{vol}(\sigma) \otimes \text{vol}(\sigma')$. By definition of the tensor s can be decomposed as

$$s := s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

where for each i , $s_i \in \sigma \otimes \sigma'$. It then follows immediately by the definition of $\text{vol}(-)$ that $s \in \text{vol}(\sigma \otimes \sigma')$.

For the other direction, fix $s \in \text{vol}(\sigma \otimes \sigma')$. We have that, by well-formedness, s can be

decomposed as

$$s := s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

where for each i , $s_i \in \sigma \otimes \sigma'$. So $s_i \upharpoonright_{A \multimap B} \in \sigma$ and $s_i \upharpoonright_{A' \multimap B'} \in \sigma'$.

Let

$$t := s_1 \upharpoonright_{A \multimap B} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1} \upharpoonright_{A \multimap B}$$

$$t' := s_1 \upharpoonright_{A' \multimap B'} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1} \upharpoonright_{A' \multimap B'}$$

It is immediate from the definition of $\text{vol}(-)$ that $t \in \text{vol}(\sigma)$, $t' \in \text{vol}(\sigma')$, and so it follows that $s \in \text{vol}(\sigma) \otimes \text{vol}(\sigma')$. $\square$

Lemma 32.4.1.

$$\text{crashcopy}_{A \otimes B} = \text{crashcopy}_A \otimes \text{crashcopy}_B$$

Proof. Note that

$$\begin{aligned} \text{crashcopy}_{A \otimes B} &= \text{vol}(\text{copy}_{A \otimes B}) && \text{(Def. of crashcopy)} \\ &= \text{vol}(\text{copy}_A \otimes \text{copy}_B) && \text{(Symm. Mon. Cat. **Safe**)} \\ &= \text{vol}(\text{copy}_A) \otimes \text{vol}(\text{copy}_B) && \text{(Prop. 32.4.1)} \\ &= \text{crashcopy}_A \otimes \text{crashcopy}_B && \text{(Def. of crashcopy)} \end{aligned}$$

$\square$

Lemma 32.4.2.

$$- \otimes - : \underline{\text{Crash}} \otimes \underline{\text{Crash}} \rightarrow \underline{\text{Crash}}$$

is a bi-semifunctor

Proof. For given

$$\sigma_1 : A_1 \multimap A_2, \quad \sigma_2 : A_2 \multimap A_3$$

$$\tau_1 : B_1 \multimap B_2, \quad \tau_2 : B_2 \multimap B_3$$

and fix $s \in (\sigma_1; \sigma_2) \otimes (\tau_1; \tau_2)$ From lemma 32.1.1 we know s can be decomposed as

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

each s_i satisfies

$$s_i \upharpoonright_{A_1 \multimap A_3} \in \text{dr}(t_i); \text{dr}(t'_i)$$

$$s_i \upharpoonright_{B_1 \multimap B_3} \in \text{dr}(u_i); \text{dr}(u'_i)$$

where

$$t := t_1 \cdot \text{\textcolor{black}{\textcolor{red}{\textbf{\textit{!}}}}} \cdot \dots \cdot \text{\textcolor{black}{\textcolor{red}{\textbf{\textit{!}}}}} \cdot t_{n+1} \in \sigma_1$$

$$t' := t_1 \cdot \cancel{t_2} \cdot \dots \cdot \cancel{t_n} \cdot t'_{n+1} \in \sigma_2$$

$$u := t_1 \cdot \cancel{t_2} \cdot \dots \cdot \cancel{t_n} \cdot u_{n+1} \in \tau_1$$

$$u' := t_1 \cdot \mathbf{z} \cdot \dots \cdot \mathbf{z} \cdot u'_{n+1} \in \tau_2$$

furthermore we have for each s_i

$$\begin{aligned} s_i &\in (\mathrm{dr}(t_i); \mathrm{dr}(t'_i)) \otimes (\mathrm{dr}(u_i); \mathrm{dr}(u'_i)) \\ &= (\mathrm{dr}(t_i) \otimes \mathrm{dr}(u_i)); (\mathrm{dr}(t'_i) \otimes \mathrm{dr}(u'_i)) \end{aligned}$$

so there exists s'_i for each i such that

$$s'_i \upharpoonright_{A_1 \otimes B_1 \multimap A_3 \otimes B_3} = s_i$$

$$s'_i \upharpoonright_{A_1 \otimes B_1 \multimap A_2 \otimes B_2} \in \text{dr}(t_i) \otimes \text{dr}(u_i)$$

$$s'_i \upharpoonright_{A_2 \otimes B_2 \multimap A_3 \otimes B_3} \in \text{dr}(t'_i) \otimes \text{dr}(u'_i)$$

let

$$\begin{aligned} r &:= s'_1 \upharpoonright_{A_1 \otimes B_1 \multimap A_2 \otimes B_2} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s'_{n+1} \upharpoonright_{A_1 \otimes B_1 \multimap A_2 \otimes B_2} \\ r' &:= s'_1 \upharpoonright_{A_2 \otimes B_2 \multimap A_3 \otimes B_3} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s'_{n+1} \upharpoonright_{A_2 \otimes B_2 \multimap A_3 \otimes B_3} \end{aligned}$$

we have $r \in \sigma_1 \otimes \tau_1, r' \in \sigma_2 \otimes \tau_2$ furthermore we know $s \in \mathbf{strat}(r); \mathbf{strat}(r') \subseteq (\sigma_1 \otimes \tau_1); (\sigma_2 \otimes \tau_2)$. The other direction is similar.

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

$$\sigma \otimes \tau \subseteq \sigma' \otimes \tau'$$

Unions are handled in the same way. □

Proposition 32.4.2. *(Crash, $-\otimes-$, 1) defines an enriched symmetric monoidal category.*

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

$$\begin{array}{ccccc} A \otimes (B \otimes C) & \xrightarrow{\alpha_{A,B,C}} & (A \otimes B) \otimes C & \quad \quad & \mathbf{1} \otimes A \xrightarrow{\lambda_A} A & \quad & A \otimes \mathbf{1} \xrightarrow{\rho_A} A \\ \sigma_A \otimes (\sigma_B \otimes \sigma_C) \downarrow & \cong & \downarrow (\sigma_A \otimes \sigma_B) \otimes \sigma_C & \quad & \mathbf{1} \otimes \sigma \downarrow & \cong & \downarrow \sigma & \quad & \sigma \otimes \mathbf{1} \downarrow & \cong & \downarrow \sigma \\ A' \otimes (B' \otimes C') & \xrightarrow{\alpha_{A',B',C'}} & (A' \otimes B') \otimes C' & \quad \quad & \mathbf{1} \otimes B \xrightarrow{\lambda_B} B & \quad & B \otimes \mathbf{1} \xrightarrow{\rho_B} B \end{array}$$

The left and right unital are straightforward. Indeed, they are given by

$$\lambda''_A := \text{vol}(\lambda_A) \qquad \rho''_A := \text{vol}(\rho_A)$$

where λ and ρ are the left and right unitals in **Conc**. It is easy to note that, up to renaming, this makes both unitals the same as **crashcopy**. Because of this, again, up to renaming, we essentially have:

$$\mathbf{1} \otimes \sigma = \{s' \in P_{(\mathbf{1} \multimap \mathbf{1}) \otimes (A \multimap B)} \mid \exists s \in \sigma. (\{\epsilon\} \otimes \text{epo}_1(s)) \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \{(\{\epsilon\} \otimes \text{epo}_{\parallel s \parallel}(s))\} = \sigma$$

which differs from σ only in the shape of the crashes. Therefore, we easily check that:

$$(\mathbf{1} \otimes \sigma); \lambda_B = \sigma; \text{crashcopy}_B = \sigma = \text{crashcopy}_A; \sigma = \lambda_A; \sigma$$

$$(\sigma \otimes \mathbf{1}); \rho_B = \sigma; \text{crashcopy}_B = \sigma = \text{crashcopy}_A; \sigma = \rho_A; \sigma$$

Let β be the braiding in **Safe** and α be the associator in **Safe**. We define the associator and braiding in **Crash** by

$$\beta''_{A,B} := \text{vol}(\beta_{A,B}) \quad \text{and} \quad \alpha''_{A,B,C} := \text{vol}(\alpha_{A,B,C})$$

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

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

where for each $s \in \sigma$

$$p_\alpha(s) := \pi'_\alpha(\text{epo}_1(s)) \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \pi'_\alpha(\text{epo}_{\|s\|}(s))$$

π'_α is the corresponding projection in **Safe** this is the key step to establish that the naturality square commutes. The reverse direction follows similarly.

For the braiding, let's first show its naturality. For a given strategy $\sigma : A \multimap C, \tau : B \multimap D$ we want to show the following diagram commutes:

$$\begin{array}{ccc} A \otimes B & \xrightarrow{\sigma \otimes \tau} & C \otimes D \\ \beta''_{A,B} \downarrow & & \downarrow \beta''_{C,D} \\ B \otimes A & \xrightarrow{\tau \otimes \sigma} & D \otimes C \end{array}$$

Note that since β is a natural isomorphism, we have that for any $s \in \sigma, t \in \tau$:

$$\begin{array}{ccc} A \otimes B & \xrightarrow{\text{dr}(\text{epo}_i(s)) \otimes \text{dr}(\text{epo}_i(t))} & C \otimes D \\ \beta_{A,B} \downarrow & & \downarrow \beta_{C,D} \\ B \otimes A & \xrightarrow{\text{dr}(\text{epo}_i(t)) \otimes \text{dr}(\text{epo}_i(s))} & D \otimes C \end{array}$$

commutes.

So for any given $u \in \text{strat}(s) \otimes \text{strat}(t); \beta''_{C,D}$ we have for each i ,

$$\begin{aligned} \text{epo}_i(u) &\in \text{dr}(\text{epo}_i(s)) \otimes \text{dr}(\text{epo}_i(t)); \beta_{C,D} \\ &= \beta_{A,B}; \text{dr}(\text{epo}_i(t)) \otimes \text{dr}(\text{epo}_i(s)) \end{aligned}$$

And we know for each i , $(\downarrow_A, \downarrow_B) \beta'_{A,B} (\downarrow_B, \downarrow_A)$ and $(\downarrow_C, \downarrow_D) \beta'_{C,D} (\downarrow_D, \downarrow_C)$, so we have

$$s \in \beta''_{A,B}; \tau \otimes \sigma$$

so $\sigma \otimes \tau; \beta''_{C,D} \subseteq \beta''_{A,B}; \tau \otimes \sigma$, other direction is similar. which lets us conclude that

$$\begin{array}{ccc} A \otimes B & \xrightarrow{\sigma \otimes \tau} & C \otimes D \\ \beta''_{A,B} \downarrow & & \downarrow \beta''_{C,D} \\ B \otimes A & \xrightarrow{\tau \otimes \sigma} & D \otimes C \end{array}$$

commutes.

The coherence diagrams follow from functoriality of $K_{\downarrow}$ – together with the fact that $K_{\downarrow}$ – distributes over $- \otimes -$ (Prop. 32.4.1), by noting that all the structural morphisms in **Crash** were defined by lifting the corresponding structural morphisms in **Conc** and **Rel**, which is also why they are isomorphisms. $\square$

32.5 Crash-Aware Linearizability

Proposition 32.5.1 (cf. Proposition 20.0.1).

$$K_{\downarrow} : \underline{\mathbf{Crash}} \rightarrow \mathbf{Crash}$$

is an enriched oplax semifunctor

Proof. Oplax semifunctor: So we want to show for any $\sigma : A \multimap B$ and $\tau : B \multimap C$ we have

$$K_{\downarrow}(\sigma; \tau) \subseteq K_{\downarrow} \sigma; K_{\downarrow} \tau$$

We have the composition is associative, crashcopy is idempotent and $\sigma \subseteq K_{\sharp} \sigma$ so

$$\begin{aligned}
K_{\sharp} (\sigma; \tau) &= \text{crashcopy}_A; \sigma; \tau; \text{crashcopy}_C \\
&\subseteq \text{crashcopy}_A; \sigma; \text{crashcopy}_B; \tau; \text{crashcopy}_C \\
&= \text{crashcopy}_A; \sigma; \text{crashcopy}_B; \text{crashcopy}_B; \tau; \text{crashcopy}_C \\
&= K_{\sharp} \sigma; K_{\sharp} \tau
\end{aligned}$$

Enrichment: Suppose $\sigma \subseteq \sigma'$ then

$$K_{\sharp} \sigma = \text{crashcopy}; \sigma; \text{crashcopy} \subseteq \text{crashcopy}; \sigma'; \text{crashcopy} = K_{\sharp} \sigma'$$

by monotonicity of composition. Similarly

$$K_{\sharp} (\cup_{i \in I} \sigma_i) = \text{crashcopy}; \cup_{i \in I} \sigma_i; \text{crashcopy} = \bigcup_{i \in I} \text{crashcopy}; \sigma_i; \text{crashcopy} = \bigcup_{i \in I} K_{\sharp} \sigma_i$$

□

Proposition 32.5.2 (cf. Proposition 20.1.1). *For $\tau : A \multimap B \in \underline{\text{Crash}}$,*

$$K_{\sharp} \tau = \{s \in P_{A \multimap B} \mid s \text{ is crash-linearizable with respect to } \tau\}$$

Proof. For one direction, let's fix $s \in K_{\sharp} \tau$ then by lemma 32.1.1 there exists $t \in \tau$ such that s and t can be decomposed as

$$s = s_1 \cdot \sharp \cdot \dots \cdot \sharp \cdot s_{n+1} \quad \text{and} \quad t = t_1 \cdot \sharp \cdot \dots \cdot \sharp \cdot t_{n+1}$$

where for each i we have $s_i \in \text{copy}_A; \text{strat}(t_i); \text{copy}_B$ which implies that $s_i \rightsquigarrow t_i$ by Prop. 13.1.1. Since this is true for all i , we have $s \xrightarrow{\sharp} \tau$.

For the other direction, fix s crash-linearizable with respect to τ . Then, there exists t in τ such that s and t can be decomposed as

$$s = s_1 \cdot \sharp \cdot \dots \cdot \sharp \cdot s_{n+1} \quad \text{and} \quad t = t_1 \cdot \sharp \cdot \dots \cdot \sharp \cdot t_{n+1}$$

and for all i we have $s_i \rightsquigarrow t_i$. So we have $s_i \in \text{copy}_A; \text{dr}(t_i); \text{copy}_B$ which implies there exists u_i such that $u_i \upharpoonright_{A,A,-,-} \in \text{copy}_A$, $u_i \upharpoonright_{-,A,B,-} \in \text{dr}(t_i)$, $u_i \upharpoonright_{-,-,B,B} \in \text{copy}_B$ and $u_i \upharpoonright_{A,-,-,B} = s_i$. Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \mathbf{f} \cdot \dots \cdot \mathbf{f} \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

we have $u \in \text{crashcopy}_A$, $u' \in \text{strat}(t) \subseteq \tau$ and $u'' \in \text{crashcopy}_B$ so we have $s \in K_{\mathbf{f}} \tau$. $\square$

Proposition 32.5.3 (cf. Proposition 21.2.1). *For $\nu' : A \in \mathbf{Safe}$ and $\nu : A \in \underline{\mathbf{Safe}}$, if $\nu' \rightsquigarrow \nu$ then $\text{vol}(\nu') \rightsquigarrow_{\mathbf{f}} \text{vol}(\nu)$.*

Proof.

$$\begin{aligned} \text{vol}(\nu') &= (\nu' \cdot \mathbf{f})^* \cdot \nu' && (\text{Def.}) \\ &\subseteq ((K_{\text{copy}} \nu) \cdot \mathbf{f})^* \cdot K_{\text{copy}} \nu && (\text{Lin. equiv. } K_{\text{copy}}) \\ &= ((\nu; \text{copy}_A) \cdot \mathbf{f})^* \cdot (\nu; \text{copy}_A) && (\text{Def. of } K_{\text{copy}}) \\ &= ((\nu \cdot \mathbf{f})^* \cdot \nu); ((\text{copy}_A \cdot \mathbf{f})^* \cdot \text{copy}_A) && (\text{Def. } -; - \text{ in } \underline{\mathbf{Crash}}) \\ &= \text{vol}(\nu); \text{crashcopy}_A = K_{\mathbf{f}} \text{vol}(\nu) && (\text{Def. vol}(-)) \end{aligned}$$

$\square$

Now, we move to locality and observational refinement. According to §6 of Oliveira Vale et al. [2024a], it is enough to prove the following lemma.

Lemma 32.5.1 (cf. Proposition 20.1.2).

- For any $\sigma : \mathbf{1} \multimap A \in \underline{\mathbf{Crash}}$ it holds that $\text{crashcopy}_1; \sigma = \sigma$.
- For $\sigma, \tau : A \multimap B$ and $\sigma', \tau' : A' \multimap B'$ we have

$$\sigma \otimes \sigma' \subseteq \tau \otimes \tau' \implies \sigma \subseteq \sigma' \wedge \tau \subseteq \tau'$$

Proof.

- For one direction, fix $s \in \text{crashcopy}_1; \sigma$, by 32.1.1 we know s can be decomposed as

$$s = s_1 \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot s_{n+1}$$

and $s_i \in t_i; u_i$ such that

$$t := t_1 \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot t_{n+1} \in \text{crashcopy}_1$$

$$u := u_1 \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot u_{n+1} \in \sigma$$

By definition of `crashcopy` we know $t_i \in \text{copy}_1$. Therefore, $t_i = \epsilon$ for all i so $t_i; u_i = u_i$, and therefore $s_i = u_i$. It then follows that $s \in \sigma$.

For the other direction, fix $s \in \sigma$, by definition we know s can be decomposed as

$$s = s_1 \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot s_{n+1}$$

Now let

$$t := \epsilon \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \epsilon$$

it is easy to check $t \in \text{crashcopy}_1$. But then, $s \in t; s$, so $s \in \text{crashcopy}_1; \sigma$.

- For given $\sigma, \tau : A \multimap B$ and $\sigma', \tau' : A' \multimap B'$ suppose $\sigma \otimes \sigma' \subseteq \tau \otimes \tau'$. We wish to show that $\sigma \subseteq \tau$. Fix $s \in \sigma$, by well-formedness s can be decomposed as

$$s = s_1 \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot s_{n+1}$$

By `\text{\textcolor{blue}{\text{\textbf{!}}}}`-receptivity it follows that

$$t := \epsilon \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \epsilon \in \sigma'$$

But $s \otimes t = \{s\}$ up to re-indexing, by definition. So, by monotonicity, $s \in s \otimes t \subseteq \tau \otimes \tau'$.

But then it follows that $s \in \tau$, so $\sigma \subseteq \tau$.

The proof of $\sigma' \subseteq \tau'$ is similar.

□

Since by now we have proven all the requirements on the embeddable subcategory of the Karoubi envelope that we have constructed, it follows that

Proposition 32.5.4 (cf. Corollaries 20.1.2 and 20.1.3). *Locality and the equivalence with observational refinement both hold for crash-aware linearizability.*

Proposition 32.5.5 (cf. Proposition 21.1.1).

- If $s' \xrightarrow{\text{t}} s$ and $s \xrightarrow{\text{tr}} t$ then $s' \xrightarrow{\text{tr}} t$
- If $s' \xrightarrow{\text{t}} s$ and $s \xrightarrow{\text{dur}} t$ then $s' \xrightarrow{\text{dur}} t$

Proof. For the first bullet: by Prop. 32.5.2, $s \xrightarrow{\text{tr}} t$ witnesses t' with $s \in K_{\sharp} t'$ and $\text{ops}(t') = t$. By enrichment and idempotence of $K_{\text{crashcopy}}$:

$$s' \in K_{\sharp} s \subseteq K_{\sharp} (K_{\sharp} t') = K_{\sharp} t'$$

giving $s' \xrightarrow{\text{t}} t'$ and hence $s' \xrightarrow{\text{tr}} t$.

For the second bullet, since crash-aware linearizability preserves the durability condition (linearizations keep the agents within each epoch), we can safely ignore the intersection with durable plays. By Prop. 32.6.3 and Prop. 32.6.4:

$$\{s'\} \subseteq K_{\sharp} s \subseteq K_{\sharp} (K \{t\})^{\sharp} = \text{copy}_A^{\sharp}; (\text{copy}_A; \{t\})^{\sharp} \subseteq \text{copy}_A; \text{copy}_A; \{t\}^{\sharp} = (K \{t\})^{\sharp} = \text{dur}(t)$$

and the claim follows. □

32.6 Crash Abstraction

We first prove a basic property about $-^b$.

Lemma 32.6.1. *For $s \in P_{A \multimap B}$,*

- if $\text{ops}(s \upharpoonright_{-,B}) \in P_B$ then $\text{ops}(s \upharpoonright_{A,-}) \in P_A$;
- $\text{ops}(s) \in P_{A \multimap B}$ if and only if $\text{ops}(s \upharpoonright_{A,-}) \in P_A$ and $\text{ops}(s \upharpoonright_{-,B}) \in P_B$.

Proof.

- Let's prove this result by contradiction. Suppose there exists $s \in P_{A \multimap B}$ such that $\text{ops}(s \upharpoonright_{-,B}) \in P_B$ and $\text{ops}(s \upharpoonright_{A,-}) \notin P_A$. First of all, by definition s can be decomposed as

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

and for each i , $s_i \in P_{A \multimap B}$

Since $\text{ops}(s \upharpoonright_{A,-}) \notin P_A$, there exists a $\alpha \in \Upsilon$ such that $\pi_\alpha(s \upharpoonright_{A,-})$ is not a valid sequential play. And, since for each i , $\pi_\alpha(s_i \upharpoonright_A)$ is a valid sequential play, there must exists i such that there is a pending O -move in $\pi_\alpha(s_i \upharpoonright_{A,-})$ and there exists $j > i$ such that $\pi_\alpha(s_j \upharpoonright_{A,-})$ is non-empty.

Since $\pi_\alpha(s_i)$ is also a valid sequential play, there must be a pending O -move also in $\pi_\alpha(s_i \upharpoonright_{-,B})$ by the switching condition, and $\pi_\alpha(s_j \upharpoonright_{-,B})$ is also non-empty, since plays are O -starting. But then, this means $\pi_\alpha(s \upharpoonright_{-,B})$ is not a valid sequential play. So $\text{ops}(s \upharpoonright_{-,B}) \notin P_B$ which is a contradiction.

- The forward direction is easy to see by definition, while the backward direction follows immediately from the first bullet point.

□

We now address the functoriality of $-^b$.

Proposition 32.6.1 (cf. Proposition 21.3.1). $-^b : \mathbf{Crash} \rightarrow \mathbf{Safe}$ defines an enriched oplax semifunctor.

Proof. Suppose $s \in (\sigma; \tau)^b$. Then, there is $s' \in \sigma \circ \tau$ such that $s = \text{ops}(s' \upharpoonright_{A,-,C})$ and in particular $\text{ops}(s' \upharpoonright_{A,-,C}) \in P_{A \multimap B}$. Note then that by Prop. 32.6.1 it follows that $\text{ops}(s' \upharpoonright_{-,B,C} \upharpoonright_{-,C}) = \text{ops}(s' \upharpoonright_{A,-,C} \upharpoonright_{-,C}) \in P_C$, so that by the same proposition it follows that $\text{ops}(s' \upharpoonright_{-,B,C} \upharpoonright_{B,-}) \in P_B$ and therefore $\text{ops}(s' \upharpoonright_{-,B,C}) \in P_{B \multimap C}$. At this point we have

$s' \upharpoonright_{A,B,-} \upharpoonright_{-,B} = s' \upharpoonright_{-,B,C} \upharpoonright_{B,-} \in P_B$ so that analogous reasoning gives that $s' \upharpoonright_{A,B,-} \in P_{A \multimap B}$. But we also have $s' \upharpoonright_{A,B,-} \in \sigma$ and $s' \upharpoonright_{-,B,C} \in \tau$, so that we have shown that $\text{ops}(s' \upharpoonright_{A,B,-}) \in \sigma^b$ and $\text{ops}(s' \upharpoonright_{-,B,C}) \in \tau^b$. We then obtain that $\text{ops}(s') \upharpoonright_{A,B,-} = \text{ops}(s' \upharpoonright_{A,B,-}) \in \sigma^b$ and $\text{ops}(s') \upharpoonright_{-,B,C} = \text{ops}(s' \upharpoonright_{-,B,C}) \in \tau^b$. Hence, $\text{ops}(s') \in \sigma^b \circ \tau^b$ and therefore $s = \text{ops}(s' \upharpoonright_{A,-,C}) = \text{ops}(s') \upharpoonright_{A,-,C} \in \sigma^b; \tau^b$.

We move on to the enrichment. Suppose $\sigma \subseteq \sigma'$, fix $s \in \sigma^b$, by definition there exists $s' \in \sigma$ such that $\text{ops}(s') = s$. Since $\sigma \subseteq \sigma'$ so $s' \in \sigma'$ so $s \in (\sigma')^b$. So we have $\sigma^b \subseteq (\sigma')^b$.

Given a family of strategies $(\sigma_i : A \multimap B)_{i \in I}$. For one direction, fix $s \in (\cup_{i \in I} \sigma_i)^b$ so there exists $s' \in \cup_{i \in I} \sigma_i$ such that $\text{ops}(s') = s$. So there exists $i \in I$ such that $s' \in \sigma_i$ and $\text{ops}(s') = s \in P_A$ which means $s \in \sigma_i^b$. So $s \in \cup_{i \in I} \sigma_i^b$. For the other direction, fix $s \in \cup_{i \in I} \sigma_i^b$, we know there exists $i \in I$ such that $s' \in \sigma_i$ and $s' \upharpoonright = s$. So $s' \in \cup_{i \in I} \sigma_i$ and $\text{ops}(s') = s \in P_A$, so $s \in (\cup_{i \in I} \sigma_i)^b$ $\square$

Proposition 32.6.2 (cf. Proposition 21.3.1). $\text{crashcopy}_A^b = \text{copy}_A$

Proof. By definition it is easy to see $\text{copy}_A \subseteq \text{crashcopy}_A^b$.

For the other direction, fix $s \in \text{crashcopy}_A^b$, by definition, s can be decomposed as

$$s = s_1 \cdot s_2 \cdot \dots \cdot s_{n+1}$$

where for each i , $s_i \in \text{copy}_A$.

By the definition of seqcopy we know for any two plays $t, t' \in \text{seqcopy}$, if $t \cdot t'$ is a valid play, then $t \cdot t' \in \text{seqcopy}$. So for any $\alpha \in \Upsilon$ we get that $\pi_\alpha(s)$ is in seqcopy_A so $s \in \text{copy}_A$ $\square$

Corollary 32.6.1. For every $\sigma : A \multimap B \in \underline{\text{Crash}}$:

$$(K_{\sharp} \sigma)^b \subseteq K_{\text{copy}} \sigma^b$$

We now move to the functoriality like properties of re-crash operation $-^\sharp$.

Proposition 32.6.3 (cf. Proposition 21.3.2). For strategies $\sigma : A \multimap B$, and $\tau : B \multimap C$, the following hold:

- if $\sigma \subseteq \sigma'$, for $\sigma' : A \multimap B$, then $\sigma^\sharp \subseteq (\sigma')^\sharp$

- Given a family $(\sigma_i : A \multimap B)_{i \in I}$ it holds that $(\cup_{i \in I} \sigma_i)^\# = \cup_{i \in I} \sigma_i^\#$
- $\sigma^\#; \tau^\# \subseteq (\sigma; \tau)^\#$

Proof.

- Suppose $\sigma \subseteq \sigma'$, fix $s \in \sigma^\#$ by definition we know $\text{ops}(s) \in \sigma$. Since $\sigma \subseteq \sigma'$, so $\text{ops}(s) \in \sigma'$ which implies $s \in (\sigma')^\#$. So $\sigma^\# \subseteq (\sigma')^\#$
- For a given family $(\sigma_i)_{i \in I}$. For one direction, fix $s \in (\cup_{i \in I} \sigma_i)^\#$ we know $\text{ops}(s) \in \cup_{i \in I} \sigma_i$ which means there exists $i \in I$ such that $\text{ops}(s) \in \sigma_i$. By definition this means $s \in \sigma_i^\#$, so $s \in \cup_{i \in I} \sigma_i^\#$.

For the other direction, let fix $s \in \cup_{i \in I} \sigma_i^\#$, by definition, there exists $i \in I$ such that $\text{ops}(s) \in \sigma_i \subseteq \cup_{i \in I} \sigma_i$. So $s \in \cup_{i \in I} \sigma_i^\#$

- fix $s \in \sigma^\#; \tau^\#$, by definition there exists $s' \in \sigma^\#; \tau^\#$ such that $s' \upharpoonright_{A, -, C} = s$, $s' \upharpoonright_{A, B, -} \in \sigma^\#$ and $s' \upharpoonright_{-, B, C} \in \tau^\#$.

Set $t := \text{ops}(s' \upharpoonright_{A, B, -})$, $u := \text{ops}(s' \upharpoonright_{-, B, C})$, it is easy to check

$$\text{ops}(s) = \text{ops}(s' \upharpoonright_{A, -, C}) \in \text{strat}(t); \text{strat}(u) \subseteq \sigma; \tau$$

so $s \in (\sigma; \tau)^\#$. Well-formedness of $\text{ops}(s)$ is guaranteed by well-formedness of $\text{ops}(s' \upharpoonright_{A, -, -})$, $\text{ops}(s' \upharpoonright_{-, -, C})$ by Prop. 32.6.1.

□

Proposition 32.6.4 (cf. Proposition 21.3.2). *For all $A \in \mathbf{Crash}$,*

$$\text{copy}_A^\# \subseteq \text{crashcopy}_A$$

Proof. Fix $s \in \text{copy}_A^\#$, by definition we know there exists $s' \in \text{copy}_A$ such that s can be decomposed as

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

where $s_1 \cdot \dots \cdot s_{n+1} = s'$ and for each i , s_i is a play in P_A . Furthermore we know for each $i \neq n+1$, $\alpha \in \Upsilon$, there is no pending O moves in $\pi_\alpha(s_i)$.

Now we want to show for all i , $\alpha \in \Upsilon$, $p \sqsubseteq_{\text{even}} \pi_\alpha(s_i)$ we have $p \upharpoonright_{A,-} = p \upharpoonright_{-,A}$ by contradiction. Suppose i is the smallest i such that there exists $\alpha \in \Upsilon$, $p \sqsubseteq_{\text{even}} \pi_\alpha(s_i)$ such that $p \upharpoonright_{A,-} \neq p \upharpoonright_{-,A}$.

Since $s'' := s_1 \cdot \dots \cdot s_i \sqsubseteq s'$ so we know $s'' \in \text{copy}_A$ which means for every $q \sqsubseteq_{\text{even}} \pi_\alpha(s'')$ we have $q \upharpoonright_{A,-} = q \upharpoonright_{-,A}$. Since there is no pending O moves in $s_1, \dots, s_{i-1}$ so $p' := \pi_\alpha(s_1 \cdot \dots \cdot s_{i-1})$ has to be an even prefix of $\pi_\alpha(s'')$ so is $p' \cdot p$. But it is easy to see $p' \cdot p \upharpoonright_{A,-} \neq p' \cdot p \upharpoonright_{-,A}$ which is a contradiction. So we know for each i , $s_i \in \text{copy}_A$. By definition, $s \in \text{crashcopy}_A$. $\square$

We also take the opportunity to show how $-^\sharp$ interacts with horizontal composition.

