A Brookes-Style Denotational Semantics for Release/Acquire Concurrency

YOTAM DVIR, Tel Aviv University, Israel
OHAD KAMMAR, University of Edinburgh, Scotland
ORI LAHAV, Tel Aviv University, Israel

We present a compositional denotational semantics for a functional language with first-class parallel composition and shared-memory operations whose operational semantics follows the Release/Acquire weak memory model (RA). The semantics is formulated in Moggi's monadic approach, and is based on Brookes-style traces. To do so we adapt Brookes's traces to Kang et al.'s view-based machine for RA, and supplement Brookes's mumble and stutter closure operations with additional operations, specific to RA. The latter provides a more nuanced understanding of traces that uncouples them from operational interrupted executions. We show that our denotational semantics is adequate and use it to validate various program transformations of interest. This is the first work to put weak memory models on the same footing as many other programming effects in Moggi's standard monadic approach.

CCS Concepts: • **Theory of computation → Denotational semantics**; **Parallel computing models**; *Functional constructs*; *Program analysis*.

Additional Key Words and Phrases: Weak memory models, Release/Acquire, Shared state, Shared memory, Concurrency, Denotational semantics, Monads, Program refinement, Program equivalence, Compiler optimizations

ACM Reference Format:
Yotam Dvir, Ohad Kammar, and Ori Lahav. 2024. A Brookes-Style Denotational Semantics for Release/Acquire Concurrency. 1, 1 (December 2024), 85 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Denotational semantics defines the meaning of programs *compositionally*, where the meaning of a program term is a function of the meanings assigned to its immediate syntactic constituents. This key feature makes denotational semantics instrumental in understanding the meaning a piece of code independently of the context under which the code will run. This style of semantics contrasts with standard operational semantics, which only executes closed/whole programs. A basic requirement of such a denotation function $[\![-]\!]$ is for it to be *adequate* w.r.t. a given operational semantics: plugging program terms M and N with equal denotations—i.e. $[\![M]\!] = [\![N]\!]$—into some program context $\Xi[-]$ that closes over their variables, results in observationally indistinguishable closed programs in the given operational semantics. Moreover, assuming that denotations have a defined order ($\leq$), a "directed" version of adequacy ensures that $[\![M]\!] \leq [\![N]\!]$ implies that all observable behaviors exhibited by $\Xi[M]$ under the operational semantics are also exhibited by $\Xi[N]$.

Authors' Contact Information: Yotam Dvir, yotamdvir@mail.tau.ac.il, Tel Aviv University, Tel Aviv, Israel; Ohad Kammar, ohad.kammar@ed.ac.uk, University of Edinburgh, Edinburgh, Scotland; Ori Lahav, orilahav@tau.ac.il, Tel Aviv University, Tel Aviv, Israel.

ACM XXXX-XXXX/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For shared-memory concurrent programming, Brookes's seminal work [13] defined a denotational semantics, where the denotation $[\![M]\!]_{\mathcal{B}}$ is a set of totally ordered traces of M closed under certain operations, called stutter and mumble. Traces consist of sequences of memory snapshots that M guarantees to provide while relying on its environment to make other memory snapshots. Brookes [12] used the insights behind this semantics to develop a model for separation logic, and Turon and Wand [51] used them to design a separation logic for refinement. Additionally, Xu et al. [53] used traces as a foundation for the Rely/Guarantee approach for verification of concurrent programs, and Liang et al. [36, 37] used a trace-based program logic for refinement.

A *memory model* decides what outcomes an execution of a program can have. Brookes [13] established the adequacy of the trace-based denotational semantics w.r.t. the operational semantics of the strongest model, known as *sequential consistency* (SC), where every memory access happens instantaneously and immediately affects all concurrent threads. However, SC is too strong to model real-world shared memory, whether it be of modern hardware, such as x86-TSO [42] and ARM [47], or of programming languages such as C/C++ [4] and Java [39]. These runtimes follow *weak memory models* that allow performant implementations, but admit more behaviors than SC.

Do weak memory models admit adequate Brookes-style denotational semantics? This question has been answered affirmatively once, by Jagadeesan et al. [25], who closely followed Brookes to define denotational semantics for x86-TSO. Other weak memory models, in particular, models of *programming languages*, and *non-multi-copy-atomic* models, where writes can be observed by different threads in different orders, were so far out of reach of Brookes's totally ordered traces, only captured by much more sophisticated models based on *partial orders* [15, 19, 24, 26, 29, 43].

In this paper we target the Release/Acquire memory model (RA, for short). This model, obtained by restricting the C/C++11 memory model to Release/Acquire atomics, is a well-studied fundamental memory model that is weaker than x86-TSO. RA, roughly speaking, ensures "causal consistency" together with "per-location-SC" and "RMW (read-modify-write) atomicity" [30, 31]. These assurances make RA sufficiently strong for implementing common synchronization idioms. RA allows more performant implementations than SC, since, in particular, it allows the reordering of a write followed by a read from a different location, which is commonly performed by hardware, and it is non-multi-copy-atomic, thus allowing less centralized architectures like POWER [48].

Our first contribution is a Brookes-style denotational semantics for RA. As Brookes's traces are totally ordered, this result may seem counterintuitive. The standard semantics for RA is a declarative (a.k.a. axiomatic) memory model, in the form of acyclicity consistency constraints over partially ordered candidate execution graphs. Since events in these graphs are not totally ordered, one might expect that Brookes's traces are insufficient. Nevertheless, our first key observation is that an *operational* presentation of RA as an interleaving semantics of a weak memory system lends itself to Brookes-style semantics. For that matter, we develop a notion of traces compatible with Kang et al.'s "view-based" machine [28], an operational semantics that is equivalent to RA's declarative formulation. Our main technical result is the (directed) adequacy of the proposed Brookes-style semantics w.r.t. that operational semantics of RA.

A main challenge when developing a denotational semantics lies in making it sufficiently abstract. While *full* abstraction is often out of reach, as a yardstick, we want our semantics to be able to justify various compiler transformations/optimizations that are known to be sound under RA [52]. Indeed, an immediate practical application of a denotational semantics is the ability to provide *local* formal justifications of program transformations, such as those performed by optimizing compilers. In this setting, to show that an optimization $N \twoheadrightarrow M$ is valid amounts to showing that replacing N by M anywhere in a larger program does not introduce new behaviors, which follows from $[\![M]\!] \leq [\![N]\!]$ given a directionally adequate denotation function $[\![-]\!]$.

To support various compiler transformations, we close our denotations under certain operations, including analogs to Brookes's stutter and mumble, but also several RA-specific operations, that allow us to relate programs which would naively correspond to rather different sets of traces. Given these closure operations, our semantics validates standard program transformations, including structural transformations, algebraic laws of parallel programming, and all known thread-local RA-valid compiler optimizations. Thus, the denotational semantics is instrumental in formally establishing validity of transformations under RA, which is a non-trivial task [19, 52].

Our second contribution is to connect the core semantics of parallel programming languages exhibiting weak behaviors to the more standard semantic account for programming languages with effects. Brookes presented his semantics for a simple imperative WHILE language, but Benton et al. [6] and Dvir et al. [20] later use Moggi's monad-based approach [40], defining a semantics for a functional, higher-order core language. In this approach the core language is modularly extended with effect constructs to denote program effects. In particular, we define parallel composition as a first-class operator. This is in contrast to most of the research of weak memory models that employ imperative languages and assume a single top-level parallel composition.

A denotational semantics given in this monadic style comes ready-made with a rich semantic toolkit for program denotation [7], transformations [5, 8–10, 23], reasoning [2, 38], etc. We challenge and reuse this diverse toolkit throughout the development. We follow a standard approach and develop specialized logical relations [45, 49] to establish the compositionality property of our proposed semantics; its soundness, which allows one to use the denotational semantics to show that certain outcomes are impossible under RA; and adequacy. This development puts weak memory models, which often require bespoke and highly specialized presentations, on a similar footing to many other programming effects.

Outline. In §2 we recall the Release/Acquire operational semantics and the trace-based denotational semantics that we use and extend in this paper. In §3 we summarize our contributions.

The rest of the paper goes into further detail. In §4 we present the programming language syntax and typing system, which in §5 we equip with an extended presentation of the RA operational semantics. In §6 we establish semantic invariants for RA that will support our definition of traces. In §7 we define our trace-based denotational semantics for RA, and in §8 we work up to and establish our main results. Finally, we conclude and discuss related work in §9.

Comparison with the conference version. Sections 1-3 and 9 cover of the conference version of this paper [21]. The rest of this paper extends the conference version. Here, definitions and theorems are formally specified and proved. The account in this manuscript also provides a more detailed discussion and more examples. By expanding in breadth and depth, we state (and prove) some results in a stronger form here, such as the denotational semantics supporting transformations involving arbitrary RMWs; and a tighter characterization of the permissible rewrite permutations.

2 Preliminaries

We recall previous work, with minor alterations. Particularly, we present a programming language and its operational semantics under the Sequential Consistency (SC) memory model (§2.1), Brookes's denotational semantics for SC (§2.2), and Kang et al.'s operational presentation of the RA memory model (§2.3). See §4 for the full specification of the programming language, and §5 for a detailed account of the RA operational semantics.

2.1 Language and Operational Semantics

The programming language we use is an extension of a functional language with shared-state constructs. We can compose program terms M and N sequentially: explicitly as $M\,;N$, or implicitly by left-to-right evaluation in the pairing construct $\langle M, N \rangle$. We can compose them in parallel as $M \parallel N$. We assume preemptive scheduling, thus imposing no restrictions on the interleaving of execution steps between parallel threads. To introduce the memory-access constructs, we present the well-known *message passing* litmus test, adapted to the functional setting:

$$(\mathtt{x} \coloneqq 1 \,;\, \mathtt{y} \coloneqq 1) \parallel \langle \mathtt{y}?, \mathtt{x}? \rangle \tag{MP}$$

Here, x and y refer to distinct shared memory locations. Assignment $\ell \coloneqq v$ stores the value v at location ℓ in memory, and dereference $\ell?$ loads a value from ℓ. The language also includes atomic read-modify-write (RMW) constructs. For example, assuming integer storable values, FAA (ℓ, v) (Fetch-And-Add) atomically adds v to the value stored in ℓ. In contrast, interleaving is permitted between the dereferencing, adding, and storing in $\ell \coloneqq (\ell? + v)$. The underlying *memory model* dictates the behavior of the memory-access constructs more precisely.

In the functional setting, execution results in a returned value: $\ell \coloneqq v$ returns the unit value $\langle\rangle$, i.e. the empty tuple; $\ell?$, and the RMW constructs such as FAA (ℓ, v), return the loaded value; $M\,;N$ returns what N returns; and $\langle M, N \rangle$, as well as $M \parallel N$, return the pair consisting of the return value of M and the return value of N. We assume left-to-right execution of pairs, so in the (MP) example $\langle \mathtt{y}?, \mathtt{x}? \rangle$ steps to $\langle v, \mathtt{x}? \rangle$ for a value v loaded from y, and $\langle v, \mathtt{x}? \rangle$ steps to $\langle v, w \rangle$ for a value w loaded from x. The left side of the parallel composition operator ($\parallel$) can step between them, affecting which v and w the right side can observe.

We can use intermediate results in subsequent computations via let binding: $\mathbf{let}\, a = M \,\mathbf{in}\, N$ binds the result of M to a in N. We execute M first, and continue execution of $N[a \mapsto V]$, i.e., substitute the resulting value V for a in N. Similarly, we deconstruct pairs by matching: $\mathbf{match}\, M \,\mathbf{with}\, \langle a, b \rangle.\, N$ binds the components of the pair that M returns to a and b in N. We define the first and second projections **fst** and **snd**, as well as the operation **swap** that swaps the pair constituents, standardly.

Traditionally, we compare weak memory models using litmus test programs, such as (MP), with which one model supports a specific *observable behavior* that the other does not. Since different models feature quite different notions of internal state, and observing the memory directly is not feasible, we ignore internal interactions. We do not consider infinite executions in this paper, so we conflate observable behaviors with *outcomes*: values that the program may evaluate to from given initial memory values. Litmus tests traditionally initialize all initial values to 0.

The literature on weak-memory models traditionally presents litmus tests imperatively using local registers a, b. We instead systematically replace registers with let-bindings. Translating the imperative style to the functional style is mechanical. For example, we compare the imperative message passing litmus tests in the two styles:

Style	Imperative	Functional
Program	$\mathtt{x} \coloneqq 1$ $\mathtt{y} \coloneqq 1$ $\parallel$ $\mathtt{a} \coloneqq \mathtt{y}$ $\mathtt{b} \coloneqq \mathtt{x}$	$\mathtt{x} \coloneqq 1\,;$ $\mathtt{y} \coloneqq 1$ $\parallel$ $\mathbf{let}\,\mathtt{a} = \mathtt{y}?\,\mathbf{in}$ $\mathbf{let}\,\mathtt{b} = \mathtt{x}?\,\mathbf{in}\,\langle \mathtt{a}, \mathtt{b} \rangle$
Relevant outcome	end state: $\mathtt{a} = 1 \wedge \mathtt{b} = 0$	return value: $\langle \langle\rangle\,, \langle 1, 0 \rangle \rangle$

The resulting functional test on the right is equivalent, using standard memory-model agnostic program transformations, to the (MP) program above.

In the strongest memory model of Sequential Consistency (SC), every value stored is immediately made available to every thread, and every dereference must load the latest stored value. The underlying memory model uses maps from locations to values for the memory state that evolves during program execution. Given an initial state, the behavior of a program in SC depends only on the choice of interleaving of steps. In (MP) the order of the two stores and the two loads ensures that $\langle \langle\rangle\,, \langle 0, 0 \rangle \rangle$, $\langle \langle\rangle\,, \langle 0, 1 \rangle \rangle$, and $\langle \langle\rangle\,, \langle 1, 1 \rangle \rangle$ are observable, but $\langle \langle\rangle\,, \langle 1, 0 \rangle \rangle$ is not.

Observable behavior as defined for whole programs is too crude to study program *terms* that can interact with the program context within which they run. Indeed, compare M_1 defined as $\mathtt{x} \coloneqq 1 ; \mathtt{y} \coloneqq \boxed{1} ; \mathtt{y}?$ versus M_2 defined as $\mathtt{x} \coloneqq 1 ; \mathtt{y} \coloneqq \boxed{\mathtt{x}?} ; \mathtt{y}?$. Under SC, the difference between them as whole programs is unobservable: starting from any initial state both return 1. Now consider them within the program context $- \parallel \mathtt{x} \coloneqq 2$. That is, compare $M_1 \parallel \mathtt{x} \coloneqq 2$ versus $M_2 \parallel \mathtt{x} \coloneqq 2$. In the first, M_1 still always returns 1; but in the second, M_2 can also return 2 by interleaving the store of 2 in x immediately after the store of 1 in x. Thus, if $[\![M]\!]$, i.e. M's denotation, were to simply map initial states to possible results according to executions of M, we could not define $[\![M \parallel N]\!]$ in terms of $[\![M]\!]$ and $[\![N]\!]$ alone, because we would have $[\![M_1]\!] = [\![M_2]\!]$ but also $[\![M_1 \parallel \mathtt{x} \coloneqq 2]\!] \neq [\![M_2 \parallel \mathtt{x} \coloneqq 2]\!]$. Therefore $[\![M]\!]$ must contain more information about M than an "input-output" relation; it must account for interference by the environment.

2.2 Brookes's Trace-based Semantics for Sequential Consistency

Brookes's [13] prominent approach defines fully-compositional denotational semantics for concurrent programs. He defined a denotational semantics $[\![-]\!]_{\mathcal{B}}$ for SC by taking $[\![M]\!]_{\mathcal{B}}$ to be a set of traces of M closed under certain closure rules as we detail below. Brookes established a (directional) adequacy theorem: if $[\![M]\!]_{\mathcal{B}} \supseteq [\![N]\!]_{\mathcal{B}}$ then the transformation $M \twoheadrightarrow N$ is valid under SC. The latter means that, when assuming SC-based operational semantics, M can be replaced by N within a program without introducing new observable behaviors for it. Thus, adequacy formally grounds the intuition that the denotational semantics soundly captures behavior of program terms.

As a particular practical benefit, we can replace formal and informal simulation arguments that justify transformations in operational semantics by cleaner and simpler proofs based on the denotational semantics. For example, a simple argument shows that $[\![\mathtt{x} \coloneqq v ; \mathtt{x} \coloneqq w]\!]_{\mathcal{B}} \supseteq [\![\mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$ holds in Brookes's semantics. Thanks to adequacy, this can justify the transformation Write-Write Elimination (WW-Elim) $\mathtt{x} \coloneqq v ; \mathtt{x} \coloneqq w \twoheadrightarrow \mathtt{x} \coloneqq w$ in SC.

Traces in SC. In Brookes's semantics, a program term denotes a set of traces, each trace consisting of a sequence of transitions. Each transition is of the form $\langle \mu, \rho \rangle$, where μ and ρ are memories, i.e. maps from locations to values. A transition describes a program term's execution relying on a memory state snapshot μ in order to guarantee the memory state snapshot ρ.

For example, $[\![\mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$ includes all traces of the form $\boxed{\langle \rho, \rho\,[\mathtt{x} \coloneqq w] \rangle}$, where $\rho\,[\mathtt{x} \coloneqq w]$ is equal to ρ except for mapping x to w. The definition is compositional: we obtain the traces in $[\![\mathtt{x} \coloneqq v ; \mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$ from sequential compositions of traces of $[\![\mathtt{x} \coloneqq v]\!]_{\mathcal{B}}$ with traces of $[\![\mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$, including all traces of the form $\boxed{\langle \mu, \mu\,[\mathtt{x} \coloneqq v] \rangle \langle \rho, \rho\,[\mathtt{x} \coloneqq w] \rangle}$. Such a trace relies on μ in order to guarantee $\mu\,[\mathtt{x} \coloneqq v]$, and then relies on ρ in order to guarantee $\rho\,[\mathtt{x} \coloneqq w]$. Allowing $\rho \neq \mu\,[\mathtt{x} \coloneqq v]$ reflects the possibility of environment interference between the two store instructions. Indeed, when denoting parallel composition $[\![M \parallel N]\!]_{\mathcal{B}}$ we include all traces obtained by interleaving transitions from a trace from $[\![M]\!]_{\mathcal{B}}$ with transitions from a trace from $[\![N]\!]_{\mathcal{B}}$. By sequencing and interleaving, one subterm's guarantee can fulfill the requirement which another subterm relies on. They may also relegate reliances and guarantees to their mutual context.

In the functional setting, executions not only modify the state but also return values. In this setting, traces are pairs, which we write as $\boxed{\xi} \therefore r$, where ξ is the sequence of transitions and r represents the final value that the program term guarantees to return [6]. For example, the semantics of dereference $[\![\mathtt{x}?]\!]_{\mathcal{B}}$ includes all traces of the form $\boxed{\langle \mu, \mu \rangle} \therefore \mu(\mathtt{x})$. Indeed, the execution of x? does not change the memory and returns the value loaded from x. In the semantics of assignment $[\![\mathtt{x} \coloneqq v]\!]_{\mathcal{B}}$, instead of $\boxed{\langle \mu, \mu\,[\mathtt{x} \coloneqq v] \rangle}$ we have $\boxed{\langle \mu, \mu\,[\mathtt{x} \coloneqq v] \rangle} \therefore \langle \rangle$.

Closure rules in SC. Were denotations in Brookes's semantics defined to *only* include the traces we explicitly mentioned above, it would not be abstract enough to justify (WW-Elim), which eliminates redundant writes. Indeed, we only saw traces with two transitions in $[\![\mathtt{x} \coloneqq v \,;\, \mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$, but in $[\![\mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$ we saw traces with one. The semantics would still be adequate, but it would lack abstraction. To achieve abstraction, Brookes introduces another main idea: closing the denotations under two closure conditions. These are:

Stutter: if $\boxed{\xi\eta} \therefore r$ is in the set, then $\boxed{\xi \langle \mu, \mu \rangle \, \eta} \therefore r$ is too. Intuitively, a program term can always guarantee what it relies on.

Mumble: if $\boxed{\xi \langle \mu, \rho \rangle \, \langle \rho, \theta \rangle \, \eta} \therefore r$ is in the set, then $\boxed{\xi \langle \mu, \theta \rangle \, \eta} \therefore r$ is too. Intuitively, a program term can always omit a guarantee to the environment, and rely on its own omitted guarantee instead of relying on the environment.

Denotations in Brookes's semantics are sets of traces that are *closed* under these conditions. For example, $[\![\mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$ is the least closed set with all traces of the form $\boxed{\langle \rho, \rho\,[\mathtt{x} \coloneqq w] \rangle} \therefore \langle\rangle$, and $[\![\mathtt{x} \coloneqq v \,;\, \mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$ is the least closed set with all sequential compositions of traces of $[\![\mathtt{x} \coloneqq v]\!]_{\mathcal{B}}$ with trace of $[\![\mathtt{x} \coloneqq w]\!]_{\mathcal{B}}$.

The closure conditions in Brookes's semantics make the traces in $[\![M]\!]_{\mathcal{B}}$ correspond precisely to *interrupted executions* of M, which are executions of M in which the memory can arbitrarily change between steps of execution. Each transition $\langle \mu, \rho \rangle$ in a trace in $[\![M]\!]_{\mathcal{B}}$ corresponds to multiple execution steps of M that transition μ into ρ, and each gap between transitions accounts for possible environment interruption. The closure rules maintain this correspondence: stutter corresponds to taking 0 steps, and mumble corresponds to taking $n + m$ steps instead of taking n steps and then m steps when the environment did not observably change the memory in between. Brookes's adequacy proof is based on this precise correspondence. In particular, the single-pair traces in $[\![M]\!]_{\mathcal{B}}$ correspond to the (uninterrupted) executions, the "input-output" relation, of M.

2.3 Overview of Release/Acquire Operational Semantics

Memory accesses in RA are more subtle than in SC. We adopt Kang et al.'s "view-based" machine [28], an operational presentation of RA proven to be equivalent to the original declarative formulation of RA [e.g. 31]. In this model, rather than the memory holding only the latest value written to every variable, the memory accumulates a set of memory update messages for each location. Each thread maintains its own *view* that captures which messages the thread can observe, by constraining the messages that the thread may read and write. The messages in the memory carry views as well, which are inherited from the thread that wrote the message, and passed to any thread that reads the message. This indirectly maintains a causal relationship between messages in memory throughout the evolution of the system.

More concretely, causality is enforced by timestamping messages, thus placing them on their location's *timeline*. A view κ associates a timestamp κ_ℓ to each location ℓ, obscuring the portion of ℓ's timeline before κ_ℓ. The view *points to* a message at ℓ with timestamp κ_ℓ. A message point to messages via the view it carries. Every message must point to itself.

To capture the atomicity of RMWs, each message occupies a half-open segment $(q, t]$ on their location's timeline, where t is the message's timestamp. A message with segment $(q, t]$ *dovetails after* a message at the same location with timestamp q, if there is one. When an RMW writes it must dovetail after the message it read. Messages are *apart* if neither dovetails after the other.

We explain our notation using the example memory in Figure 1 (top) which consists of two locations, x and y. Consider the notation of the message $\nu_3 \coloneqq \mathtt{x}{:}1@(.5, \underline{1.7}]\,\langle\!\langle \mathtt{y}@3.5 \rangle\!\rangle$ in this memory:

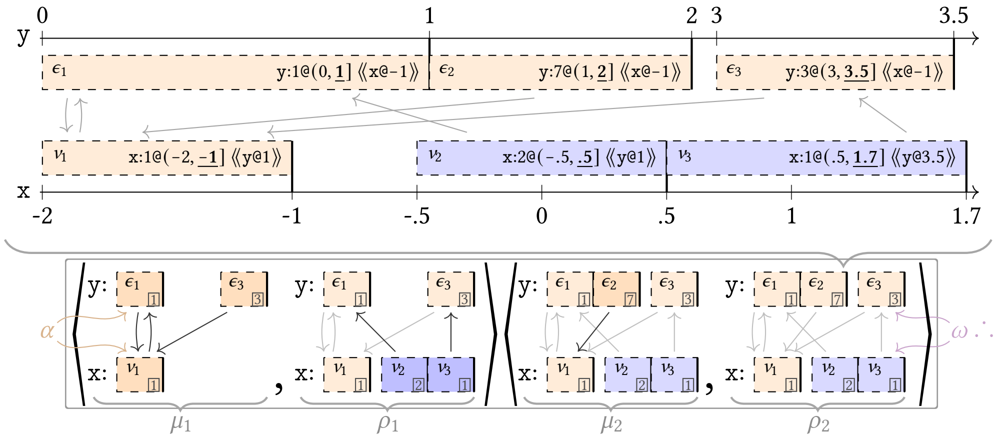


Fig. 1. Illustrations of a memory (top) and a trace (bottom), involving two memory locations, x and y. **Top:** A memory holding six messages. The timelines are deliberately misaligned and have different scales, emphasizing that timestamps for different locations are incomparable and that only the order between them is relevant. The arrows pointing between messages illustrate the graph structure that the views impose. Messages are spatially parted iff they are apart, e.g. ν_3 dovetails after ν_2, which is apart from ν_1. **Bottom:** A trace with two transitions: $\alpha \boxed{\langle \mu_1, \rho_1 \rangle \langle \mu_2, \rho_2 \rangle} \omega \therefore 5$. The memory illustrated on top is ρ_2. We highlight messages that are not part of a previous memory. The local messages are ν_2 and ν_3; the rest are environment messages.

- its location is x;
- it carries the value 1;
- it occupies the segment $(.5, 1.7]$ on x's timeline;
- it carries the view κ such that:
  - $\kappa_{\mathtt{x}} = 1.7$, so ν_3 points at itself, like every message; and
  - $\kappa_{\mathtt{y}} = 3.5$, so ν_3 points at ϵ_3.

A thread may only write a message with a timestamp visible from its view. When a thread writes to a location ℓ, it must increase the timestamp its view associates with ℓ and use its new view as the message's view. The message's segment must not overlap with any other segment on ℓ's timeline. In particular, only one message can ever dovetail after a given message. A thread can only read from messages visible from its view, and when it reads, its view increases as needed to *dominate* the view of the loaded message. Formally, a view ω dominates a view α, written $\alpha \leq \omega$, if $\alpha_\ell \leq \omega_\ell$ for every ℓ. Increasing the view in this way may obscure messages at the location of the read as well as other locations.

Revisit the (MP) litmus test: $(\mathtt{x} \coloneqq 1 \,;\, \mathtt{y} \coloneqq 1) \parallel \langle \mathtt{y}?, \mathtt{x}? \rangle$. Start with a memory with a single message holding 0 at each location, and with all views pointing to the timestamps of these message. Suppose the right thread loaded 1 from y, as Figure 2 (left) depicts. Such a message can only be available if the left thread stored it. Before storing 1 to y, the left thread stored 1 to x, obscuring the initial x message from its view. The right thread inherits this causal constraint by inheriting the view, preventing a load of 0 from x. Therefore, RA forbids the outcome $\langle \langle \rangle, \langle 1, 0 \rangle \rangle$.

In contrast, consider the *store buffering* litmus test:

$$(\mathtt{x} \coloneqq 1 \,;\, \mathtt{y}?) \parallel (\mathtt{y} \coloneqq 1 \,;\, \mathtt{x}?) \tag{SB}$$

By considering the possible interleavings, one can check that no execution in SC returns $\langle 0, 0 \rangle$. However, in RA some do. Indeed, even if the left thread stores to x before the right thread loads from x, the right thread's view allows it to load 0, as Figure 2 (right) depicts.

We can recover the SC behavior by interspersing fences between sequenced memory accesses, which we impose with FAA $(\mathtt{z}, 0)$ to a third location z. Compare (SB) to the *store buffering with*

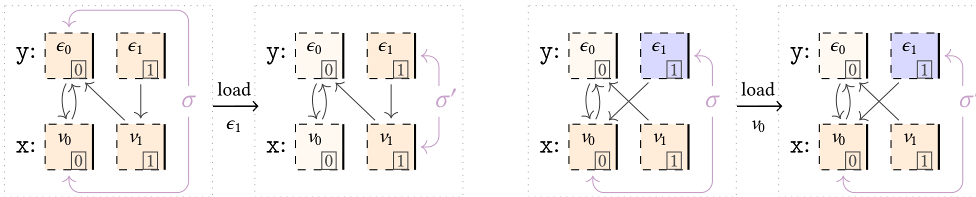


Fig. 2. Depictions of a step during an execution of a litmus test, with the view of the right thread changing from σ to σ'. The value that each message carries is in its bottom-right corner. Views are illustrated implicitly in the graph structure that they impose. Obscured messages are faded. **Left:** As the right thread in (MP) loads 1 from y, it inherits the view of ϵ_1, obscuring v_0. **Right:** The right thread in (SB) loading 0 from x. The earlier step of storing ϵ_1 did not obscure v_0.

fences litmus test:

$$(\mathrm{x} \coloneqq 1 \,; \mathrm{FAA}\,(\mathrm{z}, 0) \,; \mathrm{y}?) \parallel (\mathrm{y} \coloneqq 1 \,; \mathrm{FAA}\,(\mathrm{z}, 0) \,; \mathrm{x}?) \tag{SB+F}$$

Each FAA $(\mathrm{z}, 0)$ instruction stores a message v_i that must dovetail after the message that they load from, and inherit that message's view. They cannot both dovetail after the initial message at z because their segments cannot intersect. Thus, one of them will have to dovetail after the other. Suppose the message v_2 from the right thread dovetails after v_1 from the left. In this scenario, the view of the message v_1 points to the message ϵ_1 the left thread had previously stored at x. When the right thread loads v_1 it inherits this view, obscuring the initial message ϵ_0 at x. Therefore, when it later loads from x, it must load from ϵ_1 the value 1. Similarly, in the case of v_1 dovetailing after v_2 the left thread must load 1 from y. Thus, like in SC, no execution in RA returns $\langle 0, 0\rangle$.

3 Contribution Summary

We present our main contributions. We start with our notion of a trace, which we adapt to RA both in the structure of the trace itself, as well as in the closure rules we impose (§3.1). We then briefly explain the way in which our semantics is standard, and a few beneficial consequences of this fact (§3.2). Finally, we connect our denotational semantics to the operational semantics of RA (§3.3), showing both adequacy and sufficient abstraction.

3.1 Traces for Release/Acquire

As in Brookes's SC-traces, our RA-traces include a sequence of transitions ξ, each transition a pair of RA memories; and a return value r. We impose analogues to the stutter and mumble closure rules. The operational semantics only adds messages and never modifies them. We consequently require that every memory snapshot in the sequence ξ be contained in the subsequent one, whether it be within or across transitions. A message added within a transition is a *local message*; otherwise it is an *environment message*. We call the first memory in ξ's first transition its *opening memory*, and the second memory in ξ's last transition its *closing memory*. In addition, RA-traces include an initial view α, declaring which messages they rely on to be revealed in ξ's opening memory; and a final view ω, declaring the messages they guarantee to not obscure in ξ's closing memory. We write the trace as $\alpha \boxed{\xi} \omega \therefore r$. See Figure 1 (bottom) for an illustrated example.