Lemma 32.6.2. *For any $\sigma : A_1 \multimap B_1$, $\tau : A_2 \multimap B_2$,*

$$(\sigma \otimes \tau)^\sharp = \sigma^\sharp \otimes \tau^\sharp$$

Proof. For one direction fix $s \in (\sigma \otimes \tau)^\sharp$, we know s can be decomposed as

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

for each i we know there exists t_i, u_i such that $s_i \in t_i \otimes u_i$ such that $u_1 \cdot \dots \cdot u_{n+1} \in \sigma$ and $t_1 \cdot \dots \cdot t_{n+1} \in \tau$

By definition we can see

$$t := t_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot t_{n+1} \in \sigma^\sharp$$

$$u := u_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot u_{n+1} \in \tau^\sharp$$

so we know $s \in \text{strat}(t) \otimes \text{strat}(u) \subseteq \sigma^\sharp \otimes \tau^\sharp$

For the other direction let's fix $s \in \sigma^\sharp \otimes \tau^\sharp$ then there exists $t \in \sigma^\sharp$, $u \in \tau^\sharp$ such that $s \in \text{strat}(t) \otimes \text{strat}(u)$. Let $s' = \text{ops}(t) \otimes \text{ops}(u)$ by definition we know $s' \in \sigma \otimes \tau$ and $\text{ops}(s) = s'$ so $s \in (\sigma \otimes \tau)^\sharp$ $\square$

32.7 Strict Linearizability

We start by showing some auxiliary lemmas.

Lemma 32.7.1. *For any given $\sigma, \tau \in \mathbf{Safe}$ we have*

$$\sigma^\sharp; \text{crashcopy}_B; \tau^\sharp = \sigma^\sharp; \text{copy}_B^\sharp; \tau^\sharp$$

Proof. One direction follows from an already proved lemma.

$$\sigma^\sharp; \text{copy}_B^\sharp; \tau^\sharp \subseteq \sigma^\sharp; \text{crashcopy}_B; \tau^\sharp$$

Now let's try to prove the other direction by contradiction. Suppose $s \in \sigma^\sharp; \text{crashcopy}_B; \tau^\sharp$, assume $s \notin \sigma^\sharp; \text{copy}_B^\sharp; \tau^\sharp$. So there exists s' such that $s' \upharpoonright_{A,B,-,-} \in \sigma^\sharp, s' \upharpoonright_{-,B,B,-} \in \text{crashcopy}_B$ but $s' \upharpoonright_{-,B,B,-} \notin \text{copy}_B^\sharp, s' \upharpoonright_{-,-,B,C} \in \tau^\sharp$ and $s' \upharpoonright_{A,-,-,C} = s$

Since we already know $(\text{crashcopy})^b = \text{copy}$, it follows that $\text{copy}_B^\sharp = (\text{crashcopy}^b)^\sharp$. So we get that $s' \upharpoonright_{-,B,B,-} \notin (\text{crashcopy}^b)^\sharp$. But this means that at least one of $\text{ops}(s' \upharpoonright_{-,B,-,-})$ and $\text{ops}(s' \upharpoonright_{-,-,B,-})$ is not well-formed. But we know both $s' \upharpoonright_{A,B,-,-}$ and $s' \upharpoonright_{-,-,B,C}$ are well-formed which is a contradiction. $\square$

Lemma 32.7.2. *For any given $\sigma, \tau \in \mathbf{Safe}$ we have*

$$\sigma^\sharp; \text{crashcopy}_B; \tau^\sharp = \sigma^\sharp; \tau^\sharp$$

Proof. For one direction, note that $\sigma^\sharp; \tau^\sharp \subseteq \sigma^\sharp; \text{crashcopy}_B; \tau^\sharp$ follows from what we showed about K_4 –.

For the other direction notice that

$$\begin{aligned} \sigma^\sharp; \text{crashcopy}_B; \tau^\sharp &= \sigma^\sharp; \text{copy}_B^\sharp; \tau^\sharp \\ &\subseteq \sigma^\sharp; (\text{copy}_B; \tau)^\sharp \\ &= \sigma^\sharp; \tau^\sharp \end{aligned}$$

$\square$

Proposition 32.7.1.

$$\text{crashcopy}_A; \text{copy}_A^\# = \text{copy}_A^\#$$

Proof. One direction $\text{copy}_A^\# \subseteq \text{crashcopy}_A; \text{copy}_A^\#$ follows readily from what we showed about K_4^- .

The other direction, let's fix $s \in \text{crashcopy}_A; \text{copy}_A^\#$, then there exists s' such that $s' \upharpoonright_{A,A,-} \in \text{crashcopy}_A$, $s' \upharpoonright_{-,A,A} \in \text{copy}_A^\#$, and $s' \upharpoonright_{A,-,A} = s$.

Let's first show that $\pi_\alpha(s)$ is well-formed by contradiction. By definition we know s can be decomposed as

$$s = s_1 \cdot \bullet \cdot \dots \cdot \bullet \cdot s_{n+1}$$

If s is not well-formed, we know there exists $\alpha \in \Upsilon$ such that there exists l and $m > l$ such that $\pi_\alpha(s_l \cdot s_m)$ is not well-formed and $\pi_\alpha(s_m) \neq \epsilon$. Let m be the first m that satisfies the property.

This means either $\pi_\alpha(s_l \cdot s_m) \upharpoonright_{A,-}$ or $\pi_\alpha(s_l \cdot s_m) \upharpoonright_{-,A}$ is not well-formed. But we know $\pi_\alpha(s_l \cdot s_m) \upharpoonright_{-,A} = \pi_\alpha(s'_l \cdot s'_m) \upharpoonright_{-,A}$ which is well-formed by definition. So the only possibility left is $\pi_\alpha(s_l \cdot s_m) \upharpoonright_{A,-}$ is not well-formed.

This means there is a pending O move in $\pi_\alpha(s_l) \upharpoonright_{A,-}$ which will become a P move in s_l and this will force $\pi_\alpha(s_l) \upharpoonright_{-,A}$ has a pending O move. But we already know $\pi_\alpha(s_m) \upharpoonright_{-,A}$ is non-empty, so this is a contradiction.

Now we want to show $\text{ops}(s) \in \text{copy}_A$. We only need to show for each $\alpha \in \Upsilon$ we have $\pi_\alpha(s) \in \text{seqcopy}_A$. Let's also show this by contradiction, suppose there exists $\alpha \in \Upsilon$ and $p \sqsubseteq_{\text{even}} \pi_\alpha(s)$ such that $\pi_\alpha(p) \upharpoonright_{A,-} \neq \pi_\alpha(p) \upharpoonright_{-,A}$

So there exists $p' \sqsubseteq \pi_\alpha(s')$ such that $p' \upharpoonright_{A,-,A} = p$. Since $p' \upharpoonright_{A,A,-}$ is in seqcopy_A so $p' \upharpoonright_{A,-,-} = p' \upharpoonright_{-,A,-}$. Same we can get $p' \upharpoonright_{-,A,-} = p' \upharpoonright_{-,A,-}$. So we know $p' \upharpoonright_{A,-,-} = p' \upharpoonright_{-,A,-}$ which means $p \upharpoonright_{A,-} = p \upharpoonright_{-,A}$ which is a contradiction. $\square$

With all these lemmas proved, we are ready to show observational refinement.

Proposition 32.7.2 (cf. Proposition 21.4.1). *Let $\sigma : A \multimap B \in \mathbf{Safe}$ and $\tau : B \multimap C \in \mathbf{Safe}$, then*

$$\text{str}(\sigma); \text{str}(\tau) \subseteq \text{str}(\sigma; \tau)$$

Proof.

$$\begin{aligned}
\text{str}(\sigma); \text{str}(\tau) &= K_{\sharp} \sigma^{\sharp}; K_{\sharp} \tau^{\sharp} \\
&= \text{crashcopy}_A; \sigma^{\sharp}; \text{crashcopy}_B; \tau^{\sharp}; \text{crashcopy}_C \\
&= \text{crashcopy}_A; \sigma^{\sharp}; \tau^{\sharp}; \text{crashcopy}_C \\
&\subseteq \text{crashcopy}_A; (\sigma; \tau)^{\sharp}; \text{crashcopy}_C \\
&\subseteq K_{\sharp} (\sigma; \tau)^{\sharp} \\
&= \text{str}(\sigma; \tau)
\end{aligned}$$

□

Proposition 32.7.3 (cf. Proposition 21.4.2). *If $\nu'_A \subseteq \text{str}(\nu_A)$ then, for all $\sigma : A \multimap B \in \mathbf{Safe}$ that implements an object linearizable to $\nu_B : B$ using ν_A , i.e.*

$$\nu_A; \sigma \subseteq \nu_B$$

It holds that,

$$\nu'_A; \text{str}(\sigma) \subseteq \text{str}(\nu_B)$$

Proof. For the forward direction, we start by noting that since

$$\nu_A; \sigma \subseteq \nu_B$$

First, note that

$$\nu'_A \subseteq \text{str}(\nu_A) \subseteq \text{str}(K_{\text{copy}} \nu_A)$$

By the observational refinement property on **Conc** it follows that

$$K_{\text{copy}} \nu_A; \sigma \subseteq \nu_B$$

Then, by Prop. 32.7.2

$$\nu'_A; \text{str}(\sigma) \subseteq \text{str}(K_{\text{copy}} \nu_A); \text{str}(\sigma) = \text{str}(K_{\text{copy}} \nu_A; \sigma) \subseteq \text{str}(\nu_B)$$

□

For locality, note first that:

Proposition 32.7.4. *For $\sigma : A_1 \multimap B_1$ and $\tau : A_2 \multimap B_2$,*

$$\text{str}(\sigma \otimes \tau) = \text{str}(\sigma) \otimes \text{str}(\tau)$$

Proof.

$$\text{str}(\sigma \otimes \tau) = K_{\sharp} (\sigma \otimes \tau)^{\sharp} = K_{\sharp} (\sigma^{\sharp} \otimes \tau^{\sharp}) = K_{\sharp} \sigma^{\sharp} \otimes K_{\sharp} \tau^{\sharp} = \text{str}(\sigma) \otimes \text{str}(\tau)$$

□

Proposition 32.7.5 (Locality; cf. Proposition 21.4.3). *For $\nu'_A : A, \nu'_B : B \in \mathbf{Crash}$ and $\nu_A : A, \nu_B : B \in \mathbf{Safe}$:*

$$\nu'_A \subseteq \text{str}(\nu_A) \text{ and } \nu'_B \subseteq \text{str}(\nu_B) \text{ if and only if } \nu'_A \otimes \nu'_B \subseteq \text{str}(\nu_A \otimes \nu_B)$$

Proof. For the forward direction we have that by monotonicity:

$$\nu'_A \otimes \nu'_B \subseteq \text{str}(\nu_A) \otimes \text{str}(\nu_B) = \text{str}(\nu_A \otimes \nu_B)$$

For the reverse direction, first note that

$$\nu'_A \otimes \nu'_B \subseteq \text{str}(\nu_A \otimes \nu_B) = \text{str}(\nu_A) \otimes \text{str}(\nu_B)$$

since we have shown the tensor is an order-isomorphism the result follows. □

32.8 Durability

Lemma 32.8.1. *Let $s \in P_{A \multimap B}$ for $A, B \in \mathbf{Crash}$.*

- If $s \upharpoonright_{-,B}$ is durable then $s \upharpoonright_{A,-}$ is durable.
- s is durable if and only if $s \upharpoonright_{A,-}$ and $s \upharpoonright_{-,B}$ are both durable.

Proof.

- Let's prove this by contradiction. Suppose $s \upharpoonright_{-,B}$ is durable, but $s \upharpoonright_{A,-}$ is not. Then there exists $i \neq j$ such that $\exists \alpha \in \Upsilon. \alpha \in \Upsilon(\text{epo}_i(s \upharpoonright_{A,-})) \cap \Upsilon(\text{epo}_j(s \upharpoonright_{A,-}))$. Since both $\text{epo}_i(s), \text{epo}_j(s) \in P_{A \multimap B}$, it follows that $\pi_\alpha(\text{epo}_i(s)), \pi_\alpha(\text{epo}_j(s)) \in P_{A \multimap B}$. So by the switching condition, and since $\pi_\alpha(\text{epo}_i(s) \upharpoonright_{A,-})$ is non-empty we have that $\pi_\alpha(\text{epo}_i(s) \upharpoonright_{-,B})$ is also non-empty. The same applies for j . Again by the switching condition, $\alpha \in \Upsilon(\text{epo}_i(s \upharpoonright_{-,B})) \cap \Upsilon(\text{epo}_j(s \upharpoonright_{-,B}))$. But we know $s \upharpoonright_{-,B}$ is durable which is a contradiction.
- ($\Rightarrow$) It is easy to see by definition
- ($\Leftarrow$) Suppose both $s \upharpoonright_{A,-}$ and $s \upharpoonright_{-,B}$ are durable. Let's prove s is also durable by contradiction. Suppose s is not, so there exists $i \neq j$ such that $\exists \alpha \in \Upsilon. \alpha \in \Upsilon(\text{epo}_i(s)) \cap \Upsilon(\text{epo}_j(s))$. So $\pi_\alpha(\text{epo}_i(s) \upharpoonright_{-,B})$ and $\pi_\alpha(\text{epo}_j(s) \upharpoonright_{-,B})$ both can't be empty play. So $\alpha \in \Upsilon(\text{epo}_i(s) \upharpoonright_{-,B}) \cap \Upsilon(\text{epo}_j(s) \upharpoonright_{-,B})$ by the switching condition. But we know $s \upharpoonright_{-,B}$ is durable which is a contradiction.

□

Proposition 32.8.1 (cf. Proposition 21.5.1). *Durable strategies compose.*

Proof. Suppose $\sigma : A \multimap B, \tau : B \multimap C$, are both durable.

We already have that $\sigma; \tau : A \multimap C$ is well-defined, it remains to show that it is also durable. By definition of composition we know for any $s \in \sigma; \tau$ there exists $t \in \sigma, u \in \tau$ such that $s \upharpoonright_{A,-} = t \upharpoonright_{A,-}$ and $s \upharpoonright_{-,C} = u \upharpoonright_{-,C}$.

Since both σ, τ are durable, by proposition 32.8.1 $t \upharpoonright_{A,-}$ and $u \upharpoonright_{-,B}$ are both durable. So thanks to proposition 32.8.1 again we know s is also durable. □

This means that the restriction of **Crash** to durable strategies, **Dur**, defines a semicategory.

Lemma 32.8.2. *For sets of plays*

$$S \subseteq P_{A \multimap B} \quad T \subseteq P_{B \multimap C}$$

and

$$(S \cap P_{A \multimap B}^{\text{dur}}); (T \cap P_{B \multimap C}^{\text{dur}}) = (S; T) \cap P_{A \multimap C}^{\text{dur}}$$

Proof. For one direction, fix $s \in (S \cap P_{A \multimap B}^{\text{dur}}); (T \cap P_{B \multimap C}^{\text{dur}})$, we know there exists s' such that $s' \upharpoonright_{A,B,-} \in S \cap P_{A \multimap B}^{\text{dur}}$, $s' \upharpoonright_{-,B,C} \in T \cap P_{B \multimap C}^{\text{dur}}$ and $s' \upharpoonright_{A,-,C} = s$.

In particular we know $s' \upharpoonright_{A,B,-} \in S$ and $s' \upharpoonright_{-,B,C} \in T$ which implies $s \in S; T$. By applying proposition 32.8.1 we know $s \upharpoonright_{A,-} = s' \upharpoonright_{A,-,-}$, $s \upharpoonright_{-,C} = s' \upharpoonright_{-, -, C}$ are both durable. So also by proposition 32.8.1 we know s is durable. So $s \in (S; T) \cap P_{A \multimap C}^{\text{dur}}$.

For other direction, fix $s \in (S; T) \cap P_{A \multimap C}^{\text{dur}}$. We know there exists s' such that $s' \upharpoonright_{A,B,-} \in S$, $s' \upharpoonright_{-,B,C} \in T$, $s' \upharpoonright_{A,-,C} = s$. By proposition 32.8.1 we have that $s \upharpoonright_{-,C}$ is durable so is $s' \upharpoonright_{-, -, C}$. By proposition 32.8.1 we know $s' \upharpoonright_{-,B,-}$ has to be durable. Then by applying proposition 32.8.1 again we get $s' \upharpoonright_{A,-,-}$ is durable.

So we know $s' \upharpoonright_{A,B,-} \in S \cap P_{A \multimap B}^{\text{dur}}$ and $s' \upharpoonright_{-,B,C} \in T \cap P_{B \multimap C}^{\text{dur}}$. □

Corollary 32.8.1. *The assignment:*

$$A \longmapsto A \quad \sigma : A \multimap B \longmapsto \sigma \cap P_{A \multimap B}^{\text{dur}}$$

defines a semifunctor from Crash to Dur.

Corollary 32.8.2 (cf. Proposition 21.5.2). *durcopy is idempotent.*

Corollary 32.8.3. *For any given (lax, oplax, semi) functor $F : \mathbf{C} \rightarrow \mathbf{Crash}$, we have*

$$F(-) \cap P_{F X \multimap F Y}^{\text{dur}} : \mathbf{C} \rightarrow \mathbf{Dur}$$

also defines (lax, oplax, semi) functor respectively

Proposition 32.8.2. *A strategy $\sigma : A \multimap B \in \mathbf{Crash}$ is saturated with respect to durcopy if and only if it is a durable strategy and*

durably O -receptive:

$$\forall s \in \sigma. \forall \alpha \in \Upsilon. \forall m \in M_{A \multimap B}^{\alpha:O}. \exists i \leq \|s\|.$$

$$\text{epo}_i(s) \cdot m \in P_{A \multimap B} \wedge \forall j \neq i. \Upsilon(m) \notin \Upsilon(\text{epo}_j(s)) \implies$$

$$\text{epo}_1(s) \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \text{epo}_i(s) \cdot m \cdot \text{!} \cdot \dots \cdot \text{!} \cdot \text{epo}_{\|s\|}(s) \in \sigma$$

$$\rightsquigarrow\text{-closed: } \forall s \in \sigma. \forall t \in P_{A \multimap B}. t \rightsquigarrow_{A \multimap B} s \implies t \in \sigma$$

$$P\text{-delaying: } \forall s \in \sigma. \forall m \in M_{A \multimap B}. \lambda_{A \multimap B}(m) = P \wedge s = p \cdot m \cdot \text{!} \cdot t \Rightarrow p \cdot \text{!} \cdot t \in \sigma$$

Proof. For one direction, notice that

$$\text{durcopy}_A; \sigma; \text{durcopy}_B = (\text{crashcopy}_A; \sigma; \text{crashcopy}_B) \cap P_{A \multimap B}^{\text{dur}}$$

By 32.3.1 we know $\text{crashcopy}_A; \sigma; \text{crashcopy}_B$ satisfies O -receptive, P -delaying and $\rightsquigarrow$ -closed. So $(\text{crashcopy}_A; \sigma; \text{crashcopy}_B) \cap P_{A \multimap B}^{\text{dur}}$ will satisfy durably O -receptive and $\rightsquigarrow$ -closed.

For the other direction, suppose σ is durable, durably O -receptive, P -delaying and $\rightsquigarrow$ -closed. We want to show $\sigma = \text{durcopy}_A; \sigma; \text{durcopy}_B$.

Let σ' be the smallest strategy containing σ and satisfies O -receptive and P -delaying. By definition we know $\sigma' \cap P_{A \multimap B}^{\text{dur}} = \sigma$ and σ' satisfies $\rightsquigarrow$ -closed.

Since σ' satisfies O -receptive, P -delaying and $\rightsquigarrow$ -closed, $\sigma' = \text{crashcopy}_A; \sigma'; \text{crashcopy}_B$. So we have $(\text{crashcopy}_A; \sigma'; \text{crashcopy}_B) \cap P_{A \multimap B}^{\text{dur}} = \sigma$.

Now we have

$$\begin{aligned} \sigma &= \sigma' \cap P_{A \multimap B}^{\text{dur}} \\ &= (\text{crashcopy}_A; \sigma'; \text{crashcopy}_B) \cap P_{A \multimap B}^{\text{dur}} \\ &= (\text{crashcopy}_A \cap P_{A \multimap A}^{\text{dur}}); (\sigma' \cap P_{A \multimap B}^{\text{dur}}); (\text{crashcopy}_B \cap P_{B \multimap B}^{\text{dur}}) \\ &= \text{durcopy}_A; \sigma; \text{durcopy}_B \end{aligned}$$

□

Proposition 32.8.3. For any $A, B \in \underline{\text{Crash}}$ and $\sigma : A \multimap B \in \underline{\text{Safe}}$, $\text{dur}(\sigma) \in \underline{\text{Dur}}$.

Proof. We want to show $\text{dur}(\sigma) = \text{durcopy}_A; K_{\text{copy}} \sigma^\sharp; \text{durcopy}_B$. For the direction $\text{dur}(\sigma) \subseteq \text{durcopy}_A; \text{dur}(\sigma); \text{durcopy}_B$ follows from the analogous fact about K_{copy} – and monotonicity of $- \cap P_{A \multimap B}^{\text{dur}}$.

Now for the other direction, fix $s \in \text{durcopy}_A; \text{dur}(\sigma); \text{durcopy}_B$ first by definition s can be decomposed as

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

and for each i we know $s_i \in \text{copy}_A; s_i; \text{copy}_B$ and we know for each $i \neq j$ we have $\Upsilon(s_i) \neq \Upsilon(s_j)$

So there exists s'_i such that $s'_i \upharpoonright_{A,A,-,-} \in \text{copy}_A$, $s'_i \upharpoonright_{-,A,B,-} = s_i$, $s'_i \upharpoonright_{-,-,B,B} \in \text{copy}_B$. Since we already know for each $i \neq j$ we have $\Upsilon(s_i) \neq \Upsilon(s_j)$ so we know $\Upsilon(s'_i) \neq \Upsilon(s'_j)$.

So we know

$$\begin{aligned} t &:= s'_1 \upharpoonright_{A,A,-,-} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s'_{n+1} \upharpoonright_{A,A,-,-} \\ t' &:= s'_1 \upharpoonright_{-,A,B,-} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s'_{n+1} \upharpoonright_{-,A,B,-} \\ t'' &:= s'_1 \upharpoonright_{-,-,B,B} \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s'_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

such that $\text{ops}(t) \in \text{copy}_A$, $\text{ops}(t'') \in \text{copy}_B$ and $\text{ops}(t') \in \sigma$, so $s \in \text{dur}(\sigma)$.

For the ! -receptivity, notice that suppose $s \in \text{dur}(\sigma)$ and $s \cdot \text{!} \in P_{A \multimap B}^{\text{!}}$, it is easy to check $\text{ops}(s \cdot \text{!}) = \text{ops}(s) \in \sigma$. So $s \cdot \text{!} \in \text{dur}(\sigma)$ $\square$

Proposition 32.8.4 (cf. Proposition 21.5.5).

$$\text{dur}(\text{copy}_A) = \text{durcopy}_A$$

Proof.

$$\begin{aligned} \text{dur}(\text{copy}_A) &= (K_{\text{copy}} \text{copy}_A)^\sharp \cap P_{A \multimap A}^{\text{dur}} \\ &= \text{copy}_A^\sharp \cap P_{A \multimap A}^{\text{dur}} \end{aligned}$$

Now we only need to show $\text{copy}_A^\# \cap P_{A \multimap A}^{\text{dur}} = \text{crashcopy}_A \cap P_{A \multimap A}^{\text{dur}}$. One direction just follows from 32.6.4. Now we only need to show $\text{crashcopy}_A \cap P_{A \multimap A}^{\text{dur}} \subseteq \text{copy}_A^\# \cap P_{A \multimap A}^{\text{dur}}$.

Fix $s \in \text{crashcopy}_A \cap P_{A \multimap A}^{\text{dur}}$, by definition we know $\text{ops}(s)$ is well-formed. Furthermore since for each i , $\text{epo}_i(s) \in \text{copy}_A$ so we know $\text{ops}(s)$ is in copy_A . So $s \in \text{copy}_A^\#$. By assumption, s is durable. So $s \in \text{copy}_A^\# \cap P_{A \multimap A}^{\text{dur}}$. $\square$

Proposition 32.8.5 (cf. Proposition 21.5.4).

$$\text{dur}(-) : \underline{\text{Safe}} \rightarrow \underline{\text{Dur}}$$

behaves like an (enriched) lax semifunctor.

Proof. When $\sigma, \tau \in \text{Safe}$ we have

$$\begin{aligned} \text{dur}(\sigma); \text{dur}(\tau) &= (\sigma^\# \cap P_{A \multimap B}^{\text{dur}}); (\tau^\# \cap P_{B \multimap C}^{\text{dur}}) \\ &= (\sigma^\#; \tau^\#) \cap P_{A \multimap C}^{\text{dur}} \\ &\subseteq (\sigma; \tau)^\# \cap P_{A \multimap C}^{\text{dur}} \\ &\subseteq (K_{\text{copy}}(\sigma; \tau))^\# \cap P_{A \multimap C}^{\text{dur}} \\ &= \text{dur}(\sigma; \tau) \end{aligned}$$

$\square$

32.9 Symmetric Monoidal Structure of Dur

Proposition 32.9.1.

$$\text{dur}(-) : \text{Safe} \rightarrow \text{Dur}$$

defined by

$$\text{dur}(\sigma) := \text{vol}(\sigma) \cap P_{A \multimap B}^{\text{dur}}$$

is a functor

Proof. It follows immediately from the fact that $\text{dur}(-)$ is by definition the composition of two functors. $\square$

Lemma 32.9.1. *For sets of plays*

$$S \subseteq P_{A \multimap B} \quad T \subseteq P_{A' \multimap B'}$$

and

$$(S \cap P_{A \multimap B}^{\text{dur}}) \otimes (T \cap P_{A' \multimap B'}^{\text{dur}}) \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}} = (S \otimes T) \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}}$$

Proof. Since $S \cap P_{A \multimap B}^{\text{dur}} \subseteq S, T \cap P_{A' \multimap B'}^{\text{dur}} \subseteq T$, one direction is trivial.

For the other direction, fix $s \in (S \otimes T) \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}}$. By definition we know s can be decomposed as

$$s = s_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot s_{n+1}$$

And for each i , we know

$$s_i \in t_i \otimes u_i$$

and

$$t := t_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot t_{n+1} \in \sigma$$

$$u := u_1 \cdot \text{!} \cdot \dots \cdot \text{!} \cdot u_{n+1} \in \tau$$

Since for each i , $\Upsilon(t_i) \subseteq \Upsilon(s_i)$, and $\Upsilon(u_i) \subseteq \Upsilon(s_i)$, s is durable implies that both t and u are durable.

So $u \in \sigma \cap P_{A \multimap B}^{\text{dur}}, t \in \tau \cap P_{A' \multimap B'}^{\text{dur}}$. So $s \in (S \cap P_{A \multimap B}^{\text{dur}}) \otimes (T \cap P_{A' \multimap B'}^{\text{dur}}) \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}}$ $\square$

It immediately implies that:

Proposition 32.9.2 (cf. Proposition 21.5.8). *For any given*

$$\sigma : A \multimap B \in \mathbf{Safe} \quad \sigma' : A' \multimap B' \in \mathbf{Safe}$$

we have

$$\text{dur}(\sigma \otimes \tau) = \text{dur}(\sigma) \boxtimes \text{dur}(\sigma')$$

Proposition 32.9.3 (cf. Proposition 21.5.7). *(Dur, $-\boxtimes-$, $\mathbf{1}$) defines a symmetric monoidal*

category.

Proof. Functoriality follows from the fact that $- \boxtimes -$ is the composition of two functors.

We start by defining the structural morphisms. The left and right unital are straightforward. Indeed, they are given by

$$\lambda_A'' := \mathbf{dur}(\lambda_A) \qquad \rho_A'' := \mathbf{dur}(\rho_A)$$

where λ and ρ are the left and right unitals in **Crash**. The braiding and associator are given by

Now we have enough tools to define the braiding and associator in **Dur**

$$\beta_{A,B}'' := \mathbf{dur}(\beta_{A,B}) \quad \text{and} \quad \alpha_{A,B,C}'' := \mathbf{dur}(\alpha_{A,B,C})$$

It immediately follows from functoriality of $- \cap P_-^{\mathbf{dur}}$, the definitions of the unitals, associators and braiding, and the fact the corresponding structural morphisms are natural transformations in **Crash** that the naturality squares still commute (note that for durable σ , $\sigma \cap P^{\mathbf{dur}} = \sigma$).

The coherence diagrams follow from functoriality of $\mathbf{dur}(-)$ together with the fact that $\mathbf{dur}(-)$ distributes over $- \boxtimes -$ (Prop. 32.9.2), by noting that all the structural morphisms in **Dur** were defined by lifting the corresponding structural morphisms in **Crash**, which is also why they are isomorphisms. $\square$

32.10 Durable Linearizability

Proposition 32.10.1 (cf. Proposition 21.5.3). *For a strategy $\nu : A \in \underline{\mathbf{Crash}}$ where A is a durable game*

$$\mathbf{dur}(\nu) = \{s \in P_A^{\mathbf{dur}} \mid s \text{ is durably linearizable with respect to } \nu\}$$

Proof. Suppose $s \xrightarrow{\mathbf{dur}} t$ with $t \in \nu$. So first, by definition, s is durable. Then, $\mathbf{ops}(s) \rightsquigarrow t$, so that $\mathbf{ops}(s) \in K_{\text{copy}} \nu$. But then, since s is durable, $\mathbf{ops}(s) \in K_{\text{copy}} \nu$ so that $s \in$

$$(K_{\text{copy}} \nu)^{\sharp} \cap P_A^{\text{dur}} \subseteq \text{dur}(\nu).$$

For the other direction, suppose $s \in \text{dur}(\nu)$. Then, $s \in P_A^{\text{dur}}$ and $s \in (K_{\text{copy}} \nu)^{\sharp}$. But then, $\text{ops}(s) \in K_{\text{copy}} \nu$, so that $\text{ops}(s) \rightsquigarrow t$, and therefore $s \xrightarrow{\text{dur}} t$. $\square$

Corollary 32.10.1. *For a game $A \in \underline{\text{Crash}}$, $\nu' : A \in \underline{\text{Crash}}$ is durably linearizable to $\nu : A \in \underline{\text{Safe}}$ if and only if $\nu' \subseteq \text{dur}(\nu)$.*

Proposition 32.10.2. *For any durable $\sigma : A \in \underline{\text{Safe}}$ and $\tau : B \in \underline{\text{Safe}}$*

$$\text{dur}(\sigma \otimes \tau) = \text{dur}(\sigma) \boxtimes \text{dur}(\tau)$$

Proof.

$$\begin{aligned} \text{dur}(\sigma \otimes \tau) &= (K_{\text{copy}} (\sigma \otimes \tau))^{\sharp} \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}} \\ &= (K_{\text{copy}} \sigma \otimes K_{\text{copy}} \tau)^{\sharp} \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}} \\ &= (K_{\text{copy}} \sigma^{\sharp} \otimes K_{\text{copy}} \tau^{\sharp}) \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}} \\ &= (K_{\text{copy}} \sigma^{\sharp} \cap P_{A \multimap B}^{\text{dur}}) \otimes (K_{\text{copy}} \tau^{\sharp} \cap P_{A' \multimap B'}^{\text{dur}}) \cap P_{A \otimes A' \multimap B \otimes B'}^{\text{dur}} \\ &= \text{dur}(\sigma) \boxtimes \text{dur}(\tau) \end{aligned}$$

$\square$

Proposition 32.10.3 (cf. Proposition 21.5.9). *For $\sigma, \tau : A \multimap B \in \mathbf{Dur}$ and $\sigma', \tau' : A' \multimap B' \in \mathbf{Dur}$ we have*

$$\sigma \boxtimes \sigma' \subseteq \tau \boxtimes \tau' \implies \sigma \subseteq \tau \wedge \sigma' \subseteq \tau'$$

Proof. For given $\sigma, \tau : A \multimap B$ and $\sigma', \tau' : A' \multimap B'$ suppose $\sigma \boxtimes \sigma' \subseteq \tau \boxtimes \tau'$.

Now let's try to show $\sigma \subseteq \tau$. Fix $s \in \sigma$, by definition s can be decomposed as

$$s = s_1 \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot s_{n+1}$$

By $\text{\textcolor{blue}{\text{\textbf{!}}}}$ -receptive we know

$$t := \epsilon \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \dots \cdot \text{\textcolor{blue}{\text{\textbf{!}}}} \cdot \epsilon \in \sigma'$$

we know that $s \boxtimes t \in \tau \boxtimes \tau'$ but by definition we know it force that $s \in \tau$. So $\sigma \subseteq \tau$

The proof of $\sigma' \subseteq \tau'$ is similar. $\square$

Corollary 32.10.2 (Locality; cf. Corollary 21.5.1). *For $\nu'_A :_! A, \nu'_B :_! B \in \mathbf{Dur}$ and $\nu_A : A, \nu_B : B \in \mathbf{Safe}$:*

$$\nu'_A \xrightarrow{\text{dur}} \nu_A \text{ and } \nu'_B \xrightarrow{\text{dur}} \nu_B \text{ if and only if } \nu'_A \boxtimes \nu'_B \xrightarrow{\text{dur}} \nu_A \otimes \nu_B$$

Proof.

$$\nu'_A \boxtimes \nu'_B \subseteq \text{dur}(\nu_A \otimes \nu_B) = \text{dur}(\nu_A) \boxtimes \text{dur}(\nu_B) \iff \nu'_A \subseteq \text{dur}(\nu_A) \wedge \nu'_B \subseteq \text{dur}(\nu_B)$$

$\square$

Proposition 32.10.4 (cf. Proposition 21.5.6). *Let $A, B \in \mathbf{Crash}$. Then $\nu'_A :_! A$ is durably linearizable to $\nu_A : A$ if and only if whenever $\sigma : A \multimap B \in \mathbf{Safe}$ implements a concurrent object linearizable to ν_B using ν_A , then $\text{dur}(\sigma) : A \multimap B$ implements an object durably linearizable to ν_B using ν'_A .*

Proof. For the forward direction, we have that by assumption

$$\nu_A; \sigma \subseteq K_{\text{copy}} \nu_B$$

And by lax functoriality of $\text{dur}(-)$

$$\nu'_A; \text{dur}(\sigma) \subseteq \text{dur}(\nu_A); \text{dur}(\sigma) \subseteq \text{dur}(\nu_A; \sigma) \subseteq \text{dur}(K_{\text{copy}} \nu_B) = \text{dur}(\nu_B)$$

For the backward direction, note that

$$\nu_A; \text{copy}_A = K_{\text{copy}} \nu_A$$

So, by assumption,

$$\nu'_A; \text{dur}(\text{copy}_A) \subseteq \text{dur}(K_{\text{copy}} \nu_A)$$

and hence,

$$\nu'_A = \nu'_A; \text{durcopy}_A = \nu'_A; \text{dur}(\text{copy}_A) \subseteq \text{dur}(K_{\text{copy}} \nu_A) = \text{dur}(\nu_A)$$

□

Chapter 33

Liveness

33.1 Composition

It is easy to see that:

Lemma 33.1.1. *For any $S \subseteq P_{A \multimap B}^{\leq \infty}$ where $\forall s \in S$ there is no $\circ$ -token in s , $U_{\circ} F_{\circ} S = S$.*

Proof. Suppose $s \in U_{\circ} F_{\circ} S$. Then, there must be $s' \in F_{\circ} S$ such that $U_{\circ} s' = s$. But, if $s' \in F_{\circ} S$ then, in particular, by definition, $U_{\circ} s' \in S$. Hence, $s \in S$.

Now, if $s \in S$, then it is easy to see that either $s \in P_{A \multimap B}^{ws}$, in which case $U_{\circ} s = s$ (by assumption s has no $\circ$ -tokens) witnesses that $s \in F_{\circ} S$ so that $s \in U_{\circ} F_{\circ} S$, or there is some sequence of divergence tokens $t \subseteq (\cup_{\alpha \in \Upsilon} \alpha : \circ_A \cup \alpha : \circ_B)^*$ such that $s' = s \cdot t \in P_{A \multimap B}^{ws}$. In this latter case, s witnesses that $s' \in F_{\circ} S$ as $U_{\circ} s' = s$, and then again $s \in U_{\circ} F_{\circ} S$. $\square$

We note that the operations $F_{\circ}$ – and $U_{\circ}$ – and refinement are related to each other in the following way:

Corollary 33.1.1. *For any strategy $\sigma : A \multimap B$ and win condition $\Sigma :: A \multimap B$:*

$$F_{\circ} \sigma \sqsupseteq_L \Sigma \iff \sigma \sqsupseteq_S U_{\circ} \Sigma$$

Proof. By Def. of the refinement pre-orders and Lem. 33.1.1 we have:

$$F_{\circ} \sigma \sqsupseteq_L \Sigma \iff U_{\circ} F_{\circ} \sigma \sqsupseteq_S U_{\circ} \Sigma \iff \sigma \sqsupseteq_S U_{\circ} \Sigma$$

□

Corollary 33.1.2. *For strategies $\sigma', \sigma : A \multimap B$: $F_{\circlearrowleft} \sigma' \sqsupseteq_L F_{\circlearrowleft} \sigma \iff \sigma' \sqsupseteq_S \sigma$.*

Proof. By Prop. 33.1.1 and Prop. 33.1.1

$$F_{\circlearrowleft} \sigma' \sqsupseteq_L F_{\circlearrowleft} \sigma \iff \sigma' \sqsupseteq_S U_{\circlearrowleft} F_{\circlearrowleft} \sigma = \sigma$$

□

Lemma 33.1.2. *For any win condition, $\Sigma :: A \multimap B$, $\Sigma \subseteq F_{\circlearrowleft} U_{\circlearrowleft} \Sigma$.*

Proof.

$$s \in \Sigma \implies U_{\circlearrowleft} s \in U_{\circlearrowleft} \Sigma \implies s \in F_{\circlearrowleft} U_{\circlearrowleft} \Sigma$$

□

Corollary 33.1.3. *If $F_{\circlearrowleft} U_{\circlearrowleft} \Sigma = \Sigma$ then $\Sigma' \sqsupseteq_L \Sigma \iff \Sigma' \subseteq \Sigma$.*

Proof. Since $U_{\circlearrowleft} -$ is monotonic, the backward direction is immediate. For the forward direction, note that

$$\Sigma' \subseteq F_{\circlearrowleft} U_{\circlearrowleft} \Sigma' \subseteq F_{\circlearrowleft} U_{\circlearrowleft} \Sigma = \Sigma$$

□

This allows us to easily prove that:

Proposition 33.1.1 (cf. Proposition 23.2.1). *$- \circlearrowleft -$ is associative.*

Proof.

$$\begin{aligned} S \circlearrowleft (T \circlearrowleft R) &= F_{\circlearrowleft}(U_{\circlearrowleft} S ; U_{\circlearrowleft} F_{\circlearrowleft}(U_{\circlearrowleft} T ; U_{\circlearrowleft} R)) \\ &= F_{\circlearrowleft}(U_{\circlearrowleft} S ; (U_{\circlearrowleft} T ; U_{\circlearrowleft} R)) \\ &= F_{\circlearrowleft}((U_{\circlearrowleft} S ; U_{\circlearrowleft} T) ; U_{\circlearrowleft} R) \\ &= F_{\circlearrowleft}((U_{\circlearrowleft} F_{\circlearrowleft} (U_{\circlearrowleft} S ; U_{\circlearrowleft} T) ; U_{\circlearrowleft} R) \\ &= (S \circlearrowleft T) \circlearrowleft R \end{aligned}$$

□

Note that this is a reasonable composition operation under $\equiv_{\circ}$, as shown by:

Lemma 33.1.3. *For any $S, T \subseteq P_{A \rightarrow B}^{\text{ws}}$,*

- $F_{\circ} U_{\circ} S \equiv_{\circ} S$.
- $S \equiv_{\circ} T \iff U_{\circ} S = U_{\circ} T$.

Proof. • If $s \in S$ then note that $U_{\circ} s \in U_{\circ} S$ so that $s \in F_{\circ} U_{\circ} S$. Hence, $S \supseteq_L F_{\circ} U_{\circ} S$. Now, if $s \in F_{\circ} U_{\circ} S$ then there is $s' \in S$ such that $U_{\circ} s = U_{\circ} s'$. Hence, $F_{\circ} U_{\circ} S \supseteq_L S$.

- This is immediate from the definition, as $U_{\circ} S \subseteq U_{\circ} T$ and $U_{\circ} S \supseteq U_{\circ} T$.

□

So that $- \circ -$ is invariant under $- \equiv_{\circ} -$. That is:

Proposition 33.1.2 (cf. Proposition 23.2.2). *If $S \equiv_{\circ} S' \subseteq P_{A \rightarrow B}^{\text{ws}}$ and $T \equiv_{\circ} T' \subseteq P_{B \rightarrow C}^{\text{ws}}$ then $S \circ T \equiv_{\circ} S' \circ T'$.*

Proof.

$$S \circ T = F_{\circ} (U_{\circ} S ; U_{\circ} T) = F_{\circ} (U_{\circ} S' ; U_{\circ} T') = S' \circ T'$$

□

It is easy to see that:

Proposition 33.1.3. *For sets of plays $S_i \subseteq P_{A \rightarrow B}^{\leq \infty}$, $U_{\circ} \cup_{i \in I} S_i = \cup_{i \in I} U_{\circ} S_i$.*

Proof.

$$s \in U_{\circ} \cup_{i \in I} S_i \iff U_{\circ} s \in \cup_{i \in I} S_i s \iff \exists i. U_{\circ} s \in S_i \iff \exists i. s \in U_{\circ} S_i \iff s \in \cup_{i \in I} U_{\circ} S_i$$

□

Corollary 33.1.4. *For sets of plays $S_i \subseteq P_{A \rightarrow B}^{\leq \infty}$, $\cup_{i \in I} S_i$ is a least upper bound of $(S_i)_{i \in I}$ up-to $- \equiv_{\circ} -$.*

33.2 Sidenote: Associativity of $- \circ -$

We start with some setup for our associativity proof, some of which is useful more broadly for other proofs.

First, we observe that $s \circ t = \{s\} \circ \{t\}$. Then, by a *witness of the membership of* $s \in S ; T$ we mean the play s' satisfying the existential quantifier in the definition of $- ; -$. Due to $- \circ -$ being defined using $- ; -$, we extend the terminology to $- \circ -$.

Now, we extend the notions of interaction sequences to take into account four games (alternatively, three strategies). We define:

$$M_{A \circ B \circ C \circ D} := ((M_A^\circ + M_B^\circ + M_C^\circ + M_D^\circ)^\bullet)^\Upsilon$$

$$A \circ B \circ C \circ D := \{s \in M_{A \circ B \circ C \circ D}^\star \mid s \upharpoonright_{A,B,-,-} \in P_{A \multimap B}^{\leq \infty} \wedge s \upharpoonright_{-,B,C,-} \in P_{B \multimap C}^{\leq \infty} \wedge s \upharpoonright_{-, -, C, D} \in P_{C \multimap D}^{\leq \infty}\}$$

Proposition 33.2.1. $- ; -$ is join-preserving.

Proof. We show that

$$(\bigcup_{i \in I} S_i) ; \tau = \bigcup_{i \in I} (S_i ; T)$$

One direction is simple as we have that

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

so that

$$S_i ; T \subseteq \left(\bigcup_{i \in I} S_i \right) ; T$$

by monotonicity and hence

$$\bigcup_{i \in I} (S_i ; T) \subseteq \left(\bigcup_{i \in I} S_i \right) ; T$$

For the other direction, suppose

$$s \in \left(\bigcup_{i \in I} S_i \right) ; T$$

then

$$s \upharpoonright_{A,B,-} \in \bigcup_{i \in I} S_i \quad s \upharpoonright_{-,B,C} \in T$$

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

$$s \upharpoonright_{A,B,-} \in S_j$$

and therefore

$$s \upharpoonright_{A,-,C} \in S_j; T$$

so that

$$s \upharpoonright_{A,B,-} \in \bigcup_{i \in I} (S_i; T)$$

To finish, we show

$$S; (\cup_{i \in I} T_i) = \cup_{i \in I} (S; T_i)$$

One direction is simple as we have that

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

so that

$$S; T_i \subseteq S; \bigcup_{i \in I} T_i$$

by monotonicity and hence

$$\bigcup_{i \in I} (S; T_i) \subseteq S; \bigcup_{i \in I} T_i$$

For the other direction, suppose

$$s \in S; \bigcup_{i \in I} T_i$$

then

$$s \upharpoonright_{A,B,-} \in S \quad s \upharpoonright_{-,B,C} \in \bigcup_{i \in I} T_i$$

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

$$s \upharpoonright_{-,B,C} \in T_j$$

and therefore

$$s \upharpoonright_{A,-,C} \in S; T_j$$

so that

$$s \upharpoonright_{A,-,C} \in \bigcup_{i \in I} (S; T_i)$$

□

Corollary 33.2.1. $-; -$ is monotonic and join-preserving.

Proposition 33.2.2. For sets of plays $S \subseteq P_{A \multimap B}^{\leq \infty}$, $T \subseteq P_{B \multimap C}^{\leq \infty}$, $R \subseteq P_{C \multimap D}^{\leq \infty}$:

$$(S; T); R = S; (T; R)$$

Moreover, if $s \in (S; T); R$ or $s \in S; (T; R)$ then there exists a witness $s' \in A \circ B \circ C \circ D$ of both memberships.

Proof. First, note that the only differences between the definition of $-; -$ here and in Ch. 12 is the presence of $\bullet$ -tokens which are handled differently from other moves on projections, and the possibility of plays being infinite (note that for the sake of standard composition $\circ$ -tokens behave just like OA- or PA-moves). So assume the claim holds for s , t , and r not containing $\bullet$ -tokens. Then, given $s \in S; (T; R)$ we can take $s' = s \setminus (\cup_{\alpha \in \Upsilon} \alpha \bullet)$ and then by assumption there are s_1, s_2, s_3 such that $s' \in (s_1; s_2); s_3$. By Prop. 12.1.1 there is a witness $s'' \in A \circ B \circ C \circ D$ of this membership. Let $s_0 = s''$ and let $\alpha_0, \dots, \in v = \{\alpha \in \Upsilon \mid \alpha \bullet \in s\}$ be ordered by the order that $\alpha_i \bullet$ appear in s . Then, if $s_0 = s''$, then given s_i which result from adding $\alpha_i \bullet$ to s'' so that $s'' \upharpoonright_{A,-,-,D} = s \setminus \cup_{\alpha_j, j < i} \alpha_j \bullet$, we construct s_{i+1} as follows. It must be that $s = p \cdot \alpha_i \bullet \cdot t$. Then, we let $s_i = p' \cdot t'$ where $p' \upharpoonright_{A,-,-,D} = p$ be the maximal such prefix of s_i . Then, we make $s_{i+1} = p' \cdot \alpha_i \bullet \cdot t'$. It is easy to see then that $s_{|v|}$ witnesses that $s \in S; (T; R)$ as well as $s \in (S; T); R$. When $s \in S; (T; R)$ the proof is analogous.

Now, there is not much special about infinite plays. One need only realize that in the usual argument for building the witness in $A \circ B \circ C \circ D$ is inductive over the prefixes of the given play s , and therefore can easily be extended to instead give a chain $\{p'_0, \dots, p'_n, \dots\}$ such that $p'_i \upharpoonright_{A,-,-,D} \sqsubseteq p$ and $p'_i \sqsubset p'_{i+1}$. The limit of this chain is then the witness we require, which also shows associativity. □

Proposition 33.2.3. *For sets of plays $S \subseteq P_{A \multimap B}^{\leq \infty}$, $T \subseteq P_{B \multimap C}^{\leq \infty}$, $R \subseteq P_{C \multimap D}^{\leq \infty}$ with R satisfying $\text{O}\circ\text{P}\circ$ -receptivity, total divergence and rooted divergence:*

$$(S \wp T) \wp R = S \wp (T \wp R)$$

Proof. First we observe that

$$(S \wp T) \wp R = \left(\bigcup_{v \subseteq \Upsilon} \sharp_C^v(b_B^v S; b_B^v T) \right) \wp R = \bigcup_{\gamma \subseteq \Upsilon} \bigcup_{v \subseteq \gamma} \sharp_D^\gamma(b_C^\gamma \sharp_C^v(b_B^v S; b_B^v T); b_C^\gamma R)$$

Similarly:

$$S \wp (T \wp R) = S \wp \left(\bigcup_{\gamma \subseteq \Upsilon} \sharp_D^\gamma(b_C^\gamma T; b_C^\gamma R) \right) = \bigcup_{v \subseteq \Upsilon} \bigcup_{\gamma \subseteq \Upsilon} \sharp_D^v(b_B^v S; b_B^v \sharp_D^\gamma(b_C^\gamma T; b_C^\gamma R))$$

Fix $v, \gamma \subseteq \Upsilon$ and let $s \in \sharp_D^\gamma(b_C^\gamma \sharp_C^v(b_B^v S; b_B^v T); b_C^\gamma R)$. Now, by definition of $\sharp_D^\gamma -$, there is a subset $\gamma' \subseteq \gamma$ such that $s \setminus (\cup_{\alpha \in \gamma'} \alpha : \circ_D) \in b_C^\gamma \sharp_C^v(b_B^v S; b_B^v T); b_C^\gamma R$. Let t witness this membership. Then, there is some subset $v' \subseteq v$ such that $t|_{A, C, -} \setminus (\cup_{\alpha \in v'} \alpha : \circ_C) \in b_B^v S; b_B^v T$. Since R satisfies rooted divergence (to remove $\circ_C$ and $\circ_C \cdot \circ_D$ for $\alpha \in v'$) and by definition of v' :

$$s \setminus (\cup_{\alpha \in \gamma'} \alpha : \circ_D) \setminus (\cup_{\alpha \in v'} \alpha : \circ_C) \in b_C^{\gamma \setminus v'}(b_B^v S; b_B^v T); b_C^{\gamma \setminus v'} R \subseteq (S; T); R$$

By Prop. 33.2.2 we obtain a witness $s' \in A \circ B \circ C \circ D$. Now, we must make sure to produce evidence of membership of s by adding $\circ_D$ -tokens for every $\alpha \in v' \cup \gamma'$. Note that $v \subseteq \gamma$, as all $\alpha \in v$ either already have $\alpha : \circ_C$ in $s'|_{-, -, C, -}$, or $\pi_\alpha(s'|_{-, -, C, -})$ is infinite, or had $\alpha : \circ_C$ added by $\sharp_C^v -$. Now, let $\gamma'' \subseteq \gamma'$ be defined by:

$$\gamma'' := \{\alpha \in \gamma' \mid \pi_\alpha(s'|_{-, B, -, -}) \in P_B^\omega\}$$

Note that $v' \cap \gamma'' = \emptyset$. Indeed, as $\gamma'' \subseteq \gamma$, $\pi_\alpha(s'|_{-, -, C, -}) \in b_C \pi_\alpha(s'|_{-, -, C, -})$. But if $\alpha \in v'$, then $\pi_\alpha(s'|_{-, B, -, -}) \in b_B \pi_\alpha(s'|_{-, B, -, -})$, and, moreover, after hiding B it is possible to add a $\alpha : \circ_C$. Hence, $\pi_\alpha(s'|_{-, -, C, -})$ is finite. But since $\circ_C \notin \pi_\alpha(s'|_{-, -, C, -})$, the reason why $\alpha \in \gamma$ must be because $\pi_\alpha(s'|_{-, -, C, -})$ is infinite as well. But then, it cannot be that it was possible

to add $\odot_C$ after hiding B , which is what the set of agents in v' entails, a contradiction. As a consequence $\gamma'' \subseteq \gamma \setminus v'$. Moreover, by definition $\gamma'' \subseteq v$. We show that:

$$s \in \#_D^v(b_B^v S; b_B^v \#_D^{\gamma \setminus v'}(b_C^{\gamma \setminus v'} T; b_C^{\gamma \setminus v'} R))$$

Suppose $\alpha \in \gamma \setminus v'$. Since $s' \upharpoonright_{-, -, C, D} \in b_C^{\gamma \setminus v'} R$, it follows that $s' \upharpoonright_{-, B, C, -} \in b_C^{\gamma \setminus v'} T$. In other words, we have, shown that:

$$s' \upharpoonright_{-, B, C, -} \in b_C^{\gamma \setminus v'} T \quad s' \upharpoonright_{-, -, C, D} \in b_C^{\gamma \setminus v'} R$$

Note that from the definition of $\#_D^{\gamma}$ and γ' we know we can add $\odot_D$ tokens for agents $\alpha \in \gamma' \setminus \gamma''$ (as these do not have an infinite play in B). So, we may add $\alpha : \odot_D$ tokens for them to $s' \upharpoonright_{A, B, -, D}$ to obtain s'' such that

$$\begin{aligned} s'' \upharpoonright_{A, -, D} &= s \setminus (\cup_{\alpha \in \gamma''} \alpha : \odot_D) \setminus (\cup_{\alpha \in v'} \alpha : \odot_D) & s'' \upharpoonright_{A, B, -} &= s' \upharpoonright_{A, B, -, -} \\ s'' \upharpoonright_{-, B, D} &\in \#_D^{\gamma \setminus v'}(b_C^{\gamma \setminus v'} T; b_C^{\gamma \setminus v'} R) \end{aligned}$$

But then, note that by assumption we have:

$$s'' \upharpoonright_{A, B, -} = s' \upharpoonright_{A, B, -, -} \in b_B^v S$$

since we have not added any new events to the B component of s' it follows that $s'' \upharpoonright_{-, -, B, -} = s' \upharpoonright_{-, -, B, -}$, and hence:

$$s'' \upharpoonright_{-, B, D} \in b_B^v \#_D^{\gamma \setminus v'}(b_C^{\gamma \setminus v'} T; b_C^{\gamma \setminus v'} R)$$

Hence,

$$s \setminus (\cup_{\alpha \in \gamma''} \alpha : \odot_D) \setminus (\cup_{\alpha \in v'} \alpha : \odot_D) = s'' \upharpoonright_{A, -, D} \in b_B^v S; b_B^v \#_D^{\gamma \setminus v'}(b_C^{\gamma \setminus v'} T; b_C^{\gamma \setminus v'} R)$$

showing that, as $\gamma'' \cup v' \subseteq v$:

$$s \in \#_D^v(b_B^v S; b_B^v \#_D^{\gamma \setminus v'}(b_C^{\gamma \setminus v'} T; b_C^{\gamma \setminus v'} R))$$

as we know the addition of the $\alpha:\circlearrowleft_D$ to make s is well-formed, since s was assumed well-formed.

For the reverse direction, fix $\gamma, v \subseteq \Upsilon$ and $s \in \sharp_D^v(\flat_B^v S; \flat_B^v \sharp_D^\gamma(\flat_C^\gamma T; \flat_C^\gamma R))$. By definition, there is some subset $v' \subseteq v$ such that

$$s \setminus (\cup_{\alpha \in v'} \alpha:\circlearrowleft_D) \in \flat_B^v S; \flat_B^v \sharp_D^\gamma(\flat_C^\gamma T; \flat_C^\gamma R)$$

and some subset $\gamma' \subseteq \gamma$ such that

$$s \setminus (\cup_{\alpha \in v'} \alpha:\circlearrowleft_D) \setminus (\cup_{\alpha \in \gamma'} \alpha:\circlearrowleft_D) \in \flat_B^v S; \flat_B^v(\flat_C^{\gamma'} T; \flat_C^{\gamma'} R) \subseteq S; (T; R)$$

So let s' be a witness of this membership per Prop. 33.2.2. Let $v'' \subseteq v$ be defined by:

$$v'' := \{\alpha \in v' \mid \pi_\alpha(s' \upharpoonright_{-, -, C, -}) \in \flat_C \pi_\alpha(s' \upharpoonright_{-, -, C, -})\}$$

We show that

$$s \in \sharp_D^{\gamma \cup v''}(\flat_C^{\gamma \cup v''} \sharp_C^v(\flat_B^v S; \flat_B^v T); \flat_C^{\gamma \cup v''} R)$$

By assumption, we have:

$$s' \upharpoonright_{A, B, -, -} \in \flat_B^v S$$

But then, it follows that

$$s' \upharpoonright_{-, B, C, -} \in \flat_B^v T$$

So let

$$v''' := \{\alpha \in v' \mid \pi_\alpha(s' \upharpoonright_{-, -, C, -}) \notin P_C^\omega\}$$

Hence, $s' \upharpoonright_{A, -, C, D} \in \flat_B^v S; \flat_B^v T$. Now, note that since for $\alpha \in v'$, by well-formedness α has a pending question in C . So, first add $\alpha:\circlearrowleft_D$, for $\alpha \in v' \cap v'''$ to $s' \upharpoonright_{A, -, C, D}$ to make s'' like in

previous cases, while maintaining that:

$$\begin{aligned} s'' \upharpoonright_{A,C,-} &= s' \upharpoonright_{A,-,C,-} & s'' \upharpoonright_{-,C,D} \setminus (\cup_{\alpha \in v' \cap v'''} \alpha \circ_D) &= s' \upharpoonright_{-,-,C,D} \\ s'' \upharpoonright_{A,-,D} &= s \setminus (\cup_{\alpha \in v' \cap v'''} \alpha \circ_D) \setminus (\cup_{\alpha \in \gamma'} \alpha \circ_D) \end{aligned}$$

s'' is however not well-formed, as it responds to a pending C question with $\alpha \circ_D$ for $\alpha \in v''$. We can repair this inductively as follows. First, fix an ordering $\alpha_0, \dots, \alpha_{|v''|}$ on v'' and let $s_0 = s''$. Then, for every $i < |v''|$. Then, by definition of s'' (by induction), there is a factoring $s_i = p \cdot \alpha_i \circ_D \cdot p'$, then let $s_{i+1} = p \cdot \alpha_i \circ_C \cdot \alpha_i \circ_D \cdot p'$. Then, note that

$$s_{|v''|} \upharpoonright_{A,C,-} \in \#_C^v(b_B^v S; b_B^v T)$$

Moreover, by $\text{O}\circ\text{P}\circ$ -receptivity (when $\alpha \in v''$) or total divergence R (when $\alpha \notin v''$):

$$s_{|v''|} \upharpoonright_{-,C,D} \in R$$

Note then that we have not added or removed any moves in C for $\alpha \notin v''$. Therefore, in fact:

$$s_{|v''|} \upharpoonright_{-,C,D} \in b_C^{\gamma \cup v''} R \quad s_{|v''|} \upharpoonright_{A,C,-} \in b_C^{\gamma \cup v''} \#_C^v(b_B^v S; b_B^v T)$$

Hence,

$$s \setminus (\cup_{\alpha \in v' \setminus v'''} \alpha \circ_D) \setminus (\cup_{\alpha \in \gamma'} \alpha \circ_D) = s_{|v''|} \upharpoonright_{A,-,D} \in b_C^{\gamma \cup v''} \#_C^v(b_B^v S; b_B^v T); b_C^{\gamma \cup v''} R$$

Note then that by definition, if $\alpha \notin v'''$ then $\alpha \in \gamma$. Therefore

$$s \in \#_D^{\gamma \cup v''} (b_C^{\gamma \cup v''} \#_C^v(b_B^v S; b_B^v T); b_C^{\gamma \cup v''} R)$$

□

Proposition 33.2.4. *Any set of plays $S \subseteq P_{1 \rightarrow A}^{\leq \infty}$ is totally diverging, $\text{O}\circ\text{P}\circ$ -receptive, and satisfies rooted divergence.*

Proof. First, note that every play s of the game $\mathbf{1}$ is either empty or contains only $\bullet$ -tokens, by definition. Therefore, there is no play $s \in P_{\mathbf{1} \multimap A}$ such that $\alpha : \circ_O \in s$. Because of this, S vacuously satisfies all the properties above, which all state that if there is some play containing an $O \circ$ -token then something is true. $\square$

33.3 P-maximal extensions

Lemma 33.3.1. *For s finite, s is α :P-maximal in σ if and only if there is no P-move m such that $s \cdot m \in \sigma$.*

Proof. Since $s \sqsubseteq s$ there must be an extension $p' \sqsubseteq s$ satisfying the claim. But since it is an extension we have $s \sqsubseteq p' \sqsubseteq s$ so $p' = s$.

For the reverse direction, note that for any $p \sqsubseteq s$ we may take the extension $p' = s$ and the claim follows. $\square$

Lemma 33.3.2. *If $\pi_\alpha(s) = \epsilon$ and $s \in \sigma$ then s is α :P-maximal in σ .*

Proof. By well-formedness, if $\pi_\alpha(s) \cdot m = m$ is a well-formed play then either m is an O-move, or m is a $\bullet$ -token, and the claim follows. $\square$

Lemma 33.3.3. *If $s \in \sigma : A \multimap B$ is finite then, for any α , there exists $s' \sqsupseteq s$ such that $s' \in \text{:P-max}(\alpha)\sigma$, and $s' = s \cdot m$ with m a P-move by α or $s' = s$.*

Proof. Suppose s is not α :P-maximal (Otherwise, $s' = s \sqsupseteq s$ is already α :P-maximal). Then, by definition of α :P-maximality there is m a P-move such that $s \cdot \alpha : m \in \sigma$. But then, by well-formedness, for any n making $\pi_\alpha(s) \cdot m \cdot n$ well-formed, the move n must be either an O-move or a $\bullet$ -move. Therefore, $s' = s \cdot \alpha : m$ is α :P-maximal w.r.t. σ . $\square$

Lemma 33.3.4. *For any finite $s \in \sigma$ there exists a P-maximal extension of s in σ which only adds P-moves.*

Proof. Note that since s is finite, $\Upsilon(s)$ is finite, so let $\Upsilon(s) = \{\alpha_0, \dots, \alpha_k\}$. For any $\alpha \notin \Upsilon(s)$, it is immediate that $\pi_\alpha(s) = \epsilon$ so we will maintain this property.