RA specific closure rules. We close the denotations $[\![-]\!]_{\mathcal{A}}$ for RA under further closure rules to make the denotational semantics more abstract. For example, to justify the RA-valid (WW-Elim). The reasoning we have used to justify it under SC, by showing $[\![\mathrm{x} \coloneqq v \,; \mathrm{x} \coloneqq w]\!]_{\mathcal{B}} \supseteq [\![\mathrm{x} \coloneqq w]\!]_{\mathcal{B}}$ in Brookes's semantics, will only get us so far in RA without an additional closure rule. Replicating the process, the trace we have in $[\![\mathrm{x} \coloneqq v \,; \mathrm{x} \coloneqq w]\!]_{\mathcal{A}}$ thanks to closure under mumble has two local messages, whereas traces from $[\![\mathrm{x} \coloneqq w]\!]_{\mathcal{A}}$ only have a single local message. We did not have this problem in SC, where memory is more abstract, satisfying $\mu\,[\mathtt{x} \coloneqq v]\,[\mathtt{x} \coloneqq w] = \mu\,[\mathtt{x} \coloneqq w]$. We resolve this by closing under the absorb closure rule, which replaces two dovetailed local messages with one that carries the second message's value. To justify (WW-Elim) in RA, we appeal to absorb closure after appealing to mumble closure.

Internalized operational invariants. The validity of program transformations sometimes depends on semantic invariants of the operational semantics (§6). For example, consider the transformation $\mathtt{x?}\,;\mathtt{y?} \twoheadrightarrow \mathtt{y?}$. Consider the state S that consists of the memory at Figure 1 (top) and the view that points to v_3 and ϵ_2. The only step $\mathtt{x?}\,;\mathtt{y?}$ can take from the state S is to load v_3, inheriting the view that v_3 carries, which changes the thread's view to point to ϵ_3. Only ϵ_3 is available in the following step, which means the term returns 3. In contrast, starting from S, the term $\mathtt{y?}$ can load from ϵ_2 to return 7. This analysis does not invalidate the transformation because the state S is unreachable by an execution starting from an initial state, and should therefore be ignored when determining observable behaviors.

Just as we restrict our attention to reachable states when analyzing the operational semantics, we restrict our denotational domain to traces that possess analogous properties (§7.2). This eliminates "junk": undefinable traces that can differentiate between denotations. By forbidding junk we increase abstraction, validating more transformations. A restriction on the initial view and opening memory corresponding to the example above implies $[\![\mathtt{x?}\,;\langle\rangle]\!]_{\mathcal{A}} \supseteq [\![\langle\rangle]\!]_{\mathcal{A}}$, justifying the RA-valid Irrelevant Read Elimination (R-Elim) $\mathtt{x?}\,;\langle\rangle \twoheadrightarrow \langle\rangle$. The transformation $\mathtt{x?}\,;\mathtt{y?} \twoheadrightarrow \mathtt{y?}$ follows structurally, as we explain below.

3.2 Compositionality and the Monadic Presentation

We bridge weak-memory models with Moggi's monad-based approach to denotational semantics. This approach has a built-in semantic framework for the effect-free fragment of the language, to which effect constructs can be modularly added. Reasoning about the effect-free fragment is valid in all instances. Thanks to our contribution, this is also valid for RA weak memory. For example, $[\![\langle\rangle\,;M]\!]_{\mathcal{A}} = [\![M]\!]_{\mathcal{A}}$ and $[\![(M\,;N)\,;K]\!]_{\mathcal{A}} = [\![M\,;(N\,;K)]\!]_{\mathcal{A}}$ follow by structural reasoning. That is, without appealing to the specifics of the RA denotations. These examples justify transformations that we use, together with (R-Elim), to deduce $\mathtt{x?}\,;\mathtt{y?} \twoheadrightarrow \mathtt{y?}$:

$$\mathtt{x?}\,;\mathtt{y?} \twoheadrightarrow \mathtt{x?}\,;(\langle\rangle\,;\mathtt{y?}) \twoheadrightarrow (\mathtt{x?}\,;\langle\rangle)\,;\mathtt{y?} \twoheadrightarrow \langle\rangle\,;\mathtt{y?} \twoheadrightarrow \mathtt{y?}$$

As another example, if K is effect-free, then:

$$[\![\mathbf{if}\;K\;\mathbf{then}\;M\,;N\;\mathbf{else}\;M\,;N'\,]\!]_{\mathcal{A}} = [\![M\,;\mathbf{if}\;K\;\mathbf{then}\;N\;\mathbf{else}\;N'\,]\!]_{\mathcal{A}}$$

So-called structural transformations may otherwise require challenging ad-hoc proofs [e.g. 24, 26].

Higher order. An important aspect of a programming language is its facilitation of abstraction. Higher-order programming is a flexible instance of this aspect, in which programmable functions can take functions as input and return functions as output. Moggi's approach supports higher-order functions out-of-the-box without complicating the rest of the semantics.

Every value returned by an execution has a semantic presentation which we use as the return value in traces. The semantic and syntactic values coincide in first-order types, but different syntactic functions may have the same semantics, so the identification does not extend to the entire higher-order language.

A term is a *program* when it is *closed* (every variable occurrence is bound) and of *ground type* (resulting value is not a function). Thus a program returns a concrete result that the end-user can observe. To prove properties about higher-oder programs we use logical relations [45, 49]. Moggi's toolkit provides a standard way to define logical relations, thereby lifting properties to their higher-order counterparts.

Compositionality. In its most basic form, compositionality means that a term's denotation is defined using the denotations of its immediate subterms. In our case denotations are sets, where each elements represents a possible behavior of the term. We establish a directional generalization of compositionality:

Compositionality (Thm. 8.11)**.** *If* $[\![M]\!]_{\mathcal{A}} \subseteq [\![N]\!]_{\mathcal{A}}$ *then* $[\![\Xi\,[M]]\!]_{\mathcal{A}} \subseteq [\![\Xi\,[N]]\!]_{\mathcal{A}}$ *for every program context* $\Xi\,[-]$.

Compositionality follows from the monadic design of the denotational semantics using monotonic operators, in the same way it does for SC [e.g. 20].

3.3 Relating the Denotational Semantics to the Operational Semantics

Kang et al.'s presentation assumes top-level parallelism, a common practice in studies of weak-memory models. Restricting parallelism to the top level makes the language non-uniform and programming in it a non-compositional activity. We extend Kang et al.'s operational semantics to support first-class parallelism by organizing thread views in an evolving *view-tree*, a binary tree with view-labeled leaves, rather than in a fixed flat mapping. Thus, a *configuration* consists of a term, a memory, and a view-tree. The latter two are the configuration's *state*. In discourse, we do not distinguish between a view-leaf and its label.

Consider the Write-Read Reordering transformation (WR-Reord) $(\mathrm{x} \coloneqq v);\mathrm{y?} \twoheadrightarrow \mathbf{fst}\ \langle \mathrm{y?}, (\mathrm{x} \coloneqq v)\rangle$ that is a crucial reordering of memory accesses valid under RA but not SC. With first-class parallelism, we decompose (WR-Reord) into a combination of structural transformations, laws of parallel programming, and Write-Read Deorder (WR-Deord) $\langle (\mathrm{x} \coloneqq v)\,, \mathrm{y?}\rangle \twoheadrightarrow (\mathrm{x} \coloneqq v) \parallel \mathrm{y?}$:

$$(\mathrm{x} \coloneqq v)\,;\mathrm{y?} \overset{\text{Structural}}{\twoheadrightarrow} \mathbf{snd}\ \langle (\mathrm{x} \coloneqq v)\,, \mathrm{y?}\rangle \overset{\text{(WR-Deord)}}{\twoheadrightarrow} \mathbf{snd}\ ((\mathrm{x} \coloneqq v) \parallel \mathrm{y?})$$

$$\overset{\text{Par. Prog. Law: Symmetry}}{\twoheadrightarrow} \mathbf{snd}\ (\mathbf{swap}\ (\mathrm{y?} \parallel (\mathrm{x} \coloneqq v))\,) \overset{\text{Structural}}{\twoheadrightarrow} \mathbf{fst}\ (\mathrm{y?} \parallel (\mathrm{x} \coloneqq v)) \overset{\text{Par. Prog. Law: Sequencing}}{\twoheadrightarrow} \mathbf{fst}\ \langle \mathrm{y?}, (\mathrm{x} \coloneqq v)\rangle$$

This decomposition separates concerns: we validate each step with our semantics using independent arguments. It also highlights (WR-Deord) as the crucial step, as the rest are valid under SC.

Concrete semantics. Our contributions use a simpler semantics $[\![-]\!]_{C}$ in which the sets of traces are closed under the *concrete* rules, such as stutter and mumble, but not under the *abstract* rules, such as absorb. This is in contrast to $[\![-]\!]_{\mathcal{A}}$ which are closed under both the concrete and the abstract rules. The concrete rules saturate the trace-set with traces that account for possible interaction with the environment. They make sequential and parallel composition straightforward, a mere sequencing or interleaving of the transitions from the composed constituents. The abstract rules make the semantics more abstract, justifying more transformations.

The concrete semantics is saturated enough to include all program evaluations:

Soundness (Thm. 8.12)**.** *If a program* M *evaluates, starting in the state* $\langle \alpha, \mu\rangle$*, to a value* V*, then* $\alpha \boxed{\langle \mu, \rho\rangle}\, \omega \mathrel{.\!\cdot\!.} V \in [\![M]\!]_{C}$ *for some state* $\langle \omega, \rho\rangle$.

A crucial observation in our technical development is that we can percolate the applications of abstract closure rules "outwards":

Rewrite Commutativity (Lem. 8.3)**.** *Let τ and ϱ be traces such that from τ we can obtain ϱ using the concrete and abstract closure rules (denoted $\tau \xrightarrow{\mathfrak{ca}} \varrho$). Then there exists a trace π, such that from τ we can obtain π using only concrete closure rules (denoted $\tau \xrightarrow{\mathfrak{c}} \pi$), and from π we can obtain ϱ using only abstract closure rules (denoted $\pi \xrightarrow{\mathfrak{a}} \varrho$).*

Thanks to this combinatorial analysis, we prove that the two semantics are compatible:

Retroactive Closure (Lem. 8.7)**.** *If M is a program, then $[\![M]\!]_{\mathcal{A}}$ is equal to the closure of $[\![M]\!]_{C}$ under the abstract closure rules (denoted $[\![M]\!]_{\mathcal{A}} = [\![M]\!]_{C}^{\mathfrak{a}}$).*

As an immediate consequence, soundness holds for the abstract semantics as well.

Abstract adequacy. The abstract semantics $[\![M]\!]_{\mathcal{A}}$ includes traces that M does not exhibit concretely, but an equivalent program does. For example, $[\![\mathtt{x} \coloneqq v \,;\, \mathtt{x} \coloneqq w]\!]_{\mathcal{A}}$ includes the traces that $\mathtt{x} \coloneqq w$ exhibits concretely thanks to absorb-closure, justifying (WW-Elim).

Nevertheless, each trace in the abstract denotation corresponds to an observable behavior:

Evaluation Lemma (Lem. 8.15)**.** *For every program M and $\alpha \boxed{\langle \mu, \rho \rangle} \omega \therefore r \in [\![M]\!]_{\mathcal{A}}$ there is an evaluation of M, starting from the state $\langle \alpha, \mu \rangle$, to the value r.*

The lack of correspondence with the final state is an artifact of the concreteness-abstraction divergence between the operational and denotational semantics. Due to this divergence, it is significantly more challenging to establish this direction of the correspondence than in previous work.

The central result is (directional) adequacy, stating that denotational approximation corresponds to refinement of observable behaviors:

Adequacy (Thm. 8.13)**.** *If $[\![M]\!]_{\mathcal{A}} \subseteq [\![N]\!]_{\mathcal{A}}$, then for all program contexts $\Xi[-]$, every observable behavior of $\Xi[M]$ is an observable behavior of $\Xi[N]$: for every evaluation of $\Xi[M]$ there is an evaluation of $\Xi[N]$ from the same initial state to the same value.*

In particular, $[\![M]\!]_{\mathcal{A}} \subseteq [\![N]\!]_{\mathcal{A}}$ implies that $N \twoheadrightarrow M$ is valid under RA, because the effect of applying it is unobservable. Adequacy follows immediately from the above results. Indeed, using soundness, an observable behavior of $\Xi[M]$ corresponds to a single-transition $\tau \in [\![\Xi[M]]\!]_{\mathcal{A}}$; by the assumption and compositionality $\tau \in [\![\Xi[N]]\!]_{\mathcal{A}}$; and using the evaluation lemma, τ corresponds to an observable behavior of $\Xi[N]$.

Sufficient abstraction. Brookes's denotational semantics $[\![-]\!]_{\mathcal{B}}$ is *fully abstract*, meaning that the converse to adequacy also holds: if $N \twoheadrightarrow M$ is valid under SC, then $[\![N]\!]_{\mathcal{B}} \supseteq [\![M]\!]_{\mathcal{B}}$. However, Brookes's proof relies on an artificial program construct, **await**, that permits waiting for a specified memory snapshot and then stepping (atomically) to a second specified memory snapshot. In realistic languages, when this construct is unavailable, Brookes's full abstraction proof does not apply and full abstraction fails, as shown by Svyatlovskiy et al. [50].

Nevertheless, even without full abstraction, an adequate semantics can be abstract "enough" by ensuring that it supports known transformations. To the best of our knowledge, all transformations $N \twoheadrightarrow M$ proven to be valid under RA in the existing literature are supported by our denotational semantics, i.e. $[\![N]\!]_{\mathcal{A}} \supseteq [\![M]\!]_{\mathcal{A}}$. Structural transformations are supported by virtue of using Moggi's standard semantics. Our semantics also validates "algebraic laws of parallel programming", such as sequencing $M \parallel N \twoheadrightarrow \langle M, N \rangle$ and its generalization that Hoare and van Staden [22] recognized, $(M_1 \,;\, M_2) \parallel (N_1 \,;\, N_2) \twoheadrightarrow (M_1 \parallel N_1) \,;\, (M_2 \parallel N_2)$, which in the functional setting can take the more expressive form in which the values returned are passed on to the following computation. We present our supported transformations in Figure 3.

Laws of Parallel Programming				
Symmetry	$M \parallel N$	$\twoheadrightarrow$	$\mathbf{swap}\ (N \parallel M)$	
Generalized Sequencing				
	$(\mathbf{let}\, a = M_1\ \mathbf{in}\ M_2) \parallel (\mathbf{let}\, b = N_1\ \mathbf{in}\ N_2)$	$\twoheadrightarrow$	$\mathbf{match}\, M_1 \parallel N_1\ \mathbf{with}\ \langle a, b\rangle.\, M_2 \parallel N_2$	
Eliminations				
Irrelevant Read	$\ell? \,;\, \langle\rangle$	$\twoheadrightarrow$	$\langle\rangle$	
Write-Write	$\ell \coloneqq v \,;\, \ell \coloneqq w$	$\overset{\mathsf{Ab}}{\twoheadrightarrow}$	$\ell \coloneqq w$	
Write-Read	$\ell \coloneqq v \,;\, \ell?$	$\twoheadrightarrow$	$\ell \coloneqq v \,;\, v$	
Write-FAA	$\ell \coloneqq v \,;\, \mathrm{FAA}\,(\ell, w)$	$\overset{\mathsf{Ab}}{\twoheadrightarrow}$	$\ell \coloneqq (v + w) \,;\, v$	
Read-Write	$\mathbf{let}\, a = \ell?\ \mathbf{in}\ \ell \coloneqq (a + v) \,;\, a$	$\twoheadrightarrow$	$\mathrm{FAA}\,(\ell, v)$	
Read-Read	$\langle \ell?, \ell? \rangle$	$\twoheadrightarrow$	$\mathbf{let}\, a = \ell?\ \mathbf{in}\ \langle a, a\rangle$	
Read-FAA	$\langle \ell?, \mathrm{FAA}\,(\ell, v)\rangle$	$\twoheadrightarrow$	$\mathbf{let}\, a = \mathrm{FAA}\,(\ell, v)\ \mathbf{in}\ \langle a, a\rangle$	
FAA-Read	$\langle \mathrm{FAA}\,(\ell, v), \ell? \rangle$	$\twoheadrightarrow$	$\mathbf{let}\, a = \mathrm{FAA}\,(\ell, v)\ \mathbf{in}\ \langle a, a + v\rangle$	
FAA-FAA	$\langle \mathrm{FAA}\,(\ell, v), \mathrm{FAA}\,(\ell, w)\rangle$	$\overset{\mathsf{Ab}}{\twoheadrightarrow}$	$\mathbf{let}\, a = \mathrm{FAA}\,(\ell, v + w)\ \mathbf{in}\ \langle a, a + v\rangle$	
Others				
Irrelevant Read Introduction	$\langle\rangle$	$\twoheadrightarrow$	$\ell? \,;\, \langle\rangle$	
Read to FAA	$\ell?$	$\overset{\mathsf{Di}}{\twoheadrightarrow}$	$\mathrm{FAA}\,(\ell, 0)$	
Write-Read Deorder	$\langle (\ell \coloneqq v), \ell'? \rangle$	$\overset{\mathsf{Ti}}{\twoheadrightarrow}$	$(\ell \coloneqq v) \parallel \ell'?$	$(\ell \neq \ell')$
Write-Read Reorder	$(\ell \coloneqq v) \,;\, \ell'?$	$\overset{\mathsf{Ti}}{\twoheadrightarrow}$	$\mathbf{fst}\ \langle \ell'?, (\ell \coloneqq v)\rangle$	$(\ell \neq \ell')$

Fig. 3. A representative list of transformations the denotational semantics $[\![-]\!]_{\mathcal{A}}$ supports. Along with Symmetry, the denotational semantics supports all symmetric-monoidal laws with the binary operator ($\parallel$) and the unit $\langle\rangle$. The structural transformations supported due to the semantics being monad-based are omitted. The semantics also supports similar transformations involving RMWs other than FAA. The list mentions the abstract closure rules that the proofs appeal to.

4 Language and Typing

We consider a standard extension of Moggi's [40] computational lambda calculus with products and variants (labeled sums) further extending it with shared-memory constructs. We parameterize our language, which we call $\boldsymbol{\lambda}_{\mathrm{RA}}$, by its globally available locations, the values we store in and retrieve from these locations, and the primitives we use to mutate these values atomically through a unified *read-modify-write* construct.

Locations and Storable Values. We fix two finite sets of *(shared memory) locations* Loc, ranged over by ℓ, ℓ'; and *(storable) values* Val, ranged over by v, w, u. For example, we may take Loc and Val to be all 64-bit sequences. In concrete examples, we will use concrete names such as $\mathtt{x}, \mathtt{y}, \mathtt{z}$ for distinct locations, and numbers for values. For simplicity, we don't include primitives (such as addition) explicitly, since they require standard minor changes.

Read-modify-write (RMW). These constructs read a value from memory atomically and possibly modify it to some other computed value. Typical languages include the following constructs, which are efficiently compiled to hardware: **Compare-and-Swap:** modify when the stored value matches the parameter; **Fetch-and-Add:** increase the stored value by the parameter; and **Exchange:** modify the stored value to the parameter. For convenience, we include a single RMW construct that expresses all such operations, *as well as standard loads*. This generalization, especially bringing together loads with RMW operations, is non-standard, but makes our development more uniform.

Formally, a *modifier* is a partial function $\Phi : \mathsf{Val} \rightharpoonup \mathsf{Val}$, which represents an RMW operation that: reads a value v from memory; and if Φ is defined on v, atomically writes $\Phi(v)$ in its stead. To support parameters, an *n-ary modifier* is a partial function $\varphi_{-} : \mathsf{Val}^n \times \mathsf{Val} \rightharpoonup \mathsf{Val}$. Our language requires a family RMW, indexed by the natural numbers, consisting of sets RMW_n of n-ary modifiers which we call *primitive* modifiers. For example, the following primitive modifiers for common operations, which have efficient implementations on hardware:

Load	$\mathrm{load}(v) \coloneqq \bot$	**Fetch-and-Add**	$\mathrm{faa}_{\langle w \rangle}(v) \coloneqq v + w$
Exchange	$\mathrm{xchg}_{\langle w \rangle}(v) \coloneqq w$	**Compare-and-Swap**	$\mathrm{cas}_{\langle w,u \rangle}(v) \coloneqq \text{if } v = w \text{ then } u \text{ else } \bot$

Here: $\bot$ means 'undefined'; we omit the nullary load's parameter ($\langle\rangle$); cas requires a semantic equality comparison operator on values ($=$); and faa requires semantic addition of values ($+$).

Syntax. Given parameters Loc, Val, and RMW, Figure 4 presents $\boldsymbol{\lambda}_{\mathrm{RA}}$'s syntax, and additional syntax admitted via syntactic sugar. The types are standard, comprising tuple, sum and function types. We draw constructor names for variants from a countably infinite set Lab, ranged over by ι. We assume Lab contains Loc and Val. We identify Loc and Val with sum types **Loc** and **Val** whose constructors are the locations and values, each labeling the empty tuple type.

The core term constructs in $\boldsymbol{\lambda}_{\mathrm{RA}}$ are standard too. We treat program variables a, b, c standardly, with the usual definitions of capturing and non-capturing substitutions. Function abstraction and application are standard, and we annotate the bound variable with its type, omitting the annotation when we can infer it. Tuple constructors are standard. Variant constructors $A.\iota\, M$ are standard and we require the total sum type A to disambiguate all variant constructors, which we omit when this type can be inferred. The pattern matching constructs for tuples and variants are standard, binding (distinct) variables occurrences in each pattern, and scoping over each branch.

We index the RMW construct with a primitive modifier $\varphi \in \mathsf{RMW}$, and its first argument is a location from which to read and possibly modify, followed by a tuple supplying the parameters. The term $\mathbf{rmw}_{\varphi}\,(M; N)$ executes by evaluating M to a location ℓ, then evaluating N to a tuple of values $\vec{w} = \left\langle w_1, \ldots, w_{\varphi.\mathtt{ar}} \right\rangle$. Then, atomically, reading a value v from ℓ and overwriting it with $\varphi_{\vec{w}} v$ if it's defined. Regardless of whether $\varphi_{\vec{w}} v$ is defined, the read value v is returned. If $\varphi_{\vec{w}} v$ is defined, no other RMW can overwrite the same previous write of v.

We desugar the typical memory dereferencing primitives using our example modifier primitives:

$$M? \coloneqq \mathbf{rmw}_{\mathrm{load}}\,(M; \langle\rangle) \qquad \mathrm{FAA}\,(M, N) \coloneqq \mathbf{rmw}_{\mathrm{faa}}\,(M; \langle N \rangle)$$

$$\mathrm{XCHG}\,(M, N) \coloneqq \mathbf{rmw}_{\mathrm{xchg}}\,(M; \langle N \rangle) \quad \mathrm{CAS}\,(M, N, K) \coloneqq \mathbf{rmw}_{\mathrm{cas}}\,(M; \langle N, K \rangle)$$

Assignment $M \coloneqq N$ is standard, executing by first evaluating M to a location ℓ; evaluating N to a value v; storing the value v at the location ℓ in memory; and finally returning $\langle\rangle$. Assignment does not block overwriting, so $M \coloneqq N$ is not equivalent to $\mathrm{XCHG}\,(M, N)\,;\langle\rangle$.

The operational semantics, defined in §5, follows a call-by-value evaluation strategy, adhering to a left-to-right convention except for parallel composition $M \parallel N$. There, the executions of its threads M and N interleave, evaluating to the pair of the results of each thread.

The operational semantics hinges on the standard designation of certain terms as *values*:

$$V, W ::= \langle V_1, \ldots, V_n \rangle \mid A.\iota\, V \mid \lambda a : A.\, M \qquad \text{(Values)}$$

We write $M[a_1 \mapsto V_1, \ldots, a_n \mapsto V_n]$ for the standard capture-avoiding substitution replacing each variable a_i with V_i in M.

$M, N ::=$	term
$\quad a$	variable/identifier
$\quad \mid \lambda a : A.\, M$	function abstraction
$\quad \mid MN$	function application
$\quad \mid \langle M_1, \ldots, M_n \rangle$	tuple constructor
$\quad \mid A.\iota\, M$	variant constructor
$\quad \mid \mathbf{match}\ M\ \mathbf{with}\ \Pi$	pattern matching
$\quad \mid \mathbf{rmw}_\varphi\,(M; N)$	read-modify-write
$\quad \mid M \coloneqq N$	assignment
$\quad \mid M \parallel N$	parallel composition
$\Pi ::=$	pattern clause
$\quad \mid \langle a_1, \ldots, a_n \rangle.\, N$	tuple
$\quad \mid \{\iota_1\, a_1.N_1 \mid \cdots \mid \iota_n\, a_n.N_n\}$	variant
$V, W ::=$	value
$\quad \mid \langle V_1, \ldots, V_n \rangle$	tuple of values
$\quad \mid A.\iota\, V$	value within variant
$\quad \mid \lambda a : A.\, M$	function abstraction
$A, B ::=$	type
$\quad A \to B$	function
$\quad \mid (A_1 * \cdots * A_n)$	tuple/product
$\quad \mid \{\iota_1\ \mathbf{of}\ A_1 \mid \cdots \mid \iota_n\ \mathbf{of}\ A_n\}$	variant/sum
$G ::=$	ground type
$\quad \mid (G_1 * \cdots * G_n)$	tuple/product of ground types
$\quad \mid \{\iota_1\ \mathbf{of}\ G_1 \mid \cdots \mid \iota_n\ \mathbf{of}\ G_n\}$	variant/sum of ground types
	syntactic sugar
$\mathbf{1} \coloneqq ()$	unit
$A^n \coloneqq (A * \cdots * A)$	repeated product
$\{\iota_1 \mid \cdots \mid \iota_n\} \coloneqq \{\iota_1\ \mathbf{of}\ \mathbf{1} \mid \cdots \mid \iota_n\ \mathbf{of}\ \mathbf{1}\}$	enumeration
$A.\iota \coloneqq A.\iota\ \langle\rangle$	label
$\mathbf{let}\ a = M\ \mathbf{in}\ N \coloneqq \mathbf{match}\ \langle M \rangle\ \mathbf{with}\ \langle a \rangle.\, N$	let binding
$M ; N \coloneqq \mathbf{let}\ \_ = M\ \mathbf{in}\ N$	sequencing
$M\mathbf{?} \coloneqq \mathbf{rmw}_{\mathrm{load}}\,(M; \langle\rangle)$	load
$\mathrm{XCHG}\,(M, N) \coloneqq \mathbf{rmw}_{\mathrm{xchg}}\,(M; \langle N \rangle)$	exchange
$\mathrm{FAA}\,(M, N) \coloneqq \mathbf{rmw}_{\mathrm{faa}}\,(M; \langle N \rangle)$	fetch-and-add
$\mathrm{CAS}\,(M, N, K) \coloneqq \mathbf{rmw}_{\mathrm{cas}}\,(M; \langle N, K \rangle)$	compare-and-swap
	locations and values
$\mathbf{Loc} \coloneqq \{\mathtt{l}_1 \mid \cdots \mid \mathtt{l}_n\}$	$\mathsf{Loc} = \{\mathtt{l}_1, \ldots, \mathtt{l}_n\}$
$\mathbf{Val} \coloneqq \{\mathtt{v}_1 \mid \cdots \mid \mathtt{v}_m\}$	$\mathsf{Val} = \{\mathtt{v}_1, \ldots, \mathtt{v}_m\}$

Fig. 4. Syntax of the $\boldsymbol{\lambda}_{\mathrm{RA}}$-calculus: terms, and their subset of values; and types, and their subset of ground types. Other terms and types are admitted as syntactic sugar. The shared-state constructs that extend the core calculus are highlighted.

$$\boxed{\Gamma \vdash M : A}$$

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A} \qquad \frac{\Gamma, a : A \vdash M : B}{\Gamma \vdash \lambda a : A.\, M : A \to B} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \to B}{\Gamma \vdash NM : B}$$

$$\frac{\forall i.\ \ \Gamma \vdash M_i : A_i}{\Gamma \vdash \langle M_1, \dots, M_n \rangle : (A_1 * \cdots * A_n)} \qquad \frac{\Gamma \vdash M : A_i \qquad A = \{\iota_1 \ \mathbf{of}\ A_1 \mid \cdots \mid \iota_n \ \mathbf{of}\ A_n\}}{\Gamma \vdash A.\iota_i\, M : A}$$

$$\frac{\Gamma \vdash M : (A_1 * \cdots * A_n) \quad \Gamma, a_1 : A_1, \dots, a_n : A_n \vdash N : B}{\Gamma \vdash \mathbf{match}\, M\, \mathbf{with}\, \langle a_1, \dots, a_n \rangle.\, N : B} \qquad \frac{\Gamma \vdash M : \{\iota_1 \ \mathbf{of}\ A_1 \mid \cdots \mid \iota_n \ \mathbf{of}\ A_n\} \quad \forall i.\ \ \Gamma, a_i : A_i \vdash N_i : B}{\Gamma \vdash \mathbf{match}\, M\, \mathbf{with}\, \{\iota_1\, a_1.N_1 \mid \cdots \mid \iota_n\, a_n.N_n\} : B}$$

$$\frac{\varphi \in \mathsf{RMW}_n \qquad \Gamma \vdash M : \mathbf{Loc} \qquad \Gamma \vdash N : \mathbf{Val}^n}{\Gamma \vdash \mathbf{rmw}_\varphi\, (M; N) : \mathbf{Val}} \qquad \frac{\Gamma \vdash M : \mathbf{Loc} \qquad \Gamma \vdash N : \mathbf{Val}}{\Gamma \vdash M \coloneqq N : \mathbf{1}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash M \parallel N : (A * B)}$$

Fig. 5. Typing rules for the $\boldsymbol{\lambda}_{\text{RA}}$-calculus. Typing rules for the shared-state constructs are highlighted.

Remark. *We do not include recursion/loops in this language, which we leave to future work. While important, recursion in this higher-order setting will muddy the waters substantially, requiring us to bring into context domain theoretic concepts like least upper-bounds of ω-chains and powerdomain constructions. Even without recursion, $\boldsymbol{\lambda}_{RA}$ is expressive enough for us to discuss interesting examples and transformations.*

Type system. We present the type system in Figure 5. Each typing judgment $\Gamma \vdash M : A$ relates a type A, a term M, and a *typing context* Γ which associates to each of M's unbound variable a a type B_a, written $(a : B_a) \in \Gamma$. We write $\cdot$ for the empty context, and say that M is *closed* when $\cdot \vdash M : A$ for some type A. The *shadowing extension* of Γ by $c : C$, denoted $\Gamma, c : C$, is equal to Γ except for associating C to c. The typing rules for the shared-memory constructs are standard, and reflect their informal explanation above. In particular, for RMW the arity of the tuple must match the arity of the modifier. Each term has at most one type in a given typing context, and in that case the typing derivation is unique. We denote by $\Gamma \vdash A$ the set of terms $\{M \mid \Gamma \vdash M : A\}$.

A *program* is a closed term of *ground type*—iterated sum and product types:

$$G ::= (G_1 * \cdots * G_n) \mid \{\iota_1 \ \mathbf{of}\ G_1 \mid \cdots \mid \iota_n \ \mathbf{of}\ G_n\} \qquad \text{(Ground types)}$$

5 Operational Semantics for Release/Acquire Concurrency

We give a precise account of the "view-based" machine (§5.1) presented in §2.3. We observe that this semantics admits a non-deterministic view-forwarding step (§5.2) which our metatheory uses.

5.1 View-based Semantics

Our formalization of the operational semantics follows Kang et al. [28] and Kaiser et al. [27]. The account below grounds the explanations we gave in §2.3 more formally.

Timestamps. We maintain a per-location *timestamp* order, which constrains the memory access of threads. We use rational numbers $\mathbb{Q}$ as timestamps, ordered standardly, ranged over by t, q, p.

Views. A *view* is a location-indexed tuple of timestamps, i.e. an element $(\kappa_\ell)_{\ell \in \mathsf{Loc}}$ of the set of views $\mathsf{View} \coloneqq \mathbb{Q}^{\mathsf{Loc}}$. We let $\alpha, \kappa, \sigma, \omega$ range over views. In examples with $\mathsf{Loc} = \{\mathtt{x}, \mathtt{y}\}$, we denote by $\langle\!\langle \mathtt{x@}t \ ; \ \mathtt{y@}q \rangle\!\rangle$ the view that has t in the $\mathtt{x}$ component and q in the $\mathtt{y}$ component. We order views location-wise, i.e. $\alpha \leq \omega$ when $\forall \ell \in \mathsf{Loc}.\ \alpha_\ell \leq \omega_\ell$, and in this case say that ω *dominates* α. We also employ $\sqcup$ and $\sqcap$ for pointwise maximum and minimum of views, and denote by $\kappa[\ell{\mapsto}t]$ the view that is equal to κ everywhere except ℓ, where it equals t.

Messages. A *message* ν is a tuple in $\mathsf{Msg} \coloneqq \mathsf{Loc} \times \mathsf{Val} \times \mathbb{Q} \times \mathsf{View}$, written $\nu = \ell{:}v@(q, \kappa_\ell]\langle\!\langle \kappa \rangle\!\rangle$, where $q < \kappa_\ell$. Here, ℓ is the location of the message, v is the value of the message, q is the *initial* timestamp of the message, and κ is the view of the message. We repeat the ℓ-timestamp from the view κ in the interval $(q, \kappa_\ell]$. We say ν *dovetails after* a message at location ℓ with ℓ-timestamp q.

We use projection-notation for components of ν: $\nu.\mathtt{lc} \coloneqq \ell$, $\nu.\mathtt{vl} \coloneqq v$, $\nu.\mathtt{i} \coloneqq q$, and $\nu.\mathtt{vw} \coloneqq \kappa$. The *(final) timestamp* of the message is $\nu.\mathtt{t} \coloneqq \kappa_\ell$. In concrete examples, we reduce duplication by eliding the timestamp from the view, e.g. $\mathtt{y}{:}0@(0.5, \underline{4.2}]\ \langle\!\langle \mathtt{x@1} \rangle\!\rangle$. The message's two timestamps delimit the *segment* of the message: the interval $\nu.\mathtt{seg} \coloneqq (\nu.\mathtt{i}, \nu.\mathtt{t}]$.

We let ν, ϵ, β range over messages. We extend notation from messages to sets of messages by direct image: for example, given a set μ of messages, define $\mu.\mathtt{seg} \coloneqq \{\nu.\mathtt{seg} \mid \nu \in \mu\}$.

Memories. A *memory* is a finite non-empty set of messages. We let μ, ρ, θ range over memories, and denote the set of messages in μ at location ℓ by $\mu_\ell \coloneqq \{\nu \in \mu \mid \nu.\mathtt{lc} = \ell\}$.

Example 5.1. Figure 6 (top) illustrates a memory resulting from a program execution starting with the memory $\{\nu_1, \epsilon_1\}$: the program added messages out of the timeline order (ϵ_3 before ϵ_2); dovetailed messages ($\nu_2.\mathtt{t} = \nu_3.\mathtt{i}$); and left gaps between messages ($\nu_1.\mathtt{t} < \nu_2.\mathtt{i}$). Message views need not increase along the timeline ($\epsilon_2.\mathtt{t} \leq \epsilon_3.\mathtt{t}$ yet $\epsilon_2.\mathtt{vw} \not\leq \epsilon_3.\mathtt{vw}$).

View trees. Kang et al.'s [28] original presentation of the view-based semantics studies top-level parallelism, and thus featured a flat thread-view mappings. Here we allow nesting of parallel composition anywhere in the program, so we use a tree of views instead, whose structure changes along with the execution of the program as threads are activated and synchronize.

Formally, a *view-tree* is a binary tree with view-labeled leaves. We denote the set of view-trees by VTree, ranged over by T, R, H. We denote: by $\dot\kappa$ the leaf with label κ; by $T \frown R$ the tree whose immediate left and right subtrees are T and R; and by $T.\mathtt{lf}$ the set of labels of leaves of T. We lift the order of views leaf-wise: $\dot\kappa \leq \dot\sigma$ when $\kappa \leq \sigma$, and $T \frown R \leq T' \frown R'$ when $T \leq T'$ and $R \leq R'$.

Operational semantics. Figures 7 to 9 present the operational semantics of $\boldsymbol{\lambda}_{\text{RA}}$. A *configuration* $\langle T, \mu \rangle, M$ consists of a view-tree T capturing the views of the active threads; the current memory μ; and a closed term M. The *state* of the configuration is the pair $\langle T, \mu \rangle$. The relation $\overset{e}{\rightsquigarrow}_{\text{RA}}$ represents (atomic) steps between configurations. The label e, distinguishing the memory-accessing steps ($\bullet$) from the rest ($\circ$), is used as a proof tool (§C) and can be otherwise ignored. We denote the step relation ignoring the label $\rightsquigarrow_{\text{RA}} \coloneqq \overset{\bullet}{\rightsquigarrow}_{\text{RA}} \cup \overset{\circ}{\rightsquigarrow}_{\text{RA}}$. Let the Kleene star ($*$) denote the reflexive-transitive closure of a relation. For example, $\rightsquigarrow^*_{\text{RA}}$ is the reflexive-transitive closure of $\rightsquigarrow_{\text{RA}}$.

Sequential CBV constructs. The semantics adheres to a standard call-by-value (CBV) reduction strategy. For example, terms do not reduce under function abstractions; and as for function application, we have the AppLeft and AppRight congruence steps, and the App β-reduction step is restricted to reduce only on value arguments. The β-reductions use the $\circ$-label and view-leaves, and do not change the state; the congruence steps simply carry the label and states over.

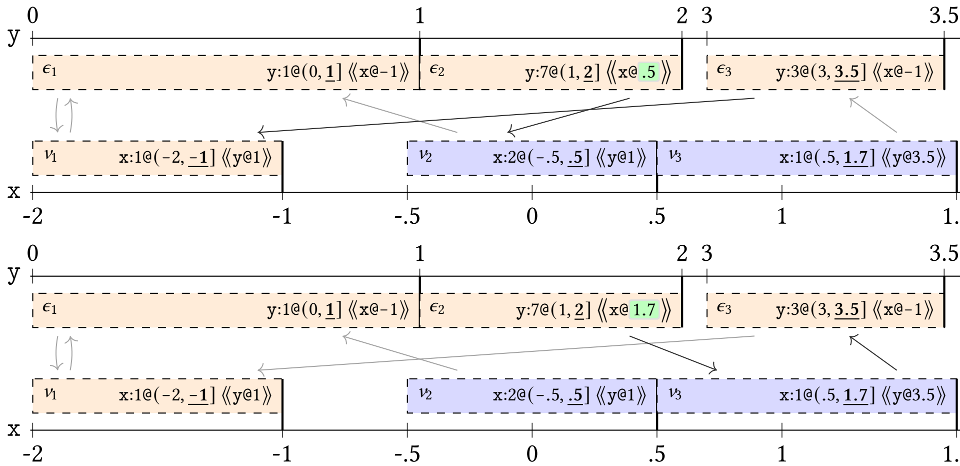


Fig. 6. Two variations on the memory illustrated in Figure 1. **Top:** This memory is well-formed. It demonstrates that the views of messages along a timeline do not have to be ordered: ϵ_2 appears earlier than ϵ_3 on y's timeline but points to a later message on x's timeline. **Bottom:** This memory is not well-formed because it contains an ascending path, in contradiction to Proposition 6.5. Intuitively, no thread could have written ϵ_2 because the view that ϵ_2 carries indicates that the thread would have already "known" about ν_3 and therefore, following the causality chain, about ϵ_3 as well. Thus, the thread would have been forbidden from picking ϵ_2's timestamp.

Parallel composition. The ParInit rule initializes a parallel composition by duplicating its view-leaf to a new node. The rules ParLeft and ParRight non-deterministically interleave the execution of the left and right threads. After both threads evaluate, ParFin joins the thread views back into a single leaf, and returns the pair of results.

Example 5.2. We show an example execution, snipping some intermediate steps:

$$\langle \mu_0, \dot{\alpha} \rangle, M\,; (N_1 \parallel N_2) \rightsquigarrow^*_{\mathrm{RA}} \langle \mu_1, \dot{\alpha}' \rangle, N_1 \parallel N_2 \rightsquigarrow_{\mathrm{RA}} \left\langle \mu_1, \dot{\alpha}' \frown \dot{\alpha}' \right\rangle, N_1 \parallel N_2$$
$$\rightsquigarrow^*_{\mathrm{RA}} \left\langle \rho, \dot{\omega}_1 \frown \dot{\omega}_2 \right\rangle, V_1 \parallel V_2 \rightsquigarrow_{\mathrm{RA}} \left\langle \rho, \omega_1 \dot{\sqcup} \omega_2 \right\rangle, \langle V_1, V_2 \rangle$$

First, M runs until it returns a value, which is discarded by the sequencing construct. Next, the parallel composition $N_1 \parallel N_2$ activates. The threads then interleave executions, each with its associated side of the view-tree, interacting via the shared memory. Finally, once each thread returns a value, they synchronize.

Assignment. The Store rule for location ℓ picks a free segment $(q, t]$ for the message it adds, where t is strictly greater than the thread's view for ℓ. The step updates this thread's view to ω by increasing the timestamp for ℓ to t; adds a message to memory with this updated view ω; and returns the unit value.

Read-modify-write. The ReadOnly and RMW rules for the **rmw** construct both: start by picking a message to the given location to read from that has the same or a larger timestamp than the thread's view; then incorporate the message's view in the thread's view; and finally return the value they read. If the given primitive modifier is undefined for the given parameters and message's value, nothing else happens (ReadOnly rule). If the modifier is defined (RMW rule), much like the Store rule, a timestamp strictly greater than the thread's view for the location is chosen to update

$$\boxed{\langle T,\mu\rangle, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, M'}$$

$$\textsc{AppLeft}\ \frac{\langle T,\mu\rangle, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, M'}{\langle T,\mu\rangle, MN \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, M'N} \quad \textsc{AppRight}\ \frac{\langle T,\mu\rangle, N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, N'}{\langle T,\mu\rangle, VN \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, VN'} \quad \textsc{App}\ \frac{}{\langle \dot\kappa,\mu\rangle, (\lambda a : A.\, M)\, V \overset{\circ}{\rightsquigarrow}_{\mathrm{RA}} \langle \dot\kappa,\mu\rangle, M[a{\mapsto}V]}$$

$$\textsc{MatchCong}\ \frac{\langle T,\mu\rangle, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, M'}{\langle T,\mu\rangle, \mathbf{match}\, M\, \mathbf{with}\, \Pi \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, \mathbf{match}\, M'\, \mathbf{with}\, \Pi} \quad \textsc{VariantCong}\ \frac{\langle T,\mu\rangle, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, M}{\langle T,\mu\rangle, A.\iota_i\, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, A.\iota_i\, M'}$$

$$\textsc{MatchVariant}\ \frac{}{\langle \dot\kappa,\mu\rangle, \mathbf{match}\, A.\iota_i\, V\, \mathbf{with}\, \{\iota_1\, a_1.N_1 \mid \cdots \mid \iota_n\, a_n.N_n\} \overset{\circ}{\rightsquigarrow}_{\mathrm{RA}} \langle \dot\kappa,\mu\rangle, N_i[a_i{\mapsto}V]}$$

$$\textsc{TupleCong}\ \frac{\langle T,\mu\rangle, M_i \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, M_i'}{\langle T,\mu\rangle, \langle V_1,\dots,V_{i-1},M_i,M_{i+1},\dots,M_n\rangle \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, \left\langle V_1,\dots,V_{i-1},M_i',M_{i+1},\dots,M_n\right\rangle}$$

$$\textsc{MatchTuple}\ \frac{}{\langle \dot\kappa,\mu\rangle, \mathbf{match}\, \langle V_1,\dots,V_n\rangle\, \mathbf{with}\, \langle a_1,\dots,a_n\rangle.\, N \overset{\circ}{\rightsquigarrow}_{\mathrm{RA}} \langle \dot\kappa,\mu\rangle, N[a_1{\mapsto}V_1,\dots,a_n{\mapsto}V_n]}$$

Fig. 7. The operational semantics rules of $\boldsymbol{\lambda}_{\mathrm{RA}}$ concerning the core calculus constructs.

$$\textsc{ParInit}\ \frac{}{\langle \dot\kappa,\mu\rangle, M \parallel N \overset{\circ}{\rightsquigarrow}_{\mathrm{RA}} \left\langle \dot\kappa \frown \dot\kappa, \mu\right\rangle, M \parallel N} \quad \textsc{ParFin}\ \frac{\omega = \kappa \sqcup \sigma}{\left\langle \dot\kappa \frown \dot\sigma, \mu\right\rangle, V \parallel W \overset{\circ}{\rightsquigarrow}_{\mathrm{RA}} \langle \dot\omega,\mu\rangle, \langle V, W\rangle}$$

$$\textsc{ParLeft}\ \frac{\langle T,\mu\rangle, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T',\mu'\rangle, M'}{\left\langle T \frown R, \mu\right\rangle, M \parallel N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \left\langle T' \frown R, \mu'\right\rangle, M' \parallel N} \quad \textsc{ParRight}\ \frac{\langle R,\mu\rangle, N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle R',\mu'\rangle, N'}{\left\langle T \frown R, \mu\right\rangle, M \parallel N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \left\langle T \frown R', \mu'\right\rangle, M \parallel N'}$$

Fig. 8. The operational semantics rules of $\boldsymbol{\lambda}_{\mathrm{RA}}$ concerning the parallel composition construct.

the thread's view, and a message is added with this updated view. In contrast to the Store rule, here the added message's segment must dovetail after the message from which the RMW read, still avoiding any existing segment in this location. Dovetailing after a message v is only possible if no existing message dovetails after v. In particular, a message can only be picked once to justify the RMW rule during an execution.

Initial states. An *initial memory* μ is a memory in which every location has exactly one message whose view contains the timestamps of the other messages. An *initial state* is a state consisting of an initial memory and a view-leaf that maps each location to the timestamp of the unique message in memory at that location. An *initial configuration* is a configuration consisting of an initial state and a closed term.

Evaluation. We're interested in the behaviors closed terms exhibit when run to completion. A configuration $\langle T,\mu\rangle, M$ *evaluates* to a value V, written $\langle T,\mu\rangle, M \Downarrow_{\mathrm{RA}} V$, when $\langle T,\mu\rangle, M \rightsquigarrow^*_{\mathrm{RA}} \langle R,\rho\rangle, V$ for some state $\langle R,\rho\rangle$. We write $\langle T,\mu\rangle, M \not\Downarrow_{\mathrm{RA}} V$ when there is no such $\langle R,\rho\rangle$. In the next

$$\textsc{Store}\ \frac{\alpha_\ell < t \quad (q,t] \cap \bigcup \mu_\ell.\mathtt{seg} = \emptyset \quad \omega = \alpha[\ell \mapsto t]}{\langle \dot{\alpha}, \mu \rangle, \ell := v \overset{\bullet}{\rightsquigarrow}_{\mathrm{RA}} \langle \dot{\omega}, \mu \uplus \{\ell{:}v@(q,\omega_\ell]\langle\!\langle \omega \rangle\!\rangle\} \rangle, \langle\rangle}$$

$$\textsc{ReadOnly}\ \frac{\ell{:}v@(q,\kappa_\ell]\langle\!\langle \kappa \rangle\!\rangle \in \mu \quad \alpha_\ell \le \kappa_\ell \quad \varphi_{\vec{w}} v = \bot \quad \omega = \alpha \sqcup \kappa}{\langle \dot{\alpha}, \mu \rangle, \mathbf{rmw}_\varphi\,(\ell; \vec{w}) \overset{\bullet}{\rightsquigarrow}_{\mathrm{RA}} \langle \dot{\omega}, \mu \rangle, v}$$

$$\textsc{RMW}\ \frac{\ell{:}v@(q,\kappa_\ell]\langle\!\langle \kappa \rangle\!\rangle \in \mu \qquad \alpha_\ell \le \kappa_\ell \qquad \varphi_{\vec{w}} v \neq \bot \qquad (\kappa_\ell, t] \cap \bigcup \mu_\ell.\mathtt{seg} = \emptyset \qquad \omega = (\alpha \sqcup \kappa)\,[\ell \mapsto t]}{\langle \dot{\alpha}, \mu \rangle, \mathbf{rmw}_\varphi\,(\ell; \vec{w}) \overset{\bullet}{\rightsquigarrow}_{\mathrm{RA}} \langle \dot{\omega}, \mu \uplus \{\ell{:}\varphi_{\vec{w}} v@(\kappa_\ell, \omega_\ell]\langle\!\langle \omega \rangle\!\rangle\} \rangle, v}$$

$$\textsc{StoreLeft}\ \frac{\langle T, \mu \rangle, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, M'}{\langle T, \mu \rangle, M := N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, M' := N} \qquad \textsc{StoreRight}\ \frac{\langle T, \mu \rangle, N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, N'}{\langle T, \mu \rangle, V := N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, V := N'}$$

$$\textsc{RMWLeft}\ \frac{\langle T, \mu \rangle, M \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, M'}{\langle T, \mu \rangle, \mathbf{rmw}_\varphi\,(M; N) \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, \mathbf{rmw}_\varphi\,(M'; N)} \qquad \textsc{RMWRight}\ \frac{\langle T, \mu \rangle, N \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, N'}{\langle T, \mu \rangle, \mathbf{rmw}_\varphi\,(V; N) \overset{e}{\rightsquigarrow}_{\mathrm{RA}} \langle T', \mu' \rangle, \mathbf{rmw}_\varphi\,(V; N')}$$

Fig. 9. The operational semantics rules of $\boldsymbol{\lambda}_{\mathrm{RA}}$ concerning the shared-state constructs.

examples, we write $M \Downarrow_{\mathrm{RA}} V$ when M may evaluate to V from every initial state, and $M \not\Downarrow_{\mathrm{RA}} V$ when it cannot evaluate to V from any initial state.

Example 5.3. We can give a more precise account of the litmus tests (SB) and (MP) from §2:

$$\mathtt{x} := 0\,; \mathtt{y} := 0\,; ((\mathtt{x} := 1\,;\quad \mathtt{y}?\quad) \parallel (\mathtt{y} := 1\,; \mathtt{x}?)) \Downarrow_{\mathrm{RA}} \langle\quad 0 \quad,\quad 0 \quad\rangle$$

$$\mathtt{x} := 0\,; \mathtt{y} := 0\,; ((\mathtt{x} := 1\,; \mathtt{y} := 1) \parallel (\quad \mathtt{y}? \quad; \mathtt{x}?)) \not\Downarrow_{\mathrm{RA}} \langle\quad \langle\rangle \quad, \langle 1, 0 \rangle \rangle$$

5.2 Non-deterministic View Forwarding

It is technically convenient to extend the operational semantics $\overset{e}{\rightsquigarrow}_{\mathrm{RA}}$ with an additional rule Adv that non-deterministically advances the view of a thread, presented in Figure 10. The Adv step advances the thread's view like the ReadOnly rule without changing the term component of the configuration. We think of this step as read-independent propagation of updates to threads. Lahav et al. [32] propose a similar extension when defining liveness conditions for RA. The effect of this step is to prohibit the thread from reading certain messages from memory, and propagating this prohibition to other threads that read values this thread writes.

$$\textsc{Adv}\ \frac{\ell{:}v@(q,\kappa_\ell]\langle\!\langle \kappa \rangle\!\rangle \in \mu \qquad \alpha_\ell \le \kappa_\ell \qquad \omega = \alpha \sqcup \kappa}{\langle \dot{\alpha}, \mu \rangle, M \overset{\circ}{\rightsquigarrow}_{\mathrm{RA}_\le} \langle \dot{\omega}, \mu \rangle, M}$$

Fig. 10. View advancement rule.

A-priori, the resulting system may exhibit more behaviors since Store and RMW steps following Adv steps will append messages with further advanced views. However, advancing views within messages only further constrains possible behaviors. We formalize this intuition using a simulation argument. One direction is straightforward: every RA-execution is an $\mathrm{RA}_\le$-execution, so $\mathrm{RA}_\le$ exhibits every behavior RA does. For the converse, we define a binary relation between configuration states $\gtrsim$ such that $\langle T, \mu \rangle \gtrsim \langle R, \rho \rangle$ when the following hold.

- The simulatee's view-tree dominates the simulator's view-tree: $T \ge R$.
- There are bijections $\phi_\ell : \mu_\ell \to \rho_\ell$ for every location ℓ such that if $\phi_\ell(\nu) = \epsilon$, then the view of the simulatee's message dominates the simulator's message $\nu.\mathtt{vw} \ge \epsilon.\mathtt{vw}$, and the messages' value and segment agree: $\nu.\mathtt{vl} = \epsilon.\mathtt{vl}$, $\nu.\mathtt{i} = \epsilon.\mathtt{i}$, $\nu.\mathtt{t} = \epsilon.\mathtt{t}$.

The relation $\gtrsim$ is a weak simulation:

Proposition 5.4. *If $\langle T, \mu \rangle \gtrsim \langle R, \rho \rangle$ and $\langle T, \mu \rangle, M \rightsquigarrow_{RA_{\leq}} \langle T', \mu' \rangle, M'$, then there exists a configuration state $\langle R', \rho' \rangle$ such that $\langle R, \rho \rangle, M \rightsquigarrow^{*}_{RA} \langle R', \rho' \rangle, M'$ and $\langle T', \mu' \rangle \gtrsim \langle R', \rho' \rangle$.*

Proof. By induction on the step. An Adv step preserves $\gtrsim$, so we take no steps in the required corresponding RA execution. In the Store case, we use the timestamp that the simulation requires, and the view that the current view tree determines. In the ReadOnly case, we load the corresponding message according to the bijection given by $\gtrsim$. The RMW case holds by combining the arguments above. The ParInit and ParFin cases preserve the memory and the order on view trees. The other base cases are β-reductions. They retain the state and thus the simulation. The congruence steps follow by induction as they propagate the state. □

Like $\rightsquigarrow_{\mathrm{RA}}$, the relation $\rightsquigarrow_{\mathrm{RA}_{\leq}}$ induces an evaluation semantics $\Downarrow_{\mathrm{RA}_{\leq}}$. By Proposition 5.4, $\Downarrow_{\mathrm{RA}}$ and $\Downarrow_{\mathrm{RA}_{\leq}}$ coincide:

Corollary 5.5. *For a configuration $\langle T, \mu \rangle, M$ and value V: $\langle T, \mu \rangle, M \Downarrow_{RA} V$ iff $\langle T, \mu \rangle, M \Downarrow_{RA_{\leq}} V$.*

Thus, we denote both by $\Downarrow$.

Remark. *Restricting ReadOnly and RMW in $RA_{\leq}$ by an additional assumption $\alpha_{\ell} = \kappa_{\ell}$ results in an equivalent evaluation semantics. Instead of loading a message using one of the unrestricted rules, we can use Adv to "prepare" the view for loading, and then load using the restricted version.*

6 Semantic Invariants

We present both known and novel $\mathrm{RA}_{\leq}$ invariants: properties that initial configurations satisfy, and are maintained along $\mathrm{RA}_{\leq}$ step. Accounting for these semantic invariants in our denotational semantics makes it more abstract, eliminating distinctions that would obstruct the justification of program transformations. During our presentation of the invariants we give intuitive explanations for why they hold, formally grounded in Theorem 6.10 and Proposition 6.13 below.

6.1 Basic memory invariants

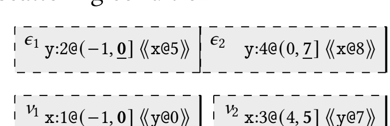
We establish basic properties of timestamps and segments in memories. A memory μ is *scattered* if segments of messages in the same location are pairwise disjoint:

$$\forall \ell \in \mathsf{Loc}\, \forall \nu, \epsilon \in \mu_{\ell}.\ \nu.\mathtt{seg} \cap \epsilon.\mathtt{seg} \neq \emptyset \implies \nu = \epsilon$$

Initial memories are scattered and execution steps preserve the fact that the memory is scattered since added messages can only occupy unused segments.

Example 6.1. The memory below (left) is scattered. We can visualize its segments along the timeline order without overlap (right) thanks to the scattering condition:

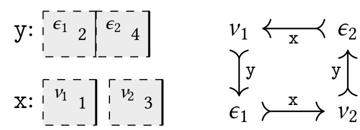
$$\left\{\begin{array}{l} \mathtt{y}{:}2@(-1, \underline{0}]\ \langle\!\langle \mathtt{x}@5 \rangle\!\rangle, \mathtt{y}{:}4@(0, \underline{7}]\ \langle\!\langle \mathtt{x}@8 \rangle\!\rangle \\ \\ \mathtt{x}{:}1@(-1, \underline{0}]\ \langle\!\langle \mathtt{y}@0 \rangle\!\rangle, \mathtt{x}{:}3@(4, \underline{5}]\ \langle\!\langle \mathtt{y}@7 \rangle\!\rangle \end{array}\right\}$$



We think of timestamps as names, i.e., abstract pointers. Formally, a view κ *points to* a message ϵ, denoted by $\kappa \rightarrowtail \epsilon$, when κ holds ϵ's timestamp at ϵ's location: $\kappa_{\epsilon.\mathtt{lc}} = \epsilon.\mathtt{t}$. A view κ *points to* memory μ, denoted by $\kappa \rightarrowtail \mu$, when it points to a μ-message in all locations: $\forall \ell \in \mathsf{Loc}\, \exists \epsilon \in \mu_{\ell}.\ \kappa \rightarrowtail \epsilon$. A message ν points to another message ϵ or memory μ when its view $\nu.\mathtt{vw}$ points to that message or memory, denoted by $\nu \rightarrowtail \epsilon$ and $\nu \rightarrowtail \mu$. A memory μ is *connected* when it is scattered, and every message within it points to it: $\forall \nu \in \mu.\ \nu \rightarrowtail \mu$.

Example 6.2. The memory from Example 6.1 is not connected: ϵ_2 doesn't point to any message in x. In contrast, the memory below (left) is connected; we visualize its timestamp orders (middle) and points-to relations (right) thanks to the connectedness condition:

$$\begin{Bmatrix} \texttt{y}{:}2@(-1,\underline{0}]\,\langle\!\langle \texttt{x}@5\rangle\!\rangle,\ \texttt{y}{:}4@(0,\underline{7}]\,\langle\!\langle \texttt{x}@0\rangle\!\rangle \\ \texttt{x}{:}1@(-1,\underline{0}]\,\langle\!\langle \texttt{y}@0\rangle\!\rangle,\ \texttt{x}{:}3@(4,\underline{5}]\,\langle\!\langle \texttt{y}@7\rangle\!\rangle \end{Bmatrix}$$



Initial memories are connected, and execution steps preserve memory connectedness, assuming that all thread views point to the current memory: when a thread adds a message to memory, it uses its own view with an advanced timestamp for the message's location, maintaining connectedness.

6.2 Causal memory invariants

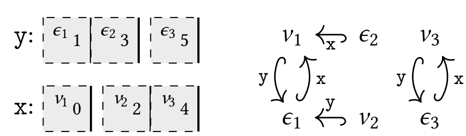
The points-to relation tracks some causal dependencies. Intuitively, events should not be caused by future events, so causal paths, i.e. paths in $\mu.\texttt{gph} \coloneqq \langle \mu, (\rightarrowtail)\backslash \mathrm{id}_\mu \rangle$, should not lead to the future along any timeline. We refine the points-to relation to enforce this.

Formally, we say that a view κ *points downwards to* a message ϵ, written $\kappa \hookrightarrow \epsilon$ when it points to it, $\kappa \rightarrowtail \epsilon$, and it dominates ϵ's view, $\kappa \geq \epsilon.\texttt{vw}$. A view *points downwards into* a scattered memory μ, denoted by $\kappa \hookrightarrow \mu$, when it points downward to a message in μ in every location, i.e.: $\forall \ell \in \mathsf{Loc}\, \exists \epsilon \in \mu_\ell.\, \kappa \hookrightarrow \epsilon$. We say that a message points downward into a memory, writing $\nu \hookrightarrow \mu$, when its view does: $\nu.\texttt{vw} \hookrightarrow \mu$. We say that a memory μ is *causally connected*, when it is connected, and every message within it points downwards into it: $\forall \nu \in \mu.\, \nu \hookrightarrow \mu$.

To simplify notations in the following examples, we omit locations from messages, instead tagging the row in the set. For example, by $5@(6,\underline{7}]\,\langle\!\langle 7\rangle\!\rangle$ in the y row we mean $\texttt{y}{:}5@(6,\underline{7}]\,\langle\!\langle \texttt{x}@7\rangle\!\rangle$.