Now, if $s_0 = s$ is P-maximal, then it is finite and we are done. Otherwise, by Lemma 33.3.3, there exists an extension with at most one more move $s_{1,1}$ which is α_1 :P-maximal

for α_1 . Note that for agents α not in $\Upsilon(s)$, we still have $\pi_\alpha(s_{1,1}) = \epsilon$ as the added move is a move by α , per lemma 33.3.3. Similarly, there is an extension $s_{1,2}$ of $s_{1,1}$ which is P-maximal for α_2 , and which is still locally empty for $\alpha \notin \Upsilon(s)$. If we repeat this argument for each $\alpha_i \in \Upsilon(s)$ in order we obtain a play $s_1 = s_{1,k}$ that is at most k moves longer and such that s has a α :P-maximal extension in $\downarrow s_1$ for every agent $\alpha \in \Upsilon$. If s_1 is not P-maximal, we repeat the argument constructing a chain $s_1 \sqsubseteq s_{2,1} \sqsubseteq s_{2,2} \sqsubseteq \dots \sqsubseteq s_{2,k} = s_2$ with each $s_{2,i}$ α_i :P-maximal. By repeating this argument, we build a chain $S = s_0 \sqsubseteq s_1 \sqsubseteq \dots s_n \sqsubseteq \dots$ such that $\sup S$ is P-maximal, as desired. $\square$

33.4 Live Linearizability

Proposition 33.4.1 (cf. Proposition 25.3.1).

$$s \in \text{P-max}_\odot^\infty(\text{copy}_A) \iff \downarrow s \subseteq \text{copy}_A \wedge \forall \alpha \in \Upsilon. \pi_\alpha(s) \upharpoonright_{A,-} = \pi_\alpha(s) \upharpoonright_{-,A} \vee \bullet \in \pi_\alpha(s)$$

Proof. We make an observation first. It is straight-forward to check that a finite play $s \in \text{copy}_A$ is α :P-maximal if and only if $\pi_\alpha(s)$ is even-length or $\bullet \in \pi_\alpha(s)$. The latter case is trivial, as by well-formedness $\pi_\alpha(s)$ ends with $\bullet$ and no move is possible after. Otherwise, as copy_A is total (i.e. copy_A defines a P-response for every pending O-move) a (finite) P-maximal play is characterized by ending with a P-move (or being empty). Hence, by alternation, $\pi_\alpha(s)$ is even-length. For the backward direction, simply note that if $\pi_\alpha(s)$ is even-length, then by alternation it is either empty or ends with P-move.

Suppose $s \in \text{P-max}_\odot(\text{copy}_A)$. Fix $\alpha \in \Upsilon$. Suppose first $\pi_\alpha(s)$ is finite. If $\bullet \in \pi_\alpha(s)$ we are done. Otherwise, $\pi_\alpha(U_\odot s)$ is even-length, so by definition of copy_A ,

$$\pi_\alpha(U_\odot s) \upharpoonright_{A,-} = \pi_\alpha(U_\odot s) \upharpoonright_{-,A}$$

Now, either $\pi_\alpha(U_\odot s) = s$, in which case we are done, or $\pi_\alpha(U_\odot s) \cdot \odot_O \cdot \odot_P = s$, in which case we see that we still have:

$$\pi_\alpha(s) \upharpoonright_{A,-} = \pi_\alpha(s) \upharpoonright_{-,A}$$

Otherwise, s is infinite. But, then, there exists an infinite chain

$$S = p_1 \sqsubseteq \dots \sqsubseteq p_n \sqsubseteq \dots$$

of α :P-maximal plays in $\downarrow s$. But then, for every i , by the observation above, $\pi_\alpha(p_i)|_{A,-} = \pi_\alpha(p_i)|_{-,A}$. Hence, $s = \sup S$ has the same property.

Now, for the backward direction, let

$$\downarrow s \subseteq \text{copy}_A \wedge \forall \alpha \in \Upsilon. \pi_\alpha(s)|_{A,-} = \pi_\alpha(s)|_{-,A} \vee \bullet \in \pi_\alpha(s)$$

If s is finite, note that by definition of copy_A and well-formedness, the next move by every α is an $\bullet$ - or O -move, so it is locally P-maximal, and therefore P-maximal. Now, suppose s is infinite. Let $p \sqsubseteq s$ and assume p is not α :P-maximal. Then, $\pi_\alpha(s)$ cannot be even-length by our observation at the start. Then, whichever is the last move of $\pi_\alpha(s)$ is either an O -move q in the target component A or an O -move a in the source component A (a P-move in A but an O -move in $A \multimap A$). In the former case, it only appears in the target A component, and in the later, it only appears in the source component due the definition of copy_A and well-formedness. But since $\pi_\alpha(s)|_{A,-} = \pi_\alpha(s)|_{-,A}$, there must be an extension of p' where the copy of q or a , depending on the case, appears in the other component. Let p' be the smallest such prefix. Then, $\pi_\alpha(p') = \pi_\alpha(p) \cdot m$ (where $m = q$ or $m = a$, depending on the case). In particular, $\pi_\alpha(p')$ is even-length and hence α :P-maximal. $\square$

Proposition 33.4.2 (cf. Proposition 24.0.1).

$$K^\perp \text{ T} = \{s \in P_{A \multimap B}^{\text{ws}} \mid \exists t \in \text{T}. s \rightsquigarrow_{A \multimap B} t\}$$

Proof. Note that, by Prop. 33.1.1:

$$\begin{aligned} U_\circ (\text{eqcopy}_A \circledast \text{T} \circledast \text{eqcopy}_A) &= U_\circ F_\circ (U_\circ \text{eqcopy}_A ; U_\circ \text{T} ; U_\circ \text{eqcopy}_B) \\ &= U_\circ \text{eqcopy}_A ; U_\circ \text{T} ; U_\circ \text{eqcopy}_B \\ &= \text{P-max}^\infty(\text{copy}_A) ; U_\circ \text{T} ; \text{P-max}^\infty(\text{copy}_B) \end{aligned}$$

Now, if $s \in \text{P-max}^\infty(\text{copy}_A) ; U_\odot \text{ T} ; \text{P-max}^\infty(\text{copy}_B)$ let

$$s' \in \text{P-max}^\infty(\text{copy}_A) ; U_\odot \text{ T} ; \text{P-max}^\infty(\text{copy}_B)$$

witness the membership of s in the interaction above. Then, let $t = s' \upharpoonright_{-,A,B,-} \in U_\odot \text{ T}$.

Then, note that

$$\downarrow s' \subseteq \downarrow \text{P-max}^\infty(\text{copy}_A) ; t ; \downarrow \text{P-max}^\infty(\text{copy}_B) = \text{copy}_A ; \downarrow t ; \text{copy}_B = K \downarrow t$$

But note that then

$$s = \downarrow s' \upharpoonright_{A,-,-,B} \in K \downarrow t$$

So by Prop. 13.1.1 it follows that $\downarrow s \rightsquigarrow \downarrow t$, so in particular $\downarrow s \rightsquigarrow \downarrow \text{T}$. Moreover, by Prop. 25.3.1 it follows that $\alpha : \bullet \in s' \upharpoonright_{-,A,B,-} \iff \alpha : \bullet \in s$. Finally, if $\alpha : \bullet \notin s$ then again by Prop. 25.3.1, then $\alpha : \bullet \notin s' \upharpoonright_{A,A,-,-}, s' \upharpoonright_{-,A,B,-}, s' \upharpoonright_{-,-,B,B}$. Hence, we have:

$$\pi_\alpha(s \upharpoonright_{A,-}) = \pi_\alpha(s' \upharpoonright_{A,-,-,-}) = \pi_\alpha(s' \upharpoonright_{-,A,-,-})$$

$$\pi_\alpha(s \upharpoonright_{-,B}) = \pi_\alpha(s' \upharpoonright_{-,-,-,B}) = \pi_\alpha(s' \upharpoonright_{-,-,-,B})$$

Conversely, assume $s \rightsquigarrow_{A \multimap B} t$ with $t \in U_\odot \text{ T}$. Then, let $\downarrow s$ be the chain:

$$s_0 \sqsubseteq s_1 \sqsubseteq \dots$$

it follows from Prop. 13.1.1 $\downarrow s \subseteq K \downarrow t$. In particular, for every s_i there is $t_i \sqsubseteq t$ such that $s_i \rightsquigarrow t_i$. By Prop. 13.3.1 (note that the result proven by Oliveira Vale et al. [2024a] applies even for $X = A \multimap B$, which they do consider a game), there is a play $s'_i \in \text{copy}_X$ for every s_i and t_i such that $s'_i \upharpoonright_{X,-} = t_i$ and $s'_i \upharpoonright_{-,X} = s_i$. Our goal is to construct s''_i such that for every i , $s''_i \sqsubseteq s'_i$, and, moreover, $s''_i \upharpoonright_{-,X} = s_i$ and $s''_i \upharpoonright_{X,-} \sqsubseteq t$. So let $s''_0 = \epsilon$. Note that by well-formedness, in fact, $s''_0 = s'_0$. Now, assume we have s''_i satisfying the inductive assumption. If $s''_i \not\sqsubseteq s'_{i+1}$ (otherwise, just let $s''_{i+1} = s'_{i+1}$), then note that there is some $\alpha : m$

such that

$$s'_{i+1} \upharpoonright_{-,X} = s_{i+1} = s_i \cdot \alpha:m = s''_i \upharpoonright_{-,X} \cdot \alpha:m$$

We split into cases depending on the polarity of m :

$\lambda_X(m) = \text{O}$ In this case, by well-formedness, $s''_{i+1} = s''_i \cdot \alpha:m$ is well-formed, and satisfies the inductive assumption.

$\lambda_X(m) = \text{P}$ For this case, we note that if $\alpha:m$ already appears in $s'_i \upharpoonright_{X,-}$ then $s''_{i+1} = s''_i \cdot \alpha:m$ is well-formed and a copycat play so the argument is the same as in the O-case. Therefore, assume $\alpha:m$ does not appear in $s''_i \upharpoonright_{X,-}$. By assumption, however, $s_i \cdot \alpha:m \rightsquigarrow t_{i+1}$ and no rule allows the removal of a P-move. Therefore, it must be that $\alpha:m$ already appears in t_{i+1} . Now, $s''_i \upharpoonright_{X,-}, t_{i+1} \sqsubseteq t$, and t_{i+1} has more moves than $s''_i \upharpoonright_{X,-}$, therefore $s''_i \upharpoonright_{X,-} \sqsubseteq t_{i+1}$. So let $t'_{i+1} = s''_i \upharpoonright_{X,-} \cdot p_{i+1} \cdot \alpha:m \sqsubseteq t_{i+1}$ be the extension that reaches the move $\alpha:m$. We show that $s''_{i+1} = s''_i \cdot p_{i+1} \cdot \alpha:m \cdot \alpha:m$ is a copycat play. First, note that any O-moves in p_{i+1} (from the point-of-view of X) must have already appeared $s''_i \upharpoonright_{-,X}$. Indeed, we have that $s_{i+1} \rightsquigarrow t_{i+1}$, and no rule allows adding an O-move to s_{i+1} . Hence, since that O-move already appears in t_{i+1} it must appear in s_i (it certainly cannot be $\alpha:m$). This shows the O-moves in p_{i+1} can be added without breaking the definition of the copycat. Moreover, if p_{i+1} contains a P-move n then note that since t_{i+1} is well-formed, its justifying O-move appears in t_{i+1} already. By the switching condition, it must appear in p_{i+1} before n or already in $s''_i \upharpoonright_{X,-} \sqsubseteq t_{i+1}$. In the former case, we are able to justify it by the previous argument, in the later it already appears in $s''_i \upharpoonright_{X,-}$. Finally, if m is a $\bullet$ -move then it actually would appear in s_{i+1} as well, in which case it could not belong to p_{i+1} .

$\lambda_X(m) = \bullet$ In this case, we have that t_{i+1} must also have a $\alpha:\bullet$. Let $s''_i \upharpoonright_{X,-} \cdot p_{i+1} \cdot \alpha:\bullet \sqsubseteq t_{i+1}$.

Then, similarly to the P-case we can argue all the moves in p_{i+1} must be justified.

Therefore, we can again use $s''_{i+1} = s''_i \cdot p_{i+1} \cdot \alpha:\bullet$.

Hence, we have $s \in K \text{ T}$. So let s' witness the membership. We simply note then that the additional conditions on $s \rightsquigarrow t$ ensure that $s' \upharpoonright_{A,A,-,-} \in \text{P-max}^\infty(\text{copy}_A)$ and $s' \upharpoonright_{-,-,B,B} \in \text{P-max}^\infty(\text{copy}_B)$ by Prop. 25.3.1. $\square$

33.5 Honest Linearizability

Proposition 33.5.1. *If $s \rightsquigarrow t$ and $\alpha \bullet \notin s$ then $\pi_\alpha(s) = \pi_\alpha(t)$.*

Proof. Let R be the relation witnessing $s \rightsquigarrow t$. Suppose $s = s_1 \cdot \alpha \cdot m \cdot s_2$. By simulation, there is t' with $(s_1 \cdot \alpha \cdot m, t') \in R$. By linearizability, $s_1 \cdot \alpha \cdot m \rightsquigarrow_A t'$. If m is a P-move, it follows that $\alpha \cdot m \in t'$ as no rule allows for removing a P-move. Otherwise, if m is an O-move then either the derivation never involves (O), and hence $m \in t'$ and we are done, or it is removed by a (O) rule. In that case $m \in s_O := t' \setminus (s_1 \cdot \alpha \cdot m)$. But in that case, by honesty, since by assumption α is not starved in s , $\pi_\alpha(t') \cdot \pi_\alpha(s_O) \sqsubseteq \pi_\alpha(t)$ so that $m \in t$ and the result follows.

Now, suppose $t = t_1 \cdot \alpha \cdot m \cdot t_2$. By simulation, there is $(s', t') \in R$ with $t_1 \cdot \alpha \cdot m \sqsubseteq t'$. By linearizability, $s' \rightsquigarrow_A t'$. If m is an O-move, then it must appear in s' and we are done (as no rule allows for the addition of an O-move). Otherwise, it is a P-move and either appears in s' or in $s_P := (t' \setminus s')$. In the former case we are done, so assume $\alpha \cdot m$ appears in s_P . Then, by honesty, it follows that some extension $s'' \sqsubseteq s$ includes $\alpha \cdot m$ and we are done. $\square$

Corollary 33.5.1. *If $s \rightsquigarrow t$ then $s \rightsquigarrow t$.*

Proof. Since $s \rightsquigarrow t \implies s \rightsquigarrow t$. In particular, there is $p \in \text{copy}_A$ such that $p|_{A,-} = t$ and $p|_{-,A} = s$. By Prop. 33.5.1 and 25.3.1 it follows that $p \in \text{eqcopy}_A$, but then $s \rightsquigarrow t$ by Prop. 24.0.1. $\square$

In fact, safe and live linearizability is also a complete proof technique for live linearizability.

Proposition 33.5.2. *For win conditions $\omega', \omega :: A$, $\omega' \rightsquigarrow \omega \implies \omega' \rightsquigarrow \omega$.*

Proof. Let $s' \in \omega'_A$. By assumption, there is $t' \in \omega_A$ such that $s' \in \{t'\}; \text{eqcopy}_A$. By Prop. 24.0.2 it follows that there is a play $p' \in \text{eqcopy}_A$ such that $p'|_{A,-} = t'$ and $p'|_{-,A} = s'$. We define the relation $R := \{(p|_{-,A}, p|_{A,-}) : p \sqsubseteq p'\}$. We verify the conditions:

Simulation: For every $s \sqsubseteq s'$, there is $p \sqsubseteq p'$ with $p|_{-,A} = s$, so $(p|_{-,A}, p|_{A,-}) \in R$. For every $q \sqsubseteq t'$, there is $p \sqsubseteq p'$ with $q \sqsubseteq p|_{A,-}$, so $(s, t) \in R$ with $q \sqsubseteq t$ for $s := p|_{-,A}$ and $t := p|_{A,-}$.

Invariant: Let $(s, t) \in R$, so $s = p \upharpoonright_{-,A}$ and $t = p \upharpoonright_{A,-}$ for some $p \sqsubseteq p'$. By Prop. 13.3.1 it follows that $s \rightsquigarrow t$. Moreover, p' satisfies that, for any $\alpha \in \Upsilon$ that is not starved in p' :

$$\pi_\alpha(t') = \pi_\alpha(p' \upharpoonright_{A,-}) = \pi_\alpha(p' \upharpoonright_{-,A}) = \pi_\alpha(s')$$

by Prop. 25.3.1. Hence

$$\pi_\alpha(s) \cdot \pi_\alpha(s_P) \sqsubseteq \pi_\alpha(s') \wedge \pi_\alpha(t) \cdot \pi_\alpha(s_O) \sqsubseteq \pi_\alpha(t')$$

for every non-starved α , establishing honesty. $\square$

Corollary 33.5.2 (cf. Proposition 24.2.1). *For win conditions $\omega', \omega :: A$, $\omega' \rightsquigarrow \omega \iff \omega' \rightsquigarrow \omega$.*

33.6 Relating Safety and Liveness

Proposition 33.6.1 (cf. Proposition 25.2.1). *For any strategy $\sigma : A \multimap B$, $\text{P-max}_\circ(\sigma)$ and $F_L \sigma$ are both total. Moreover,*

$$\Downarrow \text{P-max}_\circ(\sigma) = \sigma \qquad \Downarrow F_L \sigma = \sigma$$

and for any total win condition $\Sigma :: A \multimap B$ such that $\Downarrow \Sigma = \sigma$ it follows that

$$\text{P-max}_\circ(\sigma) \supseteq_L \Sigma \supseteq_L F_L \sigma$$

Proof. Now, let $s \in \sigma : A \multimap B$. We show $s \in \Downarrow \text{P-max}_\circ(\sigma)$. By Lem. 33.3.4 there exists a P -maximal extension $s \cdot t \in \sigma$ of s . Hence, there is some $s' \in \text{P-max}_\circ(\sigma)$ such that $U_\circ s' = s \cdot t$. But then:

$$\Downarrow \text{P-max}_\circ(\sigma) = \Downarrow U_\circ \text{P-max}_\circ(\sigma) \supseteq \Downarrow U_\circ s' = \Downarrow (s \cdot t) \ni s$$

as desired. Now, let $s \in \Downarrow \text{P-max}_\circ(\sigma)$. Then, there is $s' \in \text{P-max}_\circ(\sigma)$ such that $s \sqsubseteq U_\circ s'$. But then $U_\circ s' \in \sigma$ and by prefix-closure $s \in \sigma$. Hence $\Downarrow \text{P-max}_\circ(\sigma) = \sigma$.

Now, we show that $\mathbf{P}\text{-max}_{\circlearrowleft}(\sigma)$ is always total. Let $s \in \Downarrow \mathbf{P}\text{-max}_{\circlearrowleft}(\sigma) \subseteq \sigma$ (the inclusion holds by definition and prefix-closure). If $s \in \mathbf{P}\text{-max}(\sigma)$, then, it is easy to check that there is a t with only $\circlearrowleft$ -moves such that $s \cdot t \in \mathbf{P}\text{-max}_{\circlearrowleft}(\sigma)$. Otherwise, by Lem. 33.3.4 there is a $\mathbf{P}$ -extension t such that $s \cdot t \in \mathbf{P}\text{-max}(\sigma)$ and then by the argument from before the claim follows.

Now, if $s \in \sigma$ then note that there always exists an s' such that $U_{\circlearrowleft} s' = s$ since we can always just concatenate enough $\circlearrowleft$ -tokens to s to obtain such an s' (in particular, $F_L \sigma$ is total). But then, $\Downarrow s' = \downarrow s \subseteq \sigma$ by prefix-closure. Hence, $s' \in F_L \sigma$. But then, as we just saw $s \in \Downarrow F_L \sigma$. Conversely, if $s \in \Downarrow F_L \sigma$ then there is some $s' \in P_{A \multimap B}^{\text{ws}}$ such that $\Downarrow s' \subseteq \sigma$ and $s \in \Downarrow s'$. But then, it is immediate that $s \in \sigma$. Therefore, $\Downarrow F_L \sigma = \sigma$.

Now, if $s \in \mathbf{P}\text{-max}_{\circlearrowleft}(\sigma)$ then $U_{\circlearrowleft} s \in \sigma$, so that by prefix-closure $\Downarrow s \subseteq \sigma$ so that $s \in F_L \sigma$.

Now, suppose $\Downarrow \Sigma = \sigma$. That $\Downarrow \Sigma \subseteq F_L \sigma$ is immediate from the assumption and the definition of $F_L \sigma$.

Now, let $s \in \mathbf{P}\text{-max}_{\circlearrowleft}(\sigma)$. By definition, $U_{\circlearrowleft} s \in \mathbf{P}\text{-max}(\sigma)$. In particular, $U_{\circlearrowleft} s \in \sigma = \Downarrow \Sigma$. By totality it follows that some extension $U_{\circlearrowleft} s \cdot t \in \Sigma$. But since $U_{\circlearrowleft} s$ was $\mathbf{P}$ -maximal, it follows that t contains only $\circlearrowleft$ -tokens, so that $U_{\circlearrowleft} (s \cdot t) = U_{\circlearrowleft} s \in U_{\circlearrowleft} \Sigma$. $\square$

Lemma 33.6.1. *For any total win condition $\Sigma :: A \multimap B$, $\mathbf{P}\text{-max}_{\circlearrowleft}(\Downarrow \Sigma) \sqsupseteq_L \Sigma$.*

Proof. Let $s \in \mathbf{P}\text{-max}_{\circlearrowleft}(\Downarrow \Sigma)$ for some win condition Σ . Then, there is $p \in \Downarrow \Sigma$ such that $U_{\circlearrowleft} s = p$ and p is $\mathbf{P}$ -maximal in $\Downarrow \Sigma$. So, there exists $t \in \Sigma$ such that $p \sqsubseteq U_{\circlearrowleft} t$. Now, by totality, there is an extension containing only $\mathbf{P}$ -moves p' such that $p \cdot p' \in \Sigma$. But since s is $\mathbf{P}$ -maximal, p' may only contain $\circlearrowleft$ -tokens. Hence, $U_{\circlearrowleft} (p \cdot p') = p$. But then, $U_{\circlearrowleft} s = p = U_{\circlearrowleft} p' \subseteq U_{\circlearrowleft} \Sigma$. Hence, $\mathbf{P}\text{-max}_{\circlearrowleft}(\Downarrow \Sigma) \sqsupseteq_L \Sigma$. $\square$

Lemma 33.6.2. *If $s \in \mathbf{P}\text{-max}(-)$ then s is a limit of σ .*

Proof. Note that $s \in \alpha\text{-}\mathbf{P}\text{-max}(\sigma)$ for every α by definition of $\mathbf{P}\text{-max}(-)$, so for any prefix $p \sqsubseteq s$, by definition we have $p \cdot t = s$ for some t , and then t is the desired extension. $\square$

33.7 Symmetric Monoidal Structure for Objects

Proposition 33.7.1. *$- \otimes -$ defines a bisemifunctor with respect to $-; -$.*

Proof. Essentially the same proof as the one given by Oliveira Vale et al. [2023] extends to include infinite plays and $\bullet$. Note that for the sake of $-; -$, $\circ$ -tokens behave as OA or PA-moves. $\square$

Proposition 33.7.2. *For any set of plays $S_1 \subseteq P_{A_1 \multimap B_1}^{\leq \infty}$, $T_1 \subseteq P_{B_1 \multimap C_1}^{\leq \infty}$, and $S_2 \subseteq P_{A_2 \multimap B_2}^{\leq \infty}$, $T_2 \subseteq P_{B_2 \multimap C_2}^{\leq \infty}$,*

$$(S_1 \otimes S_2) ; (T_1 \otimes T_2) = (S_1 ; T_1) \otimes (S_2 ; T_2)$$

Proof. We provide only a sketch of the proof, and only the important direction for later, as the proof is very similar to the proof of bi-semifunctoriality of $- \otimes -$ w.r.t. $-; -$ in Oliveira Vale et al. [2024a].

We show first that $U_{\circ} -$ and $F_{\circ} -$ both commute with $- \otimes -$.

So suppose $s = U_{\circ} s'$ with $s' \in S_1 \otimes S_2$. Then, $s' \upharpoonright_{A_1 \multimap B_1, -} \in S_1$ by definition, and similarly $s' \upharpoonright_{-, A_2 \multimap B_2} \in S_2$. But then,

$$s \upharpoonright_{A_1 \multimap B_1, -} = U_{\circ} s' \upharpoonright_{A_1 \multimap B_1, -} \in U_{\circ} S_1$$

$$s \upharpoonright_{-, A_2 \multimap B_2} = U_{\circ} s' \upharpoonright_{-, A_2 \multimap B_2} \in U_{\circ} S_2$$

and therefore $s \in U_{\circ} S_1 \otimes U_{\circ} S_2$. Similarly, if $s \in U_{\circ} S_1 \otimes U_{\circ} S_2$ then

$$s \upharpoonright_{A_1 \multimap B_1, -} \in U_{\circ} S_1$$

$$s \upharpoonright_{-, A_2 \multimap B_2} \in U_{\circ} S_2$$

note then that by well-formedness a $\circ$ -move occurs only in one of the two components, and, moreover, the component where it could occur is completely determined by the component where the previous move by the same agent happened. Therefore, it follows that $s \in U_{\circ} (S_1 \otimes S_2)$.

The argument for $F_{\circ} (S_1 \otimes S_2) = F_{\circ} S_1 \otimes F_{\circ} S_2$ is quite similar. If $U_{\circ} s = s' \in S_1 \otimes S_2$ then in particular:

$$s \upharpoonright_{A_1 \multimap B_1, -} = U_{\circ} s' \upharpoonright_{A_1 \multimap B_1, -} \in U_{\circ} S_1$$

$$s \upharpoonright_{-, A_2 \multimap B_2} = U_{\circ} s' \upharpoonright_{-, A_2 \multimap B_2} \in U_{\circ} S_2$$

so $s \in F_{\circlearrowleft} S_1 \otimes F_{\circlearrowleft} S_2$. The reverse direction similarly follows by the same reasoning as in the previous case.

Finally, note that:

$$\begin{aligned}
(S_1 \otimes S_2) \circlearrowleft (T_1 \otimes T_2) &= F_{\circlearrowleft} (U_{\circlearrowleft} (S_1 \otimes S_2) ; U_{\circlearrowleft} (T_1 \otimes T_2)) \\
&= F_{\circlearrowleft} ((U_{\circlearrowleft} S_1 \otimes U_{\circlearrowleft} S_2) ; (U_{\circlearrowleft} T_1 \otimes U_{\circlearrowleft} T_2)) \\
&= F_{\circlearrowleft} ((U_{\circlearrowleft} S_1 ; U_{\circlearrowleft} T_1) \otimes (U_{\circlearrowleft} S_2 ; U_{\circlearrowleft} T_2)) \quad (\text{By Prop. 33.7.1}) \\
&= F_{\circlearrowleft} (U_{\circlearrowleft} S_1 ; U_{\circlearrowleft} T_1) \otimes F_{\circlearrowleft} (U_{\circlearrowleft} S_2 ; U_{\circlearrowleft} T_2) \\
&= (S_1 \circlearrowleft T_1) \otimes (S_2 \circlearrowleft T_2)
\end{aligned}$$

□

We already showed the following indirectly in §26, but we give a direct proof now:

Proposition 33.7.3.

$$\text{eqcopy}_A \otimes \text{eqcopy}_B = \text{eqcopy}_{A \otimes B}$$

Proof. Let $s \in \text{eqcopy}_A \otimes \text{eqcopy}_B$ and let $\alpha \in \Upsilon$. Note that every non-starved play of $t \in \text{eqcopy}$ satisfies that $\pi_{\alpha}(t)$ is even-length or infinite. Since $|s| = |s \upharpoonright_{A \multimap A, -}| + |s \upharpoonright_{-, B \multimap B}|$ it follows that s has the same property. Then, since $\text{copy}_A \otimes \text{copy}_B = \text{copy}_{A \otimes B}$ (Prop. 13.5.2), and $\Downarrow s \upharpoonright_{A, -} \subseteq \text{copy}_A$ and $\Downarrow s \upharpoonright_{-, B} \subseteq \text{copy}_B$ by assumption, we have $\Downarrow s \subseteq \text{copy}_A \otimes \text{copy}_B = \text{copy}_{A \otimes B}$. But then, every even-length prefix $p \sqsubseteq \pi_{\alpha}(s)$ satisfies

$$\pi_{\alpha}(p) \upharpoonright_{A \otimes B, -} = \pi_{\alpha}(p) \upharpoonright_{-, A \otimes B}$$

If $\pi_{\alpha}(p)$ is infinite, we are done, otherwise, it is finite but even-length (by the argument in the beginning) and again we are done.

Now, if $s \in \text{eqcopy}_{A \otimes B}$ then it is immediate that $\Downarrow s \subseteq \text{copy}_{A \otimes B} = \text{copy}_A \otimes \text{copy}_B$, and moreover, by assumption

$$\pi_{\alpha}(s) \upharpoonright_{A \otimes B, -} = \pi_{\alpha}(s) \upharpoonright_{-, A \otimes B}$$

But then

$$\begin{aligned}
\pi_\alpha(s \upharpoonright_{A \multimap A, -}) \upharpoonright_{A, -} &= \pi_\alpha(s) \upharpoonright_{A \multimap A, -} \upharpoonright_{A, -} \\
&= \pi_\alpha(s) \upharpoonright_{A \otimes B, -} \upharpoonright_{A, -} \\
&= \pi_\alpha(s) \upharpoonright_{-, A \otimes B} \upharpoonright_{A, -} \\
&= \pi_\alpha(s) \upharpoonright_{A \multimap A, -} \upharpoonright_{-, A}
\end{aligned}$$

and symmetrically for B . □

Corollary 33.7.1.

$$K^\perp (\Sigma \otimes T) = K^\perp \Sigma \otimes K^\perp T$$

Proof.

$$\begin{aligned}
K^\perp (\Sigma \otimes T) &= \text{eqcopy}_{A_1 \otimes A_2} \circ (\Sigma \otimes T) \circ \text{eqcopy}_{B_1 \otimes B_2} \\
&= (\text{eqcopy}_{A_1} \otimes \text{eqcopy}_{A_2}) \circ (\Sigma \otimes T) \circ (\text{eqcopy}_{B_1} \otimes \text{eqcopy}_{B_2}) \\
&= (\text{eqcopy}_{A_1} \circ \Sigma \circ \text{eqcopy}_{B_1}) \otimes (\text{eqcopy}_{A_2} \circ T \circ \text{eqcopy}_{B_2}) \\
&= K^\perp \Sigma \otimes K^\perp T
\end{aligned}$$

□

Proposition 33.7.4 (Locality). *Let $\langle \nu'_A, \omega'_A \rangle, \langle \nu_A, \omega_A \rangle : A$ and $\langle \nu'_B, \omega'_B \rangle, \langle \nu_B, \omega_B \rangle : B$ be objects. Then:*

$$\begin{aligned}
\langle \nu'_A, \omega'_A \rangle \otimes \langle \nu'_B, \omega'_B \rangle &\approx \langle \nu_A, \omega_A \rangle \otimes \langle \nu_B, \omega_B \rangle \\
&\iff \\
\langle \nu'_A, \omega'_A \rangle &\approx \langle \nu_A, \omega_A \rangle \wedge \langle \nu'_B, \omega'_B \rangle \approx \langle \nu_B, \omega_B \rangle
\end{aligned}$$

Proof. This proof relies crucially on the characterization of safe and live linearizability based on refinement (Prop. 24.0.1 & 13.1.1).

The proof that

$$\nu'_A \otimes \nu'_B \supseteq_S K(\nu_A \otimes \nu_B) \iff \nu'_A \subseteq K \nu_A \wedge \nu'_B \supseteq_S K \nu_B$$

is the same as that of Oliveira Vale et al. [2024a].