Example 6.3. The memory from Example 6.2 is not causally connected because $\epsilon_1 \rightarrowtail \nu_2$ while nonetheless $\epsilon_1.\texttt{vw}_\texttt{y} = 0 \not\geq 7 = \nu_2.\texttt{vw}_\texttt{y}$. The following memory is causally connected:

$$\begin{Bmatrix} \texttt{y}: & 1@(-1,\underline{0}]\,\langle\!\langle 0\rangle\!\rangle,\ 3@(0,\underline{5}]\,\langle\!\langle 0\rangle\!\rangle,\ 5@(6,\underline{7}]\,\langle\!\langle 7\rangle\!\rangle \\ \texttt{x}: & 0@(-1,\underline{0}]\,\langle\!\langle 0\rangle\!\rangle,\ 2@(4,\underline{5}]\,\langle\!\langle 0\rangle\!\rangle,\ 4@(5,\underline{7}]\,\langle\!\langle 7\rangle\!\rangle \end{Bmatrix}$$



Initial memories are causally connected, and execution steps preserve this property together with view-trees labeled solely by views that point downwards into the state's memory. In showing this property, particularly when observing steps that load a message, the following strong characterization of pointing downwards helps:

Lemma 6.4. *Let μ be a causally connected memory. A view κ points downwards into μ iff κ is the least view that dominates the views of all of the μ-messages that κ points to:*

$$\kappa \hookrightarrow \mu \iff \kappa = \bigsqcup \{\epsilon \in \mu \mid \kappa \rightarrowtail \epsilon\}.\texttt{vw}$$

Proof. For the "if" ($\Leftarrow$) direction, remembering that μ is finite, we have for every $\ell \in \mathsf{Loc}$:

$$\kappa_\ell = \left(\bigsqcup \{\epsilon \in \mu \mid \kappa \rightarrowtail \epsilon\}.\texttt{vw}\right)_\ell = \max\{\epsilon.\texttt{vw}_\ell \mid \kappa \rightarrowtail \epsilon \in \mu\}$$

In particular, $\kappa \rightarrowtail \mu$. Moreover, $\kappa \hookrightarrow \mu$ because whenever $\kappa \rightarrowtail \epsilon \in \mu$, we have $\kappa \geq \epsilon.\texttt{vw}$.

Conversely ($\Rightarrow$), let $\ell \in \mathsf{Loc}$. Since κ points to μ and μ is scattered, there exists a unique message $\nu \in \mu_\ell$ that κ points to $\kappa \rightarrowtail \nu$, i.e. $\kappa_\ell = \nu.\texttt{t} = \nu.\texttt{vw}_\ell$. Therefore, $\kappa_\ell \leq (\bigsqcup \{\epsilon \in \mu \mid \kappa \rightarrowtail \epsilon\}.\texttt{vw})_\ell$. Generalizing, $\kappa \leq \bigsqcup \{\epsilon \in \mu \mid \kappa \rightarrowtail \epsilon\}.\texttt{vw}$. Since κ points downwards $\kappa \hookrightarrow \mu$, any message ϵ in μ that κ points to $\kappa \rightarrowtail \epsilon \in \mu$, it also dominates: $\kappa \geq \epsilon.\texttt{vw}$. Therefore, $\kappa \geq \bigsqcup \{\epsilon \in \mu \mid \kappa \rightarrowtail \epsilon\}.\texttt{vw}$. □

Paths in a causally connected memory's graph descend down its timelines:

Proposition 6.5. *Let μ be a causally connected memory, with a path $\nu \rightarrowtail^* \epsilon$ in μ.gph.*

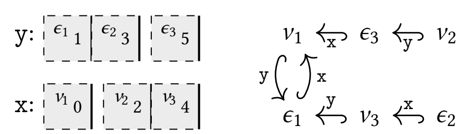
(1) Views decrease along the path: $\nu.\mathtt{vw} \geq \epsilon.\mathtt{vw}$.
(2) If there is also an opposite path $\epsilon \rightarrowtail^* \nu$, i.e., ν and ϵ are part of a cycle, then $\nu.\mathtt{vw} = \epsilon.\mathtt{vw}$.
(3) If they share a location, $\nu.\mathtt{lc} = \epsilon.\mathtt{lc}$, their timestamps decrease along the path: $\nu.\mathtt{t} \geq \epsilon.\mathtt{t}$.

Proof. Item 1 follows from the fact that the memory is causally connected by induction. The other items are direct consequences of the first. □

If a causally connected memory μ has a message in location ℓ, then μ has a timestamp-minimal message which we denote by $\min \mu_\ell$, i.e. $(\min \mu_\ell).\mathtt{t} = \min \mu_\ell.\mathtt{t}$. We say that a causally connected memory μ is *well-formed* when cycles within μ.gph consist solely of minimal messages, i.e. if $\nu \in \mu$ is part of a cycle in μ.gph, then $\nu = \min \mu_{\nu.\mathtt{lc}}$.

Example 6.6. The memory from Example 6.3 is not well-formed: ν_3 is on a cycle but not minimal. The following memory is well-formed:

$$\begin{cases} \mathtt{y}: & 1@(-1,\underline{0}]\ \langle\!\langle 0 \rangle\!\rangle, 3@(0,\underline{5}]\ \langle\!\langle 7 \rangle\!\rangle, 5@(6,\underline{7}]\ \langle\!\langle 0 \rangle\!\rangle \\ \mathtt{x}: & 0@(-1,\underline{0}]\ \langle\!\langle 0 \rangle\!\rangle, 2@(4,\underline{5}]\ \langle\!\langle 7 \rangle\!\rangle, 4@(5,\underline{7}]\ \langle\!\langle 0 \rangle\!\rangle \end{cases}$$



Initial memories are well-formed, and being well-formed is an invariant of execution steps. Indeed, messages are added one-by-one and point to existing messages, so they cannot from a new cycle; and messages are added with a larger timestamp, so minimal messages remains minimal.

Proposition 6.7. *Let μ be a well-formed memory, and let $\ell \in \mathsf{Loc}$.*

(1) Minimal messages point at minimal messages: if $\min \mu_\ell \hookrightarrow \nu$, then ν is a minimal message.
(2) Memory extension preserves minimal messages: if $\mu \subseteq \rho$ is well-formed, then $\min \mu_\ell = \min \rho_\ell$.

Proof. For (1), since μ is connected, there exists $\epsilon \in \mu_\ell$ such that $\nu \rightarrowtail \epsilon$. By Proposition 6.5, $(\min \mu_\ell).\mathtt{t} \geq \epsilon.\mathtt{t}$. By minimality $(\min \mu_\ell).\mathtt{t} = \epsilon.\mathtt{t}$, and since μ is scattered, $\epsilon = \min \mu_\ell$. Thus ν is on a cycle (with ϵ). Since μ is well-formed, ν is minimal.

For (2), since μ is well-formed, $\min \mu_\ell$ appears in a cycle in μ.gph, and thus in a cycle of the supergraph ρ.gph. Since ρ is well-formed, $\min \mu_\ell$ is minimal in ρ. □

We denote the set of well-formed memories by Mem. Figure 6 gives a positive example (top) and a negative example (bottom).

6.3 View-tree invariants

Like memories, view-trees maintain invariants during execution. In particular, the invariant that all thread views point downwards into the current memory depends on the invariants of memory, and vice-versa. Formally, we say that a view-tree T *points to/downward into* a memory μ, and write $T \rightarrowtail \mu$ and $T \hookrightarrow \mu$, when $\kappa \rightarrowtail \mu$ and $\kappa \hookrightarrow \mu$ for every $\kappa \in T.\mathtt{lf}$. We then say that a state $\langle T, \mu \rangle$ is *well-formed* when μ is well-formed and $T \hookrightarrow \mu$.

While the labels of the view-tree are related to the memory, its structure is intimately related to the syntactic structure of the configuration's term: every inner node relates to an active parallel composition. Figure 11 defines this property as an inductive relation $T \Vdash M$ specifying when T is *well-formed* for a term M. Every view-leaf is well-formed for any term. An inner node is well-formed for a parallel composition only when its immediate subtrees are well-formed for each thread. The rest of the rules reach through the term's evaluation context until they find a parallel composition sub-term. These rules follow the congruence rules from the operational semantics.

$\boxed{T \Vdash M}$

$$\textsc{Leaf}\ \frac{}{\dot\kappa \Vdash M} \qquad \textsc{Node}\ \frac{T \Vdash M \qquad R \Vdash N}{T \frown R \Vdash M \parallel N} \qquad \textsc{App-L}\ \frac{T \frown R \Vdash M}{T \frown R \Vdash MN} \qquad \textsc{App-R}\ \frac{T \frown R \Vdash N}{T \frown R \Vdash VN}$$

$$\textsc{Match}\ \frac{T \frown R \Vdash M}{T \frown R \Vdash \mathbf{match}\, M \,\mathbf{with}\, \Pi} \qquad \textsc{Variant}\ \frac{T \frown R \Vdash M}{T \frown R \Vdash A.\iota_i\, M} \qquad \textsc{Tuple}\ \frac{T \frown R \Vdash M_i}{T \frown R \Vdash \langle V_1, \dots, V_{i-1}, M_i, M_{i+1}, \dots, M_n \rangle}$$

$$\textsc{Store-L}\ \frac{T \frown R \Vdash M}{T \frown R \Vdash M \coloneqq N} \qquad \textsc{Store-R}\ \frac{T \frown R \Vdash N}{T \frown R \Vdash V \coloneqq N} \qquad \textsc{RMW-L}\ \frac{T \frown R \Vdash M}{T \frown R \Vdash \mathbf{rmw}_\varphi\, (M; N)} \qquad \textsc{RMW-R}\ \frac{T \frown R \Vdash N}{T \frown R \Vdash \mathbf{rmw}_\varphi\, (V; N)}$$

Fig. 11. The tree well-formedness rules of $\boldsymbol{\lambda}_{\mathrm{RA}}$.

Example 6.8. For $M = (M_1 \parallel M_2)\,;(N_1 \parallel N_2)$, a view leaf $\dot\kappa \Vdash M$ and an inner node $\dot\kappa_1 \frown \dot\kappa_2 \Vdash M$ are both well-formed. The evaluation context in M is $[-]\,;(N_1 \parallel N_2)$, and the active component—where reduction takes place—is $(M_1 \parallel M_2)$. The execution of $N_1 \parallel N_2$ is suspended, so we associate no views with its threads. The inner node is well-formed for the active component by Node.

For $N = V\,;(N_1 \parallel N_2)$, the inner node is *not* well-formed: $\dot\kappa_1 \frown \dot\kappa_2 \nVdash N$. The evaluation context is empty $[-]$, and the active (single) thread is $V\,;(N_1 \parallel N_2)$: the next execution step has to be MatchTuple (recall how sequencing desugars), requiring a view-leaf.

By inspecting the inductive definition of $(\Vdash)$ we find that no two rules can arrive at the same conclusion. Therefore, all of the rules are invertible: for every instantiation of every rule, if the conclusion holds, then so do all the premises. Thus when $T \Vdash M$ we can uniquely associate each subtree of T to a subterm of M. Moreover, the leaves of T are associated to threads within M such that there is no overlap.

Example 6.9. Returning to Example 6.8, by inverting $\dot\kappa_1 \frown \dot\kappa_2 \Vdash M$ we find which leaf associates to which subterm: $\dot\kappa_i \Vdash M_i$. Similarly, by inverting $\dot\kappa \Vdash M$ we find that $\dot\kappa \Vdash M_1 \parallel M_2$. Intuitively, the subthreads in the latter have not yet been activated (using ParInit).

6.4 Execution invariants

Collecting the invariants, a configuration $\langle T, \mu \rangle, M$ is *well-formed of type A* when: its state $\langle T, \mu \rangle$ is well-formed; its term is closed of type A, i.e. $\cdot \vdash M : A$; and its view-tree is well-formed for its term, i.e. $T \Vdash M$. We show that $\mathrm{RA}_{\leq}$ steps preserve well-formedness:

Theorem 6.10 (Preservation). *If $\langle T, \mu \rangle, M \rightsquigarrow_{RA_{\leq}} \langle R, \rho \rangle, N$ and $\langle T, \mu \rangle, M$ is a well-formed configuration of type A, then $\langle R, \rho \rangle, N$ is a well-formed configuration of type A.*

Proof. By induction on the step. Type-preservation is standard, so we focus on showing the other aspects of well-formedness of the configuration: that the tree is well-formed for the term and that the tree points downwards into the memory.

We start with the tree being well-formed for the term. The view-tree after the step is well-formed if it is a leaf by the Leaf rule. This covers Adv, ParFin, all of the β-reductions and memory-accessing steps. For the ParInit case, we use the Node rule and the Leaf rule for the premises.

Otherwise, we use the well-formedness rule that corresponds to the step. The well-formedness rules are invertible and derivations of well-formedness are unique, so we can apply the inverse rule before the step. For example, consider the AppLeft case ($M = M'K$ and $N = N'K$):

$$\frac{\langle T, \mu \rangle, M' \overset{e}{\rightsquigarrow}_{\mathrm{RA}_{\leq}} \langle R, \rho \rangle, N'}{\langle T, \mu \rangle, M'K \overset{e}{\rightsquigarrow}_{\mathrm{RA}_{\leq}} \langle R, \rho \rangle, N'K}\ \textsc{AppLeft}$$

By assumption $T \Vdash M'K$, so $T \Vdash M'$ by inverting the App-L rule. By the induction hypothesis, $R \Vdash N'$. If N' is a value, then R is a view-leaf by inverting the Leaf rule, and therefore $R \Vdash N'K$ by the Leaf rule. Otherwise, $R \Vdash N'K$ by the App-L rule.

The other cases are similar, with the exception of the ParLeft and ParRight cases. For both, we use the Node rule, and there is no need to distinguish the value case and use the Leaf rule.

It remains to show that the tree points downwards into the memory. We use the fact that pointwise maximum $\sqcup$ preserves pointing downwards for the cases of the ParFin, ReadOnly, and Adv rules. The view-leaf after the step points downward into the memory since it is the pointwise maximum of views that do. In the case of the Store rule, only the timestamp changes by increasing it, therefore preserving pointing downwards with respect to the other locations. With respect to the location itself, the property holds because the view-leaf points to the added message which has the same view, and views dominate themselves. For the RMW case we argue by composing the two arguments above.

For ParInit and the β-reductions, the claim is trivial because the set of views does not change. The remaining steps are congruence rules, where except for ParLeft and ParRight, the claim follows immediately from the inductive hypothesis because the states are the same in the premise. For ParLeft and ParRight, we need to also show that the other side of the view-tree points downwards into the new memory. Indeed, pointing downwards is stable under adding messages to memory, which is the only way the memory can change by taking any step. □

All initial configurations are well-formed, so executions only visit well-formed configurations.

Convention. Henceforth we restrict execution steps to be between well-formed configurations.

Execution steps maintain relationships between states beyond well-formdness. To start, we observe that the timestamp of a new message lies between some thread's initial and final views:

Lemma 6.11. *Assume $\langle T, \mu \rangle, M \rightsquigarrow_{RA_{\leq}} \langle R, \rho \rangle, N$ changed the memory, i.e. $\rho \neq \mu$. Then: the trees have the same shape; $T \leq R$; and there is a message ν such that $\rho = \mu \uplus \{\nu\}$. Moreover, there are view-leaves $\dot{\alpha}$ in T and $\dot{\omega}$ in R in corresponding positions, such that $\alpha \leq \nu.\mathtt{vw} \leq \omega$ and $\alpha_{\nu.\mathtt{lc}} < \nu.\mathtt{t}$.*

Proof. By induction on the step. The congruence cases are all immediate from the induction hypothesis. Of the others, only Store and RMW change the memory. They add a message single message, and the premises ensure the claim holds. □

The view-tree structure changes during ParInit and ParFin, so they cannot always be compared leaf-to-leaf as in Lemma 6.11. However, the sets of views that label each tree still maintain the *Egli-Milner order* induced by the view order:

Lemma 6.12 (Egli-Milner for View-leaves). *Assume $\langle T, \mu \rangle, M \rightsquigarrow^{*}_{RA_{\leq}} \langle R, \rho \rangle, N$.*

- *For every leaf view $\alpha \in T.\mathtt{lf}$, there exists a leaf $\omega \in R.\mathtt{lf}$, such that $\alpha \leq \omega$.*
- *For every leaf view $\omega \in R.\mathtt{lf}$, there exists a leaf $\alpha \in T.\mathtt{lf}$, such that $\alpha \leq \omega$.*

Proof. This property extends from a single step inductively. For a single step, we proceed by induction. The cases of the Adv rule and the memory-accessing steps the claim follows from their premises. The congruence cases and those that do not change the view-tree are all immediate from the induction hypothesis. The cases that change the tree structure, ParInit and ParFin, are immediate to check. ☐

We combine Lemmas 6.11 and 6.12 to obtain the following execution invariant:

Proposition 6.13 (Views Delimit Execution). *Assume* $\langle T, \mu \rangle, M \rightsquigarrow^*_{RA_\le} \langle R, \rho \rangle, N$. *Assume that* α *is dominated by every view in* $T.\mathtt{lf}$*, and that* ω *dominates every view in* $R.\mathtt{lf}$*. Then* $\alpha \le \omega$*; and for every added message* $\nu \in \rho \setminus \mu$*, both* $\alpha \le \nu.\mathtt{vw} \le \omega$ *and* $\alpha_{\nu.\mathtt{lc}} < \nu.\mathtt{t}$.

Proof. That $\alpha \le \omega$ follows from Lemma 6.12. For the rest, consider an added message $\nu \in \rho \setminus \mu$, and the decomposition—guaranteed by Lemma 6.11—that singles out the step that added it:

$$\langle T, \mu \rangle, M \rightsquigarrow^*_{\mathrm{RA}_\le} \langle T', \theta \rangle, M' \rightsquigarrow_{\mathrm{RA}_\le} \langle R', \theta \uplus \{\nu\} \rangle, N' \rightsquigarrow^*_{\mathrm{RA}_\le} \langle R, \rho \rangle, N$$

By Lemma 6.11 and the singled-out step in the middle, there exist $\alpha' \in T'.\mathtt{lf}$ and $\omega' \in R'.\mathtt{lf}$ such that $\alpha' \le \nu.\mathtt{vw} \le \omega'$ and $\alpha'_{\nu.\mathtt{lc}} < \nu.\mathtt{t}$. By Lemma 6.12 and the surrounding steps, there exist $\alpha'' \in T.\mathtt{lf}$ and $\omega'' \in R.\mathtt{lf}$ such that $\alpha'' \le \alpha'$ and $\omega' \le \omega''$. By assumption, $\alpha \le \alpha''$ and $\omega'' \le \omega$. Putting it all together, we have $\alpha \le \alpha' \le \nu.\mathtt{vw} \le \omega' \le \omega$ and $\alpha_{\nu.\mathtt{lc}} \le \alpha'_{\nu.\mathtt{lc}} < \nu.\mathtt{t}$. ☐

We conjecture that other standard metatheoretic results, such as progress and termination, hold with standard proofs.

6.5 Interrupted executions

To analyze program behavior under concurrent contexts, we have to take into account all possible ways in which the environment can interfere during the execution. An *interrupted execution* $\langle T, \mu \rangle, M \rightsquigarrow^*_{\mathrm{RA}_\le} \cdots \rightsquigarrow^*_{\mathrm{RA}_\le} \langle R, \rho \rangle, V$ is a sequence of executions of the form

$$\begin{aligned} \langle T, \mu \rangle, M = \langle T_1, \mu_1 \rangle, M_1 &\rightsquigarrow^*_{\mathrm{RA}_\le} \langle T_2, \rho_1 \rangle, M_2 \\ &\qquad \langle T_2, \mu_2 \rangle, M_2 \rightsquigarrow^*_{\mathrm{RA}_\le} \langle T_3, \rho_2 \rangle, M_3 \\ &\qquad\qquad \vdots \\ &\qquad\qquad \langle T_n, \mu_n \rangle, M_n \rightsquigarrow^*_{\mathrm{RA}_\le} \langle T_{n+1}, \rho_n \rangle, M_{n+1} = \langle R, \rho \rangle, V \end{aligned}$$

where $\rho_j \subseteq \mu_{j+1}$ for every $1 \le j \le n-1$. Between the executions in the sequence, the configuration may only change by adding messages to memory. We call these messages—those in $\mu_{j+1} \setminus \rho_j$—*environment messages*. Adding messages is the only interference that the environment can cause. We also have $\mu_i \subseteq \rho_i$, and we call the messages in $\rho_j \setminus \mu_j$ *local messages*. Proposition 6.13 extends to interrupted executions in a straightforward manner, replacing $\rightsquigarrow^*_{\mathrm{RA}_\le}$ with $\rightsquigarrow^*_{\mathrm{RA}_\le} \cdots \rightsquigarrow^*_{\mathrm{RA}_\le}$ and replacing *added* messages with *local* messages.

7 Denotational Semantics

Taking Moggi's monadic approach (§7.1) to denotational semantics as a basis, we design a framework for denotational semantics using Brookes-style traces (§7.2) adapted to describe behavior under RA. We then build upon this framework progressively.

First we define the *generating* denotational semantics (§7.3). The monad structure underlying this semantics does not satisfy all monad laws, and so does not fully conform to the monadic approach. It is a useful base for the next stage, as a metatheoretic tool, and it simplifies calculations.

Next, we define the *concrete* denotational semantics (§7.4). Here we do have a monad, but the denotational semantics follows the operational semantics too closely. It is insufficiently abstract, invalidating some program transformations. This semantics is useful as an intermediate step, and plays a central role in our proof of the adequacy theorem.

Finally, we define the *abstract* denotational semantics (§7.5). This semantics is the semantics we were aiming for: adequate and abstract enough to justify transformations of interest.

7.1 Monad-based Semantics

We recall Moggi's [40] approach to interpret a CBV calculus such as $\boldsymbol{\lambda}_{\text{RA}}$ using a monad. A *monad structure* $\mathcal{T} = \left\langle \underline{\mathcal{T}}, \text{return}^{\mathcal{T}}, (\gg\!=^{\mathcal{T}}) \right\rangle$ consists of three components: a set-level function $\underline{\mathcal{T}}$; a set-indexed function $\text{return}^{\mathcal{T}}$; and a two-argument set-pair-indexed function $(\gg\!=^{\mathcal{T}})$. The set-level function assigns to each set X, whose elements represent fully-evaluated semantic values, the set $\underline{\mathcal{T}}X$, whose elements represent unevaluated effectful programs returning values in X. The functions $\text{return}^{\mathcal{T}}_{\text{X}} : X \to \underline{\mathcal{T}}X$, the *unit*, represent the program fragment that returns its input without any observable side-effects. The two-argument functions $(\gg\!=^{\mathcal{T}}_{X,Y}) : (\underline{\mathcal{T}}X) \times (X \to \underline{\mathcal{T}}Y) \to \underline{\mathcal{T}}Y$, the *monadic bind*, represent the sequencing $P \gg\!=^{\mathcal{T}}_{X,Y} f$ of an X-returning program P with an Y-returning program f that depends on the result of the former program P. We often omit the monad and the set-indexing from notations, leaving them implicit.

Moggi's innovation is to take the traditional type and value semantics, following a long tradition of denotational semantics, and retain its uniform structure even for effectful computation, by using a monad structure. Each syntactic construct has a corresponding semantic construct, and the interpretation proceeds structurally over the structure of types, contexts and terms.

Type semantics. Every type A denotes a set, where: product types denote the cartesian product; variants denote tagged unions; function types use the monad structure to denote the set of parameterized computations; and typing environments denote the cartesian product:

$$[\![\Gamma]\!]_{\mathcal{T}} \coloneqq \prod_{(a:A)\in\Gamma} [\![A]\!]_{\mathcal{T}} \quad [\![A \to B]\!]_{\mathcal{T}} \coloneqq [\![A]\!]_{\mathcal{T}} \to \underline{\mathcal{T}}[\![B]\!]_{\mathcal{T}} \quad [\![(A_1 * \cdots * A_n)]\!]_{\mathcal{T}} \coloneqq [\![A_1]\!]_{\mathcal{T}} \times \cdots \times [\![A_n]\!]_{\mathcal{T}}$$

$$[\![\{\iota_1 \text{ of } A_1 \mid \cdots \mid \iota_n \text{ of } A_n\}]\!]_{\mathcal{T}} \coloneqq (\{\iota_1\} \times [\![A_1]\!]_{\mathcal{T}}) \cup \cdots \cup (\{\iota_n\} \times [\![A_n]\!]_{\mathcal{T}})$$

In particular, denotations of ground types $[\![G]\!]_{\mathcal{T}}$ do not depend on the monad structure. For example, $[\![\mathbf{Val}]\!]_{\mathcal{T}}$ is in a bijection with the (storable) values Val, and we will identify them.

Value semantics. Every value $\Gamma \vdash V : A$ denotes a function $[\![V]\!]^{\mathsf{v}}_{\mathcal{T}} : [\![\Gamma]\!]_{\mathcal{T}} \to [\![A]\!]_{\mathcal{T}}$, taking as argument a *semantic environment* $\gamma \in [\![\Gamma]\!]_{\mathcal{T}}$ supplying a semantic value to each variable in Γ:

$$[\![b]\!]^{\mathsf{v}}_{\mathcal{T}}(\gamma_a)_{(a:A)\in\Gamma} \coloneqq \gamma_b \quad [\![A.\iota\, V]\!]^{\mathsf{v}}_{\mathcal{T}}\gamma \coloneqq \left\langle \iota, [\![V]\!]^{\mathsf{v}}_{\mathcal{T}}\gamma \right\rangle \quad [\![\langle V_1, \ldots, V_n\rangle]\!]^{\mathsf{v}}_{\mathcal{T}}\gamma \coloneqq \left\langle [\![V_1]\!]^{\mathsf{v}}_{\mathcal{T}}\gamma, \ldots, [\![V_n]\!]^{\mathsf{v}}_{\mathcal{T}}\gamma \right\rangle$$

$$[\![\lambda b : B.\, M]\!]^{\mathsf{v}}_{\mathcal{T}}(\gamma_a)_{(a:A)\in\Gamma} \coloneqq \lambda\gamma_b.\, [\![M]\!]^{\mathsf{v}}_{\mathcal{T}}(\gamma_a)_{(a:A)\in\Gamma, b:B}$$

Closed values $\cdot \vdash V : A$ denote functions from the singleton $[\![\cdot]\!]_{\mathcal{T}} \coloneqq \{()\}$ to $[\![A]\!]_{\mathcal{T}}$, so we write $[\![V]\!]^{\mathsf{v}}_{\mathcal{T}}$ for $[\![V]\!]^{\mathsf{v}}_{\mathcal{T}}()$. The semantics of closed ground values do not use the monad structure.

Term semantics. Every term $\Gamma \vdash M : A$ denotes a function $[\![M]\!]^{\mathsf{c}}_{\mathcal{T}} : [\![\Gamma]\!]_{\mathcal{T}} \to \underline{\mathcal{T}}[\![A]\!]_{\mathcal{T}}$. The monadic bind expresses left-to-right evaluation order, and the unit expresses pure computation, e.g.:

$$[\![MN]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \coloneqq [\![M]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg\!= \lambda g.\, [\![N]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg\!= \lambda r.\, g(r)$$

$$[\![\langle M_1, \ldots, M_n\rangle]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \coloneqq [\![M_1]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg\!= \lambda r_1. \cdots [\![M_n]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg\!= \lambda r_n.\, \text{return}\,\langle r_1, \ldots, r_n\rangle$$

Monad laws. While a monad structure suffices to define these interpretations, it does not suffice to guarantee they behave as expected. For example, a nested tuple of values $V \coloneqq \langle\langle 1, 2\rangle, 3\rangle$ has the value semantics $[\![V]\!]^{\mathsf{v}}_{\mathcal{T}} = \langle\langle 1, 2\rangle, 3\rangle$ and the term semantics:

$$[\![V]\!]^{\mathsf{c}}_{\mathcal{T}} = (\text{return}\, 1 \gg\!= \lambda r.\, (\text{return}\, 2 \gg\!= \lambda s.\, \text{return}\,\langle r, s\rangle)) \gg\!= \lambda r'.\, (\text{return}\, 3 \gg\!= \lambda s'.\, \text{return}\,\langle r', s'\rangle)$$

We would expect the two semantics to relate via $[\![V]\!]^{\mathsf{c}}_{\mathcal{T}} = \text{return } [\![V]\!]^{\mathsf{v}}_{\mathcal{T}}$, but a mere monad structure will not guarantee it. A *monad* is a monad structure satisfying:

$$\text{return}\, r \gg= f = f(r) \qquad \text{(Left Neutrality)}$$
$$P \gg= \text{return} = P \qquad \text{(Right Neutrality)}$$
$$(P \gg= f) \gg= g = P \gg= \lambda r.\, (f(r) \gg= g) \qquad \text{(Associativity)}$$

As Moggi shows, a monad does guarantee the value and term semantics agree in this way.

The metatheory also uses the monad laws extensively, such as in the following lemma, which relates substitutions to standard denotations via typing context extension. Denote by $\Delta \leq \Gamma$ the statement that $(a : A) \in \Gamma$ whenever $(a : A) \in \Delta$; and define $\Gamma \setminus \Delta$ by $(a : A) \in \Gamma \setminus \Delta$ iff $(a : A) \in \Gamma$ and $(a : A) \notin \Delta$. Let $\mathsf{Sub}_\Delta \coloneqq \prod_{(a:A)\in\Delta} \{V \mid \cdot \vdash V : A\}$ be the set of variable substitutions for Δ. For $\Theta \in \mathsf{Sub}_\Delta$, denote by ΘM the standard simultaneous substitution by Θ in M.

Lemma 7.1 (Substitution Lemma). *Given a monad $\mathcal{T}$, assume $\Gamma \vdash M : A$ and let $\Theta \in \mathsf{Sub}_\Delta$ for some $\Delta \leq \Gamma$. For all $\gamma \in [\![\Gamma]\!]_{\mathcal{T}}$, if $\forall (b : B) \in \Delta.\, \gamma b = [\![\Theta_b]\!]^{\mathsf{v}}_{\mathcal{T}}$, then $[\![M]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma = [\![\Theta M]\!]^{\mathsf{c}}_{\mathcal{T}}(\gamma_b)_{(b:B)\in\Gamma\setminus\Delta}$.*

Structural transformations. Using the monad we can justify a wide class of simple transformations called structural transformations. For example, given $\vdash V : \{\textbf{true} \mid \textbf{false}\}$ and $\Gamma \vdash M : A$:

$$[\![\textbf{match}\, V\, \textbf{with}\, \{\textbf{true}.M \mid \textbf{false}.M\}]\!]^{\mathsf{c}}_{\mathcal{T}} = [\![M]\!]^{\mathsf{c}}_{\mathcal{T}}$$

Even though M may use program effects, we can prove the equality by reasoning synthetically with the monad and the semantic constructs, keeping $[\![M]\!]^{\mathsf{c}}_{\mathcal{T}}$ indeterminate.

Adding the effects. An advantage of Moggi's approach is its compatibility with effects. To define the denotations of shared-memory constructs (right), we equip $\mathcal{T}$ with more structure, one component per construct (left):

$$[\![\text{store}_{\ell,v}]\!]_{\mathcal{T}} \in \underline{\mathcal{T}}\mathbf{1} \qquad [\![M \coloneqq N]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \coloneqq [\![M]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg= \lambda\ell.\, [\![N]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg= \lambda v.\, [\![\text{store}_{\ell,v}]\!]_{\mathcal{T}}$$
$$[\![\text{rmw}_{\ell,\Phi}]\!]_{\mathcal{T}} \in \underline{\mathcal{T}}\mathsf{Val} \qquad [\![\textbf{rmw}_\varphi\,(M;N)]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \coloneqq [\![M]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg= \lambda\ell.\, [\![N]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \gg= \lambda\vec{v}.\, [\![\text{rmw}_{\ell,\varphi_{\vec{v}}}]\!]_{\mathcal{T}}$$
$$(|||^{\mathcal{T}}_{X,Y}) : \underline{\mathcal{T}}X \times \underline{\mathcal{T}}Y \to \underline{\mathcal{T}}\,(X \times Y) \qquad [\![M \parallel N]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \coloneqq [\![M]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \mathbin{|||} [\![N]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma$$

7.2 Trace-based Semantics

We instantiate the monad-based semantics to our case using traces, the semantic counterpart to interrupted executions. Their core component is a sequence of memory-transitions, summarizing which messages the behavior they describe relies on and guarantees. A *(memory)-transition* is pair $\langle\mu, \rho\rangle$ of memories, such that $\mu \subseteq \rho$.

We capture the evolving reliances and guarantees about memory messages with a *chronicle*: a possibly empty finite sequence of transitions $\xi = \langle\mu_1, \rho_1\rangle \dots \langle\mu_n, \rho_n\rangle$ where $\rho_j \subseteq \mu_{j+1}$ for every j. When ξ is non-empty, we denote its *opening* and *closing* memories by $\xi.\mathtt{o} \coloneqq \mu_1$ and $\xi.\mathtt{c} \coloneqq \rho_n$. Its *local messages* are the ones added within transitions: $\xi.\mathtt{own} \coloneqq \bigcup_{i=1}^{n}(\rho_i \setminus \mu_i)$. The other messages $\rho_n \setminus \xi.\mathtt{own}$ are its *environment messages.* Let Chro be the set of chronicles, ranged over by ξ, η.

In the operational semantics, a thread's view may obscure some messages. The trace captures only an *initial view* that declares which messages may be relied on to be available at the beginning, and a *final view* that declares which messages are guaranteed to be available at the end. Together, these are the *delimiting views.*

Finally, a trace includes a semantic representation of the returned value. Given a set X representing semantic return values, an *X-pre-trace* is an element $\tau \in \mathsf{View} \times \mathsf{Chro} \times \mathsf{View} \times X$, written $\tau = \alpha\boxed{\xi}\omega \mathrel{.\!\raisebox{0.5ex}{.}\!.} r$, whose chronicle ξ is non-empty. We retrieve the components of the pre-trace τ

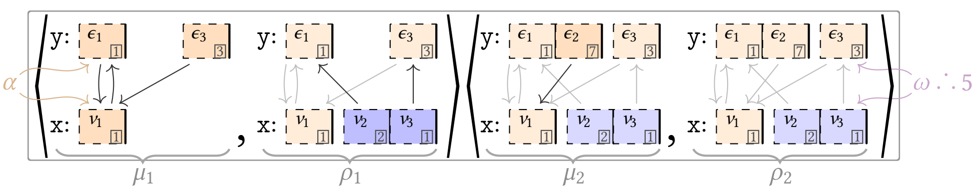
using $\tau.\mathtt{ivw} \coloneqq \alpha$ (initial view), $\tau.\mathtt{ch} \coloneqq \xi$ (chronicle), $\tau.\mathtt{fvw} \coloneqq \omega$ (final view), and $\tau.\mathtt{ret} \coloneqq r$ (returned value). Let τ, π, ϱ range over pre-traces.

An X-pre-trace $\tau = \alpha\boxed{\xi}\omega \mathrel{.\cdot.} r$ is an X-*trace* when:

- each transition in ξ consists of well-formed memories;
- the initial view is dominated by the final view, and these views points downwards into the opening and closing memories respectively: $\xi.\mathtt{o} \hookleftarrow \alpha \leq \omega \hookrightarrow \xi.\mathtt{c}$; and
- the view and the segment of every local message are both bound by the delimiting views, i.e.: $\forall\, \nu \in \xi.\mathtt{own}.\, \alpha \leq \nu.\mathtt{vw} \leq \omega \wedge \alpha_{\nu.\mathtt{lc}} < \nu.\mathtt{t}$.

These conditions reflect the execution invariants of Theorem 6.10 and Proposition 6.13. We denote the set of X-traces by $\mathsf{Trace}X$.

Example 7.2. We present a trace $\alpha\boxed{\langle\mu_1, \rho_1\rangle\ \langle\mu_2, \rho_2\rangle}\omega \mathrel{.\cdot.} 5$, also depicted in Figure 1 (bottom):



The trace's closing memory ρ_2 holds six messages. The arrows pointing between messages illustrate the graph structure that the views impose. Messages are spatially parted iff they are apart, e.g. ν_3 dovetails after ν_2, which is apart from ν_1. We highlight messages that are not part of a previous memory. The local messages are ν_2 and ν_3; the rest are environment messages.

Closure rules. Semantics of terms $P \in \underline{\mathcal{T}}X$ in trace semantics are sets of traces, representing the possible behaviors, including possible environment interference. As subsets, they carry a natural inclusion order. We write $[\![M]\!]^{\mathsf{c}}_{\mathcal{T}} \subseteq [\![N]\!]^{\mathsf{c}}_{\mathcal{T}}$ for containment in any context: $\forall\, \gamma \in [\![\Gamma]\!]_{\mathcal{T}}.\, [\![M]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma \subseteq [\![N]\!]^{\mathsf{c}}_{\mathcal{T}}\gamma$. Intuitively, this property means that every behavior of M is a behavior of N.

We will be looking at sets of traces closed under certain closure rules reflecting the way in which traces represent possible behaviors. A *closure/rewrite rule* $\times$ is a binary relation between pre-traces. When the relation holds, written $\tau \xrightarrow{\times} \pi$, we say that the *source* τ $\times$*-rewrites* to the *target* π. Let $\star$ be a set of closure rules. We write $\tau \xrightarrow{\star} \pi$ when $\tau \xrightarrow{\times} \pi$ for some $\times \in \star$. A set $U \subseteq \mathsf{Trace}X$ is $\star$*-closed* when $\tau \in U$ and $\tau \xrightarrow{\star} \pi \in \mathsf{Trace}X$ implies $\pi \in U$. The $\star$*-closure* of a set $U \subseteq \mathsf{Trace}X$, denoted $U^{\star}$, is the least $\star$-closed superset of U. Thus U is $\star$-closed iff $U = U^{\star}$. We denote the set of countable $\star$-closed subsets of E by $\mathcal{P}^{\star}_{\mathrm{ctbl}}(E) \coloneqq \{U \in \mathcal{P}_{\mathrm{ctbl}}(E) \mid U = U^{\star}\}$. We $\star$*-close* a function ϕ that returns sets of traces by composition with the closure: $\phi^{\star} \coloneqq -^{\star} \circ \phi$. We say that a function ϕ is *pointwise* $\star$*-closed* when $\phi = \phi^{\star}$. We say that a function ϕ between sets of traces is $\star$*-closed* when its restriction to $\star$-closed subsets is pointwise $\star$-closed.

Monad structure. Given a choice of closure rules $\star$, we define the $\star$*-monad structure* $\mathcal{T}$ as follows. The set-level function of $\mathcal{T}$'s monad structure sends every set X to the set of countable $\star$-closed sets of X-traces: $\underline{\mathcal{T}}X \coloneqq \mathcal{P}^{\star}_{\mathrm{ctbl}}(\mathsf{Trace}X)$. The unit $\star$-closes over all single-transition traces that maintain the view and the memory. The bind appends traces with compatible intermediate views:

$$\mathrm{return}^{\mathcal{T}}_{X} r \coloneqq \left\{\kappa\boxed{\langle\mu,\mu\rangle}\kappa \mathrel{.\cdot.} r \in \mathsf{Trace}X \mid \kappa \in \mathsf{View}, \mu \in \mathsf{Mem}\right\}^{\star}$$

$$P \gg\!\!=^{\mathcal{T}}_{X,Y} f \coloneqq \left\{\alpha\boxed{\xi\eta}\omega \mathrel{.\cdot.} s \in \mathsf{Trace}Y \mid \alpha\boxed{\xi}\kappa \mathrel{.\cdot.} r \in P,\ \kappa \leq \sigma,\ \sigma\boxed{\eta}\omega \mathrel{.\cdot.} s \in fr\right\}^{\star}$$

Parallel composition. $(|||^{\mathcal{T}})$ interleaves chronicles and pairs the returned values. The delimiting views must bound the views of the resulting traces, so we take the greatest lower bound of the initial views, and the least upper bound of the final views. To define these bounds, denote the set

Table 1. Summary of the ⋆-monad structures $\mathcal{T}$ that we define.

Model	$\mathcal{T}$	⋆	Monad?	Noteworthy properties	Section
Null	$\mathcal{N}$	$\emptyset$	No	Straightforward denotations	§7.2
Generating	$\mathcal{G}$	$\mathfrak{g}$	No	Simple to calculate (Proposition 7.5) and allows deferring closure (Lemma 8.5)	§7.3
Concrete	$\mathcal{C}$	$\mathfrak{gc}$	Yes	Strongly corresponds to $\mathrm{RA}_{\leq}$ operational semantics (Theorem 8.12 and Lemma 8.14)	§7.4
Abstract	$\mathcal{A}$	$\mathfrak{gca}$	Yes	Adequate (Theorem 8.13) and abstract (Table 3)	§7.5

of views pointing downward into a well-formed memory μ by $- \hookrightarrow \mu := \{\kappa \in \mathsf{View} \mid \kappa \hookrightarrow \mu\}$. This set is finite since Loc and μ are finite, and each κ mentions only timestamps that appear in μ; and it has a minimum: the view that points to all the initial messages $\lambda\ell.\min \mu_\ell.\mathtt{t}$. Consider a non-empty subset of views $U \subseteq - \hookrightarrow \mu$. Since $- \hookrightarrow \mu$ is finite and closed under $\sqcup$, the subset U has a least upper bound $\sup_\mu U := \bigsqcup U$. Since $- \hookrightarrow \mu$ has a minimal element—the view pointing to the minimal messages—U also has a greatest lower bound $\inf_\mu U := \bigsqcup \{\kappa \in \mathsf{View} \mid \bigsqcap U \geq \kappa \hookrightarrow \mu\}$. Though $\bigsqcap U$ bounds U below, it may not be in $- \hookrightarrow \mu$.

Example 7.3. For μ the memory from Example 6.6, $\alpha_1 := \langle\!\langle \mathtt{x@5}\ ;\ \mathtt{y@7} \rangle\!\rangle$ and $\alpha_2 := \langle\!\langle \mathtt{x@7}\ ;\ \mathtt{y@5} \rangle\!\rangle$, we have $\alpha_1 \hookrightarrow \mu$ and $\alpha_2 \hookrightarrow \mu$, but $\alpha_1 \sqcap \alpha_2 = \langle\!\langle \mathtt{x@5}\ ;\ \mathtt{y@5} \rangle\!\rangle \not\hookrightarrow \mu$. Here, $\inf_\mu \{\alpha_1, \alpha_2\} = \langle\!\langle \mathtt{x@0}\ ;\ \mathtt{y@0} \rangle\!\rangle$.

Denote by $\xi_1 \parallel \xi_2$ the set of all the interleavings of ξ_1 and ξ_2 that form chronicles. We define:

$$P_1 \,|||^{\mathcal{T}}_{X_1,X_2}\, P_2 := \left\{ \begin{array}{l} \inf_{\xi.\mathtt{o}} \{\alpha_1, \alpha_2\} \boxed{\xi} \sup_{\xi.\mathtt{c}} \{\omega_1, \omega_2\} \mathrel{.\!\raisebox{0.4ex}{.}\!.} \langle r_1, r_2 \rangle \in \mathsf{Trace}\,(X_1 \times X_2) \\ \mid \xi \in (\xi_1 \parallel \xi_2) \wedge \forall i \in \{1,2\} \,.\, \alpha_i \boxed{\xi_i} \omega_i \mathrel{.\!\raisebox{0.4ex}{.}\!.} r_i \in P_i \end{array} \right\}^{\star}$$

Memory access. Mirroring the operational semantics, we interpret:

$$[\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{T}} := \left\{ \kappa \boxed{\langle \mu, \mu \uplus \{\ell{:}v@(q,t]\langle\!\langle \kappa[\ell \mapsto t] \rangle\!\rangle\} \rangle} \kappa[\ell \mapsto t] \mathrel{.\!\raisebox{0.4ex}{.}\!.} \langle\rangle \in \mathsf{Trace1} \;\middle|\; t, q \in \mathbb{Q} \right\}^{\star}$$

$$[\![\mathrm{rmw}_{\ell,\Phi}]\!]_{\mathcal{T}} := [\![\mathrm{rmw}^{\mathrm{RO}}_{\ell,\Phi}]\!]_{\mathcal{T}} \cup [\![\mathrm{rmw}^{\mathrm{RMW}}_{\ell,\Phi}]\!]_{\mathcal{T}} \text{ where:}$$

$$[\![\mathrm{rmw}^{\mathrm{RO}}_{\ell,\Phi}]\!]_{\mathcal{T}} := \left\{ \kappa \boxed{\langle \mu, \mu \rangle} \kappa \mathrel{.\!\raisebox{0.4ex}{.}\!.} v.\mathtt{vl} \in \mathsf{TraceVal} \;\middle|\; \kappa \rightarrowtail v \in \mu_\ell \,,\, \Phi\,(v.\mathtt{vl}) = \bot \right\}^{\star}$$

$$[\![\mathrm{rmw}^{\mathrm{RMW}}_{\ell,\Phi}]\!]_{\mathcal{T}} := \left\{ \begin{array}{l} \kappa \boxed{\langle \mu, \mu \uplus \{\epsilon\} \rangle} \kappa[\ell \mapsto t] \mathrel{.\!\raisebox{0.4ex}{.}\!.} v.\mathtt{vl} \in \mathsf{TraceVal} \\ \mid \kappa \rightarrowtail v \in \mu_\ell \,,\, \epsilon = \ell{:}\Phi\,(v.\mathtt{vl})\,@(v.\mathtt{t}, t]\langle\!\langle \kappa[\ell \mapsto t] \rangle\!\rangle \end{array} \right\}^{\star}$$

Requiring the resulting pre-traces to form traces ensures the constraints on their timestamps and segment hold. Assignment stores a new message. The RMW interpretation loads a message, and stores a new message depending on the modifier's result.

This semantics restricts loading to messages that the initial view already points to. This restriction will make the denotations from §7.3 more convenient to use. The semantics in §7.4 will include traces that load messages with later timestamps, thanks to closure under the rewind closure rule.

Monotonicity. To accommodate reasoning about refinement, we establish that the trace monad operators are monotonic with respect to set inclusion:

Proposition 7.4. *Let $P_i, Q_i \in \underline{\mathcal{T}}X_i$ and $f, g : X_1 \to \underline{\mathcal{T}}X_2$. If $P_i \subseteq Q_i$ and $\forall r \in X_1.\, fr \subseteq gr$, then:*

$$P_1 \gg\!=^{\mathcal{T}} f \subseteq Q_1 \gg\!=^{\mathcal{T}} g \qquad P_1 \,|||^{\mathcal{T}}\, P_2 \subseteq Q_1 \,|||^{\mathcal{T}}\, Q_2$$

Proof. The $(-)^{\star}$ operator is monotonic by virtue of being a closure operator. Thus, it suffices to show inclusion for the underlying sets (as if $\star = \emptyset$). This proof is straightforward from the set-definitions, where traces are suitable combinations of traces in the operands. □

Table 2. Summary of all closure rules: generating ($\mathfrak{g}$), concrete ($\mathfrak{c}$), and abstract ($\mathfrak{a}$).

	Name	Source Trace	Closure Relation	Target Trace	Condition	Figure(s)
$\mathfrak{g}$	Loosen	$\alpha \boxed{\xi \left(\eta \overline{\uplus} \{\epsilon\}\right)} \omega$	$\xrightarrow{\mathsf{Ls}}$	$\alpha \boxed{\xi \left(\eta \overline{\uplus} \{\nu\}\right)} \omega$	$\nu \preceq_{\mathtt{vw}} \epsilon$	12L
	Expel	$\alpha \boxed{\xi \left(\eta \overline{\uplus} \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\}\right)} \omega$	$\xrightarrow{\mathsf{Ex}}$	$\alpha \boxed{\xi \left(\eta \overline{\uplus} \{\nu, \epsilon\}\right)} \omega$	$\nu \mathrel{-\!\!\subset} \epsilon$	13L
	Condense	$\alpha \boxed{\xi \left(\eta \overline{\uplus} \{\nu, \epsilon\}\right)} \omega$	$\xrightarrow{\mathsf{Cn}}$	$\left(\alpha \boxed{\xi \left(\eta \overline{\uplus} \{\nu\}\right)} \omega\right) [\uparrow\epsilon]$	$\nu \mathrel{-\!\!\in} \epsilon$	14L, 15L
$\mathfrak{c}$	Stutter	$\alpha \boxed{\xi\eta} \omega$	$\xrightarrow{\mathsf{St}}$	$\alpha \boxed{\xi \langle \mu, \mu \rangle \eta} \omega$		
	Mumble	$\alpha \boxed{\xi \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta} \omega$	$\xrightarrow{\mathsf{Mu}}$	$\alpha \boxed{\xi \langle \mu, \theta \rangle \eta} \omega$		
	Forward	$\alpha \boxed{\xi} \kappa$	$\xrightarrow{\mathsf{Fw}}$	$\alpha \boxed{\xi} \omega$	$\kappa \leq \omega$	16L
	Rewind	$\kappa \boxed{\xi} \omega$	$\xrightarrow{\mathsf{Rw}}$	$\alpha \boxed{\xi} \omega$	$\alpha \leq \kappa$	16R
$\mathfrak{a}$	Tighten	$\alpha \boxed{\xi \langle \mu, \rho \uplus \{\nu\} \rangle \eta \overline{\uplus} \{\nu\}} \omega$	$\xrightarrow{\mathsf{Ti}}$	$\alpha \boxed{\xi \langle \mu, \rho \uplus \{\epsilon\} \rangle \eta \overline{\uplus} \{\epsilon\}} \omega$	$\nu \preceq_{\mathtt{vw}} \epsilon$	12R, 17
	Absorb	$\alpha \boxed{\xi \langle \mu, \rho \uplus \{\nu, \epsilon\} \rangle \eta \overline{\uplus} \{\nu, \epsilon\}} \omega$	$\xrightarrow{\mathsf{Ab}}$	$\alpha \boxed{\xi \left\langle \mu, \rho \uplus \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\} \right\rangle \eta \overline{\uplus} \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\}} \omega$	$\nu \mathrel{-\!\!\subset} \epsilon$	13R, 18
	Dilute	$\left(\alpha \boxed{\xi \langle \mu, \rho \uplus \{\nu\} \rangle \eta \overline{\uplus} \{\nu\}} \omega\right) [\uparrow\epsilon]$	$\xrightarrow{\mathsf{Di}}$	$\alpha \boxed{\xi \langle \mu, \rho \uplus \{\nu, \epsilon\} \rangle \eta \overline{\uplus} \{\nu, \epsilon\}} \omega$	$\nu \mathrel{-\!\!\in} \epsilon$	14R, 15R, 19

Our monad structures. In the degenerate case of taking no closure rules $\star \coloneqq \emptyset$, we call the resulting $\star$-monad structure the *null model* $\mathcal{N}$. This model invalidates both identity axioms. Indeed, $(\mathrm{return}^{\mathcal{N}} r \mathbin{\rangle\!\rangle\!=} \mathrm{return}^{\mathcal{N}}) \neq \mathrm{return}^{\mathcal{N}} r$, because only the traces from the left side of the inequation have two transitions. The induced denotational semantics is insufficiently abstract. For example, the inequation above implies that $[\![\langle\rangle \,;\, \langle\rangle]\!]^{\mathsf{c}}_{\mathcal{N}} \neq [\![\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{N}}$, showing that this model fails to satisfy even the most basic semantic equivalences. Still, we will find that less abstract models provide stepping stones to more abstract ones.

Each following subsection (§7.3-7.5) defines an additional monad structure, summarized in Table 1. Each structure builds on the previous one by adding closure rules, summarized in Table 2. This compact table packs many side conditions and new notation, which we explain as we present the rules. In presenting these closure rules we omit the return value, because they all maintain it.

7.3 Generating Denotations

We identify a set of closure rules $\mathfrak{g} \coloneqq \{\mathsf{Ls}, \mathsf{Ex}, \mathsf{Cn}\}$ under which the operations of the null model are closed: $\mathrm{return}^{\mathcal{N}}$ is pointwise closed under $\mathfrak{g}$; if f is pointwise $\mathfrak{g}$-closed, then $\mathbin{\rangle\!\rangle\!=}^{\mathcal{N}} f$ is $\mathfrak{g}$-closed; and similarly for the effect operations. Let the *generating model* $\mathcal{G}$ be the $\mathfrak{g}$-monad structure.

Proposition 7.5. *For all $P_i \in \underline{\mathcal{G}} X_i$ and $f : X_1 \to \underline{\mathcal{G}} X_2$:*

$$P_1 \mathbin{\rangle\!\rangle\!=}^{\mathcal{N}} f \;=\; P_1 \mathbin{\rangle\!\rangle\!=}^{\mathcal{G}} f \qquad\qquad P_1 \mathbin{|||}^{\mathcal{N}} P_2 \;=\; P_1 \mathbin{|||}^{\mathcal{G}} P_2$$

Moreover, $\mathrm{return}^{\mathcal{N}} = \mathrm{return}^{\mathcal{G}}$, $[\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{N}} = [\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{G}}$, *and* $[\![\mathrm{rmw}_{\ell,\Phi}]\!]_{\mathcal{N}} = [\![\mathrm{rmw}_{\ell,\Phi}]\!]_{\mathcal{G}}$.

This proposition means that we can calculate in $\mathcal{G}$ as concretely as in $\mathcal{N}$; we need not worry about traces obtained from the set-definitions by applying some arbitrarily long chain of closures.

The difference between denotations in $\mathcal{G}$ and in $\mathcal{N}$ lies in the higher-order fragment. For example, traces in $[\![\lambda f : \mathbf{1} \to \mathbf{1}.\, f\,\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{T}}$ have return value $\lambda f \in [\![\mathbf{1}]\!] \to \underline{\mathcal{T}}\,[\![\mathbf{1}]\!].\, f\,\langle\rangle$. For $\mathcal{T} = \mathcal{N}$ the return value is defined on functions $f \in [\![\mathbf{1}]\!] \to \underline{\mathcal{N}}\,[\![\mathbf{1}]\!]$, which may not be pointwise $\mathfrak{g}$-closed. In contrast, for $\mathcal{T} = \mathcal{G}$ the return value is only defined on $f \in [\![\mathbf{1}]\!] \to \underline{\mathcal{G}}\,[\![\mathbf{1}]\!]$.

We provide operational intuition for each $\mathfrak{g}$-closure rule by drawing explicit connections with interrupted executions. This intuition should be taken with a grain of salt: the abstract model (§7.5) uses these rules as well, but its traces do not correspond to interrupted executions as they do here.

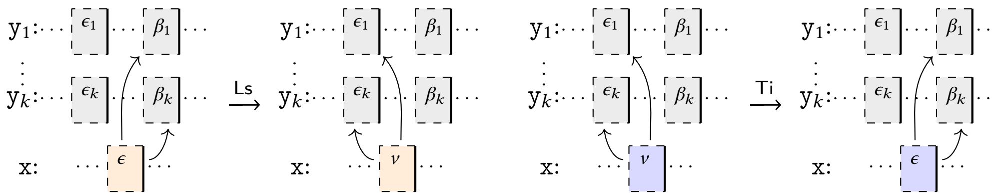


Fig. 12. Schematic depictions of the loosen (left) and tighten (right) closure rules, focusing on a particular memory in the trace. For every i, the messages β_i and ϵ_i may coincide, dovetail, or be apart on $\mathtt{y}_i$'s timeline. **Left:** The environment message ϵ is "loosened" to ν. **Right:** The local message ν is "tightened" to ϵ.

Loosen. When a program relies on a message ϵ from the environment, it relies on the message's view being small enough to allow the behavior that follows. For example, allowing threads that read from ϵ to still read and write certain messages in certain positions on the timeline. In addition, the program relies on the message's timestamp—which is part of the view—to be big enough for it not to be obscured when needed. Figure 12 (left) depicts this rule.

Define the loosen (Ls) closure rule:

$$\text{Assuming } \nu \leq_{\mathtt{vw}} \epsilon,\ \alpha \boxed{\xi\left(\eta \mathbin{\overline{\uplus}} \{\epsilon\}\right)} \omega \xrightarrow{\mathsf{Ls}} \alpha \boxed{\xi\left(\eta \mathbin{\overline{\uplus}} \{\nu\}\right)} \omega \qquad \text{(Loosen)}$$

Here, we use the partial order on messages $\nu \leq_{\mathtt{vw}} \epsilon$ defined by requiring that they may only differ in their timestamps for other memory locations for which ν's timestamps must precede ϵ's:

$$\nu \leq_{\mathtt{vw}} \epsilon \iff \nu.\mathtt{lc} = \epsilon.\mathtt{lc} \wedge \nu.\mathtt{vl} = \epsilon.\mathtt{vl} \wedge \nu.\mathtt{seg} = \epsilon.\mathtt{seg} \wedge \nu.\mathtt{vw} \leq \epsilon.\mathtt{vw}$$

If the source in (Loosen) is a trace, then the target is a trace iff either η is empty or $\nu \hookrightarrow \left(\eta \mathbin{\overline{\uplus}} \{\nu\}\right).\mathtt{o}$.

Intuitively, the source behavior can only use the view in ϵ by incorporating it into its view and the view of its local messages using the max ($\sqcup$) operation on views. Since allowing threads to silently increase their own view does not change the observed behavior, we would still be able to guarantee the same local messages if the environment message has a smaller view. To make this intuition more precise, we outline a simulation argument in the case the program exhibits the source behavior through an interrupted execution that matches the trace exactly. We do not turn it into a formal proof, since the abstract model §7.5 violates this simplifying assumption anyway.

Given an interrupted execution, we can replace an environment message ϵ with a message $\nu \leq_{\mathtt{vw}} \epsilon$ and obtain an interrupted execution of the same program. Whenever a thread with view α loads ϵ via the ReadOnly step in the original interrupted execution, its view becomes $\omega \coloneqq \alpha \sqcup \epsilon.\mathtt{vw}$. In the new interrupted execution, we instead use the Adv rule to compensate for the earlier view in ν, once for every other location ℓ, and forward the view to the message at location ℓ with timestamp ω_ℓ. Then we are able to load ϵ via ReadOnly, since the message has the same timestamp and the thread's view at the location $\epsilon.\mathtt{lc} = \nu.\mathtt{lc}$ hasn't changed during the Adv steps. The RMW-modifier still fails in the new execution because ν and ϵ hold the same value and the decision whether to modify it depends only on the value and the parameters, not the view. Loading via the RMW rule is similar, where the modifier still succeeds with the same modification. We choose the same timestamp for the new message we dovetail to ν, and it inherits the current view: ω. Steps via other rules remain the same.

The operations of $\mathcal{N}$ are {Ls}-closed since the inclusion of a trace never relies on the view of an environment message other than it being dominated $\epsilon.\mathtt{vw} \leq \kappa$ by another view κ. Since $\nu \leq_{\mathtt{vw}} \epsilon$, we have $\nu.\mathtt{vw} \leq \epsilon.\mathtt{vw} \leq \kappa$ and ν and ϵ are otherwise identical, so the trace will be included in the result of the operation.

Expel. The expel (Ex) closure rule replaces an environment message ϵ' with two dovetailing messages ν and ϵ that, together, occupy the same segment. Moreover, ϵ' and ϵ have the same

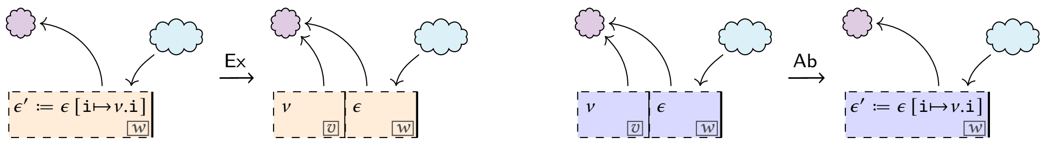


Fig. 13. Schematic depictions of the expel (left) and absorb (right) closure rules, that focus on the segment of the dovetailed messages together with all pointers into and out of them, within a particular memory snapshot. The left cloud represents the subset of the memory that the messages in focus are pointing to, showing their views are the same. The right cloud represents views that point to each of the dovetailing messages, possibly including the initial and final view, as well as other messages. Thus, no view may point to ν. A condition that is not depicted is that all the messages must appear in the same places in the chronicle. **Left:** The environment message ν is "expelled" from the message ϵ', which becomes ϵ. **Right:** The local message ν is "absorbed" into the message ϵ, which becomes ϵ'.

view and value, so we can obtain ϵ' from ϵ by modifying its initial timestamp, $\epsilon' = \epsilon\,[\mathtt{i}{\mapsto}\nu.\mathtt{i}]$, as Figure 13 (left) depicts. This rule ensures that the value v is available at the same timestamp with the same carried view, and that no more of the timeline is occupied. Formally:

$$\text{Assuming } \nu \prec\!\!\subset \epsilon,\ \alpha\boxed{\xi\left(\eta \mathbin{\overline{\uplus}} \{\epsilon\,[\mathtt{i}{\mapsto}\nu.\mathtt{i}]\}\right)}\omega \xrightarrow{\mathsf{Ex}} \alpha\boxed{\xi\left(\eta \mathbin{\overline{\uplus}} \{\nu, \epsilon\}\right)}\omega \qquad \text{(Expel)}$$

Here, $\nu \prec\!\!\subset \epsilon$ is the *monotone dovetailing* relation, defined to hold when ϵ dovetails after ν and ϵ's view dominates ν's view:

$$\nu \prec\!\!\subset \epsilon \iff \nu.\mathtt{lc} = \epsilon.\mathtt{lc} \wedge \nu.\mathtt{t} = \epsilon.\mathtt{i} \wedge \nu.\mathtt{vw} \leq \epsilon.\mathtt{vw}$$

The final condition means the rule is more relaxed than Figure 13 depicts, where $\nu.\mathtt{vw} = \epsilon.\mathtt{vw}$. This difference is immaterial to the model, because we can recover the relaxed version by applying loosen after the strict version.

As was the case for loosen, if the source in (Expel) is a trace, then the target is a trace iff either η is empty or $\nu \hookrightarrow \left(\eta \mathbin{\overline{\uplus}} \{\nu\}\right).\mathtt{o}$.

To justify the rule for interrupted executions, suppose ϵ' is an environment message in an interrupted execution. By replacing ϵ' with ν and ϵ, we obtain another interrupted execution, in which the environment added these two messages. Throughout the interrupted execution, no view ever points to ν, as if ν was not there other than making its segment unavailable.

The operations of $\mathcal{N}$ are $\{\mathsf{Ex}\}$-closed since they never rely on the absence of messages except for the availability of segments, which this rule preserves.

Condense. In the condense (Cn) closure rule, the source behavior may include an environment message ϵ dovetailing after a message ν that carries the same value and view. The target behavior removes ϵ, and modifies ν to a message ν' that occupies the same segment as the two messages combined, as Figure 14 (left) depicts.