Now, for the win condition side, note that we showed:

$$K^=(\omega_A \otimes \omega_B) = K^= \omega_A \otimes K^= \omega_B$$

Hence, by the derivation just given and order-isomorphism (Prop. 26.3.1) we have

$$\begin{aligned} \omega'_A \otimes \omega'_B \subseteq K^=(\omega_A \otimes \omega_B) &\iff \omega'_A \otimes \omega'_B \subseteq K^= \omega_A \otimes K^= \omega_B \\ &\iff \omega'_A \subseteq K^= \omega_A \wedge \omega'_B \subseteq K^= \omega_B \end{aligned}$$

□

Proposition 33.7.5. *For any strategies $\sigma : A \multimap C$ and $\tau : B \multimap D$:*

$$\text{P-max}_{\circlearrowleft}^\infty(\sigma \otimes \tau) \supseteq_L \text{P-max}_{\circlearrowleft}^\infty(\sigma) \otimes \text{P-max}_{\circlearrowleft}^\infty(\tau)$$

Proof. First, we note that (see the proof of Prop. 33.7.2):

$$\text{P-max}_{\circlearrowleft}^\infty(\sigma \otimes \tau) = F_{\circlearrowleft} \text{P-max}^\infty(\sigma \otimes \tau)$$

$$\text{P-max}_{\circlearrowleft}^\infty(\sigma) \otimes \text{P-max}_{\circlearrowleft}^\infty(\tau) = F_{\circlearrowleft} \text{P-max}^\infty(\sigma) \otimes F_{\circlearrowleft} \text{P-max}^\infty(\tau) = F_{\circlearrowleft} (\text{P-max}^\infty(\sigma) \otimes \text{P-max}^\infty(\tau))$$

By Prop. 33.1.2 it follows that:

$$\text{P-max}_{\circlearrowleft}^\infty(\sigma \otimes \tau) \supseteq_L \text{P-max}_{\circlearrowleft}^\infty(\sigma) \otimes \text{P-max}_{\circlearrowleft}^\infty(\tau) \iff \text{P-max}^\infty(\sigma \otimes \tau) \supseteq_S \text{P-max}^\infty(\sigma) \otimes \text{P-max}^\infty(\tau)$$

We argue that $s|_{A \multimap C, -} \notin \text{P-max}(\sigma)$ or $s|_{-, B \multimap D} \notin \text{P-max}(\tau)$ implies $s \notin \text{P-max}(\sigma \otimes \tau)$. First, suppose $s|_{A \multimap C, -}$ is finite and assume that there is a P-move $m \in A \multimap C$ such that $s|_{A \multimap C, -} \cdot m \in \sigma$. Then, the last move by $\Upsilon(m)$ in $s|_{A \multimap C, -}$ must be an O-move. But if

that is the case, then this must be the last move by $\Upsilon(m)$ in s , by the switching condition for $- \otimes -$. But if that is the case, then either s is finite and $s \cdot m \in \sigma \otimes \tau$, or s is infinite, and $\Upsilon(m)$ has no $\Upsilon(m)$:P-maximal extension in s . Otherwise, $s \upharpoonright_{A \multimap C, -}$ is infinite, in which case s is also infinite. So, assume $s \upharpoonright_{A \multimap C, -}$ is infinite but not P-maximal. Then, there is an α and a prefix $p \sqsubseteq s \upharpoonright_{A \multimap C, -}$ with no locally P-maximal extension. In particular, the last move by α in p must be an O-move. But then, by the switching condition on tensor, p also has no locally P-maximal extension in s . In all cases $s \notin \text{P-max}(\sigma \otimes \tau)$. The argument for the τ case is symmetric. $\square$

Corollary 33.7.2. *If $\langle \nu, \omega \rangle$ and $\langle \nu', \omega' \rangle$ are complete then $\langle \nu \otimes \nu', \omega \otimes \omega' \rangle$ is complete.*

Proof.

$$\text{P-max}_{\odot}^{\infty}(\nu \otimes \nu') \supseteq_L \text{P-max}_{\odot}^{\infty}(\nu) \otimes \text{P-max}_{\odot}^{\infty}(\nu') \supseteq_L \omega \otimes \omega'$$

$\square$

33.8 Parallel Implementations

Lemma 33.8.1 (cf. Lemma 26.1.2). *A tight object is nearly receptive and complete if and only if it is nearly receptive P-deterministic and complete.*

Proof. A version of this result is mentioned in [Gowers and Laird, 2018]. Our particular proof is inspired by [Girard, 1987a].

The backward direction is trivial. For the forward direction, we show that if an $M = \langle \sigma, \Sigma \rangle$ is complete then it is nearly P-deterministic and complete. The result then follows by decomposing each M_i into its own nearly P-deterministic and winning decomposition.

First, we say $s \in P_{A \multimap B}^{\text{ws}}$ is coherent with $t \in P_{A \multimap B}^{\text{ws}}$, written $s \supset t$, when the first point where they differ is at a P-position. I.e., if $p \cdot m \sqsubseteq s$ and $p \cdot n \sqsubseteq s$ and $m \neq n$ then both are O-moves, or at least one of them is a $\bullet$ -moves. Note that $\supset$ is a reflexive and symmetric relation. We call a subset $C \subseteq P_{A \multimap B}^{\text{ws}}$ a clique when it is pairwise coherent (i.e. $\forall s, t \in C. s \supset t$). It is easy to check that cliques form a partial order (ordered by inclusion $- \subseteq -$), and that moreover it is non-empty ($\{\epsilon\}$ is always a clique), and that any chain $(C_i)_{i \in I}$ of cliques has least upper bound (just take $C = \cup_{i \in I} C_i$). Therefore, by Zorn's

lemma there is such a thing as a maximal clique. It is also easy to check that if $p \sqsubseteq s$ and $s \supset t$ then $p \supset t$. In particular, $p \supset s$. Given a subset $S \subseteq P_{A \multimap B}^{\text{ws}}$ we will write $\mathcal{C}(S)$ for the domain of cliques of S .

Now, let C be a maximal clique within Σ . Then (noting that in the sequential case $\sqsupseteq_L$ is equivalent to $\sqsubseteq$):

- $\Downarrow C$ is O-receptive. Suppose $s \in \Downarrow C$ and $s \cdot m \in P_{A \multimap B}$ with m an O-move. Since $\Downarrow C \subseteq \Downarrow \Sigma \subseteq \sigma$ (by tightness) it follows that $s \in \sigma$ so that by O-receptivity $s \cdot m \in \sigma$. Now, by Prop. 33.3.4 there is some P-move n such that $s \cdot m \cdot n \in \text{P-max}_{\circ}^{\infty}(\sigma) \subseteq \Sigma$. Suppose now that $s \cdot m \cdot n \notin C$. By maximality, it follows that there is some $t \in C$ such that $s \cdot m \cdot n \asymp t$ (where, $\asymp := \neg \supset$). Now, by definition, there is some $s' \in C$ such that $s \sqsubseteq U_{\circ} s'$. In particular, $s' \supset t$. Note that since s is in P-position, and so is s' , it follows that $s \sqsubseteq s'$. If $s \cdot m \cdot n \sqsubseteq s'$, then $s \cdot m \cdot n \supset t$, a contradiction. Therefore, $s = s'$. Since $s \supset t$ and $s \cdot m \cdot n \asymp t$ it follows that $s \cdot m \sqsubseteq t$ (otherwise $s \cdot m \cdot n \supset t$). In particular, $s \cdot m \in \Downarrow C$.
- $\langle \Downarrow C, C \rangle$ is P-deterministic. Suppose $s \cdot m \in \Downarrow C \cup C$ with m a P-move. Then, assume $s \cdot n \in \Downarrow C \cup C$. Note then that it must be that $s \cdot m, s \cdot n \in C$. But then, $s \cdot m \supset s \cdot n$ and hence $m = n$ or one of them is a $\bullet$ -move.
- $\langle \Downarrow C, C \rangle$ is complete. Suppose $s \in \text{P-max}(\Downarrow C)$. In particular, there is $s' \in C$ such that $s \sqsubseteq U_{\circ} s'$. If s is in P-position then $s \sqsubseteq s'$ and hence $s \supset s'$ so that $s \in C$. Otherwise, s is in O-position so $s \cdot \circ \in \text{P-max}_{\circ}(\Downarrow C)$. If $s \cdot m \sqsubseteq s'$ for a non- $\circ$ P-move m then $s \cdot m \sqsubseteq U_{\circ} s'$ and then $s \notin \text{P-max}(\Downarrow C)$. Therefore, it must be that $s \cdot \circ \sqsubseteq s'$ and hence $s \cdot \circ \in C$. Now, if $s \in \uparrow \Downarrow C$ was infinite, we note that every prefix of s in P-position is in C , so for any $t \in C$, either $t \sqsubseteq s$ or, otherwise, let p be the maximal common prefix of s and t . Then, $p \supset t$ so $s \supset t$, and hence $s \in C$.

We claim that

$$M = \cup_{C \in \mathcal{C}(\Sigma). C \text{ is maximal}} \langle \Downarrow C, C \rangle$$

That

$$\cup_{C \in \mathcal{C}(\Sigma). C \text{ is maximal}} \langle \Downarrow C, C \rangle \sqsupseteq_O M$$

is immediate, as by definition every $C \subseteq \Sigma$, and therefore $\Downarrow C \subseteq \Downarrow \Sigma \subseteq \sigma$ by tightness.

Now, if $s \in \Sigma$ then $\{s\}$ is a clique so by Zorn's lemma there is some maximal clique $C \in \mathcal{C}(\Sigma)$ s.t. $\{s\} \subseteq C$. Moreover, if $s \in \sigma$ then there is an extension $s' \in \text{P-max}_{\circlearrowleft}(\sigma)$ such that $s \sqsubseteq s'$. But then, by completeness, $s' \in \Sigma \subseteq C$ for some maximal C (see above). Then, $s \in \Downarrow s' \subseteq \Downarrow C$. $\square$

Proposition 33.8.1 (cf. Proposition 26.1.3). *Sequential implementations compose.*

Proof. Let $M = \cup_{i \in I} \langle \sigma_i, \Sigma_i \rangle$ and $N = \cup_{j \in J} \langle \tau_j, \mathbf{T}_j \rangle$ be the deterministic decompositions of implementations $M : A \multimap B$ and $N : B \multimap C$. Then, by join-preservation:

$$M ;; N = \cup_{i \in I} \cup_{j \in J} \langle \sigma_i, \Sigma_i \rangle ;; \langle \tau_j, \mathbf{T}_j \rangle$$

so it is enough to show that each

$$\langle \sigma_i, \Sigma_i \rangle ;; \langle \tau_j, \mathbf{T}_j \rangle$$

is a deterministic implementation.

So let instead $\langle \sigma, \Sigma \rangle$ and $\langle \tau, \mathbf{T} \rangle$ be deterministic implementations.

Receptivity: First, let $s \in \sigma ; \tau$ and assume m is an O-move such that $s \cdot m \in P_{A \multimap C}^{\leq \infty}$. So let s' be the play in $s \circledast \tau$ witnessing the membership of s in $\sigma ; \tau$. There are two cases to consider depending on whether m is an O-move in A or C . If it is in C , by the switching condition the last move in s' (if it exists) is a P-move in C and therefore $s' \cdot m$ is well-formed. Since τ is O-receptive, it follows that $s' \cdot m \in \sigma \circledast \tau$. Hence, $s \cdot m \in \sigma ; \tau$. The argument is analogous if m is a move in A (note that it would be a P-move in A , and therefore it still is an O-move in $\sigma : A \multimap B$). Now, if m is instead a $\bullet$ -move then note that $s' \cdot \bullet$ witnesses that $s \cdot \bullet \in \sigma ; \tau$ by $\bullet$ -receptivity of both σ and τ .

Complete: By Prop. 26.1.1:

$$\text{P-max}_{\circlearrowleft}^{\infty}(\sigma ; \tau) = \text{P-max}_{\circlearrowleft}^{\infty}(\sigma) \circledast \text{P-max}_{\circlearrowleft}^{\infty}(\tau) \sqsupseteq_L \Sigma \circledast \mathbf{T}$$

Determinism: By Prop. 33.8.2 and the fact that we already showed that their composition is complete, it is enough to separately show that $\sigma ; \tau$ and $\Sigma \not\prec T$ are deterministic. Moreover, by Prop. 26.1.1 $\Sigma = \text{P-max}_{\odot}^{\infty}(\sigma)$ and $T = \text{P-max}_{\odot}^{\infty}(\tau)$. In particular,

$$\Downarrow \text{P-max}_{\odot}^{\infty}(\sigma) \not\prec \text{P-max}_{\odot}^{\infty}(\tau) \supseteq_S \Downarrow \text{P-max}_{\odot}^{\infty}(\sigma) ; \Downarrow \text{P-max}_{\odot}^{\infty}(\tau) = \sigma ; \tau$$

by Prop. 27.1.1 and Prop. 25.3.1. But then, by Prop. 33.8.2 it is enough to show $\Sigma \not\prec T$ is deterministic.

So let $s, s' \in \Sigma \odot T$. Let $p \sqsubseteq s, s'$ be their maximal common prefix. If $p = s$ or $p = s'$ it is immediate that $s \supset s'$. Therefore, let $p \cdot m \sqsubseteq s$ and $p \cdot n \sqsubseteq s'$ with m and n P-moves (otherwise, $s \supset s'$ once more). Let s_m and s_n be the unique witnesses of the memberships $p \cdot m, p \cdot n \in \sigma \odot T$ as described by Lemma 33.8.2. If either witness s_x (with $x \in \{m, n\}$) is such that s_x is an interaction with an infinite suffix of only B moves and $s_x \upharpoonright_{A,-,C} \cdot \odot_C = p \cdot x$ then we have $x = \odot_C$ and $s_x \upharpoonright_{A,-,C} = p$. Meanwhile we have $s_{\neg x} \upharpoonright_{A,-,C} = p \cdot (\neg x)$ or $s_{\neg x} \upharpoonright_{A,-,C} \cdot \odot_C = p \cdot (\neg x)$ (where $\neg x$ is whichever of m and n x is not). In the second case we are done, as $m = n = \odot_C$. In the first case, it follows that there is some prefix $s' \sqsubseteq s_{\neg x}$ such that $s' \upharpoonright_{A,-,C} = p$. But then, by minimality of s_x it follows that $s_x \sqsubseteq s' \sqsubseteq s_{\neg x}$. Since s_x has an infinite suffix in B then so does $s_{\neg x}$. But then either $s_{\neg x} \upharpoonright_{A,B,-} \cdot (\neg x)$ or $s_{\neg x} \upharpoonright_{-,B,C} \cdot (\neg x)$ is not well-defined, a contradiction.

So consider the remaining case that both $s \cdot m \in \Sigma ; T$ and $s \cdot n \in \Sigma ; T$. Let p' be the largest common prefix of s_m and s_n (note that one exists as ϵ is a common prefix). There are four cases to consider, depending on whether $p' \sqsubset s_x$ or $p' = s_x$ for $x \in \{m, n\}$. Note that if $p' = s_m$ and $p' = s_n$ then $s_m = s_n$ and then $m = n$ (as they end with m and n). By swapping the roles of m and n if necessary, there are only two other cases to consider. Either $s_m = p' \sqsubset s_n$ or $p' \sqsubset s_m$ and $p' \sqsubset s_n$. If $s_m = p' \sqsubset s_n$ then, there exists p'' such that $p' = p'' \cdot m$ and there is t such that $s_n = p'' \cdot m \cdot t$. Moreover, $s_m \upharpoonright_{A,-,C} = p \cdot m$. Therefore, $s_n \upharpoonright_{A,-,C} = p \cdot m \cdot t \upharpoonright_{A,-,C}$. Then $p \cdot m \cdot t \upharpoonright_{A,-,C} = s_n \upharpoonright_{A,-,C} = p \cdot n$. But then, $m = n$. Assume now that $p' \sqsubset s_m$ and $p' \sqsubset s_n$. Then, there are $m' \neq n'$ such that $p' \cdot m' \sqsubseteq s_m$ and $p' \cdot n' \sqsubseteq s_n$. Note that by

well-formedness, m' and n' appear in the same position. Hence, if m' is a move in B , or a P-move in C , or an O-move in A then it is either a P-move in $A \multimap B$ or a P-move in $B \multimap C$, in which case $m' = n'$ by determinism of either σ or τ . Otherwise it could be an O-move in C or a P-move in A . In that case, by the switching condition, both m' and n' must be moves in the same component (C or A), and hence $p' \upharpoonright_{A,-,C} \cdot m' \sqsubseteq s \cdot m$ and $p' \upharpoonright_{A,-,C} \cdot n' \sqsubseteq p \cdot n$ so $m' = n'$, a contradiction. If $m' = \bullet$ (the case for $n' = \bullet$ is analogous) then $p' \upharpoonright_{A,-,C} \cdot \bullet \sqsubseteq p \cdot m$ and by well-formedness $m = \bullet$ contradicting that it is a P-move. In all cases, we either reach that $m = n$ or a contradiction, so determinism follows. □

Corollary 33.8.1. *Given a functor $F : \mathbf{Seq}_{\text{det}} \rightarrow \mathbf{Seq}_{\text{det}}$, the operation F^Φ defines a functor $F^\Phi : \mathbf{UImpl} \rightarrow \mathbf{UImpl}$.*

Proof. That $\mathcal{P}(F)$ is indeed a functor follows from:

$$\mathcal{P}(F) S ; \mathcal{P}(F) T = \cup_{\sigma \in S} \cup_{\tau \in T} F \sigma ; F \tau = \cup_{\sigma \in S} \cup_{\tau \in T} F (\sigma ; \tau) = \mathcal{P}(F) (S ; T)$$

Note that when F is join-preserving we have:

$$\mathcal{P}(F) \cup_{i \in I} S_i = \cup_{\sigma \in \cup_{i \in I} S_i} F \sigma = \cup_{i \in I} \cup_{\sigma \in S_i} F \sigma = \cup_{i \in I} \mathcal{P}(F)(S_i)$$

Then, by Prop. 26.2.1 and Prop. 26.1.1:

$$\begin{aligned}
F^\Phi M ; ; F^\Phi N &= (\|_{\alpha \in \Upsilon} \mathcal{O}(\mathcal{P}(F) S[\alpha])) ; ; (\|_{\alpha \in \Upsilon} \mathcal{O}(\mathcal{P}(F) T[\alpha])) \\
&= \|_{\alpha \in \Upsilon} (\mathcal{O}(\mathcal{P}(F) S[\alpha]) ; ; \mathcal{O}(\mathcal{P}(F) T[\alpha])) \\
&= \|_{\alpha \in \Upsilon} \mathcal{O}(\mathcal{P}(F) S[\alpha] ; \mathcal{P}(F) T[\alpha]) \\
&= \|_{\alpha \in \Upsilon} \mathcal{O}(\mathcal{P}(F) (S[\alpha] ; T[\alpha])) \\
&= F^\Phi \|_{\alpha \in \Upsilon} \mathcal{O}(S[\alpha] ; T[\alpha]) \\
&= F^\Phi \|_{\alpha \in \Upsilon} (\mathcal{O}(S[\alpha]) ; ; \mathcal{O}(T[\alpha])) \\
&= F^\Phi (\|_{\alpha \in \Upsilon} \mathcal{O}(S[\alpha]) ; ; \|_{\alpha \in \Upsilon} \mathcal{O}(T[\alpha])) \\
&= F^\Phi (M ; ; N)
\end{aligned}$$

□

Proposition 33.8.2. *Suppose $\kappa : F \Rightarrow G$ is a natural transformation between endofunctors in $\mathbf{Seq}_{\mathbf{det}}$. Then, $\mathcal{O}(\kappa)^\Phi$ is a natural transformation $\mathcal{O}(\kappa)^\Phi : F^\Phi \Rightarrow G^\Phi$ in $\mathbf{UImpl}$.*

Proof. First, we show $\mathcal{P}(\kappa) : \mathcal{P}(F) \Rightarrow \mathcal{P}(G)$. Indeed, let $S : A \multimap B \in \mathcal{P}(\mathbf{Seq}_{\mathbf{det}})$.

$$\mathcal{P}(\kappa_A) ; \mathcal{P}(G) S = \{\kappa_A\} ; \cup_{\sigma \in S} G \sigma = \cup_{\sigma \in S} \kappa_A ; G \sigma = \cup_{\sigma \in S} F \sigma ; \kappa_B = \mathcal{P}(F) S ; \mathcal{P}(\kappa_B)$$

Let $M : A \multimap B$ decompose as:

$$M = \|_{\alpha \in \Upsilon} \mathcal{O}(S[\alpha])$$

By Prop. 26.1.1, Prop. 26.2.1, Prop. 26.2.1 and Cor. 33.8.1:

$$\begin{aligned}
\kappa_A^\Phi \circ G^\Phi M &= \|_{\alpha \in \Upsilon} \iota_\alpha(\mathcal{O}(\kappa_A)) \circ \mathcal{O}(G S[\alpha]) \\
&= \|_{\alpha \in \Upsilon} \mathcal{O}(\iota_\alpha(\kappa_A)) \circ \mathcal{O}(G S[\alpha]) \\
&= \|_{\alpha \in \Upsilon} \mathcal{O}(\iota_\alpha(\kappa_A) ; G S[\alpha]) \\
&= \|_{\alpha \in \Upsilon} \mathcal{O}(F S[\alpha] ; \iota_\alpha(\kappa_B)) \\
&= F^\Phi M \circ \kappa_B^\Phi
\end{aligned}$$

□

Lemma 33.8.2.

- If σ is deterministic then so is $\text{P-max}_{\odot}^{\infty}(\sigma)$.
- If Σ is deterministic then so is $\Downarrow\Sigma$.
- If $\langle\sigma, \Sigma\rangle$ is complete then it is deterministic if and only if σ and Σ are both deterministic.

Proof.

- Suppose $s \cdot m \cdot t \in \text{P-max}_{\odot}^{\infty}(\sigma)$ and $s \cdot n \cdot t' \in \text{P-max}_{\odot}^{\infty}(\sigma)$ with s in O-position. If either m or n is not a $\odot$ -token, then s is not P-maximal so neither can be a $\odot$ -token, and hence $s \cdot m \in \sigma$ and $s \cdot n \in \sigma$ so $m = n$. If either one is a $\odot$ -token, then s is P-maximal, so both are $\odot$ -tokens.
- Suppose $s \cdot m \in \Downarrow\Sigma$ and $s \cdot n \in \Downarrow\Sigma \cup \Sigma$. Note then that in any case $s \cdot m, s \cdot n \in \Sigma$ so $s \cdot m \subset s \cdot n$ so $m = n$.
- The forward direction is straight-forward. The interesting case for the backward direction is when $s \cdot m \in \sigma$ and $s \cdot n \in \Sigma$. In this case, we have $s \cdot m \in \text{P-max}_{\odot}^{\infty}(\sigma) \subseteq \Sigma$ so the claim follows from the determinism of Σ .

□

Lemma 33.8.3. *Let $\sigma : A \multimap B$ and $\tau : B \multimap C$ be deterministic sequential strategies. Then, if $s \in \sigma; \tau$ then there exists a unique $s' \in \sigma \circ \tau$ minimal under prefixing such that $s' \upharpoonright_{A, -, C} = s$.*

Proof. The existence of some such s' follows immediately from the definition of $-\circ-$. So let s' and t' be two such candidates. Let p' be the largest common prefix of s' and t' and, for the sake of contradiction, assume $p' \neq s'$ (the case for $p' \neq t'$ is symmetric). Since p' is a strict prefix of s' it follows that there is some move m such that $p' \cdot m \sqsubseteq s'$. Now, if $p' = t'$ then we have $t' \cdot m \sqsubseteq s'$. Then s' is not minimal. Hence, p' is also a strict prefix of t' , and

therefore there is a move n such that $p' \cdot n \sqsubseteq t'$. Note that m and n are moves that can happen in the same position. Note then that we have $p' \upharpoonright_{A,-,C} \cdot m \upharpoonright_{A,-,C} = p' \upharpoonright_{A,-,C} \cdot n \upharpoonright_{A,-,C}$ as $s' \upharpoonright_{A,-,C} = t' \upharpoonright_{A,-,C}$. Hence, $m \upharpoonright_{A,-,C} = n \upharpoonright_{A,-,C}$. Because of this, either $m = n$, in which case we contradict the maximality of p' , or both m and n are moves in B . If they are both moves in B , since they occur in the same position, they must be either both O- or both P-moves in B . In the first case, note that m and n are P-moves in τ so that $m = n$ by P-determinism, and in the later $m = n$ as they are P-moves in σ . $\square$

Corollary 33.8.2. *Let $\Sigma : A \multimap B$ and $T : B \multimap C$ be deterministic sequential win conditions. Then, if $s \in \Sigma \circ T$ then there exists a unique $s' \in \Sigma \circ T$ minimal under prefixing such that either $s' \upharpoonright_{A,-,C} = s$ or $s' \upharpoonright_{A,-,C} \cdot \circ = s$ and s' has an infinite suffix in B .*

Proof. Suppose $s \in \Sigma \circ T$ and let $s' \in \Sigma \circ T$ be a minimal witness. If both $s' \upharpoonright_{A,B,-}$ and $s' \upharpoonright_{-,B,C}$ are finite, this is the same as Prop. 33.8.3. If s' is infinite, then we note that for every $p \sqsubseteq s'$, we have $p \in \Downarrow \Sigma \circ \Downarrow T$. By Lemma 33.8.2 we can take a chain $p_1 \sqsubseteq p_2 \sqsubseteq \dots \sqsubseteq s'$ such that every p_i ends with a move in A or C , and is therefore minimal, so that they are unique by Lem. 33.8.3. Hence, s is unique. Finally, suppose $s' \upharpoonright_{A,-,C}$ is finite but $s' \upharpoonright_{A,B,-}$ or $s' \upharpoonright_{-,B,C}$ is infinite. In this case, it must be that s' ends with infinite chattering in B , so that $s = s' \upharpoonright_{A,-,C} \cdot \circ$ by definition of $- \circ -$. But, in that case, we can use the same argument as the infinite case to obtain that the s' itself is unique. $\square$

Proposition 33.8.3 (cf. Proposition 26.1.1). *Suppose σ and τ are deterministic implementations. Then,*

$$\text{P-max}_{\circ}^{\infty}(\sigma; \tau) = \text{P-max}_{\circ}^{\infty}(\sigma) \circ \text{P-max}_{\circ}^{\infty}(\tau)$$

Proof. By Prop. 27.1.1 we have that $\text{P-max}_{\circ}(\sigma; \tau) \subseteq \text{P-max}_{\circ}^{\infty}(\sigma) \circ \text{P-max}_{\circ}^{\infty}(\tau)$. So it is enough to prove that $\uparrow(\sigma; \tau) \subseteq \text{P-max}_{\circ}^{\infty}(\sigma) \circ \text{P-max}_{\circ}^{\infty}(\tau)$.

Let $s \in \uparrow(\sigma; \tau) \subseteq \sigma$. It follows there is a sequence of P-maximal plays $p_1 \sqsubseteq p_2 \sqsubseteq \dots \sqsubseteq s$ such that each $p_i \in \sigma; \tau$. By taking minimal witnesses p'_i as implied by Lem. 33.8.3 we obtain a chain $p'_1 \sqsubseteq \dots p'_i \dots$ by uniqueness, so that $s' = \sup s$ must satisfy $s' \upharpoonright_{A,-,C} = s$ or $s' \upharpoonright_{A,-,C} \cdot \circ = s$. In particular, we must have $s' \in \text{P-max}_{\circ}^{\infty}(\sigma) \circ \text{P-max}_{\circ}^{\infty}(\tau)$, or $s' \cdot \circ_A \cdot \circ_B \cdot \circ_C \in \text{P-max}_{\circ}^{\infty}(\sigma) \circ \text{P-max}_{\circ}^{\infty}(\tau)$ and the claim follows.

Now, assume $s \in \text{P-max}_{\odot}^{\infty}(\sigma) \circledast \text{P-max}_{\odot}^{\infty}(\tau)$. In particular, $s \in \sigma \circledast \tau$, so let $s' \in \sigma \circledast \tau$ be the witness provided by Lem. 33.8.2. If s is infinite we are done, as it is immediate that $s \in \text{P-max}_{\odot}^{\infty}(\sigma \circledast \tau)$. Otherwise, let s be finite. If s' is finite, then since both $s' \upharpoonright_{A,B,-}$ and $s' \upharpoonright_{-,B,C}$ end with P-moves (or $\bullet$ -moves), then $s' \upharpoonright_{A,-,C}$ ends with a P-move (or a $\bullet$ -move). Otherwise, s' has an infinite suffix in B , in which case, by definition of $- \circledast -$ s ends with $\odot_C$ and is therefore P-maximal. In all cases $s \in \text{P-max}(\sigma \circledast \tau)$. $\square$

Proposition 33.8.4 (cf. Proposition 26.1.2). *Given $\sigma : A \multimap B \in \mathbf{Seq}_{\text{det}}$, the mapping $\sigma \mapsto \mathcal{O}(\sigma) := \langle \sigma^{\bullet}, \text{P-max}_{\odot}^{\infty}(\sigma^{\bullet}) \rangle$ is functorial, and its image always defines a deterministic implementation.*

Proof. For convenience, let $\Sigma = \text{P-max}_{\odot}^{\infty}(\sigma^{\bullet})$.

That $\sigma^{\bullet}$ is O-receptive, $\bullet$ -receptive is almost immediate.

Now let $s \in \sigma^{\bullet}; \tau^{\bullet}$. Let s' be the witness of $s \in \sigma^{\bullet}; \tau^{\bullet}$. Now, if $\bullet$ appears in $s = s' \upharpoonright_{A,-,C}$ then by definition of the projections and well-formedness $s' = p' \cdot \bullet$ for some p' . Note then that $p' \in \sigma \circledast \tau$ and then $s = p' \upharpoonright_{A,-,C} \cdot \bullet \in (\sigma; \tau)^{\bullet} \subseteq (\sigma \circledast \tau)^{\bullet}$. If $\bullet \notin s$ then $s \in \sigma; \tau \subseteq (\sigma; \tau)^{\bullet}$.

If $s \in (\sigma; \tau)^{\bullet}$ then either $s \in \sigma; \tau \subseteq \sigma^{\bullet}; \tau^{\bullet}$ or $s = p \cdot \bullet$ and $p \in \sigma; \tau$. So let p' witness that $p \in \sigma; \tau$. It is easy then to check that $p' \cdot \bullet$ then witnesses that $s \in \sigma^{\bullet}; \tau^{\bullet}$.

By double inclusion we have $\sigma^{\bullet}; \tau^{\bullet} = (\sigma; \tau)^{\bullet}$.