To formally capture how the views in the trace change in this rule, we define *pulling* a view κ *along* a message ϵ in location ℓ to be the view $\kappa\,[\uparrow\epsilon]$, which is equal to κ unless the timestamp κ_ℓ is ϵ's initial timestamp, in which case it becomes ϵ's final timestamp (depicted on the right):

$$\begin{aligned} \ell &\coloneqq \epsilon.\mathtt{lc} \\ i &\coloneqq \epsilon.\mathtt{i} \\ t &\coloneqq \epsilon.\mathtt{t} \end{aligned} \qquad \kappa\,[\uparrow\epsilon] \coloneqq \begin{cases} \kappa_\ell = i: & \kappa[\ell{\mapsto}t] \\ \text{otherwise:} & \kappa \end{cases} \qquad \begin{matrix} \kappa_\ell & \kappa[\uparrow\epsilon]_\ell \\ \downarrow & \downarrow \\ (i, & t] \end{matrix}$$

We extend the pulling operation to messages, memories, chronicles, (pre-)traces, and view trees, by pulling the view associated with these objects. In particular, if ϵ dovetails after ν, then pulling ν along ϵ merges them into one contiguous message $\nu\,[\uparrow\epsilon]$ which has ν's view and value; and $\kappa\,[\uparrow\epsilon]$ points to ϵ iff κ points to ν or ϵ.

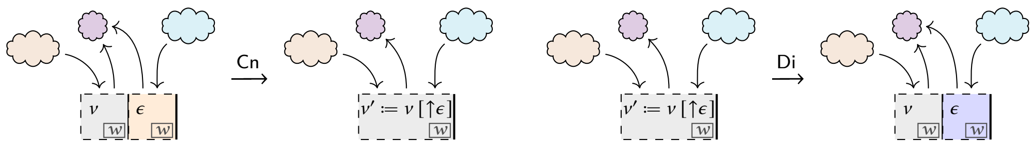


Fig. 14. Schematic depictions of the condense (left) and dilute (right) closure rules, in the style of Figure 13. A condition that is not depicted is that ν and ν' must appear in the same places in the chronicle, and ϵ may not appear before them. The views that point to ν' in the source can point either to ν or to ϵ in the target. **Left:** The message ν turns into ν' by "condensing" the environment message ϵ. **Right:** The message ν' turns into ν by "diluting" out the local message ϵ.

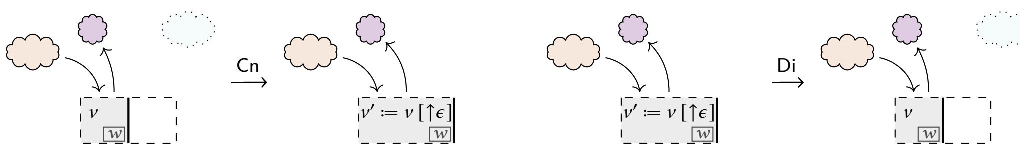


Fig. 15. Schematic depictions of the condense (left) and dilute (right) closure rules as in Figure 14, focusing this time on a memory without ϵ. **Left:** Since ϵ is to appear as an environment message in the chronicle, it can appear since the opening memory, not appear even in the closing memory, or somewhere in between. **Right:** Since ϵ is to appear as a local message, it cannot appear in the opening memory, and must appear in the closing memory.

The closure rule, formally:

$$\text{Assuming } \nu \mathrel{-\!\!\Subset} \epsilon,\ \alpha\boxed{\xi\left(\eta \mathbin{\overline{\uplus}} \{\nu, \epsilon\}\right)}\omega \xrightarrow{\mathsf{Cn}} \left(\alpha\boxed{\xi\left(\eta \mathbin{\overline{\uplus}} \{\nu\}\right)}\omega\right)[\uparrow\epsilon] \qquad \text{(Condense)}$$

Here, $\nu \mathrel{-\!\!\Subset} \epsilon$ is the monotone *repetitive* dovetailing relation, defined to hold when ϵ dovetails monotonically after ν and the messages have the same value:

$$\nu \mathrel{-\!\!\Subset} \epsilon \iff \nu \mathrel{-\!\!\subset} \epsilon \wedge \nu.\mathtt{vl} = \epsilon.\mathtt{vl}$$

As was the case for expel, relaxing the condition that the views must be equal-up-to-timestamp, as Figure 14 (left) depicts, is admissible; this time by applying loosen before the strict version.

The decomposition of the chronicle in the rule determines where ϵ first appears (if at all), but ν can first appear earlier. This situation is depicted in Figure 15 (left).

When η is empty the target differs from the source iff there is a message at $\epsilon.\mathtt{i} = \nu.\mathtt{t}$. In this case, assuming the source is a trace, for the target to be a trace $\epsilon.\mathtt{seg}$ must be available, otherwise there will be a memory that is not scattered. If $\epsilon.\mathtt{seg}$ is available, then the target will be a trace, because pulling along a free segment retains the well-formed memory properties. For example, pointing downwards is preserved due to the following lemma:

Lemma 7.6. $\forall \epsilon \in \mathsf{Msg}\,\forall \kappa, \sigma \in \mathsf{View}.\ \kappa_{\epsilon.\mathtt{lc}}, \sigma_{\epsilon.\mathtt{lc}} \notin \epsilon.\mathtt{seg} \setminus \{\epsilon.\mathtt{t}\} \implies \kappa \leq \sigma \implies \kappa\,[\uparrow\epsilon] \leq \sigma\,[\uparrow\epsilon]$.

To summarize, if the source in (Condense) is a trace, then the target is a trace iff either ξ is empty, $\epsilon.\mathtt{i} \notin \xi.\mathtt{c}.\mathtt{t}$, or $\epsilon.\mathtt{seg} \cap \bigcup \xi.\mathtt{c}.\mathtt{seg} = \emptyset$.

If we have an interrupted execution with two messages ν and ϵ as in condense, we will also have an interrupted execution without the environment message ϵ, and with ν' instead of ν. The new interrupted execution uses ν' whenever the original uses either ν or ϵ.

The operations of $\mathcal{N}$ are $\{\mathsf{Cn}\}$-closed. This fact is harder to demonstrate compared to the previous rules. Considerations involving the value available to load, and the segment available to store, are similar. If a message dovetailed after ϵ in the source, it dovetails after ν' in the target. Thus, if a message was added due to an RMW in the source, the condition to dovetail after a message that holds the loaded value is still met in the target. There are also new considerations due to the closure rule affecting the entire trace rather than just one or two messages. For instance, to show that bind ($\rangle\!\rangle\!=^{\mathcal{N}}$) preserves the rule, we replace an application of condense after binding the traces with applications of condense (with the same messages) on each of the traces before binding. This replacement is subtle because the delimiting views change, and thus the condition $\kappa \leq \sigma$ imposed on binding the traces changes to $\kappa\,[\uparrow\epsilon] \leq \sigma\,[\uparrow\epsilon]$. The condition still holds due to Lemma 7.6 since neither κ nor σ point into the *interior* of $\epsilon.\mathtt{seg}$, because no message has a timestamp there. This insight resolves similar subtleties for the other $\mathcal{N}$-constructs.

7.4 Concrete Denotations

Brookes [13] pioneered two closure rules to make denotations abstract and support desired program transformations: stuttering and mumbling. To define our next model, we adapt these closure rules to our setting, as well as add two additional ones: $\mathfrak{c} \coloneqq \{\mathsf{St}, \mathsf{Mu}, \mathsf{Fw}, \mathsf{Rw}\}$. We denote the union of closure-rule sets by juxtaposition, e.g. $\mathfrak{gc} \coloneqq \mathfrak{g} \cup \mathfrak{c}$. We denote by C the $\mathfrak{gc}$-monad structure, and call this model the *concrete model*. Like the generating model, C still maintains a close correspondence to the operational semantics $\mathrm{RA}_{\leq}$. However, C is a monad, a crucial element in the proof of the adequacy theorem.

Proposition 7.7. *C is a monad.*

We describe each of the $\mathfrak{c}$-closure rules below, providing justifications from two viewpoints: rely/guarantee intuitions, and correspondence to interrupted executions. The rely/guarantee intuitions will serve us better in the abstract model (§7.5), which also uses these rules, because its traces do not correspond to interrupted executions as they do here.

We motivate each closure rule $\mathsf{x} \in \mathfrak{c}$ with an example of a transformation $M \twoheadrightarrow N$ supported thanks to that rule. For simplicity and emphasis, we demonstrate this with $[\![M]\!]^{\mathsf{c}\,\star}_{\mathcal{G}} \supseteq [\![N]\!]^{\mathsf{c}}_{\mathcal{G}}$ where $\mathsf{x} \in \star \subseteq \mathfrak{c}$, rather than $[\![M]\!]^{\mathsf{c}}_{C} \supseteq [\![N]\!]^{\mathsf{c}}_{C}$ (which will also hold).

Stutter. A program can always make the same memory guarantees on which it relies. This is captured by stutter (St), which inserts a transition with equal components somewhere:

$$\alpha\boxed{\xi\eta}\omega \xrightarrow{\mathsf{St}} \alpha\boxed{\xi\,\langle\mu,\mu\rangle\,\eta}\omega \qquad \text{(Stutter)}$$

If the source of (Stutter) is a trace, then the target is a trace iff μ is well-formed and the initial view points to the opening memory: $\alpha \rightarrowtail \mu$ (this may fail to hold if ξ is empty).

We can also understand stutter using interrupted executions. Given an interrupted execution, a sequence of 0 steps $\langle T,\mu\rangle, M \rightsquigarrow^{*} \langle T,\mu\rangle, M$ can be inserted anywhere as long as $\langle T,\mu\rangle$ is well-formed and μ contains the previous memory if there is one, and is contained in the subsequent memory if there is one.

As a concrete (contrived) example, consider the transformation $\langle\rangle\,;\langle\rangle \twoheadrightarrow \langle\rangle\,;\langle\rangle\,;\langle\rangle$. We can use stutter to validate it. Indeed, $[\![\langle\rangle\,;\langle\rangle]\!]^{\mathsf{c}\,\{\mathsf{St}\}}_{\mathcal{G}} \supseteq [\![\langle\rangle\,;\langle\rangle\,;\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}}$, even though $[\![\langle\rangle\,;\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}} \not\supseteq [\![\langle\rangle\,;\langle\rangle\,;\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}}$.

Mumble. A program can omit a guarantee and rely on that guarantee internally. This is captured by mumble (Mu), which combines transitions with the same memory at their common edge:

$$\alpha\boxed{\xi\,\langle\mu,\rho\rangle\,\langle\rho,\theta\rangle\,\eta}\omega \xrightarrow{\mathsf{Mu}} \alpha\boxed{\xi\,\langle\mu,\theta\rangle\,\eta}\omega \qquad \text{(Mumble)}$$

If the source in (Mumble) is a trace then so is its target.

We can also understand mumble using interrupted executions. If we have an interrupted execution of the form $\ldots\langle T,\mu\rangle, M \rightsquigarrow^{*}_{\mathrm{RA}_{\leq}} \langle R,\rho\rangle, N \quad \langle R,\rho\rangle, N \rightsquigarrow^{*}_{\mathrm{RA}_{\leq}} \langle H,\theta\rangle, K\ldots$ that is compatible with the source trace, then the shorter interrupted execution $\ldots\langle T,\mu\rangle, M \rightsquigarrow^{*}_{\mathrm{RA}_{\leq}} \langle H,\theta\rangle, K\ldots$ is compatible with the target trace.

Fig. 16. Schematic depictions of the rewind and forward closure rules, focusing on a single location, where the initial/final view points to ν before and points to ϵ after. The messages ν and ϵ may coincide, dovetail, or be apart. **Left:** The final view ω is "forwarded" to ω'. **Right:** The initial view α is "rewound" to α'.

As a concrete example, we use mumble to validate the transformation $\ell? \,;M \twoheadrightarrow M$, which also demonstrates the importance of the internalized invariants from §6, i.e. the use of traces rather than pre-traces. Indeed, $\alpha\boxed{\langle\mu,\rho\rangle\,\xi}\omega \mathrel{.\!\!\because} r \in [\![M]\!]^{\mathsf{c}}_{\mathcal{G}}$ is a trace, so $\alpha \rightarrowtail \mu$. Therefore, there is some $\alpha\boxed{\langle\mu,\mu\rangle}\alpha \mathrel{.\!\!\because} v \in [\![\ell?]\!]^{\mathsf{c}}_{\mathcal{G}}$. So $\alpha\boxed{\langle\mu,\mu\rangle\,\langle\mu,\rho\rangle\,\xi}\omega \mathrel{.\!\!\because} r \in [\![\ell?\,;M]\!]^{\mathsf{c}}_{\mathcal{G}}$. Thus $\alpha\boxed{\langle\mu,\rho\rangle\,\xi}\omega \mathrel{.\!\!\because} r \in [\![\ell?\,;M]\!]^{\mathsf{c}\,\{\mathsf{Mu}\}}_{\mathcal{G}}$.

Forward. If a program fragment can operate and guarantee a certain set of messages remain visible, it can operate in the same way and guarantee a subset of these messages remain visible. The final view serves to guarantee revealed messages to subsequent computation, so we reflect this fact by forward (Fw), which increases the final view:

$$\text{Assuming } \kappa \leq \omega,\ \alpha\boxed{\xi}\kappa \xrightarrow{\mathsf{Fw}} \alpha\boxed{\xi}\omega \qquad \text{(Forward)}$$

Figure 16 (left) depicts the rule. If the source of (Forward) is a trace, then the target is a trace iff the final view points downwards into the closing memory: $\omega \hookrightarrow \xi.\mathsf{c}$.

We can also understand forward using interrupted executions. If we have an interrupted execution of the form ... $\langle T,\mu\rangle, M \rightsquigarrow^{*}_{\mathrm{RA}_{\leq}} \langle R,\rho\rangle, N$, we can append Adv steps to the final sequence of steps to obtain ... $\langle T,\mu\rangle, M \rightsquigarrow^{*}_{\mathrm{RA}_{\leq}} \langle R',\rho\rangle, N$, where $R \leq R' \hookrightarrow \rho$.

As a concrete example, we use stutter and forward to validate the transformation $\langle\rangle \twoheadrightarrow \ell?;\langle\rangle$. To show that $[\![\langle\rangle]\!]^{\mathsf{c}\,\{\mathsf{St},\mathsf{Fw}\}}_{\mathcal{G}} \supseteq [\![\ell?\,;\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}}$, we first use stutter to compensate for the additional transition and then use forward to compensate for the difference between the initial and final views.

Rewind. If a program fragment can operate by relying on a certain set of visible messages, it can operate in the same way by relying on a superset of these messages being visible. The initial view serves to guarantee revealed messages from previous computation, so we reflect this fact by rewind (Rw), which decreases the initial view:

$$\text{Assuming } \alpha \leq \kappa,\ \kappa\boxed{\xi}\omega \xrightarrow{\mathsf{Rw}} \alpha\boxed{\xi}\omega \qquad \text{(Rewind)}$$

Figure 16 (right) depicts this rule. If the source of (Rewind) is a trace, then the target is a trace iff the initial view points downwards into the opening memory: $\alpha \hookrightarrow \xi.\mathsf{o}$.

We can also understand rewind using interrupted executions, as we did for forward. Instead of appending Adv steps to the final sequence, we prepend Adv steps to the initial sequence.

As a concrete example, we use rewind and stutter to validate the transformation $M \twoheadrightarrow \langle\rangle\,;M$.

7.5 Abstract Denotations

Finally, we define the *abstract model*, $\mathcal{A}$ as the $\mathfrak{gca}$-monad structure, where $\mathfrak{a} \coloneqq \{\mathsf{Ti}, \mathsf{Ab}, \mathsf{Di}\}$ are closure rules presented below. This model fulfills the basic requirement of a monadic model:

Proposition 7.8. *$\mathcal{A}$ is a monad.*

By including the additional closure rules of $\mathfrak{a}$ we depart from the operational interpretations that we have used for the previous rules. This departure allows us to obtain the abstraction that the concrete model lacks. We took a parsimonious approach, only proposing rules that we need to justify program transformations that the RA model is expected to validate. With each closure rule, we present a program transformations whose validation uses that particular rule, though other $\mathfrak{gc}$-closures are often required as well.

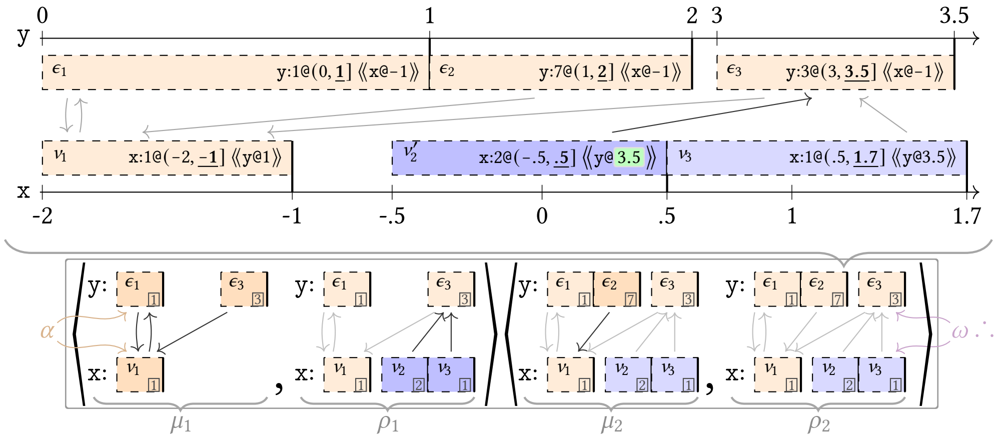


Fig. 17. A possible result from rewriting the trace from Figure 1 using tighten. Since ν_2 is local in the trace from Figure 1, tighten can advance its view to point to ϵ_3 instead of ϵ_1. The same replacement is applied throughout the trace's sequence, not just the closing memory.

Tighten. The role of the view that a message carries, other than providing the timestamp, is to constrain the loading thread by increasing its view when it loads the message. Considering a local message ν, its view serves to guarantee that loading it would not obscure any message within a certain portion of the memory. Therefore, replacing ν by ϵ that only differs in its view, where $\nu \le_{\mathtt{vw}} \epsilon$, as Figure 12 (right) depicts, means that only a sub-portion of the memory is guaranteed not to become obscured by loading the message, and keeps everything else the same. This replacement of the carried view is the effect of the tighten (Ti) closure rule. Formally:

$$\text{Assuming } \nu \le_{\mathtt{vw}} \epsilon,\ \alpha \boxed{\xi \langle \mu, \rho \uplus \{\nu\} \rangle\, \eta \,\overline{\uplus}\, \{\nu\}}\, \omega \xrightarrow{\mathsf{Ti}} \alpha \boxed{\xi \langle \mu, \rho \uplus \{\epsilon\} \rangle\, \eta \,\overline{\uplus}\, \{\epsilon\}}\, \omega \qquad \text{(Tighten)}$$

See Figure 17 for a concrete example.

As a concrete benefit of tighten, consider the write-read-reordering transformation:

$$\underbrace{\ell \coloneqq v\,;\mathbf{let}\ a = \langle \ell'?, \ell'? \rangle\ \mathbf{in}\ a}_{M} \twoheadrightarrow \underbrace{\mathbf{let}\ a = \langle \ell'?, \ell'? \rangle\ \mathbf{in}\ \ell \coloneqq v\,;a}_{N} \qquad \text{where } \ell \neq \ell'$$

This transformation is not valid under SC, but it is valid under RA. The reason it is valid is non-trivial. For example, consider an execution of M that loads two different values from distinct messages ν and ϵ, which are the only messages at ℓ'. The message M stores afterwards must carry a view κ with the increased ℓ'-timestamp: $\kappa_{\ell'} = \epsilon.\mathtt{t} > \nu.\mathtt{t}$. The term N cannot simulate this exactly: even if N advances the view, with the RA$_{\le}$-rule Adv, to store the same view κ as M, the first message N needs to load ν will be obscured.

We validate this transformation using the denotational semantics: $[\![M]\!]_C^{\mathsf{c}\,\{\mathsf{Ti}\}} \supseteq [\![N]\!]_C^{\mathsf{c}}$.

Absorb. Applying absorb (Ab) removes a local message ν and decreases the initial timestamp of a local message ϵ dovetailing after ν with the same view, such that the resulting ϵ' covers the segment of ν. Figure 13 (right) depicts this. In this way, the rule weakens its memory guarantee to the environment because it has less messages available to load from, without strengthening the guarantee by way of making any more of the location's timeline available. No view can point to ν before applying this rule, otherwise the resulting pre-trace would not be a trace. The rule is formally specified as follows, where we abbreviate by denoting $\epsilon_{\mathtt{i}}^{t} \coloneqq \epsilon\,[\mathtt{i} \mapsto t]$:

$$\text{Assuming } \nu \mathrel{-\!\!\subset} \epsilon,\ \alpha \boxed{\xi \langle \mu, \rho \uplus \{\nu, \epsilon\} \rangle\, \eta \,\overline{\uplus}\, \{\nu, \epsilon\}}\, \omega \xrightarrow{\mathsf{Ab}} \alpha \boxed{\xi \left\langle \mu, \rho \uplus \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\} \right\rangle \eta \,\overline{\uplus}\, \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\}}\, \omega \qquad \text{(Absorb)}$$

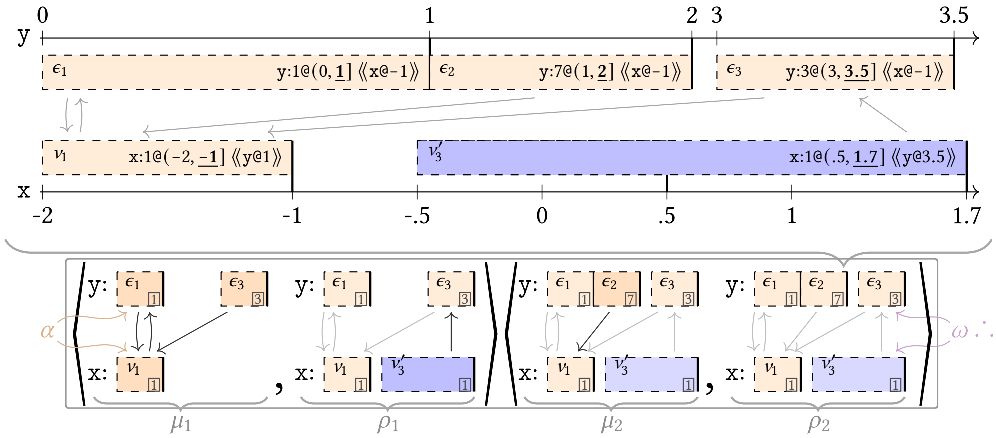


Fig. 18. A trace resulting from absorb-rewriting the trace from Figure 17. The dovetailed messages ν_2 and ν_3 are local in the trace from Figure 1, added within the same transition, so by absorb-rewriting they can be replaced by ν_3' obtained by stretching ν_3's segment to cover ν_2's segment.

See Figure 18 for a concrete example. As in expel, we relax the condition of equal views, admissible due to tighten.

The transformation $\ell := w ; \ell := v \twoheadrightarrow \ell := v$ is a concrete example where this rule is useful, in which we use absorb to compensate for the extra message on the left: $[\![\ell := w \,;\, \ell := v]\!]_C^{\mathsf{c}\,\{\mathsf{Ab}\}} \supseteq [\![\ell := v]\!]_C^{\mathsf{c}}$. In more detail, consider a trace on the right $\tau \in [\![\ell := v]\!]_C^{\mathsf{c}}$ with local message β. Pick: a timestamp t from the interior of $\beta.\mathtt{seg}$, a trace $\pi \in [\![\ell := w]\!]_C^{\mathsf{c}}$ with local message that has the segment $(\beta.\mathtt{i}, t]$ and a trace $\varrho \in [\![\ell := v]\!]_C^{\mathsf{c}}$ with local message that has the segment $(t, \beta.\mathtt{t}]$. After binding to obtain the sequencing of τ with π in $[\![\ell := w \,;\, \ell := v]\!]_C^{\mathsf{c}}$, use mumble to combine the transitions, then absorb to replace these two messages with β, resulting in τ.

Dilute. Formally, the dilute (Di) rule is specified as follows:

$$\text{Assuming } \nu \mathrel{-\!\!\in} \epsilon,\ \left(\alpha \boxed{\xi \left\langle \mu, \rho \uplus \{\nu\} \right\rangle \eta \,\overline{\uplus}\, \{\nu\}} \omega\right) [\uparrow\epsilon] \xrightarrow{\mathsf{Di}} \alpha \boxed{\xi \left\langle \mu, \rho \uplus \{\nu, \epsilon\} \right\rangle \eta \,\overline{\uplus}\, \{\nu, \epsilon\}} \omega \quad \text{(Dilute)}$$

See Figure 19 for a concrete example.

We restrict to the case that $\nu.\mathtt{vw} = \epsilon.\mathtt{vw}$ when explaining the rule. The rest can be seen as a formal extension which is admissible in the presence of tighten, as with condense and loosen.

Figure 14 (right) depicts how, in memories with ϵ, views that point to $\nu\,[\uparrow\epsilon]$ in the source point to either ν or ϵ in the target. Without ϵ, the views all point to ν, as Figure 15 (right) depicts.

In the source behavior of this closure rule the program relies on and guarantees $\nu\,[\uparrow\epsilon]$ whenever it appears in the first or second component of a transition, respectively. In the target behavior the program relies on and guarantees ν at the same stages, and the program relies on the segment of ϵ to be unoccupied until the program guarantees ϵ itself. When the segment of ϵ is unoccupied, the difference in timestamp is inconsequential because both messages are ordered the same with respect to the other messages on the timeline. The addition of ϵ to the segment is also inconsequential, insofar as offering the same value and view; and it fills up the remaining unoccupied portion of the segment of $\nu\,[\uparrow\epsilon]$, ensuring it is unavailable in the target behavior as well.

As a concrete example of this rule in use, consider the transformation $\ell? \twoheadrightarrow \mathrm{FAA}\,(\ell, 0)$. A trace from the target $\tau \in [\![\mathrm{FAA}\,(\ell, 0)]\!]_C^{\mathsf{c}}$ has a local message ϵ that dovetails after an existing environment message ν. There is a matching trace $\pi \in [\![\ell?]\!]_C^{\mathsf{c}}$ in the source without that local message. By closure under $\mathfrak{g}$, we can apply the condense closure rule on π, pulling ν along ϵ, to obtain a trace $\varrho \in [\![\ell?]\!]_C^{\mathsf{c}}$

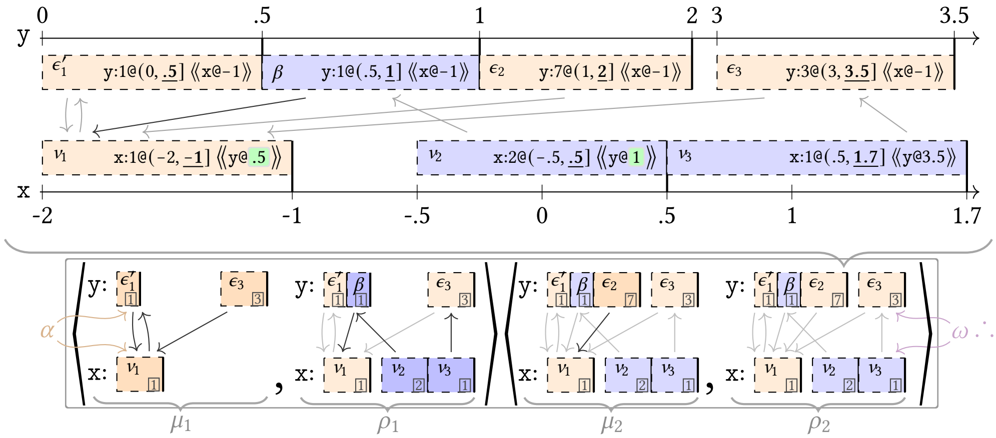


Fig. 19. A possible result from rewriting of the trace from Figure 1 using dilute. The message ϵ_1 from Figure 1 was replaced with ϵ_1', with the same value 1. The local message β—which takes up the rest of the missing space left behind by ϵ_1—always appears with ϵ_1', dovetailing after it and carrying the same value. The message ϵ_2, that used to dovetail after ϵ_1, now dovetails after β.

with the environment message $\nu\ [\uparrow\epsilon]$ (condense applies even when pulling along a message ϵ that is not in the trace). Then, we can apply dilute to ϱ to add the local ϵ, resulting in τ.

8 Metatheory

The difference between the different monad structures from §7 are due to the abstraction afforded to them by the closure rules. Ultimately, it is the monad $\mathcal{A}$ that we are interested in, as it is the one over which we define satisfactory denotational semantics. To prove the results that justify this, we first relate the different monad structures using properties of the closure rules and their interactions (§8.1). Then, focusing on $\mathcal{A}$, we prove (directional) compositionality (§8.2) and soundness (§8.3). These results facilitate the main result: (directional) adequacy (§8.4). Finally, we exhibit the abstraction of the denotational semantics with various transformations it supports (§8.5).

8.1 Rewrite Castling

A complicating aspect of these trace models is how intricately rewrites between traces interact. For example, an application of forward may only be possible after using stutter to add a transition to the end of the chronicle with stutter, in which the messages that the final view is intended to point to exist. So given a sequence of rewrites between traces $\tau \xrightarrow{\mathsf{St}} \pi \xrightarrow{\mathsf{Fw}} \varrho$, there may be no trace π' such that $\tau \xrightarrow{\mathsf{Fw}} \pi' \xrightarrow{\mathsf{St}} \varrho$.

Example 8.1. Let $\alpha \coloneqq \langle\!\langle \mathtt{x@0} \ ; \ \mathtt{y@0} \rangle\!\rangle$ and $\omega \coloneqq \langle\!\langle \mathtt{x@3} \ ; \ \mathtt{y@3} \rangle\!\rangle$ be views. Using the simplified notation for memories from §6, let:

$$\mu \coloneqq \begin{Bmatrix} \mathtt{y} : 5@(-1,\underline{0}]\ \langle\!\langle 0 \rangle\!\rangle,\ 7@(1,\underline{3}]\ \langle\!\langle 0 \rangle\!\rangle \\ \mathtt{x} : 6@(-1,\underline{0}]\ \langle\!\langle 0 \rangle\!\rangle \end{Bmatrix}$$

$$\rho \coloneqq \begin{Bmatrix} \mathtt{y} : 5@(-1,\underline{0}]\ \langle\!\langle 0 \rangle\!\rangle,\ 7@(1,\underline{3}]\ \langle\!\langle 0 \rangle\!\rangle \\ \mathtt{x} : 6@(-1,\underline{0}]\ \langle\!\langle 0 \rangle\!\rangle,\ 8@(1,\underline{3}]\ \langle\!\langle 3 \rangle\!\rangle \end{Bmatrix} = \mu \uplus \left\{ \mathtt{x}{:}8@(1,\underline{3}]\ \langle\!\langle \mathtt{y@3} \rangle\!\rangle \right\}$$

Consider the following sequence of trace rewrites:

$$\underbrace{\alpha\,\boxed{\langle\mu,\mu\rangle}\,\alpha \mathrel{.\cdot.} 9}_{\tau} \xrightarrow{\mathsf{St}} \underbrace{\alpha\,\boxed{\langle\mu,\mu\rangle\,\langle\rho,\rho\rangle}\,\alpha \mathrel{.\cdot.} 9}_{\pi} \xrightarrow{\mathsf{Fw}} \underbrace{\alpha\,\boxed{\langle\mu,\mu\rangle\,\langle\rho,\rho\rangle}\,\omega \mathrel{.\cdot.} 9}_{\varrho}$$

We also have the sequence:

$$\underbrace{\alpha\,\boxed{\langle\mu,\mu\rangle}\,\alpha \mathrel{.\cdot.} 9}_{\tau} \xrightarrow{\mathsf{Fw}} \underbrace{\alpha\,\boxed{\langle\mu,\mu\rangle}\,\omega \mathrel{.\cdot.} 9}_{\pi'} \xrightarrow{\mathsf{St}} \underbrace{\alpha\,\boxed{\langle\mu,\mu\rangle\,\langle\rho,\rho\rangle}\,\omega \mathrel{.\cdot.} 9}_{\varrho}$$

but π' is not a trace because $\omega \not\rightarrowtail \mu$, so this is not a sequence of trace rewrites.

Other closure-rule pairs permit this kind of rearrangement. For example, loosen and stutter: if $\tau \xrightarrow{\mathsf{Ls}} \pi \xrightarrow{\mathsf{St}} \varrho$ is a sequence of trace rewrites, then there exists a trace π' such that $\tau \xrightarrow{\mathsf{St}} \pi' \xrightarrow{\mathsf{Ls}} \varrho$.

Example 8.2. Keeping α, ω, μ, and ρ from Example 8.1, let:

$$\theta := \begin{Bmatrix} \mathtt{y} : 5@(-1,\underline{0}]\,\langle\!\langle 0\rangle\!\rangle,\ 7@(1,\underline{3}]\,\langle\!\langle 0\rangle\!\rangle \\ \mathtt{x} : 6@(-1,\underline{0}]\,\langle\!\langle 0\rangle\!\rangle,\ 8@(1,\underline{3}]\,\langle\!\langle 0\rangle\!\rangle \end{Bmatrix} = \mu \uplus \left\{\mathtt{x}{:}8@(1,\underline{3}]\,\langle\!\langle \mathtt{y}@0\rangle\!\rangle\right\}$$

Consider the following sequence of trace rewrites:

$$\underbrace{\alpha\,\boxed{\langle\rho,\rho\rangle}\,\omega \mathrel{.\cdot.} 9}_{\tau} \xrightarrow{\mathsf{Ls}} \underbrace{\alpha\,\boxed{\langle\theta,\theta\rangle}\,\omega \mathrel{.\cdot.} 9}_{\pi} \xrightarrow{\mathsf{St}} \underbrace{\alpha\,\boxed{\langle\theta,\theta\rangle\,\langle\theta,\theta\rangle}\,\omega \mathrel{.\cdot.} 9}_{\varrho}$$

We also have the sequence:

$$\underbrace{\alpha\,\boxed{\langle\rho,\rho\rangle}\,\omega \mathrel{.\cdot.} 9}_{\tau} \xrightarrow{\mathsf{St}} \underbrace{\alpha\,\boxed{\langle\rho,\rho\rangle\,\langle\rho,\rho\rangle}\,\omega \mathrel{.\cdot.} 9}_{\pi'} \xrightarrow{\mathsf{Ls}} \underbrace{\alpha\,\boxed{\langle\theta,\theta\rangle\,\langle\theta,\theta\rangle}\,\omega \mathrel{.\cdot.} 9}_{\varrho}$$

where π' is a trace, so this is a sequence of trace rewrites.

More generally, every sequence of rewrites can be rearranged such that $\mathfrak{g}$-rewrites appear first, then $\mathfrak{c}$-rewrites, and finally $\mathfrak{a}$-rewrites. This property will play a pivotal rule in our development of the metatheory. It is an immediate consequence of the following lemma. We write $\mathsf{x} \rightleftarrows \mathsf{y}$ when $\xrightarrow{\mathsf{x}}\xrightarrow{\mathsf{y}} \subseteq \xrightarrow{\mathsf{y}}\xrightarrow{\mathsf{x}}$, where the rewrites $\xrightarrow{\mathsf{x}}$ and $\xrightarrow{\mathsf{y}}$ are restricted to traces.

Lemma 8.3 (Rewrite Castling). *If* $\mathsf{x} \in \mathfrak{a}$ *and* $\mathsf{y} \in \mathfrak{g}\mathfrak{c}$, *or* $\mathsf{x} \in \mathfrak{c}\mathfrak{a}$ *and* $\mathsf{y} \in \mathfrak{g}$, *then* $\mathsf{x} \rightleftarrows \mathsf{y}$.

Proof. The proof proceeds by case analysis on x and y, each encapsulated in diagram(s) such as the two in Figure 20. The detailed proof, including all of the diagrams, is in §F. Specifically, see diagrams 31 and 42 for larger and more detailed versions of those in Figure 20.

Each diagram shows the assumed rewrite sequence $\tau \xrightarrow{\mathsf{x}} \pi \xrightarrow{\mathsf{y}} \varrho$ on the left, with the conditions that are known because they were required for the rewrites to be applicable. For example, Figure 20 (left) shows that $\nu \mathrel{-\!\!\subset} \epsilon$ holds due to the assumed x=Ab-rewrite, and $\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}} \mathrel{-\!\!\Subset} \hat{\epsilon}$ holds due to the assumed y=Cn-rewrite. The deduced sequence $\tau \xrightarrow{\mathsf{y}} \pi' \xrightarrow{\mathsf{x}} \varrho$ on the right, with the conditions that need to hold for the rewrites to be applicable. For example, Figure 20 (left) shows that $\epsilon \mathrel{-\!\!\Subset} \hat{\epsilon}$ needs to hold to apply the y=Cn-rewrite, and $\nu\,[\uparrow\hat{\epsilon}] \mathrel{-\!\!\subset} \epsilon\,[\uparrow\hat{\epsilon}]$ needs to hold to apply the x=Ab-rewrite.

The conditions are enough to show that the closure rules apply for pre-traces, but for the sequence to be valid, we must verify that π' is a trace. This is done by inferring from the fact that it was x-rewritten from the trace τ, and y-rewritten to the trace π, using the conditions we have collected as we presented the closure rules in §7.3-7.5. For example, in Figure 20 (left), we need to show

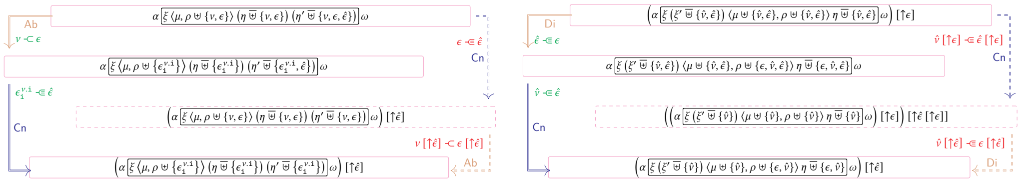


Fig. 20. Two cases from the proof of Rewrite Castling in which "active" messages overlap.

that the pre-trace $\left(\alpha \boxed{\xi_1 \left(\eta' \uplus \{\nu, \epsilon\}\right)} \omega\right) [\uparrow\hat{\epsilon}]$ is a trace, where $\xi_1 \coloneqq \xi \left\langle \mu, \rho \uplus \{\nu, \epsilon\}\right\rangle \left(\eta \uplus \{\nu, \epsilon\}\right)$. Since it is the target of an application of a Cn-rewrite, by the condition for condense from §7.3 (also summarized in Lemma F.1), it is a trace iff either ξ_1 is empty, $\hat{\epsilon}.\mathtt{i} \notin \xi_1.\mathtt{c}.\mathtt{t}$, or $\hat{\epsilon}.\mathtt{seg} \cap \bigcup \xi_1.\mathtt{c}.\mathtt{seg} = \emptyset$. We show the final one of these criteria. We use the fact that $\left(\alpha \boxed{\xi_2 \left(\eta' \uplus \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\}\right)} \omega\right) [\uparrow\hat{\epsilon}]$, where $\xi_2 \coloneqq \xi \left\langle \mu, \rho \uplus \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\}\right\rangle \left(\eta \uplus \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\}\right)$, is by assumption a trace and the target of a Cn-rewrite. Therefore, either ξ_2 is empty, $\hat{\epsilon}.\mathtt{i} \notin \xi_2.\mathtt{c}.\mathtt{t}$, or $\hat{\epsilon}.\mathtt{seg} \cap \bigcup \xi_2.\mathtt{c}.\mathtt{seg} = \emptyset$. By the transition $\left\langle \mu, \rho \uplus \left\{\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}\right\}\right\rangle$ appears in ξ_2, it is non-empty, and $\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}} \in \xi_2.\mathtt{c}$. Therefore, $\hat{\epsilon}.\mathtt{seg} \cap \bigcup \xi_2.\mathtt{c}.\mathtt{seg} = \emptyset$. Since $\xi_2.\mathtt{c}$ differs from $\xi_1.\mathtt{c}$ by messages that occupy the same total segment, $\bigcup \xi_2.\mathtt{c}.\mathtt{seg} = \bigcup \xi_1.\mathtt{c}.\mathtt{seg}$, and we are done: $\hat{\epsilon}.\mathtt{seg} \cap \bigcup \xi_1.\mathtt{c}.\mathtt{seg} = \emptyset$.

The cases in Figure 20 are among the more interesting cases in which the activities of x and y overlap. The left diagram shows a sub-case of Ab ⇄ Cn in which the absorbing message (ϵ) also serves as the condensing message. On the right, a sub-case of Di ⇄ Cn in which the diluted message ($\hat{\epsilon}$) is also the message that is being condensed. This case is particularly tricky because the pulls need to be commuted, as in $(- [\uparrow\epsilon]) [\uparrow\hat{\epsilon} [\uparrow\epsilon]] = (- [\uparrow\hat{\epsilon}]) [\uparrow\epsilon [\uparrow\hat{\epsilon}]]$. □

When defining the closure rules, we could have restricted $\nu \prec\!\!\subset \epsilon$ (and similarly $\nu \prec\!\!\Subset \epsilon$) to messages with equal views: $\nu.\mathtt{vw} = \epsilon.\mathtt{vw}$, resulting in the same semantics. For example, to apply the restricted version of absorb, one first applies tighten, which is also an $\mathfrak{a}$-rewrite, to make the views equal. In fact, we used this slightly simpler presentation in the abridged version of this paper [21].

Using the simpler presentation would require a less tidy statement of Rewrite Castling. For example, we would not have Di ⇄ Ls, because it may be that we "dilute" an environment message and then "loosen" it. After castling, using the restricted version of dilute, we would need to then "tighten" the new local message to recover the resulting trace from the original rewrite sequence.

Example 8.4. Keeping α, ω, and μ from Example 8.1, let:

$$\rho_q \coloneqq \begin{Bmatrix} \mathtt{y} : 5@(-1, \underline{0}] \langle\!\langle 0 \rangle\!\rangle, & 7@(1, \underline{3}] \langle\!\langle 0 \rangle\!\rangle \\ \mathtt{x} : 6@(-1, \underline{0}] \langle\!\langle 0 \rangle\!\rangle, & 8@(1, \underline{3}] \langle\!\langle q \rangle\!\rangle \end{Bmatrix} = \mu \uplus \left\{\mathtt{x}{:}8@(1, \underline{3}] \langle\!\langle \mathtt{y}@q \rangle\!\rangle\right\}$$

$$\theta_{q,t} \coloneqq \begin{Bmatrix} \mathtt{y} : 5@(-1, \underline{0}] \langle\!\langle 0 \rangle\!\rangle, & 7@(1, \underline{3}] \langle\!\langle 0 \rangle\!\rangle & \\ \mathtt{x} : 6@(-1, \underline{0}] \langle\!\langle 0 \rangle\!\rangle, & 8@(1, \underline{2}] \langle\!\langle q \rangle\!\rangle, & 8@(2, \underline{3}] \langle\!\langle t \rangle\!\rangle \end{Bmatrix} = \mu \uplus \left\{\mathtt{x}{:}8@(1, \underline{2}] \langle\!\langle \mathtt{y}@q \rangle\!\rangle, \mathtt{x}{:}8@(2, \underline{3}] \langle\!\langle \mathtt{y}@t \rangle\!\rangle\right\}$$

Consider the following sequence of trace rewrites, where Di$^=$ is the restricted version of dilute:

$$\underbrace{\alpha \boxed{\langle \rho_3, \rho_3 \rangle} \alpha \therefore 9}_{\tau} \xrightarrow{\mathsf{Di}^=} \underbrace{\alpha \boxed{\langle \rho_3, \theta_{3,3} \rangle} \alpha \therefore 9}_{\pi} \xrightarrow{\mathsf{Ls}} \underbrace{\alpha \boxed{\langle \rho_0, \theta_{0,3} \rangle} \alpha \therefore 9}_{\varrho}$$

We also have the sequence:

$$\underbrace{\alpha \boxed{\langle \rho_3, \rho_3 \rangle} \alpha \therefore 9}_{\tau} \xrightarrow{\mathsf{Ls}} \underbrace{\alpha \boxed{\langle \rho_0, \rho_0 \rangle} \alpha \therefore 9}_{\pi'} \xrightarrow{\mathsf{Di}^=} \underbrace{\alpha \boxed{\langle \rho_0, \theta_{0,0} \rangle} \alpha \therefore 9}_{\pi''} \xrightarrow{\mathsf{Ti}} \underbrace{\alpha \boxed{\langle \rho_0, \theta_{0,3} \rangle} \alpha \therefore 9}_{\varrho}$$

which recovers ϱ only after an application of tighten.

As a corollary to Rewrite Castling, we can castle $\mathfrak{ca}$-rewrites out of the $\mathcal{G}$-operators:

Lemma 8.5 (Deferral of Closure). *Let $\mathfrak{c} \subseteq \star \subseteq \mathfrak{ca}$. For all $P_i \in \underline{\mathcal{G}}X_i$ and $f : X_1 \to \underline{\mathcal{G}}X_2$:*

$$\left(P_1^\star \gg\!=^{\mathcal{G}} f^\star\right)^\star = \left(P_1 \gg\!=^{\mathcal{G}} f\right)^\star \qquad \left(P_1^\star \mid\mid\mid^{\mathcal{G}} P_2^\star\right)^\star = \left(P_1 \mid\mid\mid^{\mathcal{G}} P_2\right)^\star$$

Proof. In the proof we rely on the fact that $\mathfrak{g}$ has a counterpart for every closure rule in $\mathfrak{a}$: Ls $\leftrightarrow$ Ti; Ex $\leftrightarrow$ Ab; Cn $\leftrightarrow$ Di. Figures 12 to 15 depict this correspondence when comparing the left and right sides of the figure: the messages that need to be local to apply an $\mathfrak{a}$-closure need to be environment messages in their $\mathfrak{g}$-counterpart, and the rewrite goes in the opposite direction. For example, instead of Ab-rewriting some trace $\tau \in P_1$ and then "binding" it with a trace $\pi \in f(\tau.\mathtt{vl})$, we can instead mirror its effect by Ex-rewriting π to make its messages match τ's, bind *those* together, and then Ab-rewrite *after* the bind.

The detailed proof is in §A. □

Deferral of Closure also applies to $\mathcal{C}$ and $\mathcal{A}$ instead of $\mathcal{G}$, since $\underline{\mathcal{G}}X \supseteq \underline{\mathcal{C}}X \supseteq \underline{\mathcal{A}}X$. Since calculations in $\mathcal{G}$ are relatively simple, this lemma is quite convenient to have.

Example 8.6. Deferral of Closure helps show the associativity law holds for both $\mathcal{C}$ and $\mathcal{A}$. The associativity law for $\mathcal{N}$ is easy to verify directly. It implies the associativity law for $\mathcal{G}$ by Proposition 7.5. To show the associativity law for $\mathcal{C}$, we specialize Deferral of Closure to $\star = \mathfrak{c}$, and restrict to $P \in \underline{\mathcal{C}}X$, $f : X \to \underline{\mathcal{C}}Y$, and $g : Y \to \underline{\mathcal{C}}Z$, obtaining:

$$\left(P \gg\!=^{\mathcal{C}} f\right) \gg\!=^{\mathcal{C}} g = \left(\left(P \gg\!=^{\mathcal{G}} f\right)^{\mathfrak{c}} \gg\!=^{\mathcal{G}} g\right)^{\mathfrak{c}} = \left(\left(P \gg\!=^{\mathcal{G}} f\right) \gg\!=^{\mathcal{G}} g\right)^{\mathfrak{c}}$$

$$P \gg\!=^{\mathcal{C}} \left(\lambda r.\, f(r) \gg\!=^{\mathcal{C}} g\right) = \left(P \gg\!=^{\mathcal{G}} \left(\lambda r.\, f(r) \gg\!=^{\mathcal{G}} g\right)^{\mathfrak{c}}\right)^{\mathfrak{c}} = \left(P \gg\!=^{\mathcal{G}} \left(\lambda r.\, f(r) \gg\!=^{\mathcal{G}} g\right)\right)^{\mathfrak{c}}$$

The associativity law for $\mathcal{A}$ follows from the same argument, by specializing to $\star = \mathfrak{ca}$.

When calculating denotations of terms, we can use Deferral of Closure to similarly delay taking the closure. Specifically, for programs (closed terms of ground type), we can delay all the way through to the top level. Relating $\mathcal{C}$ to $\mathcal{A}$ in this way is a key step in our proof of adequacy.

Lemma 8.7 (Retroactive Closure). *If M is a program, then $[\![M]\!]^{\mathfrak{c}}_{\mathcal{A}} = [\![M]\!]^{\mathfrak{c}\,\mathfrak{a}}_{\mathcal{C}}$.*

The proof, using Rewrite Castling and Deferral of Closure, is in §A. It follows a standard logical-relation argument to account for higher-order terms that may appear as subterms of the program.

The rest of §8 focuses mainly on the model $\mathcal{A}$. To emphasize this fact, and to avoid clutter, we henceforth omit $\mathcal{A}$ from its semantic notations. In particular, we write $[\![M]\!]^{\mathfrak{c}}$ rather than $[\![M]\!]^{\mathfrak{c}}_{\mathcal{A}}$.

8.2 Compositionality

To state compositionality, and later adequacy, we need a few technical concepts involving capturing and capture-avoiding substitution in $\boldsymbol{\lambda}_{\mathrm{RA}}$ and its semantics. We extend $\boldsymbol{\lambda}_{\mathrm{RA}}$ with well-typed *metavariables*: these are binding-aware identifiers $\Gamma \vdash \mathbf{M} : A$. Metavariables represent "holes" into which we can slot well-typed terms $\Gamma \vdash M : A$, in an operation called *metavariable substitution*. When such a metavariable appears in a term, it is accompanied by an explicit value substitution governing which values to substitute when we slot a term into it. Metavariable substitution captures the variables of which the metavariables are aware.

Example 8.8. Consider the following metavariable that is aware of a context with two variables: $a : \mathsf{Loc}, b : \mathsf{Val} \vdash \mathbf{M} : \mathbf{1}$. The term $\vdash \mathbf{M}[a \mapsto \mathtt{x}, b \mapsto 42] : \mathbf{1}$ contains this metavariable and no other variables. Metasubstituting the open term $a : \mathsf{Loc}, b : \mathsf{Val} \vdash a \coloneqq b$ for $\mathbf{M}$ yields $\vdash \mathtt{x} \coloneqq 42 : \mathbf{1}$.

This treatment of metavariables and their substitution is tedious but standard given the binding structure in the syntax. A *(term) context* $\Delta \vdash \Xi\,[\Gamma \vdash - : A] : B$ is a term of type B with variables from Δ and one meta-variable $\Gamma \vdash - : A$ of type A that assumes a binding context Γ. It is a *program context* when Δ is empty and $B = G$ is ground.

The recursive definition of a term's denotation only uses the denotations of its subterms, so the semantics is automatically compositional. Abbreviating $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$ into $\Gamma \vdash M, N : A$:

Proposition 8.9 (Compositionality). *Let $\Delta \vdash \Xi\,[\Gamma \vdash - : A] : B$ be a term context and assume $\Gamma \vdash M, N : A$. If $[\![M]\!]^{\mathsf{c}} = [\![N]\!]^{\mathsf{c}}$ then $[\![\Xi\,[M]]\!]^{\mathsf{c}} = [\![\Xi\,[N]]\!]^{\mathsf{c}}$.*

However, we are interested in a directional version of this, dealing not only with set equality but also with set inclusion. Simply replacing $=$ with $\subseteq$ in Proposition 8.9 results in a false claim. This is because the language is higher-order, so only a "nested" form of containment holds.

Example 8.10. We have $[\![\ell \coloneqq 2]\!]^{\mathsf{c}} \subseteq [\![\ell \coloneqq 1\,; \ell \coloneqq 2]\!]^{\mathsf{c}}$, yet $[\![\lambda\ell.\,\ell \coloneqq 2]\!]^{\mathsf{c}} \not\subseteq [\![\lambda\ell.\,\ell \coloneqq 1\,; \ell \coloneqq 2]\!]^{\mathsf{c}}$. So Proposition 8.9 with $\subseteq$ instead of $=$ is indeed false. These latter sets are in fact disjoint, their traces having different return values: $\lambda\ell.\,[\![\ell \coloneqq 2]\!]^{\mathsf{c}}$ versus $\lambda\ell.\,[\![\ell \coloneqq 1\,; \ell \coloneqq 2]\!]^{\mathsf{c}}$. These values are related by pointwise containment.

We therefore restrict the directional version of compositionality to programs:

Theorem 8.11 (Directional Compositionality). *Let $\cdot \vdash \Xi\,[\Gamma \vdash - : A] : G$ be a program context and assume $\Gamma \vdash M, N : A$. If $[\![M]\!]^{\mathsf{c}} \subseteq [\![N]\!]^{\mathsf{c}}$ then $[\![\Xi\,[M]]\!]^{\mathsf{c}} \subseteq [\![\Xi\,[N]]\!]^{\mathsf{c}}$.*

The proof is in §B. It uses a logical relation to account for higher-order terms. On ground types, the logical relation simplifies to set containment.

8.3 Soundness

A basic part of the correspondence between the denotational and the operational semantics is its soundness, in the sense that the denotation of a program has traces corresponding to evaluations. More specifically, program evaluation is reflected in the denotation of the program by a single-transition trace, using the greatest lower bound of the initial view tree as the initial view:

Theorem 8.12 (Soundness). *For a program M, if $\langle T, \mu\rangle, M \Downarrow V$, then there exist μ' and ω such that $\inf_{\mu} T \boxed{\langle \mu, \mu' \rangle} \omega \therefore V \in [\![M]\!]^{\mathsf{c}}_{C}$.*

The proof is in §C. It first shows soundness by straightforward induction for a model with a different model, which uses a more intensional notion of a trace. These traces keep track of the initial view tree, and their transitions correspond to memory-accessing steps (•-labeled steps) in the operational semantics. This different model is then related to C by a logical relation.

By Retroactive Closure, $[\![M]\!]^{\mathsf{c}}_{C} \subseteq [\![M]\!]^{\mathsf{c}}$. So, though stated only for C, Soundness holds for $\mathcal{A}$ too.

Impossible outcomes. The contrapositive presentation of Soundness states that certain evaluations of a program can be ruled out by inspecting its denotation. For example, the impossible evaluations of (MP) from Example 5.3 can be shown indirectly by calculating its denotation.

8.4 Adequacy

Adequacy uses contextual refinements to formalize how denotations capture behavior within any context. For $\Gamma \vdash M, N : A$, we say that M *contextually refines* N, denoted by $\Gamma \vdash M \sqsubseteq N : A$, or by $M \sqsubseteq N$ for short, when $\langle \dot{\alpha}, \mu \rangle, \Xi[M] \Downarrow V$ implies that $\langle \dot{\alpha}, \mu \rangle, \Xi[N] \Downarrow V$ for every program context $\cdot \vdash \Xi[\Gamma \vdash - : A] : G$, initial configuration state $\langle \dot{\alpha}, \mu \rangle$, and value V.

Theorem 8.13 (Directional Adequacy). *If* $[\![M]\!]^{c} \subseteq [\![N]\!]^{c}$ *then* $M \sqsubseteq N$.

The proof begins by examining the tight correspondence between traces in denotations over C and interrupted executions. Formally, we say that M *executes through* τ to V, written $M : \tau : V$, when there is an interrupted execution from M to V such that $\tau.\mathtt{vl} = [\![V]\!]^{v}_{C}$, which starts with the view-leaf $\tau.\mathtt{ivw}$, passes exactly through the memory transitions of $\tau.\mathtt{ch}$, and ends with the view-leaf $\tau.\mathtt{fvw}$. By the Fundamental Lemma, the precise statement and proof of which we relegate to §D: for every $\tau \in [\![M]\!]^{c}_{C}$ there exists an appropriate value V such that $M : \tau : V$. In particular:

Lemma 8.14 (Concrete Traces Lemma). *For a program* M*, if* $\tau \in [\![M]\!]^{c}_{C}$ *then* $M : \tau : \tau.\mathtt{vl}$.

Traces in denotations over $\mathcal{A}$ do not enjoy this correspondence, due to the model's abstraction. However, a looser correspondence holds, between denotations of programs to their evaluations:

Lemma 8.15 (Evaluation Lemma). *For a program* M*, if* $\alpha \boxed{\langle \mu, \rho \rangle} \omega \therefore r \in [\![M]\!]^{c}$ *then* $\langle \dot{\alpha}, \mu \rangle, M \Downarrow r$.

Proof. By Retroactive Closure the trace is obtained by $\mathfrak{a}$-rewriting a trace in $[\![M]\!]^{c}_{C}$. We proceed by induction on the length of this rewrite sequence. In the base case, we use the Concrete Traces Lemma, which for a single-transition trace degenerates to an uninterrupted execution. For the inductive step, we observe that $\mathfrak{a}$-rewrites preserve evaluation. We leave the details to §D. □

We are finally prepared to prove the Directional Adequacy theorem:

Proof of Directional Adequacy. Assume $[\![M]\!]^{c} \subseteq [\![N]\!]^{c}$. Let $\cdot \vdash \Xi[\Gamma \vdash - : A] : G$ be a program context and assume $\langle \dot{\alpha}, \mu \rangle, \Xi[M] \Downarrow V$. By Soundness followed by Retroactive Closure, $\tau \in [\![\Xi[M]]\!]^{c}_{C} \subseteq [\![\Xi[M]]\!]^{c}$ for some τ of the form $\alpha \boxed{\langle \mu, - \rangle} - \therefore V$. By Directional Compositionality and the assumption, $\tau \in [\![\Xi[N]]\!]^{c}$. By the Evaluation Lemma, $\langle \dot{\alpha}, \mu \rangle, \Xi[N] \Downarrow V$. □

8.5 Validating Transformations

Using Directional Adequacy, we can validate $M \twoheadrightarrow N$ in our model by showing that $[\![M]\!]^{c} \supseteq [\![N]\!]^{c}$. As mentioned in §7.1, we validate the structural transformations by virtue of using standard denotational semantics. For others, thanks to Deferral of Closure and closure preserving containment, we can use the $\mathcal{G}$ operators instead of the $\mathcal{A}$ operators, simplifying calculations.

Figure 3 lists various transformations that we support in this way. Table 3 includes a more general collection, with accompanying proofs in §E. The table is organized such that the general pattern appears first, followed by specific instantiations and corollaries.

For handling the RMW modifiers, we use additional notations. For modifiers $\Phi, \Psi \in \mathsf{Val} \rightharpoonup \mathsf{Val}$:

- The *domain of definition* of Φ is $\mathrm{dom}\,\Phi \coloneqq \{v \in \mathsf{Val} \mid \Phi v \neq \bot\}$.
- We say that Ψ is an *expansion* of Φ, denoted by $\Phi \leq \Psi$, if $\Phi v \neq \Psi v$ occurs only when $\Phi v = \bot$ and $\Psi v = v$. Intuitively, this means that Φ and Ψ are the same, except that in some cases in which Φ reads and does not write, Ψ overwrites the read value with itself.
- We denote by Φ^{id} the unique expansion of Φ that is total: $\Phi^{\mathrm{id}} v \coloneqq \text{if } \Phi v{=}\bot \text{ then } v \text{ else } \Phi v$. Intuitively, Φ^{id} overwrites the read value with itself whenever Φ reads but does not write.
- We let $\left(\Psi \circ^{\mathrm{id}} \Phi\right) v \coloneqq \text{if } \Phi v{=}\bot \text{ then } \Psi v \text{ else } \Psi^{\mathrm{id}} (\Phi v)$. Intuitively, $\Psi \circ^{\mathrm{id}} \Phi$ composes the modification of Φ followed by the modification of Ψ, only failing if both do.

Some optimizations involving modifiers assume the language can express corresponding constructs. For example, the Write-RMW Elimination instantiated with $\varphi = \mathrm{faa}$ requires addition (+),

Table 3. Transformations that $\mathcal{A}$ validates. The list mentions the $\mathfrak{a}$-closures that the proofs appeal to.