Prop. 26.1.1 shows that $\text{P-max}_{\odot}^{\infty}(-)$ is functorial in this case too, and the claim follows. $\square$

Proposition 33.8.5. *Deterministic decompositions of implementations are unique up-to bijection of indexing sets.*

Proof. Let $M = \langle \sigma, \Sigma \rangle$ and let $M = \cup_{i \in I} M_i$ be a deterministic decomposition of M . We show that any $M_i = \langle \Downarrow C, C \rangle$ where C is a maximal clique of Σ in the sense of the proof of Prop. 26.1.2. So let $M_i = \langle \tau, T \rangle$ and let $s \in \Sigma$ be pairwise coherent with every $t \in T$. Then, let p be the maximal $p \in T$ such that $p \sqsubseteq s$ (note that since $\epsilon \in \tau$, $\epsilon \in T$). If p is in $\bullet$ -position or $\odot$ -position, then $p \sqsubseteq s \implies s = p$. Hence, $s \in T$. So let p be in P-position. Then, either $p = s$ and $s \in T$ or there is an O-move or $\bullet$ -move such that $p \cdot m \sqsubseteq s$. If $m = \bullet$ then by receptivity we have $p \cdot m \in \tau$ and therefore $p \cdot m \in T$. If m is an O-move

then similarly we have $p \cdot m \in \tau$ by receptivity, so that there is some P-move n such that $p \cdot m \cdot n \in T$ by completeness. Now, $s \supset p \cdot m \cdot n$. So $p \cdot m \cdot n \sqsubseteq s$, contradicting the maximality of p . Therefore, $s \in T$. Hence, T must be a maximal clique of Σ . But $T = \text{P-max}_{\circlearrowleft}^{\infty}(\tau)$ by Prop. 26.1.1. So $\tau = \Downarrow \text{P-max}_{\circlearrowleft}^{\infty}(\tau) = \Downarrow T$ by Prop. 25.3.1. Hence, the decomposition is unique up-to a bijection over the indexing set. $\square$

Proposition 33.8.6. *For every deterministic implementation M there exists $\sigma \in \mathbf{Seq}_{\text{det}}$ such that $\mathcal{O}(\sigma) = M$.*

Proof. Note that by Prop. 26.1.1 we have $M = \langle \sigma', \text{P-max}_{\circlearrowleft}^{\infty}(\sigma') \rangle$. Note that since σ' is $\bullet$ -receptive, $\sigma' = (\sigma' \setminus \bullet)^{\bullet}$. So let $\sigma = \sigma' \setminus \bullet$. It is enough to argue that σ is a strategy in $\mathbf{Seq}_{\text{det}}$. But that it is non-empty, prefix-closed, O-receptive and P-deterministic follows immediately from the corresponding properties of σ' , as they are not affected by removing $\bullet$ -tokens. $\square$

Proposition 33.8.7. *$\mathcal{O}(\text{seqcopy}_A)$ is the neutral element for sequential implementations.*

Proof. seqcopy_A is the neutral element for O-receptive sequential strategies as seen in Corollary 6.3.1, and in particular it follows that $\{\text{seqcopy}_A\}$ is the neutral element for $\mathcal{P}(\mathbf{Seq}_{\text{det}})$. Now, given $M : A \multimap B$ a sequential implementation, we have that since $\mathcal{O}(-)$ is fully faithful, there is a set $S \subseteq \mathbf{Seq}_{\text{det}}$ such that $\mathcal{O}(S) = M$. Hence:

$$\begin{aligned} \mathcal{O}(\text{seqcopy}_A) ; ; M ; ; \mathcal{O}(\text{seqcopy}_B) &= \mathcal{O}(\text{seqcopy}_A) ; ; \mathcal{O}(S) ; ; \mathcal{O}(\text{seqcopy}_B) \\ &= \mathcal{O}(\text{seqcopy}_A ; S ; \text{seqcopy}_B) \\ &= \mathcal{O}(S) \\ &= M \end{aligned}$$

$\square$

Proposition 33.8.8 (cf. Proposition 26.2.1). *For sets of plays*

$$S = \|\alpha \in \Upsilon S[\alpha] \subseteq P_{A \multimap B}^{\leq \infty} \qquad T = \|\alpha \in \Upsilon T[\alpha] \subseteq P_{B \multimap C}^{\leq \infty}$$

it holds that $S \circ T = \parallel_{\alpha \in \Upsilon} S[\alpha] \circ T[\alpha]$.

Proof. Note that:

$$S \circ T = F_{\circ} ((U_{\circ} (\parallel_{\alpha \in \Upsilon} S[\alpha]); U_{\circ} (\parallel_{\alpha \in \Upsilon} T[\alpha]))$$

So if we can show that $U_{\circ}$ – commutes with parallel composition we obtain that:

$$F_{\circ} ((U_{\circ} (\parallel_{\alpha \in \Upsilon} S[\alpha]) ; U_{\circ} (\parallel_{\alpha \in \Upsilon} T[\alpha])) = F_{\circ} (((\parallel_{\alpha \in \Upsilon} U_{\circ} S[\alpha]) ; (\parallel_{\alpha \in \Upsilon} U_{\circ} T[\alpha]))$$

Essentially the same argument as Prop. 12.1.2, extended to infinite plays in the usual way, gives that:

$$F_{\circ} (((\parallel_{\alpha \in \Upsilon} U_{\circ} S[\alpha]) ; (\parallel_{\alpha \in \Upsilon} U_{\circ} T[\alpha]))) = F_{\circ} \parallel_{\alpha \in \Upsilon} U_{\circ} S[\alpha] \circ U_{\circ} T[\alpha]$$

Finally, if we show $F_{\circ}$ – commutes with parallel composition we get to:

$$F_{\circ} \parallel_{\alpha \in \Upsilon} U_{\circ} S[\alpha] \circ U_{\circ} T[\alpha] = \parallel_{\alpha \in \Upsilon} F_{\circ} (U_{\circ} S[\alpha] \circ U_{\circ} T[\alpha]) = \parallel_{\alpha \in \Upsilon} S[\alpha] \circ T[\alpha]$$

which is the desired claim. Therefore, we spend the rest of the proof showing $F_{\circ}$ – and $U_{\circ}$ – commute.

Note that

$$\begin{aligned} s \in U_{\circ} \parallel_{\alpha \in \Upsilon} S[\alpha] &\iff s \setminus (\cup_{\alpha \in \Upsilon} \alpha \circ) \in \parallel_{\alpha \in \Upsilon} S[\alpha] \\ &\iff \forall \alpha \in \Upsilon. \pi_{\alpha}(s \setminus (\cup_{\alpha \in \Upsilon} \alpha \circ)) \in S[\alpha] \\ &\iff \forall \alpha \in \Upsilon. \pi_{\alpha}(s) \setminus \alpha \circ \in S[\alpha] \\ &\iff \forall \alpha \in \Upsilon. \pi_{\alpha}(s) \in U_{\circ} S[\alpha] \\ &\iff s \in \parallel_{\alpha \in \Upsilon} U_{\circ} S[\alpha] \end{aligned}$$

So

$$U_{\circ} \parallel_{\alpha \in \Upsilon} S[\alpha] = \parallel_{\alpha \in \Upsilon} U_{\circ} S[\alpha]$$

Now, assuming all $S[\alpha]$ are $\odot$ -token free:

$$\begin{aligned}
s \in F_{\odot} \parallel_{\alpha \in \Upsilon} S[\alpha] &\iff \exists s' \in \parallel_{\alpha \in \Upsilon} S[\alpha]. s' = U_{\odot} s \\
&\iff \exists s'. (\forall \alpha \in \Upsilon. \pi_{\alpha}(s') \in S[\alpha]) \wedge s' = U_{\odot} s \\
&\iff \forall \alpha \in \Upsilon. \pi_{\alpha}(U_{\odot} s) \in S[\alpha] \\
&\iff \forall \alpha \in \Upsilon. U_{\odot} \pi_{\alpha}(s) \in S[\alpha] \\
&\iff \forall \alpha \in \Upsilon. s \in F_{\odot} S[\alpha]
\end{aligned}$$

and hence

$$F_{\odot} \parallel_{\alpha \in \Upsilon} S[\alpha] = \parallel_{\alpha \in \Upsilon} F_{\odot} S[\alpha]$$

□

Proposition 33.8.9 (cf. Proposition 26.2.3). *For any parallel strategy $\sigma = \parallel_{\alpha \in \Upsilon} \sigma[\alpha]$, $\text{P-max}_{\odot}^{\infty}(\sigma) = \parallel_{\alpha \in \Upsilon} \text{P-max}_{\odot}^{\infty}(\sigma[\alpha])$.*

Proof. By the proof of Prop. 26.2.1:

$$\text{P-max}_{\odot}^{\infty}(\sigma) = F_{\odot} \text{P-max}^{\infty}(\sigma)$$

and similarly:

$$\parallel_{\alpha \in \Upsilon} \text{P-max}_{\odot}^{\infty}(\sigma[\alpha]) = \parallel_{\alpha \in \Upsilon} F_{\odot} \text{P-max}^{\infty}(\sigma[\alpha]) = F_{\odot} \parallel_{\alpha \in \Upsilon} \text{P-max}^{\infty}(\sigma[\alpha])$$

By Prop. 33.1.2 it follows that

$$\text{P-max}_{\odot}^{\infty}(\sigma) = \parallel_{\alpha \in \Upsilon} \text{P-max}_{\odot}^{\infty}(\sigma[\alpha]) \iff \text{P-max}^{\infty}(\sigma) = \parallel_{\alpha \in \Upsilon} \text{P-max}^{\infty}(\sigma[\alpha])$$

First, let $s \in \parallel_{\alpha \in \Upsilon} \text{P-max}_{\odot}^{\infty}(\sigma[\alpha]_i)$ and let $p \sqsubseteq s$. Then, either p is already α :P-maximal, or, otherwise, we note that $\pi_{\alpha}(s)$ is P-maximal. In particular, there is some prefix $p' \sqsubseteq s$ such that $p \sqsubseteq p'$ and p' is α :P-maximal. Indeed, if $\pi_{\alpha}(s)$ is infinite then this is true by taking any extension of p such that ends with a P-move by α . If it is finite, then we note that we may take any p' such that $\pi_{\alpha}(p') = \pi_{\alpha}(s)$. Now, by assumption, there is no P-move

m such that $\pi_\alpha(p') \cdot m \in \sigma[\alpha]$. By well-formedness, there cannot be any P-move m in $\sigma[\alpha]$ such that $\pi_\alpha(p') \cdot m \in \sigma[\alpha]$. It then follows that there is no P-move m such that $p' \cdot m \in \sigma$ by definition of a parallel strategy.

Now, suppose $s \in \text{P-max}^\infty(\sigma)$. Let $p_\alpha \sqsubseteq \pi_\alpha(s)$. Then, there is some minimal prefix $p \sqsubseteq s$ such that $\pi_\alpha(p) = p_\alpha$. But then, there is some extension p' such that no P-move $\alpha:m$ satisfies that $p' \cdot \alpha:m \in \sigma$. But suppose there was a P-move m such that $\pi_\alpha(p') \cdot m \in \sigma[\alpha]$. Then, by definition of a parallel strategy, it follows that $p' \cdot \alpha:m \in \sigma$, a contradiction. Hence $p'_\alpha = \pi_\alpha(p')$ satisfies the desired condition for P-maximality in $\sigma[\alpha]$. $\square$

Corollary 33.8.3 (cf. Corollary 26.2.3). *Any concurrent implementation is nearly complete.*

Proof. Let

$$M = \parallel_{\alpha \in \Upsilon} \cup_{i \in I[\alpha]} \langle \sigma_i, \Sigma_i \rangle$$

be the parallel-deterministic decomposition of a concurrent implementation M . By Prop. 26.2.2 we may commute the two forms of non-determinism, coming from the parallel composition and the choice:

$$M = \cup_{i \in I} \parallel_{\alpha \in \Upsilon} \langle \sigma_{i(\alpha)}, \Sigma_{i(\alpha)} \rangle = \cup_{i \in I} \langle \parallel_{\alpha \in \Upsilon} \sigma_{i(\alpha)}, \parallel_{\alpha \in \Upsilon} \Sigma_{i(\alpha)} \rangle$$

where we let $I = \prod_{\alpha \in \Upsilon} I[\alpha]$.

By Prop. 26.2.3 it follows that for all $i \in I$:

$$\text{P-max}_\odot^\infty(\parallel_{\alpha \in \Upsilon} \sigma_{i(\alpha)}) = \parallel_{\alpha \in \Upsilon} \text{P-max}_\odot^\infty(\sigma_{i(\alpha)}) \supseteq_L \parallel_{\alpha \in \Upsilon} \Sigma_{i(\alpha)}$$

and therefore M is nearly complete. $\square$

Proposition 33.8.10 (cf. Proposition 26.2.4). *Concurrent implementations define a category.*

Proof. Let $M = \parallel_{\alpha \in \Upsilon} M[\alpha]$ and $N = \parallel_{\alpha \in \Upsilon} N[\alpha]$ by concurrent implementations. Then, by

Prop. 26.2.1:

$$M ; ; N = (\|_{\alpha \in \Upsilon} M[\alpha]) ; ; (\|_{\alpha \in \Upsilon} N[\alpha]) = \|_{\alpha \in \Upsilon} (M[\alpha] ; ; N[\alpha])$$

and then by Prop. 26.1.3 it follows that $M ; ; N$ is a concurrent implementation.

Moreover, let $O_{\text{copy}} = \mathcal{O}(\text{seqcopy})$. Then:

$$\begin{aligned} \langle \text{copy}_A, \text{eqcopy}_A \rangle ; ; M ; ; \langle \text{copy}_B, \text{eqcopy}_B \rangle &= \|_{\alpha \in \Upsilon} O_{\text{copy}} ; ; \|_{\alpha \in \Upsilon} M[\alpha] ; ; \|_{\alpha \in \Upsilon} O_{\text{copy}} \\ &= \|_{\alpha \in \Upsilon} (\|_{\alpha \in \Upsilon} O_{\text{copy}} ; ; M[\alpha] ; ; \|_{\alpha \in \Upsilon} O_{\text{copy}}) \\ &= \|_{\alpha \in \Upsilon} M[\alpha] \\ &= M \end{aligned}$$

□

33.9 Composing Completeness and Tightness

Proposition 33.9.1 (cf. Proposition 27.1.1).

- If $\sigma : A \multimap B$ is sufficiently $\mathcal{O}$ -receptive and $\tau : B \multimap C$ is $\mathcal{O}$ -receptive then

$$\text{P-max}_{\mathcal{O}}(\sigma; \tau) \sqsubseteq_L \text{P-max}_{\mathcal{O}}^\infty(\sigma) \circledast \text{P-max}_{\mathcal{O}}^\infty(\tau)$$

- Similarly, if $\sigma : A \multimap B$ is $\mathcal{O}$ -receptive and $\tau : B \multimap C$ is sufficiently $\mathcal{O}$ -receptive then

$$\text{P-max}_{\mathcal{O}}(\sigma; \tau) \sqsubseteq_L \text{P-max}_{\mathcal{O}}^\infty(\sigma) \circledast \text{P-max}_{\mathcal{O}}^\infty(\tau)$$

- For any $\Sigma :: A \multimap B$ and $T :: B \multimap C$:

$$\Downarrow(\Sigma \circledast T) \sqsubseteq_S \Downarrow \Sigma ; \Downarrow T$$

Proof. Let $\text{P-max}^\infty(\sigma) := U_{\mathcal{O}} \text{P-max}_{\mathcal{O}}^\infty(\sigma)$. Moreover, by “P-maximal” we will include as well the infinite plays of $\text{P-max}^\infty(\sigma)$ when we refer to them below.

- We note first that, by Lemma 33.1.1

$$\begin{aligned}
\text{P-max}_{\circlearrowleft}^{\infty}(\sigma) \circledast \text{P-max}_{\circlearrowleft}^{\infty}(\tau) &= F_{\circlearrowleft} \text{P-max}^{\infty}(\sigma) \circledast F_{\circlearrowleft} \text{P-max}^{\infty}(\tau) \\
&= F_{\circlearrowleft} (U_{\circlearrowleft} F_{\circlearrowleft} \text{P-max}^{\infty}(\sigma) ; U_{\circlearrowleft} F_{\circlearrowleft} \text{P-max}^{\infty}(\tau)) \\
&= F_{\circlearrowleft} (\text{P-max}^{\infty}(\sigma) ; \text{P-max}^{\infty}(\tau))
\end{aligned}$$

Hence

$$\begin{aligned}
\text{P-max}_{\circlearrowleft}(\sigma ; \tau) \supseteq_L \text{P-max}_{\circlearrowleft}^{\infty}(\sigma) \circledast \text{P-max}_{\circlearrowleft}^{\infty}(\tau) &\iff \\
U_{\circlearrowleft} F_{\circlearrowleft} (\text{P-max}_{\circlearrowleft}(\sigma ; \tau)) \subseteq U_{\circlearrowleft} F_{\circlearrowleft} (\text{P-max}^{\infty}(\sigma) ; \text{P-max}^{\infty}(\tau)) &\iff \\
\text{P-max}^{\infty}(\sigma ; \tau) \subseteq \text{P-max}^{\infty}(\sigma) ; \text{P-max}^{\infty}(\tau) &
\end{aligned}$$

We start by showing that $\text{P-max}_{\circlearrowleft}(\sigma ; \tau) \supseteq_L \text{P-max}^{\infty}(\sigma) ; \text{P-max}^{\infty}(\tau)$. So, let $s \in \sigma ; \tau$ be P-maximal as witnessed by the interaction $s' \in \sigma \circledast \tau$. Assume either $s' \upharpoonright_{A,B,-}$ or $s' \upharpoonright_{-,B,C}$ is not P-maximal in σ or τ respectively. We fix a total ordering of $\Upsilon = \{\alpha_0, \dots, \alpha_n, \dots\}$. At this point, note that since s' is finite, by the switching condition, there is an upper-bound $N \in \mathbb{N}$ such that for any $n > N$, α_n does not appear in s' . We will construct an interaction s'' by constructing first a chain of interactions $s'_{i,j,b}$ where $i \in \mathbb{N}$, $j \in \leq N$ stands for $\alpha_j \in \Upsilon$, and b for a “case” to be explained now. We alternate between two cases: ($b = 1$) $s'_{i,j,b} \upharpoonright_{A,B,-}$ is not P-maximal and ($b = 2$) $s'_{i,j,b} \upharpoonright_{A,B,-}$ is P-maximal but $s'_{i,j,b} \upharpoonright_{-,B,C}$ is not. Therefore, $s_{i,j,b}$ will satisfy that (for $i > 0$): (0) $s'_{i,j,b} \in \sigma \circledast \tau$ and $s'_{i,j,b} \upharpoonright_{A,-,C} = s$ (1) $s'_{i',j',b'} \sqsubseteq s'_{i,j,b}$ for any (i', j', b') smaller than (i, j, b) in lexicographic order; (2) $s'_{i,j,1} \upharpoonright_{A,B,-}$ is P-maximal in σ ; (3) $s'_{i,j,2} \upharpoonright_{-,B,C}$ is α_j -P-maximal in τ ; (4) for all j , $\pi_{\alpha_k}(s'_{i,j,b}) = \epsilon$ for $k > N$. We start by letting $s'_{0,0,1} = s'$. If $s'_{0,0,1} \upharpoonright_{A,B,-}$ (1) is not P-maximal, then we let t be a sequence of P-moves making $s'_{0,0,1} \upharpoonright_{A,B,-} \cdot t$ P-maximal in σ (see Prop. 33.3.4). Note that since s is P-maximal, all moves in t must be in B . Since τ is O-receptive, and t only has O-moves from the point of view of τ , we also have $s'_{0,0,1} \upharpoonright_{-,B,C} \cdot t \in \tau$. Hence, $s'_{0,0,1} \cdot t \in \sigma \circledast \tau$ and $(s'_{0,0,1} \cdot t) \upharpoonright_{A,-,C} = s$. So we let $s'_{1,0,1} = s'_{0,0,1} \cdot t$ and start the construction proper. So, assume we have $s'_{i,j,b}$ with $i > 0$ satisfying invariants (0)-(3). If $s'_{i,j,b} \upharpoonright_{A,B,-}$ and

$s'_{i,j,b} \upharpoonright_{-,B,C}$ are both P-maximal (in σ and τ , resp.), we are done.

Otherwise, assume first we are in case $(b = 1, j \leq N)$. Then, if $s'_{i,j,1} \upharpoonright_{-,B,C}$ is α_j :P-maximal, we let $s'_{i,j,2} := s'_{i,j,1}$. Otherwise, there must be a P-move m such that $s'_{i,j,1} \upharpoonright_{-,B,C} \cdot m \in \tau$ is α_j :P-maximal (Lemma 33.3.3). Since s is P-maximal, m must be a move in B . Now, since $s'_{i,j,1} \upharpoonright_{A,B,-}$ is P-maximal and σ is sufficiently O-receptive, it follows that $s'_{i,j,1} \upharpoonright_{A,B,-} \cdot m \in \sigma$. Hence, we have $s'_{i,j,1} \cdot m \in \sigma \circ \tau$ and $(s' \cdot m) \upharpoonright_{A,-,C} = s$. So we let $s'_{i,j,2} := s'_{i,j,1} \cdot m$.

Now, suppose we are in case $(b = 2, j \leq N)$. This works similarly to our setup, we let t be P-extension of $s'_{i,j,2} \upharpoonright_{A,B,-}$ such that $s'_{i,j,2} \upharpoonright_{A,B,-} \cdot t$ is P-maximal. Since $s'_{i,j,2} \upharpoonright_{A,-,C} = s$ is P-maximal, it follows that all moves in t are in B . Since τ is O-receptive, it follows that $s'_{i,j,2} \cdot t$ is a valid interaction. So, if $j < N$ we let $s'_{i,j+1,1} := s'_{i,j,2} \cdot t$, and if $j = N$ we let $s'_{i+1,0,1} := s'_{i,j,2} \cdot t$.

Now, let $s'' = \sup \{s_{i,j,b} \mid i \in \mathbb{N} \wedge j \leq N \wedge b \in \{1, 2\}\}$. By repeating this argument we either obtain an interaction $s'' \in \text{P-max}(\sigma) \circ \text{P-max}(\tau)$ such that $s'' \upharpoonright_{A,-,C} = s$ or an infinite sequence s'' such that

$$\forall p \sqsubseteq s'' \upharpoonright_{A,B,-} \cdot p \in \sigma$$

$$\forall p \sqsubseteq s'' \upharpoonright_{-,B,C} \cdot p \in \tau$$

so that $s'' \in \text{P-max}^\infty(\sigma) \circ \text{P-max}^\infty(\tau)$ by definition of $\text{P-max}^\infty(-)$ and the claim follows.

Note that for any agent α_k , $s'' = \sup \{s_{i,j,b} \mid i \in \mathbb{N} \wedge j = k \wedge b \in \{1, 2\}\}$ witness the requirement on the chains that appears in the definition of a limit.

- The proof of this is analogous to the proof above.
- We note first that, by Lemma 33.1.1

$$\Downarrow(\Sigma \circ T) = \Downarrow_{U_\circ} F_\circ (U_\circ \Sigma ; U_\circ T) = \Downarrow(U_\circ \Sigma ; U_\circ T)$$

So if $s \in \Downarrow(\Sigma \circ T)$ then there exists a $s' \in U_\circ \Sigma ; U_\circ T$ such that $p \sqsubseteq s' \upharpoonright_{A,-,C}$. So take the smallest prefix $p' \sqsubseteq s'$ such that $p' \upharpoonright_{A,-,C} = s$ (the prefix of s' such that the

last event of p is the last event of s'). Then,

$$p' \upharpoonright_{A,B,-} \sqsubseteq s' \upharpoonright_{A,B,-} \in U_{\circlearrowleft} \Sigma \qquad p' \upharpoonright_{-,B,C} \sqsubseteq s' \upharpoonright_{-,B,C} \in U_{\circlearrowleft} \mathsf{T}$$

so $p' \in \Downarrow \Sigma \circ \Downarrow \mathsf{T}$ and the claim follows. □

Proposition 33.9.2 (cf. Proposition 27.2.1). *Every nearly complete sufficiently O-receptive $\bullet$ -receptive object is a nearly complete deterministic sufficiently O-receptive $\bullet$ -receptive object.*

Proof. The backward direction is immediate. We prove only the forward direction. Note that it is enough to show that if a single object $M = \langle \sigma, \Sigma \rangle$ is a complete sufficiently O-receptive object then it is nearly complete deterministic sufficiently O-receptive object. The result for a nearly such object then follows by applying this result to each M_i in its decomposition.

Now, the proof follows mostly along the lines of the proof of Lemma 26.1.2. We define a notion of coherence for well-scheduled concurrent plays:

Definition 33.9.1. Given games A, B we define the *coherence relation* $s \supset_{A \multimap B} t$ over $P_{A \multimap B}^\infty$, to hold when the alternating decompositions

$$U_{\circlearrowleft} s = O_1 \cdot P_2 \cdot \dots \cdot O_{2n-1} \cdot P_{2n}$$

$$U_{\circlearrowleft} t = O'_1 \cdot P'_2 \cdot \dots \cdot O'_{2n-1} \cdot P'_{2n}$$

satisfy that if $p \cdot X_i \sqsubseteq s$ and $p \cdot Y_i \sqsubseteq t$ with X_i and Y_j concurrency classes, then if $X_i \neq Y_j$ then either (1) $X_i \subseteq Y_i$ and $X_{i+1} = \epsilon$ or (2) $Y_i \subseteq X_i$ and $Y_{i+1} = \epsilon$ or (3) $i = j$ and both are odd (i.e. both are opponent classes).

It is easy to check that the coherence relation is reflexive and symmetric. Taking the same convention as Lemma 26.1.2, we call a subset $C \subseteq P_{A \multimap B}^{ws}$ a clique if it is pairwise coherent. It is well-known that cliques form a domain [Girard, 1987a] whose ordering is

$- \subseteq -$, and in particular are a DCPO. Moreover, given $S \subseteq P_{A \multimap B}^{\text{ws}}$ we may consider instead the domain of cliques over S , which we write $\mathcal{C}(S)$. From this, we derive several crucial facts about $\mathcal{C}(\Sigma)$. First, the domain of cliques is non-empty as $\{\epsilon\}$ is always a clique. Moreover, from the DCPO structure, we obtain that any chain $(C_i)_{i \in I}$ of cliques $\mathcal{C}(\Sigma)$ has a least upper bound given by $\cup_{i \in I} C_i$. Zorn's lemma then gives that there are maximal cliques over $\mathcal{C}(S)$.

So let C be a maximal clique within Σ . Define, $\langle C \rangle$ as the set:

$$\langle C \rangle := \{s \in \sigma \mid s \in \Downarrow C\}$$

We prove that $\langle \langle C \rangle, C \rangle$ is a reasonable candidate for a decomposition of some object:

- $\langle \langle C \rangle, C \rangle$ is complete. Let $s \in \text{P-max}(\langle C \rangle)$. By definition, $s \in \langle C \rangle \subseteq \sigma$. By completeness, s admits a P-extension $s \cdot t \in U_{\circlearrowleft} \Sigma$. Now, if $s \cdot t \in U_{\circlearrowleft} C$ then note that since s is P-maximal in $\langle C \rangle$, $t = \epsilon$ so $s \in U_{\circlearrowleft} C$, as desired. Otherwise, let $s' \in \Sigma$ be a play such that $U_{\circlearrowleft} s' = s \cdot t$. It then follows that there is some $t' \in C$ such that $s' \succ t'$ (otherwise, $s \in U_{\circlearrowleft} C$ already). Hence, $U_{\circlearrowleft} s' \succ U_{\circlearrowleft} t'$. In particular, there is a prefix $p \cdot X \sqsubseteq s \cdot t$ and $p \cdot Y \sqsubseteq U_{\circlearrowleft} t'$ witnessing the incoherence. Note then that since $s \supset U_{\circlearrowleft} t'$, it must be that $s \sqsubseteq p \cdot X, p \cdot Y$, and, moreover, X is a P-class (as t can only contain P-moves). Hence, p must end with an O-class, so that Y must be a P-class as well. But then, if t' has prefix p' such that $U_{\circlearrowleft} p' = p \cdot Y$ we are done as $p \cdot Y$ is a P-extension of s . Otherwise, $p \cdot Y$ is a strictly bigger play than s , so we may repeat the argument with $p \cdot Y$ replaced by s . Since only finitely many P-moves can be added to s until we reach a P-extension, it follows that eventually we obtain a P-extension of s in C .
- $\langle C \rangle$ is $\bullet$ -receptive. Suppose $s \in \langle C \rangle \sqsupseteq_S \sigma$. Then, since σ is $\bullet$ -receptive, if $s \cdot \alpha \cdot \bullet$ is well-formed, then it is in σ . But then, $s \cdot \alpha \cdot \bullet \supset t$ if and only if $s \supset t$, by definition of $\supset$. Hence, $s \cdot \alpha \cdot \bullet \in \langle C \rangle$.
- $\langle C \rangle$ is sufficiently O-receptive. Suppose first that there is $s' \in C$ such that $U_{\circlearrowleft} s' = s$ and $s \in \langle C \rangle$. So s is an O-receptive position in σ by assumption. Suppose now $s \in \langle C \rangle$ is P-maximal. We argue that $s \in \langle C \rangle \subseteq \sigma$ is P-maximal in σ or there is $s' \in \Sigma$ such

that $U_{\odot} s' = s$. It will then follow that s is an $\mathbf{O}$ -receptive position by assumption. So, assume s is not $\mathbf{P}$ -maximal in σ . Then, there exists a $\mathbf{P}$ -extension $s \cdot t \in \sigma$ such that $s \cdot t$ is $\mathbf{P}$ -maximal. By completeness, there is $s' \in \Sigma$ such that $U_{\odot} s' = s \cdot t$. Since s is $\mathbf{P}$ -maximal in $\langle C \rangle$ it follows that it must be that s' is incoherent with some play of C . Since incoherence can only be caused by $\mathbf{P}$ -classes, there must be a play $t' \in C$ such that $U_{\odot} t' = s$ and $s' \succsim t'$. So s is an $\mathbf{O}$ -receptive position. Infinite limits are in C because the coherence relation is prefix-closed.

- $\langle \langle C \rangle, C \rangle$ is $\mathbf{P}$ -deterministic. Immediate from the definition of coherence.

Now, we show

$$\bigcup_{C \in \mathcal{C}(\Sigma). C \text{ is maximal}} \langle \langle C \rangle, C \rangle = M$$

That

$$\bigcup_{C \in \mathcal{C}(\Sigma). C \text{ is maximal}} \langle \langle C \rangle, C \rangle = M$$

is immediate as $C \subseteq \Sigma$ by definition and $\langle C \rangle \subseteq \sigma$ by tightness.

For the reverse refinement, we have that $\{s\}$ is a clique for any $s \in \Sigma$, so by Zorn's lemma there is some maximal clique C such that $\{s\} \subseteq C$. Moreover, if $s \in \sigma$, then by completeness there is some $\mathbf{P}$ -extension $s' \in \Sigma$ of s , and then $s \in \langle C' \rangle$ where C' is some clique such that $\{s'\} \subseteq C'$. $\square$

Lemma 33.9.1. *Every nearly complete $\sim'$ -saturated sufficiently $\mathbf{O}$ -receptive $\bullet$ -receptive object is a nearly complete $\sim'$ -saturated weakly deterministic sufficiently $\mathbf{O}$ -receptive $\bullet$ -receptive object.*

Proof. As in the proof of Prop. 27.2.1 it suffices to show that a complete $\sim'$ -saturated sufficiently $\mathbf{O}$ -receptive $\bullet$ -receptive object $\langle \sigma, \Sigma \rangle$ has a valid decomposition. By applying Prop. 27.2.1 we obtain a decomposition $\bigcup_{i \in I} \langle \sigma_i, \Sigma_i \rangle$ satisfying all the claims except for the $\sim'$ -saturation. We argue that decomposition with components $\langle \sigma'_i, \Sigma'_i \rangle$ given by $\langle [\sigma_i]_{\sim'}, [\Sigma'_i]_{\sim'} \rangle$ where $[S]_{\sim'} := \{s \mid \exists t \in S. s \sim' t\}$ is the $\sim'$ -closure of S gives the desired decomposition.

Note that $\langle \sigma'_i, \Sigma'_i \rangle \sqsupseteq_O \langle \sigma, \Sigma \rangle$ because $\langle \sigma, \Sigma \rangle$ is $\sim'$ -closed. That $\langle \sigma'_i, \Sigma'_i \rangle$ is still complete follows from the fact that $s \sim' t$ with s $\mathbf{P}$ -maximal, implies t is also $\mathbf{P}$ -maximal, which

then gives the desired P-extension to show completeness. The same reasoning shows sufficient receptivity. $\bullet$ -receptivity is straight-forward by the $\bullet$ -receptivity of $\langle \sigma_i, \Sigma_i \rangle$. Weak determinism holds by definition and from the (strong) determinism of $\langle \sigma_i, \Sigma_i \rangle$. $\square$

Lemma 33.9.2 (cf. Lemma 27.2.1). *If $\langle \sigma, \Sigma \rangle : A \multimap B$, $\langle \tau, T \rangle : B \multimap C$ is an $\sim'$ -saturated weakly deterministic object then:*

$$\uparrow(\sigma ; \tau) \supseteq_L \text{P-max}_{\circlearrowleft}^{\infty}(\Sigma) \circledcirc \text{P-max}_{\circlearrowleft}^{\infty}(T)$$

Proof. Our first observation is that if $s \in \sigma \circledcirc \tau$ and $s \upharpoonright_{A,-,C}$ has the alternating decomposition $O_1 \cdot \dots \cdot X_n$ then we may find a play $\langle s \rangle \in \sigma \circledcirc \tau$ such that $\langle s \rangle \upharpoonright_{A,-,C} = s \upharpoonright_{A,-,C}$, $\langle s \rangle \upharpoonright_{-,B,-} = s \upharpoonright_{-,B,-}$ and, moreover:

$$\langle s \rangle = O_1 \cdot O'_1 \cdot \dots \cdot P'_{k_1} \cdot P_2 \cdot O'_{k_1+1} \cdot \dots \cdot P'_{k_2} \cdot O_3 \cdot O'_{k_2+1} \cdot \dots \cdot P'_{k_3} \cdot P_4 \cdot \dots \cdot X_n$$

in such a way that if $O_{k_l+1} = \epsilon$ then P_{k_l} is the maximal concurrency class satisfying this property.

There are two ways the shape above could be invalidated: a class O_i could be interleaved with moves in B , or the class P_{i+1} could be interleaved with moves in B . We show that in each of those cases, we can obtain a longer play “closer” to satisfying the shape (except for the O_i class being necessarily preceded by a P class in A , what we handle later). It is straight-forward to then turn this into a proof by induction.

Suppose the shape above is not satisfied at O_i in the sense that $p \cdot p_O \cdot t_{A,C} \cdot m_O \cdot t = s$ with $p_O \cdot m_O \subseteq O_i$ and $t_{A,C}$ a play with no moves in A or C . Then, we note that by the switching condition, the preceding move by the same agent as m_O must appear in p . Hence, $p \cdot p_O \cdot m_O \cdot t_{A,C}$ is a play in $\sigma \circledcirc \tau$ with the same B and $A \multimap C$ projections as s .

For the second situation, suppose $s = p \cdot p_P \cdot t_{A,C} \cdot m_P \cdot t$ with t_A containing only moves in B and $p_P \cdot m_P \subseteq P_{i+1}$. In this case, we note that t_A may not contain any moves by the same agent as a move in p_P . Hence, $p \cdot t_{A,C} \cdot p_P \cdot m_P \cdot t$ is still valid.

Making these replacements ensures that we obtain a play satisfying all the constraints above, except for the last class in B between two classes in $A \multimap C$ needing to be a P -class

(and maximal in some cases). For that we note that if we have $s = p \cdot X'_i \cdot Y_k \cdot X_{i+1}' \cdot t$ (where Y_k is a class in B) and X'_i and X'_{i+1} are not P- and O-classes respectively, then either (1) they both have the same polarity or (2) X'_i is an O-class and X'_{i+1} is a P-class. In case (1) we note that if they are both P-classes then they are all O-moves in $A \multimap B$, so $p \cdot X'_i \cdot X'_{i+1} \cdot Y_i \cdot t$ is valid. In case (2) we similarly see that $p \cdot Y_i \cdot X'_i \cdot X'_{i+1} \cdot t$ is valid.

Now, the only invariant left to satisfy is that P'_i is always maximal when O'_{i+1} is empty. This follows by determinism of σ as if O'_{i+1} is empty then there is always a P-extension which can be concatenated to P'_i while maintaining the validity of the play.

So assume given a limit $s \in \uparrow(\sigma ; \tau)$. By definition, there is a chain $s_1 \sqsubseteq \dots \sqsubseteq s_n \sqsubseteq \dots \in \sigma ; \tau$ such that $\sup_{i \in \mathbb{N}} s_i = s$. By definition, there is a corresponding sequence $S = (s'_i \in \sigma \circ \tau)_{i \in \mathbb{N}}$ such that $s'_i \upharpoonright_{A, -, C} = s_i$ where each s'_i may be assumed minimal under prefixing. In a sequential setting we would be then able to prove that the sequence S defines a chain. In the concurrent case, this is not possible without some care. So instead of showing it defines a chain, we will show that it can be modified into a chain. First, let the alternating decomposition of s be:

$$s = O_1 \cdot P_2 \cdot \dots \cdot O_{2n-1} \cdot P_{2n} \cdot \dots$$

Then, define the sequence $(t_i)_{i \in \mathbb{N}}$ by letting t_i be the smallest play s'_j such that $O_1 \dots X_i \sqsubseteq s_i \upharpoonright_{A, -, C}$. Then, we consider the sequence $(t'_i)_{i \in \mathbb{N}}$ with $t'_i := \langle t_i \rangle$. We claim this forms a chain (notice that by the argument above it is still a valid chain over $\sigma \circ \tau$).

Now, suppose t'_i is not a prefix of t'_j (with $i < j$). Then, there is a concurrency class of t'_i that is different from the corresponding concurrency class of t'_j . Notice that the first such concurrent class cannot be a concurrency class in B , as $t'_i \upharpoonright_{-, B} \sqsubseteq t'_j \upharpoonright_{-, B} \sqsubseteq s$. So suppose

$$O_1 \cdot \dots \cdot X_k \sqsubseteq t'_i$$

$$O_1 \cdot \dots \cdot Y_l \sqsubseteq t'_j$$

Then the concurrency class preceding X_k and Y_l is the same, so that by alternation of the concurrency classes both X_k and Y_l have the same polarity. In particular, this is a

proponent concurrency class in either σ or τ , and therefore $Y_l = X_k$ by weak determinism (plus the fact that they only involve moves in the same component), a contradiction.

Therefore, $(t'_i)_{i \in \mathbb{N}}$ forms a chain of plays in $\sigma \circ \tau$, and in particular $(t'_i|_{A,-})_{i \in \mathbb{N}}$ is a chain in σ and $(t'_i)_{i \in \mathbb{N}}$ is a chain in τ . Hence, $s \in \text{P-max}_{\circlearrowleft}^{\infty}(\sigma); \text{P-max}_{\circlearrowleft}^{\infty}(\tau)$ (note that since $s|_{-,B}$ is well-scheduled, so is $\sup_{i \in \mathbb{N}} t'_i$). $\square$

Proposition 33.9.3 (cf. Proposition 27.3.1). *O-receptive precise objects compose.*

Proof. Let the objects decompose as $M = \cup_{i \in I} \langle \sigma_i, \Sigma_i \rangle$ and $N = \cup_{j \in J} \langle \tau_j, T_j \rangle$. Then, by monotonicity:

$$M ; ; N \equiv_O \cup_{i \in I} \cup_{j \in J} \langle \sigma_i, \Sigma_i \rangle ; ; \langle \tau_j, T_j \rangle$$

We re-establish all the necessary claims. Note that by Lemma 27.2.1 we need not worry about determinism, as we can apply the lemma again to decompose back into deterministic components.

That the composition $\langle \sigma_i, \Sigma_i \rangle ; ; \langle \tau_j, T_j \rangle$ is tight follows by lemma 27.1.1. Completeness follows from Lemma 27.1.1 together with Lemma 27.2.2. $\bullet$ -receptivity and O-receptivity follow from the same argument as in Prop. 12.3.1. $\sim'$ -saturation follows from the fact that these are precisely the strategies saturated under the idempotent: $\{s \in \text{copy}_A \mid s|_{A,-} = s|_{-,A}\}$ and similarly for the liveness side. $\equiv_{\sim}$ -closedness follows from the analogous argument with the idempotent $\{s \in \text{copy}_A \mid s|_{A,-} \equiv_{\sim} s|_{-,A}\}$ (see Prop. 12.2.1 for how these arguments by saturation under idempotents work). $\square$

Proposition 33.9.4 (cf. Proposition 27.4.2). *If $\langle \sigma, \Sigma \rangle : A \multimap B$ and $\langle \tau, T \rangle : B \multimap C$ are relaxedly tight and either σ or τ is saturated then $\langle \sigma, \Sigma \rangle ; ; \langle \tau, T \rangle$ is relaxedly tight.*

Proof. Note that, by Prop. 27.1.1:

$$\begin{aligned}
\Downarrow(\Sigma \circledast T) &\sqsubseteq_S \Downarrow\Sigma ; \Downarrow T \\
&\sqsubseteq_S K \sigma ; K \tau \\
&= \text{copy}_A ; \sigma ; \text{copy}_B ; \text{copy}_B ; \tau ; \text{copy}_C \\
&= \text{copy}_A ; \sigma ; \tau ; \text{copy}_C \\
&= K (\sigma ; \tau)
\end{aligned}$$

and hence $\Downarrow(\Sigma \circledast T) \rightsquigarrow \sigma ; \tau$. □

Chapter 34

Program Logics Soundness

34.1 Safety

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

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

Given a unary predicate P and a binary predicate R , we define the binary predicate:

$$x (P \mid R) y \iff x \in P \wedge x R y$$

Then, we define the judgment

$$\text{safe}_\alpha(\mathcal{R}, \mathcal{G}, P, C, Q)$$

inductively as follows:

$$\begin{array}{c}
\frac{\text{rely}(\mathcal{R}, P) \mid \text{ID} \subseteq Q}{\text{safe}_\alpha(\mathcal{R}, \mathcal{G}, P, \text{ret } v, Q)} \text{ DONE} \\
\\
\frac{\begin{array}{l}
\forall B, C'. C \mapsto_B C' \Rightarrow \exists P^O. \mathcal{R}, \mathcal{G} \models_\alpha \{\text{rely}(\mathcal{R}, P)\} B \{P^O\} \\
\wedge \exists P^P. \mathcal{R}, \mathcal{G} \models_\alpha \{\text{rely}(\mathcal{R}, P^O \circ \text{rely}(\mathcal{R}, P))\} B \{P^P\} \\
\wedge \text{safe}_\alpha(\mathcal{R}, \mathcal{G}, P^P \circ \text{rely}(\mathcal{R}, P^O \circ \text{rely}(\mathcal{R}, P)), C', Q)
\end{array}}{\text{safe}_\alpha(\mathcal{R}, \mathcal{G}, P, C, Q)} \text{ STEP}
\end{array}$$

A straight-forward proof by induction shows that:

Lemma 34.1.1. *If*

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

then

$$\text{safe}_\alpha(\mathcal{R}, \mathcal{G}, P, C, Q)$$

Proposition 34.1.1 (Soundness). *If $\xi \vdash M : \nu_E \Rightarrow \nu_F$ then*

$$\xi \cap ([\mathcal{V}_E]_S ; [M]_S) \rightsquigarrow [\mathcal{V}_F]_S$$

Proof. By the definition of linked semantics it suffices to show that for any play

$$s \in [\text{Link } \mathcal{V}_E; M]_S \cap \xi$$

we have $s|_F \in K_{\text{copy}} \nu_F$.

By definition

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

where for each $\alpha \in A$

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

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

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

and hence for every $f \in F$:

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

We prove the result by induction on the length of

$$\langle c_0, \Delta_0, \iota_0, \epsilon, \text{st}_{E,0} \rangle \xrightarrow[\mathcal{V}_E]{s \quad M} \langle c, \Delta, \iota, s, \text{st}_E \rangle$$

for which we maintain the invariant that

$$s \upharpoonright_F \in \nu'_F$$

and that there is a position ρ_F such that

$$s \upharpoonright_F \dashrightarrow \rho_F$$

and that there are pre-conditions P_{α} for every $\alpha \in A$ such that

$$(\Delta, \iota, s, \text{st}_E, \rho) \in P_{\alpha} \quad \text{stable}(\mathcal{R}[\alpha], P_{\alpha})$$

and moreover:

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

$$c(\alpha) = (C, B) \Rightarrow \exists f \in F. \text{safe}_{\alpha}(\mathcal{R}[\alpha], \mathcal{G}[\alpha], P_{\alpha}, C, \text{returned}_{\alpha}(f) \circ Q[\alpha]^f)$$

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

$$s \upharpoonright_F \dashrightarrow \rho_F$$

and by the definition of possibility and Prop. 13.1.1 it follows that

$$s \upharpoonright_F \in K_{\text{copy}} \nu_F$$

We now start the proof proper. We will not bother with the invariant $s \upharpoonright_F \in \nu'_F$ from the definitions of `invoke`, `return` and `PRIM`. In the case where

$$\langle c, \Delta, \iota, s, \text{st}_E \rangle = \langle c_0, \Delta_0, \iota_0, \epsilon, \text{st}_{E,0} \rangle$$

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 $c(\alpha) = \text{idle}$. Now note that $(\Delta, \iota, \epsilon, \text{st}_E, \rho) \in \text{idle}_\alpha$ for every $\alpha \in A$ by definition. Moreover

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

by assumption that `CONCIMPL` holds. By definition:

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

Furthermore,

$$(\Delta, \iota, \epsilon, \text{st}_E, \rho) \in P_\alpha \quad \text{stable}(\mathcal{R}[\alpha], P_\alpha)$$

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

For the inductive step we have that

$$\langle c_0, \Delta_0, \iota_0, \epsilon, \text{st}_{E,0} \rangle \xrightarrow{s \ M}_{\mathcal{V}_E} \langle c, \Delta, \iota, s, \text{st}_E \rangle \xrightarrow{\alpha:m \ M}_{\mathcal{V}_E} \langle c', \Delta', \iota', s', \text{st}'_E \rangle$$

Moreover, we have

$$s \upharpoonright_F \dashrightarrow \rho_F$$

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

$$(\Delta, \iota, s, \text{st}_E, \rho) \in P_\alpha \quad \text{stable}(\mathcal{R}[\alpha], P_\alpha)$$

and moreover:

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

$$c(\alpha) = (C, B) \Rightarrow \exists f \in F. \text{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], P_\alpha, C, \text{returned}_\alpha(f) \circ Q[\alpha]^f)$$

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

$c(\alpha) = \text{idle}$ Note that in this case, $c' = c[\alpha \mapsto (M[\alpha]^f, \perp)]$ for some $f \in F$, $s' = s \cdot \alpha:f$. By the invariant, $P_\alpha \subseteq \text{idle}_\alpha$, and in particular $(\Delta, \iota, s, \text{st}_E, \rho) \in \text{idle}_\alpha$. Let $(\Delta', \iota', s', \text{st}_E, \rho')$ be such that ρ' is any ρ' such that

$$(s, \rho) \text{ invoke}_\alpha(f) (s', \rho')$$

Note that as $(\Delta, \iota, s, \text{st}_E, \rho) \in \text{idle}_\alpha$ it immediately follows that there is exactly one such ρ' (given by just appending $\alpha:f$ to ρ). We argue that:

$$\{s' \upharpoonright_F\} \dashrightarrow \rho'$$

By definition,

$$\rho' = \rho \cdot \alpha:f$$

Now, by induction there is t_P such that

$$s \upharpoonright_F \cdot t_P \rightsquigarrow_{\dagger F} \rho$$

but then

$$s \upharpoonright_F \cdot \alpha:f \cdot t_P \rightsquigarrow_{\dagger F} s \upharpoonright_F \cdot t_P \cdot \alpha:f \rightsquigarrow_{\dagger F} \rho \cdot \alpha:f = \rho'$$

it follows that

$$\{s' \upharpoonright_F\} \dashrightarrow \rho'$$

Note moreover that as $(\Delta, \iota, s, \text{st}_E, \rho) \in P_\alpha$, by induction

$$(\Delta, \iota, s, \text{st}_E, \rho) \in P_\alpha \subseteq P[\alpha] \subseteq P[\alpha]^f$$

and by construction

$$(\Delta, \iota, s, \text{st}_E, \rho) \text{ invoke}_\alpha(f) (\Delta', \iota', s', \text{st}_E, \rho')$$

so that

$$(\Delta', \iota', s', \text{st}_E, \rho') \in \text{invoke}_\alpha(f) \circ P[\alpha]^f$$

In addition, by assumption

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

By Lemma 34.1.1 it follows that

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

and

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

so if we let

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

Then it is almost immediate from the definition of *safe* that

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

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 $(\Delta', \iota', s', \text{st}_E, \rho') \in P_{\alpha'}$. For that, note that by induction $P_{\alpha'}$ is stable and by assumption $\mathcal{R}[\alpha']$ contains $\text{invoke}_\alpha(f) \subseteq \text{invoke}_{\alpha'}(-)$ so that $P_{\alpha'}$ is stable under $\text{invoke}_\alpha(f)$.

Now, $(\Delta', \iota', s', \text{st}_E, \rho') \in \text{idle}_{\alpha'} \iff (\Delta, \iota, s, \text{st}_E, \rho) \in \text{idle}_{\alpha'}$ by definition. It is obvious that if $(\Delta', \iota', s', \text{st}_E, \rho') \in \text{idle}_{\alpha'}$ then all the conditions are still satisfied by induction. Finally, if $(\Delta', \iota', s', \text{st}_E, \rho') \notin \text{idle}_{\alpha'}$ then there is an operation f' for which it holds that

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

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

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

$c(\alpha) = (\text{ret } v, \perp)$ In this case it must be that $c' = c[\alpha \mapsto \text{idle}]$, $s' = s \cdot \alpha : v$ for some $v \in \text{ar}(f)$ and $\Delta' = \Delta[\alpha : \emptyset]$. By induction there exists $f \in F$ such that

$$\text{safe}_{\alpha}(\mathcal{R}[\alpha], \mathcal{G}[\alpha], P_{\alpha}, \text{ret } v, \text{returned}_{\alpha}(f) \circ Q[\alpha]^f)$$

In this case safe_{α} consists of a DONE rule, and therefore:

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

In particular

$$(\Delta, \iota, s, \text{st}_E, \rho) \in P_{\alpha} \subseteq \text{rely}(\mathcal{R}, P_{\alpha}) \subseteq \text{return}_{\alpha}(f) \circ Q[\alpha]^f$$

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

$$(\Delta, \iota, s, \text{st}_E, \rho) \text{ return}_{\alpha}(f) (\Delta', \iota', s', \text{st}_E, \rho')$$

Moreover, by induction

$$s \upharpoonright_F \dashrightarrow \rho$$

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

$$s' \upharpoonright_F = s \upharpoonright_F \cdot \alpha : v$$

Hence

$$s' \upharpoonright_F \dashrightarrow_{\nu_F} \rho' = \rho$$

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

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

Then, by construction and by LOCALIMPL:

$$(\Delta', \iota', s', \text{st}_E, \rho') \in \text{return}_\alpha(f) \circ \text{returned}_\alpha(f) \circ Q[\alpha]^f \circ \text{invoke}_\alpha(f) \circ P[\alpha]^f \subseteq P'_\alpha$$

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

Moreover,

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

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 $(\Delta', \iota', s', \text{st}_E, \rho') \in P'_{\alpha'}$ but $P'_{\alpha'}$ is stable under $\text{return}_\alpha(-)$ by assumption $(\Delta, \iota, s, \text{st}_E, \rho) \in P_{\alpha'}$ so that $(\Delta', \iota', s', \text{st}_E, \rho') \in P'_{\alpha'}$.

$c(\alpha) = (C, B)$ **and** $C \neq \text{ret } v$ In this case, we have that $C \rightarrow_{B'} C'$ and $(\Delta', s') \in \llbracket B' \rrbracket(\Delta, s)$.

The interesting case is when B' is a command issuing an effect from E , so we assume

$s' = s \cdot \alpha:m$ where m is the move resulting from B' . Moreover, there is some $f \in F$

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

Now, notice that it follows by safe_α that

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

and

$$\exists P^P. \mathcal{R}, \mathcal{G} \models_{\alpha} \{\text{rely}(\mathcal{R}[\alpha], P^O \circ \text{rely}(\mathcal{R}, P_{\alpha}))\} \quad B' \{P^P\}$$

and

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

Now, by assumption $(\Delta, \iota, s, \text{st}_E, \rho) \in P_{\alpha}$ and $s|_E \cdot \alpha : m \in \nu_E$. By the two commit applications (the O-step and P-step), it follows that there is some ρ' such that

$$(\Delta, \iota, s, \text{st}_E, \rho) \xrightarrow{P^P \circ P^O} (\Delta', \iota', s \cdot \alpha : m, \text{st}'_E, \rho') \quad \rho \dashrightarrow \rho'$$

and

$$(\Delta, \iota, s, \text{st}_E, \rho) \mathcal{G} (\Delta', \iota', s \cdot \alpha : m, \text{st}'_E, \rho')$$

by assumption

$$s|_F \dashrightarrow \rho$$

so that

$$s'|_F = s|_F \dashrightarrow \rho \dashrightarrow \rho'$$

Moreover, if we set $P'_{\alpha} = P^P \circ \text{rely}(\mathcal{R}, P^O \circ \text{rely}(\mathcal{R}, P_{\alpha}))$ then

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

and moreover

$$(\Delta', \iota', s', \text{st}'_E, \rho') \in P'_{\alpha}$$

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 $(\Delta', \iota', s', \text{st}'_E, \rho') \in P_{\alpha'}$. But as

$$(\Delta, \iota, s, \text{st}_E, \rho) \mathcal{G}[\alpha] (\Delta', \iota', s', \text{st}'_E, \rho')$$

and

$$(\Delta, \iota, s, \text{st}_E, \rho) \in P_{\alpha'}$$

it follows from assumption that

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

and by stability that

$$(\Delta', \iota', s', \text{st}'_E, \rho') \in P_{\alpha'}$$

as desired. □

34.2 Liveness

Lemma 34.2.1 (Soundness- ω_F). *If $\xi \vdash M : \langle \nu_E, \omega_E \rangle \Rightarrow \langle \nu_F, \omega_F \rangle$ is true, then $\omega'_F \leadsto \omega_F$.*

Lemma 34.2.2. *If $\xi \vdash M : \langle \nu_E, \omega_E \rangle \Rightarrow \langle \nu_F, \omega_F \rangle$ is true, then for any $s \in \llbracket M \rrbracket_{\leq \omega}$ such that $s|_E \in \nu_E$ and any $s_i \in \downarrow s$, there exists ρ_i such that invariants (I1)–(I4) hold for s_i and ρ_i for any i , and moreover:*

$$\forall (p, \pi) \in \rho_i. s_i|_F \leadsto p \tag{34.1}$$

$$\forall (p_{i+1}, \pi_{i+1}) \in \rho_{i+1}. \exists (p_i, \pi_i) \in \rho_i. p_i \sqsubseteq p_{i+1} \tag{34.2}$$

Proof. By induction over the execution s_i . These properties are all maintained by the program logic rules, so they hold at each step. □

Proof of Lemma 34.2.1. It suffices to show that for any trace $s \in \llbracket M \rrbracket_{\omega}^{\nu_E}$, there is some trace $t \in \omega_F$ such that $\downarrow s|_F \leadsto \downarrow U_{\odot} t$ since $s \in \llbracket M \rrbracket_{\omega}^{\nu_E} \iff s|_F \in U_{\odot} \omega'_F$.

Let $s \in \llbracket M \rrbracket_{\omega}^{\nu_E}$ such that I1–I4 hold. By Lemma 34.2.2, we know for the chain $\downarrow s$, there exists $(\rho_i)_{i < \omega}$ such that (I1), (34.1), (I2), (I3), (I4), (34.2) holds for all s_i and ρ_i .

We then prove the conclusion by discussing whether s is finite.

Case 1: s is finite. Since s is finite, we know $s \in \downarrow s$ and there exists $\rho_{|s|}$ with all the conditions satisfied. As $\rho_{|s|}$ is non-empty, we take some $(p, \pi) \in \rho_{|s|}$. By (34.1), we know $t = s \upharpoonright_F \rightsquigarrow p$ and we only need to show that they contain the same events for non-starving threads. Since $s \upharpoonright_F = t \in \omega'_F$, we know it must be P-maximal, i.e., for any α , $\text{last}(\pi_\alpha(s \upharpoonright_F))$ is either a $\circ$, an overlay return event, or a $\bullet$. Then, by the definition of linearizability, we know $\pi_\alpha(p) = \pi_\alpha(s \upharpoonright_F) = \pi_\alpha(t)$. Therefore, we proved this case.

Case 2: s is infinite. As a result, $(p_i)_{i \in \omega}$ is also infinite. Every thread in Υ either always eventually makes progress, eventually diverges forever during some invocation, or gets starved.

- For threads that will always eventually make progress, they may have ρ_i to be diverging at some point but gets converged and linearized at some later point. We need to remove their invocations events from the linearized trace when they are linearized as diverging invocations.
- For thread that eventually diverges forever, they may also contain invocations that get linearized as diverging but eventually converge. We treat them similarly to the previous case. For those invocations that truly diverge, we collect them using the set below. By tracking the invocation event and its index in the concrete trace, we are then able to determine which invocation in the linearized trace is truly diverging and keep them in the final linearization.

$$D \triangleq \{(\alpha, j, f) \mid \forall k > j. \pi_\alpha(s_j \upharpoonright_F) = \pi_\alpha(s_{j-1} \upharpoonright_F) \cdot \alpha.f = \pi_\alpha(s_k \upharpoonright_F)\}$$

- For threads that starve, we ignore all their diverging linearizations.

Therefore, we have a way to identify linearized diverging invocations that should be kept in the linearized traces. We then construct a transformation from possibilities to sets of linearized traces.

$$T(\rho_i) = \{p \mid \exists (p', \pi) \in \rho_i. p = |p'|_{\pi, s_i, D}\}$$

where $|p'|_{\pi, s_i, D}$ is defined below given $p' = p'' \cdot \alpha : f \wedge p = |p''|_{\pi, s_i, D}$.

$$|p'|_{\pi, s_i, D} = \begin{cases} \epsilon & p' = \epsilon \\ p \cdot \alpha : f & \left(\begin{array}{l} \pi[\alpha] = \text{div}(f) \wedge s_i = s_j \cdot s' \wedge \alpha \notin s' \upharpoonright_F \wedge \\ \text{last}(s_j) = \alpha : f \wedge (\alpha, j, f) \in D \end{array} \right) \\ p & \text{otherwise} \end{cases}$$

This operation keeps diverging pending invocations that are inside D while removing others.

Notice that the transformation T satisfies the following conditions.

$$T(\rho_0) = \{\epsilon\} \quad \forall i, p \in T(\rho_{i+1}). \exists p' \in T(\rho_i). p' \sqsubseteq p \quad T(\rho_i) \neq \emptyset$$

The second condition is true because ρ_{i+1} is obtained by performing possibility updates from ρ_i .

- If the last update is S-INVOKE or S-RETURN, the prefix relation is preserved because they only append events to the previous linearized trace.
- If the last update is P-DIVERGE, the prefix relation is still preserved. But notice that not all P-DIVERGE updates are respected by the transformation T . T may ignore some P-DIVERGE updates and does not modify p , which still preserves the prefix relation.
- If the last update is P-CONVERGE, we then know the converged invocation is not inside D . Therefore, the invocation will not appear in the previous transformed trace because it was removed previously and we still have the prefix relation.

Therefore, there exists an infinite chain $(p_i)_{i < \omega}$, where $p_i \in T(\rho_i)$ and $p_i \sqsubseteq p_{i+1}$. Moreover, by (I2), we know for any p_i , there exists some $w_i \in \omega_F$ such that $p_i \sqsubseteq U_{\circlearrowright} w_i$. We then show there exists some $p \in \omega_F$ such that for any p_i there is $p_i \sqsubseteq U_{\circlearrowright} p$. Assume for all $p \in \omega_F$ there exists p_i and p_j ($i < j$) such that (1) $p_i \not\sqsubseteq U_{\circlearrowright} p$ and $p_j \sqsubseteq U_{\circlearrowright} p$, there is a contradiction as $p_i \sqsubseteq p_j \sqsubseteq U_{\circlearrowright} p$, or (2) $p_i \sqsubseteq U_{\circlearrowright} p$ and $p_j \not\sqsubseteq U_{\circlearrowright} p$, there is also a contradiction as $p_i \sqsubseteq p_j \sqsubseteq U_{\circlearrowright} w_i$. We take this p as the linearization of s and show $\downarrow s \upharpoonright_F \rightsquigarrow \downarrow U_{\circlearrowright} p$.

• **Linearizability.** For any $t_k \sqsubseteq s \upharpoonright_F$, there exists $s_i \sqsubseteq s$ such that $s_i \upharpoonright_F = t_k$ and $s_{i-1} \neq t_k$.

We take p_i that corresponds to s_i and have $s_i \upharpoonright_F \rightsquigarrow p_i$ and $p_i \sqsubseteq p$.

• **Honesty.** If $s_i \upharpoonright_F \rightsquigarrow p_i$ is a dishonest linearization, then either there is some pending invocations missing in p_i or some linearized responses missing in $s_i \upharpoonright_F$.

– **O-Honesty.** For the first case, we find ρ_i corresponding to s_i . We denote by s_O the currently uncommitted invocations from ρ_i , i.e., those with $\pi[\alpha] = \text{lin}(f)$. If the missing invocation $\alpha:f$ is not in D , then it is missing because it either has not been linearized by the prover at the moment of s_i or it is a diverging invocation removed by the transformation T . Since it is not in D , we may find some $k > i$ such that $\text{last}(\pi_\alpha(s_k)) = \alpha:v$ is a response. According to the RETURN rule, all $(p', \pi') \in \rho_k$ must have the response linearized, which also indicates that the invocation f is linearized in p' . We then take the transformed trace p_k corresponding to $T(\rho_k)$ and have $p_k \sqsubseteq p$ and

$$\pi_\alpha(p_i \cdot s_O) = \pi_\alpha(p_i) \cdot \alpha:f \sqsubseteq \pi_\alpha(p_k) \sqsubseteq \pi_\alpha(p).$$

If the missing invocation f is in D , then there is no guarantee that the operation will terminate and we need to use the honesty metric to ensure the divergence is eventually linearized in p . To construct the metric, we need to use a more detailed continuation definition and a different (but equivalent to the original one) small-step semantics.

The continuation K is idle when the current thread is idle. When the current thread is not idle, its continuation is always a tuple of currently focused command C and a list of loop continuations L . A loop continuation L is a tuple of loop body and commands following the loop.

$$K \in \text{Cont} ::= \text{idle} \mid (C, \vec{L}) \quad \text{where } L \in \text{Com} \times \text{Com}$$

Most of the continuation reduction rules are lifting previous ones in Section 9.3, but those for loops are different. Encountering a loop will push the loop continuation into the stack, while finishing the loop would either restart the loop by instantiating the latest loop body

or exit the loop by popping a loop continuation out of the stack.

$$\begin{aligned}
(C^*, \vec{L}) &\mapsto (\text{skip}, \vec{L} :: (C, \text{skip})) & (C^*; C', \vec{L}) &\mapsto (\text{skip}, \vec{L} :: (C, C')) \\
(\text{skip}, \vec{L} :: (C, C')) &\mapsto (C, \vec{L} :: (C, C')) & (\text{skip}, \vec{L} :: (C, C')) &\mapsto (C', \vec{L})
\end{aligned}$$

For the threadpool configuration $(c, \Delta, \iota, s_i, \text{st}_E)$ at position s_i along the execution s , we define its termination metric as follows.

$$\mu(\alpha, c, \Delta, \iota, s_i) \triangleq (|c[\alpha]|, \vec{H}, |\text{fst}(c[\alpha])|, \iota[\alpha])$$

- The first component $|c[\alpha]|$ is total lines of code in post-loop continuations.

$$|c[\alpha]| = \begin{cases} |\text{fst}(c[\alpha])| & |\text{snd}(c[\alpha])| = 0 \\ \sum_{(C) \in \text{snd}(c[\alpha])} |C| & \text{otherwise} \end{cases}$$

- The second component is a list of loop metrics, where H_i is the metric evaluation function that the prover used for the i -th loop in the current stack.

$$\vec{H} \triangleq [H_1(\alpha, \iota, s_i, s), H_2(\alpha, \iota, s_i, s), \dots, H_l(\alpha, \iota, s_i, s)]$$

Its ordering is defined below.

$$\vec{H} < \vec{H}' \iff H_l < H'_l \vee |\vec{H}| < |\vec{H}'|$$

It decreases if the metric for the innermost loop decreases or some loop finishes.

- The third component is the remaining lines of code in the current loop iteration.
- The last component is the scheduler counter for the current thread.

The ordering of the entire $\mu(\alpha, c, \Delta, \iota, s_i)$ is given by lexical order. The metric $\mu(\alpha, c, \Delta, \iota, s_i)$ is strictly decreasing if for all $(p, \pi) \in \rho_i$, $\pi[\alpha] \neq \text{div}(f)$. For any step that updates $(c, \Delta, \iota, s_i, \text{st}_E)$:

- If it is not a step made by α , then the first and third component obviously do not change. The second component does not increase because any environment step should not increase the loop metric. The last component decreases by scheduler counter update.
- If it is a step made by α that encounters a new loop: The first component decreases as the loop body gets removed from the post-loop continuation.
- If it starts a new iteration of the latest loop. By ITER rule, H_l decreases and so does the second component.
- If it exits the latest loop, the size of the second component decreases.

Therefore, the metric goes to bottom if the prover never linearized the current invocation as diverging, meaning the operation terminates, which contradicts the assumption. As a result, there exists a point s_j ($j > i$), such that for any $(p'_j, \pi_j) \in \rho_j$, $\pi_j[\alpha] = \text{div}(f)$. Thus, p_j is the linearization that contains the pending invocation as the diverging invocation as it is not removed from p'_j in the transformation, i.e.,

$$\pi_\alpha(p_i \cdot s_O) = \pi_\alpha(p_i) \cdot \alpha:f = \pi_\alpha(p_j) \sqsubseteq \pi_\alpha(p)$$

Therefore, we prove the O-honesty.

– **P-Honesty.** For the second case, the prover has linearized some response $\alpha:v$ that has not returned yet and $\alpha:v$ is missing in s_i . We denote by s_P the currently committed but not yet returned responses from ρ_i , i.e., those with $\pi[\alpha] = \text{lin}(f, v)$. Notice that although prover can first linearize the response in some possibilities but remove them later, we are considering the linearized trace from a possibility that is kept forever. So the response observed here is never removed and we may utilize the metric provided by the prover to show the termination. Therefore, the condition $\forall(p', \pi') \in \rho. \pi'[\alpha] = \text{div}(-)$ will never be satisfied after s_i for the current operation, and $\mu(\alpha, c, \Delta, \iota, s_i)$ is ensured to be strictly decreasing and eventually reaching $(0, \square, 0, -)$ at some point s_j , which implies the current

operation has terminated at s_j . Thus, we have

$$\pi_\alpha(s_i \cdot s_P) = \pi_\alpha(s_i) \cdot \alpha \cdot v = \pi_\alpha(s_j) \sqsubseteq \pi_\alpha(s)$$

and prove the P-honesty.

• **Progress.** Similarly, we may construct a global metric using $\mathcal{D}$ to show that for any $p' \sqsubseteq p$, there exists some s_i and p_i such that $p' \sqsubseteq p_i$ because p_i is constantly increasing along p and will eventually surpass p' . Since $s_i \upharpoonright_F \rightsquigarrow p_i$, we know the linearization p is tight.

□

Bibliography

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

Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. In Rudrapatna Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 291–301, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-47507-1.

Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.

Samson Abramsky and Guy McCusker. Linearity, sharing and state: A fully abstract game semantics for idealized algol with active expressions. In Peter W. O’Hearn and Robert D. Tennent, editors, *Algol-like Languages*, pages 297–329. Birkhäuser Boston, Boston, MA, 1997. ISBN 978-1-4757-3851-3. doi: 10.1007/978-1-4757-3851-3_10. URL https://doi.org/10.1007/978-1-4757-3851-3_10.

Samson Abramsky and Guy McCusker. Game semantics. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, pages 1–55, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-642-58622-4.

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for pcf. *Inf. Comput.*, 163(2):409–470, 2000. ISSN 0890-5401. doi: 10.1006/inco.2000.2930. URL <http://www.sciencedirect.com/science/article/pii/S0890540100929304>.

- Peter Aczel. *Non-Well-Founded Sets*, volume 14 of *CSLI Lecture Notes*. Stanford University, 1988.
- Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.
- Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. ISSN 0020-0190. doi: [https://doi.org/10.1016/0020-0190\(85\)90056-0](https://doi.org/10.1016/0020-0190(85)90056-0). URL <https://www.sciencedirect.com/science/article/pii/0020019085900560>.
- André Arnold and Maurice Nivat. Comportements de processus. In *Actes du Colloque AFCET “Les Mathématiques de l’Informatique”*, pages 35–68, Paris, 1982.
- Hagit Attiya and Constantin Enea. Putting Strong Linearizability in Context: Preserving Hyperproperties in Programs That Use Concurrent Objects. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:17, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-126-9. doi: 10.4230/LIPIcs.DISC.2019.2. URL <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2019.2>.
- R.J.R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981. ISSN 0022-0000. doi: [https://doi.org/10.1016/0022-0000\(81\)90005-2](https://doi.org/10.1016/0022-0000(81)90005-2). URL <https://www.sciencedirect.com/science/article/pii/0022000081900052>.
- Naama Ben-David, Michal Friedman, and Yuanhao Wei. Brief Announcement: Survey of Persistent Memory Correctness Conditions. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41:1–41:4, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-255-6. doi: 10.4230/LIPIcs.DISC.2022.41. URL <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2022.41>.

- Nick Benton. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices*, 39(1):14–25, 2004.
- Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl. ISBN 978-3-939897-98-9. doi: 10.4230/LIPIcs.OPODIS.2015.20. URL <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2015.20>.
- Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi: 10.1145/3473586. URL <https://doi.org/10.1145/3473586>.
- Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56(1):183–220, 1992. ISSN 0168-0072. doi: [https://doi.org/10.1016/0168-0072\(92\)90073-9](https://doi.org/10.1016/0168-0072(92)90073-9). URL <https://www.sciencedirect.com/science/article/pii/0168007292900739>.
- Stefan Bodenmüller, John Derrick, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. A fully verified persistency library. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 26–47, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-50521-8. doi: 10.1007/978-3-031-50521-8_2. URL https://doi.org/10.1007/978-3-031-50521-8_2.
- Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing, PODC '93*, pages 41–51, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897916131. doi: 10.1145/164051.164056. URL <https://doi.org/10.1145/164051.164056>.
- Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski. An abstract, certified account of operational game semantics. In Viktor Vafeiadis, editor, *Programming*

- Languages and Systems*, pages 172–199, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-91118-7.
- Stephen Brookes. A grainless semantics for parallel programs with shared mutable data. *Electronic Notes in Theoretical Computer Science*, 155:277 – 307, 2006. ISSN 1571-0661. doi: <https://doi.org/10.1016/j.entcs.2005.11.060>. URL <http://www.sciencedirect.com/science/article/pii/S1571066106001976>. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).
- Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227 – 270, 2007. ISSN 0304-3975. doi: <https://doi.org/10.1016/j.tcs.2006.12.034>. URL <http://www.sciencedirect.com/science/article/pii/S0304397506009248>. Festschrift for John C. Reynolds’s 70th birthday.
- Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying concurrent problems: Beyond linearizability and up to tasks. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, DISC 2015, pages 420–435, Berlin, Heidelberg, 2015. Springer-Verlag. ISBN 9783662486528. doi: 10.1007/978-3-662-48653-5_28. URL https://doi.org/10.1007/978-3-662-48653-5_28.
- Simon Castellán, Pierre Clairambault, Silvain Rideau, and Glynn Winskel. Games and Strategies as Event Structures. *Logical Methods in Computer Science*, Volume 13, Issue 3, September 2017. doi: 10.23638/LMCS-13(3:35)2017. URL <https://lmcs.episciences.org/3966>.
- Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Parameterised linearisability. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming*, pages 98–109, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43951-7.
- Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nikolai Zeldovich. Argosy: verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*,

- PLDI 2019, pages 1054–1068, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314585. URL <https://doi.org/10.1145/3314221.3314585>.
- Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 243–258, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359632. URL <https://doi.org/10.1145/3341301.3359632>.
- Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 423–439. USENIX Association, July 2021. ISBN 978-1-939133-22-9.
- Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815402. URL <https://doi.org/10.1145/2815400.2815402>.
- Pierre Clairambault, Julian Gutierrez, and Glynn Winskel. The winning ways of concurrent games. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pages 235–244, 2012. doi: 10.1109/LICS.2012.34.
- Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 207–231, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.
- Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D’Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 504–528, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14107-2.

- Emanuele D’Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, 43(4), nov 2021. ISSN 0164-0925. doi: 10.1145/3477082. URL <https://doi.org/10.1145/3477082>.
- Emanuele D’Osualdo, Azalea Raad, and Viktor Vafeiadis. The path to durable linearizability. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571219. URL <https://doi.org/10.1145/3571219>.
- Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 173–188, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-71316-6.
- Ivana Filipovic, Peter O’Hearn, Noam Rinetzkyl, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, dec 2010. ISSN 0304-3975. doi: 10.1016/j.tcs.2010.09.021. URL <https://doi.org/10.1016/j.tcs.2010.09.021>.
- Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 442–451, 2018.
- Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity. *Logical Methods in Computer Science*, 17, 2021.
- Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 388–402, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15375-4.
- Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: a separation logic framework for

- verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–31, 2022.
- Philippe Gaucher. Flows revisited: the model category structure and its left determinedness. *Cahiers de topologie et géométrie différentielle catégoriques*, LXI(2):208–226, 2020. URL <https://hal.archives-ouvertes.fr/hal-01919037>.
- Dan R. Ghica. Diagrammatic reasoning for delay-insensitive asynchronous circuits. In Bob Coecke, Luke Ong, and Prakash Panangaden, editors, *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky: Essays Dedicated to Samson Abramsky on the Occasion of His 60th Birthday*, pages 52–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38164-5. doi: 10.1007/978-3-642-38164-5_5. URL https://doi.org/10.1007/978-3-642-38164-5_5.
- Dan R. Ghica. The far side of the cube. In Alessandra Palmigiano and Mehrnoosh Sadrzadeh, editors, *Samson Abramsky on Logic and Structure in Computer Science and Beyond*, pages 219–250. Springer International Publishing, Cham, 2023. ISBN 978-3-031-24117-8. doi: 10.1007/978-3-031-24117-8_6. URL https://doi.org/10.1007/978-3-031-24117-8_6.
- Dan R. Ghica and Andrzej S. Murawski. Angelic semantics of fine-grained concurrency. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, pages 211–225, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24727-2.
- Dan R. Ghica and Nikos Tzevelekos. A system-level game semantics. *Electronic Notes in Theoretical Computer Science*, 286:191–211, 2012. ISSN 1571-0661. doi: <https://doi.org/10.1016/j.entcs.2012.08.013>. URL <https://www.sciencedirect.com/science/article/pii/S1571066112000436>. Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).
- Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987a. ISSN 0304-3975. doi: [https://doi.org/10.1016/0304-3975\(87\)90045-4](https://doi.org/10.1016/0304-3975(87)90045-4). URL <https://www.sciencedirect.com/science/article/pii/0304397587900454>.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987b.

Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 373–382, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306911. doi: 10.1145/1993636.1993687. URL <https://doi.org/10.1145/1993636.1993687>.

Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 453–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22012-8.

Éric Goubault, Jérémy Ledent, and Samuel Mimram. Concurrent Specifications Beyond Linearizability. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-098-9. doi: 10.4230/LIPIcs.OPODIS.2018.28. URL <http://drops.dagstuhl.de/opus/volltexte/2018/10088>.

W. John Gowers and James D. Laird. A Fully Abstract Game Semantics for Countable Nondeterminism. In Dan R. Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-088-0. doi: 10.4230/LIPIcs.CSL.2018.24. URL <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2018.24>.

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. Certikos: a certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing

- Machinery. ISBN 9781450311793. doi: 10.1145/2103799.2103803. URL <https://doi.org/10.1145/2103799.2103803>.
- Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. *SIGPLAN Not.*, 50(1):595–608, January 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676975. URL <https://doi.org/10.1145/2775051.2676975>.
- Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI’16, pages 653–669, USA, 2016. USENIX Association. ISBN 9781931971331.
- Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 646–661, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192381. URL <https://doi.org/10.1145/3192366.3192381>.
- Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. Building certified concurrent os kernels. *Commun. ACM*, 62(10):89–99, September 2019. ISSN 0001-0782. doi: 10.1145/3356903. URL <https://doi.org/10.1145/3356903>.
- Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS’04)*, ICDCS ’04, pages 400–407, USA, 2004. IEEE Computer Society. ISBN 0769520863. doi: 10.1109/ICDCS.2004.1281605.
- Rachid Guerraoui and Eric Ruppert. Linearizability is not always a safety property. In Guevara Noubir and Michel Raynal, editors, *Networked Systems*, pages 57–69, Cham, 2014. Springer International Publishing. ISBN 978-3-319-09581-3.

- Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local Linearizability for Concurrent Container-Type Data Structures. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-017-0. doi: 10.4230/LIPIcs.CONCUR.2016.6. URL <http://drops.dagstuhl.de/opus/volltexte/2016/6180>.
- Graham Harman. An outline of object-oriented philosophy. *Science Progress*, 96(2):187–199, 2013. doi: 10.3184/003685013X13691199842803.
- Graham Harman. *Object-Oriented Ontology: A New Theory of Everything*. Pelican Books, London, 2018. ISBN 978-0-241-26915-2.
- Russell Harmer and Guy McCusker. A fully abstract game semantics for finite nondeterminism. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science, LICS '99*, page 422, USA, 1999. IEEE Computer Society. ISBN 0769501583.
- Eashan Hatti, Arthur Oliveira Vale, Zhongye Wang, Yueyang Feng, and Zhong Shao. A complete program logic for compositional linearizability. In *40th European Conference on Object-Oriented Programming (ECOOP 2026)*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. To appear.
- Susumu Hayashi. Adjunction of semifunctors: Categorical structures in nonextensional λ calculus. *Theoretical Computer Science*, 41:95–104, 1985. ISSN 0304-3975. doi: [https://doi.org/10.1016/0304-3975\(85\)90062-3](https://doi.org/10.1016/0304-3975(85)90062-3). URL <https://www.sciencedirect.com/science/article/pii/0304397585900623>.
- Nir Hemed, Noam Rinetzkky, and Viktor Vafeiadis. Modular verification of concurrency-aware linearizability. In *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings 29*, pages 371–387. Springer, 2015.
- Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm.

- In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138407. doi: 10.1145/1007912.1007944. URL <https://doi.org/10.1145/1007912.1007944>.
- Maurice Herlihy and Nir Shavit. On the nature of progress. In Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems*, pages 313–328, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25873-2.
- Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL <https://doi.org/10.1145/78969.78972>.
- C. A. R. Hoare. *Proof of correctness of data representations*, pages 65–82. Prentice-Hall, Inc., USA, 1989. ISBN 0132840278.
- R. Hoofman and I. Moerdijk. A remark on the theory of semi-functors. *Mathematical Structures in Computer Science*, 5(1):1–8, 1995. doi: 10.1017/S096012950000061X.
- Martin Hyland. Game semantics. In Andrew M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 131–184. Cambridge University Press, 1997. doi: 10.1017/CBO9780511526619.005.
- Martin Hyland and C. H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000. doi: 10.1006/inco.2000.2917.
- Martin Hyland, Misao Nagayama, John Power, and Giuseppe Rosolini. A category theoretic formulation for engeler-style models of the untyped λ -calculus. *Electronic Notes in Theoretical Computer Science*, 161:43–57, 2006. ISSN 1571-0661. doi: <https://doi.org/10.1016/j.entcs.2006.04.024>. URL <https://www.sciencedirect.com/science/article/pii/S1571066106003999>. Proceedings of the Third Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT 2004).
- Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Il-

- cinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53426-7. doi: 10.1007/978-3-662-53426-7_23. URL https://doi.org/10.1007/978-3-662-53426-7_23.
- Prasad Jayanti, Siddhartha Jayanti, Ugur Yavuz, and Lizzie Hernandez. A universal, sound, and complete forward reasoning technique for machine-verified proofs of linearizability. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi: 10.1145/3632924. URL <https://doi.org/10.1145/3632924>.
- André Joyal. Remarques sur la théorie des jeux à deux personnes. *Gazette des sciences mathématiques du Québec*, 1(4), 1977.
- André Joyal and Myles Tierney. *An Extension of the Galois Theory of Grothendieck*, volume 51 of *Memoirs of the American Mathematical Society*. American Mathematical Society, Providence, RI, 1984. ISBN 0-8218-2312-4.
- Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *SIGPLAN Not.*, 50(1):637–650, jan 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676980. URL <https://doi.org/10.1145/2775051.2676980>.
- Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi: 10.1017/S0956796818000151.
- Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: Prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371113. URL <https://doi.org/10.1145/3371113>.
- Alan Kay. prototypes vs classes was: Re: Sun’s HotSpot. Post to the Squeak-dev mailing list, October 1998. URL <https://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html>. Accessed: 2026-02-12.

- Alan Kay. Dr. Alan Kay on the meaning of “object-oriented programming”. Email to Stefan Ram, 2003. URL http://www.purl.org/stefan_ram/pub/doc_kay_oop_en. Accessed: 2026-02-12.
- G. M. Kelly. *Basic Concepts of Enriched Category Theory*, volume 64 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1982.
- Artem Khyzha and Ori Lahav. Abstraction for crash-resilient objects. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 262–289, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99336-8. doi: 10.1007/978-3-030-99336-8_10. URL https://doi.org/10.1007/978-3-030-99336-8_10.
- Artem Khyzha, Alexey Gotsman, and Matthew Parkinson. A generic logic for proving linearizability. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 426–443, Cham, 2016. Springer International Publishing. ISBN 978-3-319-48989-6.
- Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. 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*, pages 639–667, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 978-3-662-54433-4. doi: 10.1007/978-3-662-54434-1_24. URL https://doi.org/10.1007/978-3-662-54434-1_24.
- Jérémie Koenig. Grounding game semantics in categorical algebra. In Kohei Kishida, editor, *Proceedings of the Fourth International Conference on Applied Category Theory, ACT 2021*, 2021.
- Jérémie Koenig and Zhong Shao. Refinement-based game semantics for certified abstraction layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS ’20*, pages 633–647, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371049. doi: 10.1145/3373718.3394799. URL <https://doi.org/10.1145/3373718.3394799>.

- Marta Z. Kwiatkowska. *On topological characterization of behavioural properties*, pages 153–177. Oxford University Press, Inc., USA, 1991. ISBN 0198537603.
- James Laird. A game semantics of idealized csp. *Electronic Notes in Theoretical Computer Science*, 45:232–257, 2001. ISSN 1571-0661. doi: [https://doi.org/10.1016/S1571-0661\(04\)80965-4](https://doi.org/10.1016/S1571-0661(04)80965-4). URL <https://www.sciencedirect.com/science/article/pii/S1571066104809654>. MFPS 2001, Seventeenth Conference on the Mathematical Foundations of Programming Semantics.
- L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. doi: 10.1109/TSE.1977.229904.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL <https://doi.org/10.1145/359545.359563>.
- Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439.
- Dongjae Lee, Minki Cho, Jinwoo Kim, Soonwon Moon, Youngju Song, and Chung-Kil Hur. Fair operational semantics. *Proceedings of the ACM on Programming Languages*, 7 (PLDI):811–834, 2023.
- Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. Lilo: A higher-order, relational concurrent separation logic for liveness. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):1267–1294, 2025.
- Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL <http://doi.acm.org/10.1145/1538788.1538814>.
- Mohsen Lesani, Li-Yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: Verified transactional objects. *Proc. ACM Program.*

- Lang.*, 6(OOPSLA1), apr 2022. doi: 10.1145/3527324. URL <https://doi.org/10.1145/3527324>.
- Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 459–470, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462189. URL <https://doi.org/10.1145/2491956.2462189>.
- Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 385–399, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837635. URL <https://doi.org/10.1145/2837614.2837635>.
- Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158108. URL <https://doi.org/10.1145/3158108>.
- Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. Characterizing progress properties of concurrent objects via contextual refinements. In Pedro R. D'Argenio and Hernán Melgratti, editors, *CONCUR 2013 – Concurrency Theory*, pages 227–241, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40184-8.
- Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014a. Association for Computing Machinery. ISBN 9781450328869. doi: 10.1145/2603088.2603123. URL <https://doi.org/10.1145/2603088.2603123>.
- Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of*

- the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10, 2014b.
- Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. ISSN 0164-0925. doi: 10.1145/197320.197383. URL <https://doi.org/10.1145/197320.197383>.
- Nancy Lynch and Frits Vaandrager. Forward and backward simulations. *Inf. Comput.*, 128(1):1–25, jul 1996. ISSN 0890-5401. doi: 10.1006/inco.1996.0060. URL <https://doi.org/10.1006/inco.1996.0060>.
- Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 1998.
- Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 3–10, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 1595935754. doi: 10.1145/1190216.1190220. URL <https://doi.org/10.1145/1190216.1190220>.
- Paul-Andre Melliès. Asynchronous games 4: A fully complete model of propositional linear logic. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, LICS '05, pages 386–395, USA, 2005a. IEEE Computer Society. ISBN 0769522661. doi: 10.1109/LICS.2005.6. URL <https://doi.org/10.1109/LICS.2005.6>.
- Paul-André Melliès. Sequential algorithms and strongly stable functions. *Theoretical Computer Science*, 343(1):237 – 281, 2005b. ISSN 0304-3975. doi: <https://doi.org/10.1016/j.tcs.2005.05.015>. URL <http://www.sciencedirect.com/science/article/pii/S0304397505003178>. Game Theory Meets Theoretical Computer Science.
- Paul-André Melliès. Categorical semantics of linear logic. In *In: Interactive Models of Computation and Program Behaviour, Panoramas et Synthèses 27, Société Mathématique de France 1196*, 2009.

- Paul-André Melliès. Categorical combinatorics of scheduling and synchronization in game semantics. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290336. URL <https://doi.org/10.1145/3290336>.
- Paul-André Melliès and Samuel Mimram. Asynchronous games: Innocence without alternation. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 395–411, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74407-8.
- Paul-André Melliès and Léo Stefanescu. An asynchronous soundness theorem for concurrent separation logic. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 699–708, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355834. doi: 10.1145/3209108.3209116. URL <https://doi.org/10.1145/3209108.3209116>.
- Paul-André Melliès and Léo Stefanescu. Concurrent separation logic meets template games. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, pages 742–755, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371049. doi: 10.1145/3373718.3394762. URL <https://doi.org/10.1145/3373718.3394762>.
- Paul-André Melliès and Nicolas Tabareau. An algebraic account of references in game semantics. *Electronic Notes in Theoretical Computer Science*, 249:377–405, 2009. ISSN 1571-0661. doi: <https://doi.org/10.1016/j.entcs.2009.07.099>. URL <https://www.sciencedirect.com/science/article/pii/S1571066109003211>. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).
- Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 3–16, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676970. URL <https://doi.org/10.1145/2676726.2676970>.

- John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, feb 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL <https://doi.org/10.1145/103727.103729>.
- Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
- M.-A. Moens, U. Berni-Canani, and Francis Borceux. On regular presheaves and regular semi-categories. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 43(3):163–190, 2002. URL http://www.numdam.org/item/CTGDC_2002__43_3_163_0/.
- James H. Morris, Jr. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1969. URL <https://dspace.mit.edu/handle/1721.1/64850>. Ph.D. dissertation.
- Andrzej S. Murawski and Nikos Tzevelekos. Game semantics for interface middleweight java. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 517–528, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325448. doi: 10.1145/2535838.2535880. URL <https://doi.org/10.1145/2535838.2535880>.
- Andrzej S. Murawski and Nikos Tzevelekos. Higher-order linearisability. *Journal of Logical and Algebraic Methods in Programming*, 104:86–116, 2019. ISSN 2352-2208. doi: <https://doi.org/10.1016/j.jlamp.2019.01.002>. URL <https://www.sciencedirect.com/science/article/pii/S2352220817302250>.
- Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In Zhong Shao, editor, *Programming Languages and Systems*, pages 290–310, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54833-8.
- Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 396, New York, NY, USA, 1994.

- Association for Computing Machinery. ISBN 0897916549. doi: 10.1145/197917.198176. URL <https://doi.org/10.1145/197917.198176>.
- Peter W. O’Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-28644-8.
- Peter W. O’Hearn and Uday S. Reddy. Objects, interference, and the yoneda embedding. *Theoretical Computer Science*, 228(1):253–282, 1999. ISSN 0304-3975. doi: [https://doi.org/10.1016/S0304-3975\(98\)00360-0](https://doi.org/10.1016/S0304-3975(98)00360-0). URL <https://www.sciencedirect.com/science/article/pii/S0304397598003600>.
- Arthur Oliveira Vale, Paul-André Mellès, Zhong Shao, Jérémie Koenig, and Léo Stefanescu. Layered and object-based game semantics. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498703. URL <https://doi.org/10.1145/3498703>.
- Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. A compositional theory of linearizability. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571231. URL <https://doi.org/10.1145/3571231>.
- Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. A compositional theory of linearizability. *J. ACM*, 71(2), apr 2024a. ISSN 0004-5411. doi: 10.1145/3643668. URL <https://doi.org/10.1145/3643668>.
- Arthur Oliveira Vale, Zhongye Wang, Yixuan Chen, Peixin You, and Zhong Shao. Compositionality and observational refinement for linearizability with crashes. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024b. doi: 10.1145/3689792. URL <https://doi.org/10.1145/3689792>.
- Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. Funtal: reasonably mixing a functional language with assembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 495–509, New York, NY, USA, 2017. Association for Computing Machinery.

- ery. ISBN 9781450349888. doi: 10.1145/3062341.3062347. URL <https://doi.org/10.1145/3062341.3062347>.
- James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In Zhong Shao, editor, *Programming Languages and Systems*, pages 128–148, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54833-8.
- Robin Piedeleu. *Picturing Resources in Concurrency*. PhD thesis, University of Oxford, 2019.
- Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977. doi: 10.1109/SFCS.1977.32.
- Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-X86 architecture. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019a. doi: 10.1145/3371079. URL <https://doi.org/10.1145/3371079>.
- Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019b. doi: 10.1145/3360561. URL <https://doi.org/10.1145/3360561>.
- M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. doi: 10.1147/rd.32.0114.
- Uday S. Reddy. A linear logic model of state. Technical report, Dept. of Computer Science, UIUC, Urbana, IL, 1993.
- Uday S. Reddy. Passivity and independence. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 342–352. IEEE Computer Society, 1994.
- Uday S. Reddy. Global state considered unnecessary: An introduction to object-based semantics. *LISP Symb. Comput.*, 9(1):7–76, 1996.

- Uday S. Reddy. Objects and classes in algol-like languages. *Inf. Comput.*, 172(1):63–97, February 2002. ISSN 0890-5401. doi: 10.1006/inco.2001.2927. URL <https://doi.org/10.1006/inco.2001.2927>.
- Uday S. Reddy. Automata-theoretic semantics of idealized algol with passive expressions. *Electronic Notes in Theoretical Computer Science*, 298:325–348, 2013. ISSN 1571-0661. doi: <https://doi.org/10.1016/j.entcs.2013.09.020>. URL <https://www.sciencedirect.com/science/article/pii/S1571066113000662>. Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIX.
- Uday S. Reddy and Brian P. Dunphy. An automata-theoretic model of idealized algol. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, pages 337–350, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31585-5.
- Christian Retoré. A non-commutative extension of classical linear logic. In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications*, TLCA '97, pages 300–318, Berlin, Heidelberg, 1997. Springer-Verlag. ISBN 3540626883.
- Silvain Rideau and Glynn Winskel. Concurrent strategies. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 409–418, 2011. doi: 10.1109/LICS.2011.13.
- Jan J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.
- Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, July 2009.
- Gerhard Schellhorn, John Derrick, and Heike Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Logic*, 15(4), sep 2014. ISSN 1529-3785. doi: 10.1145/2629496. URL <https://doi.org/10.1145/2629496>.

- Dana Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Oxford University Computing Laboratory, 1970.
- Dana Scott. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136. Springer, 1972.
- Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. Conditional contextual refinement. *Proceedings of the ACM on Programming Languages*, 7(POPL):1121–1151, 2023.
- Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In Zhong Shao, editor, *Programming Languages and Systems*, pages 149–168, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54833-8.
- Edward Szpilrajn. Sur l’extension de l’ordre partiel. *Fundamenta Mathematicae*, 16(1): 386–389, 1930. doi: 10.4064/fm-16-1-386-389. URL <http://eudml.org/doc/212499>.
- R Kent Treiber et al. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research , 1986.
- Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP ’13*, pages 377–390, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323260. doi: 10.1145/2500365.2500600. URL <https://doi.org/10.1145/2500365.2500600>.
- Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74407-8.
- Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP ’06*, pages 129–

- 136, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595931899. doi: 10.1145/1122971.1122992. URL <https://doi.org/10.1145/1122971.1122992>.
- Simon Friis Vindum, Dan Frumin, and Lars Birkedal. Mechanized verification of a fine-grained concurrent queue from meta’s folly library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 100–115, 2022.
- Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. FliT: a library for simple and efficient persistent algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’22, pages 309–321, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392044. doi: 10.1145/3503221.3508436. URL <https://doi.org/10.1145/3503221.3508436>.
- Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive os kernels. In *International Conference on Computer Aided Verification*, pages 59–79. Springer, 2016.
- Yu Zhang, Jérémie Koenig, Zhong Shao, and Yuting Wang. Unifying compositional verification and certified compilation with a three-dimensional refinement algebra. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi: 10.1145/3704900. URL <https://doi.org/10.1145/3704900>.