Generalized Sequencing (E.1)

$$(\textbf{let}\, a = M_1 \,\textbf{in}\, M_2) \parallel (\textbf{let}\, b = N_1 \,\textbf{in}\, N_2) \twoheadrightarrow \textbf{match}\, M_1 \parallel N_1 \,\textbf{with}\, \langle a, b\rangle.\, M_2 \parallel N_2$$

Sequencing $M \parallel N \twoheadrightarrow \langle M, N\rangle$

Irrelevant Read Introduction (E.2) $\langle\rangle \twoheadrightarrow \ell? \,;\, \langle\rangle$

Irrelevant Read Elimination (E.3) $\ell? \,;\, \langle\rangle \twoheadrightarrow \langle\rangle$

Write-Write Elimination (E.10)

$$\ell \coloneqq w \,;\, \ell \coloneqq v \overset{\text{Ab}}{\twoheadrightarrow} \ell \coloneqq v$$

Write-Read Deorder (E.5) $(\ell \neq \ell')$

$$\langle \ell \coloneqq v, \ell'?\rangle \overset{\text{Ti}}{\twoheadrightarrow} \ell \coloneqq v \parallel \ell'?$$

RMW Expansion (E.6) $(\varphi_{\vec{v}} \leq \psi_{\vec{w}})$

$$\begin{aligned} \textbf{rmw}_{\varphi}\,(\ell; \vec{v}) &\overset{\text{Di}}{\twoheadrightarrow} \textbf{rmw}_{\psi}\,(\ell; \vec{w}) \\ \ell? &\overset{\text{Di}}{\twoheadrightarrow} \text{CAS}\,(\ell, v, v) \\ \text{CAS}\,(\ell, v, v) &\overset{\text{Di}}{\twoheadrightarrow} \text{FAA}\,(\ell, 0) \end{aligned}$$

Atomic Store (E.4)

$$\ell \coloneqq v \twoheadrightarrow \text{XCHG}\,(\ell, v) \,;\, \langle\rangle$$

Symmetric-Monoidal Laws, e.g.

$$M \parallel N \twoheadrightarrow \textbf{match}\, N \parallel M \,\textbf{with}\, \langle b, a\rangle.\, \langle a, b\rangle$$

Write-RMW Elimination (E.9)

$$\begin{aligned} \ell \coloneqq v \,;\, \textbf{rmw}_{\varphi}\,(\ell; \vec{w}) &\overset{\text{Ab}}{\twoheadrightarrow} \ell \coloneqq \varphi^{\text{id}}_{\vec{w}} v \,;\, v \\ \ell \coloneqq v \,;\, \ell? &\twoheadrightarrow \ell \coloneqq v \,;\, v \\ \ell \coloneqq v \,;\, \text{CAS}\,(\ell, v, u) &\overset{\text{Ab}}{\twoheadrightarrow} \ell \coloneqq u \,;\, v \\ \ell \coloneqq v \,;\, \text{CAS}\,(\ell, w, u) &\twoheadrightarrow \ell \coloneqq v \,;\, v \qquad (v \neq w) \\ \ell \coloneqq v \,;\, \text{FAA}\,(\ell, w) &\overset{\text{Ab}}{\twoheadrightarrow} \ell \coloneqq v + w \,;\, v \\ \ell \coloneqq v \,;\, \text{XCHG}\,(\ell, w) &\overset{\text{Ab}}{\twoheadrightarrow} \ell \coloneqq w \,;\, v \end{aligned}$$

RMW-Write Elimination (E.11) $(\operatorname{dom} \psi_{\vec{w}} \supseteq \operatorname{dom} \varphi_{\vec{u}})$

$$\begin{aligned} &\textbf{let}\, a = \textbf{rmw}_{\varphi}\,(\ell; \vec{u}) \,\textbf{in} \\ &\quad \textbf{match}\, (\psi_{\vec{w}})\, a \,\textbf{with} \\ &\quad \{\iota_\bot\, \_.a \mid \iota_\top\, v.\ell \coloneqq v \,;\, a\} \overset{\text{Ab}}{\twoheadrightarrow} \textbf{rmw}_{\psi}\,(\ell; \vec{w}) \\ &\textbf{let}\, a = \ell? \,\textbf{in}\, (\textbf{if}\, a = v \\ &\quad \textbf{then}\, \ell \coloneqq w \,\textbf{else}\, \langle\rangle\,) \,;\, a \twoheadrightarrow \text{CAS}\,(\ell, v, w) \\ &\textbf{let}\, a = \ell? \,\textbf{in}\, \ell \coloneqq a + v \,;\, a \twoheadrightarrow \text{FAA}\,(\ell, v) \\ &\textbf{let}\, a = \ell? \,\textbf{in}\, \ell \coloneqq v \,;\, a \twoheadrightarrow \text{XCHG}\,(\ell, v) \end{aligned}$$

RMW-RMW Elimination (E.8) $\left\langle \textbf{rmw}_{\varphi}\,(\ell; \vec{v}), \textbf{rmw}_{\psi}\,(\ell; \vec{w})\right\rangle \overset{\text{Ab}}{\twoheadrightarrow} \textbf{let}\, a = \textbf{rmw}_{\zeta}\,(\ell; \vec{u}) \,\textbf{in}\, \left\langle a, \varphi^{\text{id}}_{\vec{v}} a\right\rangle \quad (\zeta_{\vec{u}} = \psi_{\vec{w}} \circ^{\text{id}} \varphi_{\vec{v}})$

$\langle \ell?, \ell?\rangle \twoheadrightarrow \textbf{let}\, a = \ell? \,\textbf{in}\, \langle a, a\rangle$ $\quad\langle \text{FAA}\,(\ell, v), \text{FAA}\,(\ell, w)\rangle \twoheadrightarrow \textbf{let}\, a = \text{FAA}\,(\ell, v + w) \,\textbf{in}\, \langle a, a + v\rangle$

$\langle \ell?, \text{CAS}\,(\ell, v, w)\rangle \twoheadrightarrow \textbf{let}\, a = \text{CAS}\,(\ell, v, w) \,\textbf{in}\, \langle a, a\rangle$ $\quad\langle \text{XCHG}\,(\ell, w), \ell?\rangle \twoheadrightarrow \textbf{let}\, a = \text{XCHG}\,(\ell, w) \,\textbf{in}\, \langle a, w\rangle$

and the RMW-Write Elimination instantiated with $\varphi = \text{cas}$ requires branching on value comparison ($\textbf{if} - = - \textbf{then} - \textbf{else} -$). Under this assumption, for every modifier Φ both Φ and Φ^{id} are represented by closed, pure (effect-free) terms, of type $\textbf{Val} \to \{\iota_\bot \textbf{ of } \mathbf{1} \mid \iota_\top \textbf{ of Val}\}$ and $\textbf{Val} \to \textbf{Val}$ respectively. These are used implicitly in Table 3 when modifiers appear in syntax positions. For example, $\text{cas}_{\langle 3,2\rangle}$ is represented by $\lambda a.\, \textbf{if}\, a = 3 \,\textbf{then}\, \iota_\top 2 \,\textbf{else}\, \iota_\bot$; and $\text{cas}^{\text{id}}_{\langle 3,2\rangle}$ by $\lambda a.\, \textbf{if}\, a = 3 \,\textbf{then}\, 2 \,\textbf{else}\, a$.

The listed memory-access transformations are stated in ground terms, but imply more general variants. For example, we state Write-Write Elimination as $\ell \coloneqq w \,;\, \ell \coloneqq v \twoheadrightarrow \ell \coloneqq v$, from which we can deduce e.g., $\lambda a : \textbf{Loc}.\, a \coloneqq w \,;\, a \coloneqq v \twoheadrightarrow \lambda a : \textbf{Loc}.\, a \coloneqq v$. The proof uses the standard semantics: structural transformations include any pure computations that result in the same value, and in particular, we can replace the locations and (storable) values with pure computations that result in them, or program variables of the same type.

All told, we claim that our adequate denotational semantics is sufficiently abstract. This claim supports the case that Moggi's semantic toolkit can successfully scale to handle the intricacies of RA concurrency by adapting Brookes's traces.

9 Related Work and Concluding Remarks

Our work follows the approach of Brookes [13] and its extension to higher-order functions using monads by Benton et al. [6]. Brookes developed a denotational semantics for shared memory concurrency under standard sequentially consistency [35], and established full abstraction w.r.t. a language that has a global atomic **await** instruction that locks the entire memory. The concepts behind this approach had been used in multiple related developments [e.g. 12, 36, 37, 51]. We hope that our work that targets RA will pave the way for similar extensions.

Jagadeesan et al. [25] adapted Brookes's semantics to the x86-TSO memory model [42]. They showed that for x86-TSO it suffices to include the final store buffer at the end of the trace and add two additional simple closure rules that emulate non-deterministic propagation of writes from store buffers to memory, and identify observably equivalent store buffers. The x86-TSO model, however, is much closer to sequential consistency than RA, which we study in this paper. In particular, unlike RA, x86-TSO is "multi-copy-atomic" (writes by one thread are made globally visible to *all* other threads at the same time) and successful RMW operations are immediately globally visible. Additionally, the parallel composition construct in Jagadeesan et al. [25] is rather strong: threads are forked and joined only when the store buffers are empty. Being non-multi-copy-atomic, RA requires a more delicate notion of traces and closure rules, but it has more natural meta-theoretic properties, which one would expect from a programming language concurrency model: sequencing, a.k.a. thread-inlining, is unsound under x86-TSO [see 25, 33] but sound under RA (see Figure 3).

Burckhardt et al. [14] developed a denotational semantics for hardware weak memory models, including x86-TSO, following an alternative approach. They represent sequential code blocks by sequences of operations that the code performs, and close them under reorderings and eliminations that characterize the memory model. This approach does not validates important optimizations, such as Read-Read Elimination. Moreover, RA is not characterizable in this way [33].

Dodds et al. [19] developed a fully abstract denotational semantics for RA, extended with fences and non-atomic accesses. Their semantics is based on RA's *declarative* (a.k.a. axiomatic) formulation as acyclicity criteria on execution graphs. Roughly speaking, their denotation of code blocks (that they assume to be sequential) quantifies over all possible context execution graphs and calculates for each context the "happens-before" relation between context actions that is induced by the block. They further use a finite approximation of these histories to atomically validate refinement in a model checker. While we target RA as well, there are two crucial differences between Dodds et al.'s work and ours. First, we employ Brookes-style totally ordered traces and use interleaving-based operational presentation of RA. Second, and more importantly, we strive for a compositional semantics where denotations of compound programs are defined as functions of denotations of their constituents, which is not the case for Dodds et al.'s definitions. Their model can nonetheless validate transformations by checking them locally without access to the full program.

Others present non-compositional techniques and tools to check refinement under weak memory models between whole-thread sequential programs that apply for any concurrent context. Poetzl and Kroening [46] considered the SC-for-DRF model, using locks to avoid races. Their approach matches source to target by checking that they perform the same state transitions from lock to subsequent unlock operations and that the source does not allow more data-races. Morisset et al. [41] and Chakraborty and Vafeiadis [16] addressed this problem for the C/C++11 model, of which RA is a central fragment, by implementing matching algorithms between source and target that validate that all transformations between them have been independently proven to be safe under C/C++11.

Cho et al. [18] introduced a specialized semantics for *sequential* programs that can be used for justifying compiler optimizations under weak memory concurrency. They showed that behavior refinement under their sequential semantics implies refinement under any (sequential or parallel) context in the Promising Semantics 2.1 [17]. Their work focuses on optimizations of race-free accesses that are similar to C11's "non-atomics" [4, 34]. It cannot be used to establish the soundness of program transformations that we study in this paper. Adding non-atomics to our model is an important future work.

Denotational approaches were developed for models much weaker than RA [15, 24, 26, 29, 43] that allow the infamous Read-Write Reorder and thus, for a high-level programming language, require addressing the challenge of detecting semantic dependencies between instructions [3]. These approaches are based on summarizing multiple partial orders between actions that may arise when a given program is executed under some context. In contrast, we use totally ordered traces by relating to RA's interleaving operational semantics. In particular, Kavanagh and Brookes [29] use partial orders, Castellan, Paviotti et al. [15, 43] use event structures, and Jagadeesan et al., Jeffrey et al. [24, 26] employ "Pomsets with Preconditions" which trades compositionality for supporting non-multi-copy-atomicity, as in RA. These approaches do not validate certain access eliminations, nor Irrelevant Load Introduction, which our model validates.

An exciting aspect of our work is the connection between memory models to Moggi's monadic approach [40]. For SC, Abadi and Plotkin, Dvir et al. [1, 20] have made an even stronger connection via algebraic theories [44]. These allow to modularly combine shared memory concurrency with other computational effects. Birkedal et al. [11] develop semantics for a type-and-effect system for SC memory which they use to enhance compiler optimizations based on assumptions on the context that come from the type system. We hope to the current work can serve as a basis to extend such accounts to weaker models.

Acknowledgments

Supported by the Israel Science Foundation (grant number 814/22) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811); and by a Royal Society University Research Fellowship and Enhancement Award.

References

[1] Martín Abadi and Gordon Plotkin. 2010. A Model of Cooperative Threads. *Log. Methods Comput. Sci.* 6, 4 (2010). https://doi.org/10.2168/LMCS-6(4:2)2010
[2] Alejandro Aguirre, Shin-ya Katsumata, and Satoshi Kura. 2022. Weakest preconditions in fibrations. *Mathematical Structures in Computer Science* 32, 4 (2022). https://doi.org/10.1017/S0960129522000330
[3] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP (LNCS, Vol. 9032)*. Springer. https://doi.org/10.1007/978-3-662-46669-8_12
[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM. https://doi.org/10.1145/1926385.1926394
[5] Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract effects and proof-relevant logical relations. In *POPL*. ACM.
[6] Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-dependent transformations for concurrent programs. In *PPDP*. ACM. https://doi.org/10.1145/2967973.2968602
[7] Nick Benton, John Hughes, and Eugenio Moggi. 2000. Monads and Effects. In *APPSEM*.
[8] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*. ACM.
[9] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*. ACM.
[10] Nick Benton and Benjamin Leperchey. 2005. Relational Reasoning in a Nominal Semantics for Storage. In *TLCA*. Springer.
[11] Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *CSL (LIPIcs, Vol. 16)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.CSL.2012.107
[12] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007). https://doi.org/10.1016/j.tcs.2006.12.034
[13] Stephen D. Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Inf. Comput.* 127, 2 (1996). https://doi.org/10.1006/inco.1996.0056
[14] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. 2010. Verifying Local Transformations on Relaxed Memory Models. In *CC (LNCS, Vol. 6011)*. Springer. https://doi.org/10.1007/978-3-642-11970-5_7
[15] Simon Castellan. 2016. Weak memory models using event structures. In *JFLA*. Saint-Malo, France. https://hal.inria.fr/hal-01333582

[16] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *CGO*. ACM. https://doi.org/10.1145/2854038.2854051

[17] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular data-race-freedom guarantees in the promising semantics. In *PLDI*. ACM. https://doi.org/10.1145/3453483.3454082

[18] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential reasoning for optimizing compilers under weak memory concurrency. In *PLDI*. ACM. https://doi.org/10.1145/3519939.3523718

[19] Mike Dodds, Mark Batty, and Alexey Gotsman. 2018. Compositional Verification of Compiler Optimisations on Relaxed Memory. In *ESOP (LNCS, Vol. 10801)*. Springer. https://doi.org/10.1007/978-3-319-89884-1_36

[20] Yotam Dvir, Ohad Kammar, and Ori Lahav. 2022. An Algebraic Theory for Shared-State Concurrency. In *APLAS (LNCS, Vol. 13658)*. Springer. https://doi.org/10.1007/978-3-031-21037-2_1

[21] Yotam Dvir, Ohad Kammar, and Ori Lahav. 2024. A Denotational Approach to Release/Acquire Concurrency. In *ESOP (LNCS)*. Springer, 121–149. https://doi.org/10.1007/978-3-031-57267-8_5

[22] Tony Hoare and Stephan van Staden. 2014. The laws of programming unify process calculi. *Sci. Comput. Program.* 85 (2014). https://doi.org/10.1016/j.scico.2013.08.012

[23] Martin Hofmann. 2008. Correctness of effect-based program transformations. In *Formal Logical Methods for System Security and Correctness*. IOS Press.

[24] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428262

[25] Radha Jagadeesan, Gustavo Petri, and James Riely. 2012. Brookes Is Relaxed, Almost!. In *FOSSACS (LNCS, Vol. 7213)*. Springer. https://doi.org/10.1007/978-3-642-28729-9_12

[26] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022). https://doi.org/10.1145/3498716

[27] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[28] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. ACM. https://doi.org/10.1145/3009837.3009850

[29] Ryan Kavanagh and Stephen Brookes. 2018. A Denotational Semantics for SPARC TSO. In *MFPS (ENTCS, Vol. 341)*. Elsevier. https://doi.org/10.1016/j.entcs.2018.03.025

[30] Ori Lahav. 2019. Verification under Causally Consistent Shared Memory. *ACM SIGLOG News* 6, 2 (April 2019). https://doi.org/10.1145/3326938.3326942

[31] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*. ACM. https://doi.org/10.1145/2837614.2837643

[32] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making weak memory models fair. *Proc. ACM Program. Lang.* 5, OOPSLA (2021). https://doi.org/10.1145/3485475

[33] Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM (LNCS, Vol. 9995)*. https://doi.org/10.1007/978-3-319-48989-6_29

[34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*. ACM. https://doi.org/10.1145/3062341.3062352

[35] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979). https://doi.org/10.1109/TC.1979.1675439

[36] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*. ACM. https://doi.org/10.1145/2103656.2103711

[37] Hongjin Liang, Xinyu Feng, and Ming Fu. 2014. Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations. *ACM Trans. Program. Lang. Syst.* 36, 1 (2014). https://doi.org/10.1145/2576235

[38] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2019. The next 700 Relational Program Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (Dec. 2019). https://doi.org/10.1145/3371072

[39] Jeremy Manson, William W. Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL*. ACM. https://doi.org/10.1145/1040305.1040336

[40] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991). https://doi.org/10.1016/0890-5401(91)90052-4

[41] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*. ACM. https://doi.org/10.1145/2491956.2491967

[42] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs (LNCS, Vol. 5674)*. Springer. https://doi.org/10.1007/978-3-642-03359-9_27

[43] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *ESOP (LNCS, Vol. 12075)*. Springer. https://doi.org/10.1007/978-3-030-44914-8_22
[44] Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *FOSSACS*. Springer.
[45] Gordon D Plotkin. 1973. Lambda-definability and logical relations. *Univ. of Edinburgh School of Artificial Intelligence Memorandum SAI-RM-4* (1973).
[46] Daniel Poetzl and Daniel Kroening. 2016. Formalizing and Checking Thread Refinement for Data-Race-Free Execution Models. In *TACAS (LNCS, Vol. 9636)*. Springer. https://doi.org/10.1007/978-3-662-49674-9_30
[47] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158107
[48] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. ACM. https://doi.org/10.1145/1993498.1993520
[49] Lau Skorstengaard. 2019. An Introduction to Logical Relations. arXiv:1907.11133 [cs.PL] https://arxiv.org/abs/1907.11133 Based on Amal Ahmed's OPLSS 2015 course.
[50] Mikhail Svyatlovskiy, Shai Mermelstein, and Ori Lahav. 2024. Compositional Semantics for Shared-Variable Concurrency. *Proc. ACM Program. Lang.* 8, PLDI (2024). https://doi.org/10.1145/3656399
[51] Aaron Joseph Turon and Mitchell Wand. 2011. A separation logic for refining concurrent objects. In *POPL*. ACM. https://doi.org/10.1145/1926385.1926415
[52] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *POPL*. ACM. https://doi.org/10.1145/2676726.2676995
[53] Qiwen Xu, Willem P. de Roever, and Jifeng He. 1997. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects Comput.* 9, 2 (1997). https://doi.org/10.1007/BF01211617

A Proofs of Deferral of Closure and Retroactive Closure

The proof of Rewrite Castling is in §F. Below are proofs of other claims from §8.1.

Proof of Deferral of Closure. Since $(-)^\star$ is a closure operator, it is monotonic, so the $\supseteq$ containment follows from the monotonicity of $\left(\rangle\!\rangle\!=^{\mathcal{G}}\right)$ and $\left(|||^{\mathcal{G}}\right)$ (Proposition 7.4). Moreover, for the $\subseteq$ containment, suffice it we show that $P_1^\star \rangle\!\rangle\!=^{\mathcal{G}} f^\star \subseteq \left(P_1 \rangle\!\rangle\!=^{\mathcal{G}} f\right)^\star$ and $P_1^\star\,|||^{\mathcal{G}} P_2^\star \subseteq \left(P_1\,|||^{\mathcal{G}} P_2\right)^\star$.

Denote by P^n the set of traces obtained by $\star$-rewriting n times a trace from P, and similarly for f^n. So it is sufficient to show that for all $n_1, n_2 \in \mathbb{N}$, $P_1^{n_1} \rangle\!\rangle\!=^{\mathcal{G}} f^{n_2} \subseteq \left(P_1 \rangle\!\rangle\!=^{\mathcal{G}} f\right)^\star$ and $P_1^{n_1}\,|||^{\mathcal{G}} P_2^{n_2} \subseteq \left(P_1\,|||^{\mathcal{G}} P_2\right)^\star$. We show this by induction on $n_1 + n_2$, where the base case $P_1 \rangle\!\rangle\!=^{\mathcal{G}} f \subseteq \left(P_1 \rangle\!\rangle\!=^{\mathcal{G}} f\right)^\star$ and $P_1\,|||^{\mathcal{G}} P_2 \subseteq \left(P_1\,|||^{\mathcal{G}} P_2\right)^\star$ holds since $(-)^\star$ is a closure operator.

For the induction step, the induction hypothesis is that the claim holds for $n_1 + n_2 \leq m$, and we must show it holds for $n_1 + n_2 = m + 1$. So either $n_1 = n_1' + 1$ or $n_2 = n_2' + 1$. We focus on the claim for $\left(|||^{\mathcal{G}}\right)$, since we find that proving the claim for $\left(\rangle\!\rangle\!=^{\mathcal{G}}\right)$ to be similar and somewhat easier.

Let $\tau \in P_1^{n_1}\,|||^{\mathcal{G}} P_2^{n_2}$. So $\tau = \inf_{\xi.\circ} \{\alpha_1, \alpha_2\} \boxed{\xi} \omega_1 \sqcup \omega_2 \mathrel{.\!\cdot\!.} \langle r_1, r_2 \rangle$ where $\tau_i \coloneqq \alpha_i \boxed{\xi_i} \omega_i \mathrel{.\!\cdot\!.} r_i \in P_i^{n_i}$ and $\xi \in \xi_1 \parallel \xi_2$. Assume w.l.o.g. that $n_1 = n_1' + 1$. So there is some $\tau_1' \in P_1^{n_1'}$ and $\mathsf{x} \in \star$ such that $\tau_1' \xrightarrow{\mathsf{x}} \tau_1$. By case analysis on x, we show that there exists $\tau' \in P_1^{n_1'}\,|||^{\mathcal{G}} P_2^{n_2}$ such that τ' $\star$-rewrites to τ. By the induction hypothesis $\tau' \in \left(P_1\,|||^{\mathcal{G}} P_2\right)^\star$, and so $\tau \in \left(P_1\,|||^{\mathcal{G}} P_2\right)^\star$.

For the $\mathsf{x} \in \star \cap \mathfrak{c}$ cases, we construct τ' from τ_1' and τ_2. The procedure depends on x:

Rw. So $\tau_1' = \alpha_1' \boxed{\xi_1} \omega_1 \mathrel{.\!\cdot\!.} r_1$ where $\alpha_1 \leq \alpha_1'$. We take

$$\tau' \coloneqq \inf_{\xi.\circ} \left\{\alpha_1', \alpha_2\right\} \boxed{\xi} \omega_1 \sqcup \omega_2 \mathrel{.\!\cdot\!.} \langle r_1, r_2 \rangle$$

Since $\inf_{\xi.\mathsf{o}} \{\alpha_1, \alpha_2\} \leq \inf_{\xi.\mathsf{o}} \{\alpha_1', \alpha_2\}$, we have $\tau' \xrightarrow{\mathsf{Rw}} \tau$.

Fw. Similar to Rw.

St. So $\tau_1' = \alpha_1 \boxed{\eta_1 \eta_1'} \omega_1 \therefore r_1$ where $\xi_1 = \eta_1 \langle \mu, \mu \rangle \eta_1'$. Since $\xi \in \xi_1 \parallel \xi_2$, there exist η, η' such that $\xi = \eta \langle \mu, \mu \rangle \eta'$, where η includes the transitions from η_1 and η' includes the transitions from η_1'. Formally, there exist η_2, η_2' such that $\xi_2 = \eta_2 \eta_2'$, $\eta \in \eta_1 \parallel \eta_2$ and $\eta' \in \eta_1' \parallel \eta_2'$. In particular, $\eta\eta' \in \eta_1\eta_1' \parallel \eta_2\eta_2' = \eta_1\eta_1' \parallel \xi_2$. Denoting $\xi' \coloneqq \eta\eta'$, we take

$$\tau' \coloneqq \inf_{\xi'.\mathsf{o}} \{\alpha_1, \alpha_2\} \boxed{\xi'} \omega_1 \sqcup \omega_2 \therefore \langle r_1, r_2 \rangle$$

We have $\xi.\mathsf{o} \subseteq \xi'.\mathsf{o}$, so $\inf_{\xi.\mathsf{o}} \{\alpha_1, \alpha_2\} \leq \inf_{\xi'.\mathsf{o}} \{\alpha_1, \alpha_2\}$. So $\tau' \xrightarrow{\mathsf{Rw}} \tau''$, where

$$\tau'' \coloneqq \inf_{\xi.\mathsf{o}} \{\alpha_1, \alpha_2\} \boxed{\xi'} \omega_1 \sqcup \omega_2 \therefore \langle r_1, r_2 \rangle$$

Since $\xi = \eta \langle \mu, \mu \rangle \eta'$, we have $\tau'' \xrightarrow{\mathsf{St}} \tau$.

Mu. So $\tau_1' = \alpha_1 \boxed{\eta_1 \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta_1'} \omega_1 \therefore r_1$ where $\xi_1 = \eta_1 \langle \mu, \theta \rangle \eta_1'$. Since $\xi \in \xi_1 \parallel \xi_2$, there exist η, η' such that $\xi = \eta \langle \mu, \theta \rangle \eta'$, where η includes the transitions from η_1 and η' includes the transitions from η_1'. Formally, there exist η_2, η_2' such that $\xi_2 = \eta_2\eta_2'$, $\eta \in \eta_1 \parallel \eta_2$ and $\eta' \in \eta_1' \parallel \eta_2'$. In particular, $\eta \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta' \in \eta_1 \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta_1' \parallel \eta_2\eta_2' = \eta_1 \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta_1' \parallel \xi_2$. Denoting $\xi' \coloneqq \eta \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta'$, we take

$$\tau' \coloneqq \inf_{\xi'.\mathsf{o}} \{\alpha_1, \alpha_2\} \boxed{\xi'} \omega_1 \sqcup \omega_2 \therefore \langle r_1, r_2 \rangle$$

Since $\xi = \eta \langle \mu, \theta \rangle \eta'$, and $\xi'.\mathsf{o} = \xi.\mathsf{o}$ and $\eta'.\mathsf{o} = \eta.\mathsf{o}$, we have $\tau' \xrightarrow{\mathsf{Mu}} \tau$.

For the $\mathsf{x} \in \star \cap \mathfrak{a}$ cases, we construct τ' from τ_1' and a τ_2' defined such that $\tau_2 \xrightarrow{\mathsf{y}} \tau_2'$ for some $\mathsf{y} \in \mathfrak{g}$. By iterating Rewrite Castling n_2 times to percolate $\xrightarrow{\mathsf{y}}$ through the $\star$-rewrite sequence that resulted in τ_2, we find that $\tau_2' \in P_2^{n_2}$. This is because $P_2 \in \underline{\mathcal{G}} X_2$. The procedure depends on x:

Ti. So $\tau_1' = \alpha_1 \boxed{\eta_1 \langle \mu, \rho \uplus \{\nu\} \rangle \eta_1' \,\overline{\uplus}\, \{\nu\}} \omega_1 \therefore r_1$ where $\xi_1 = \eta_1 \langle \mu, \rho \uplus \{\epsilon\} \rangle \eta_1' \overline{\uplus} \{\epsilon\}$ and $\nu \leq_{\mathsf{vw}} \epsilon$. Since $\xi \in \xi_1 \parallel \xi_2$, there are $\eta, \eta', \eta_2, \eta_2'$ such that $\xi = \eta \langle \mu, \rho \uplus \{\epsilon\} \rangle (\eta' \uplus \{\epsilon\})$, $\xi_2 = \eta_2(\eta_2' \uplus \{\epsilon\})$, $\eta \in \eta_1 \parallel \eta_2$ and $\eta' \uplus \{\epsilon\} \in \eta_1' \uplus \{\epsilon\} \parallel \eta_2' \uplus \{\epsilon\}$. Taking the same order of interleaving, $\eta' \uplus \{\nu\} \in \eta_1' \uplus \{\nu\} \parallel \eta_2' \uplus \{\nu\}$. Therefore, we have $\xi' \in \xi_1' \parallel \xi_2'$, where

$$\xi' \coloneqq \eta \langle \mu, \rho \uplus \{\nu\} \rangle (\eta' \uplus \{\nu\}), \quad \xi_1' \coloneqq \eta_1 \langle \mu, \rho \uplus \{\nu\} \rangle \eta_1' \,\overline{\uplus}\, \{\nu\}, \text{ and } \quad \xi_2' \coloneqq \eta_2(\eta_2' \uplus \{\nu\})$$

Define $\tau_2' \coloneqq \alpha_2 \boxed{\xi_2'} \omega_2 \therefore r_2$. Since $\tau_2 \xrightarrow{\mathsf{Ls}} \tau_2'$, indeed $\tau_2' \in P_2^{n_2}$. We take

$$\tau' \coloneqq \inf_{\xi'.\mathsf{o}} \{\alpha_1, \alpha_2\} \boxed{\xi'} \omega_1 \sqcup \omega_2 \therefore \langle r_1, r_2 \rangle$$

Since $\xi.\mathsf{o} = \xi'.\mathsf{o}$, we have $\tau' \xrightarrow{\mathsf{Ti}} \tau$.

Ab. Similar to Ti, using $\tau_2 \xrightarrow{\mathsf{Ex}} \tau_2'$.

Di. So $\tau_1' = \left(\alpha_1 \boxed{\eta_1 \langle \mu, \rho \uplus \{\nu\} \rangle \eta_1' \,\overline{\uplus}\, \{\nu\}} \omega_1 \therefore r\right) [\uparrow\epsilon]$, where $\xi = \eta_1 \langle \mu, \rho \uplus \{\nu, \epsilon\} \rangle \eta_1' \,\overline{\uplus}\, \{\nu, \epsilon\}$. The reasoning in this case proceeds similarly, using $\tau_2 \xrightarrow{\mathsf{Cn}} \tau_2'$ and interleaving τ_1' with τ_2' to take

$$\tau' \coloneqq \inf_{\xi'.\mathsf{o}} \{\alpha_1 [\uparrow\epsilon], \alpha_2 [\uparrow\epsilon]\} \boxed{\xi'} \omega_1 [\uparrow\epsilon] \sqcup \omega_2 [\uparrow\epsilon] \therefore \langle r_1, r_2 \rangle$$

We have $\xi.\mathsf{o} = \xi'.\mathsf{o}$ again too. Moreover, ξ is the chronicle of a trace, and ϵ appears in it. So no view that appears in the trace can point into the interior of ϵ's segment. Otherwise, since view must point to timestamps of messages, we would have a memory that is not scattered. We show $\inf_{\xi.\mathsf{o}} \{\alpha_1, \alpha_2\} [\uparrow\epsilon] \leq \inf_{\xi.\mathsf{o}} \{\alpha_1 [\uparrow\epsilon], \alpha_2 [\uparrow\epsilon]\}$. Indeed, for $\kappa \hookrightarrow \xi.\mathsf{o}$, assume $\kappa \leq \alpha_i$. Therefore, $\kappa [\uparrow\epsilon] \leq \alpha_i [\uparrow\epsilon]$, and so $\kappa [\uparrow\epsilon] \leq \inf_{\xi.\mathsf{o}} \{\alpha_1 [\uparrow\epsilon], \alpha_2 [\uparrow\epsilon]\}$. Thus in particular for $\kappa = \inf_{\xi.\mathsf{o}} \{\alpha_1, \alpha_2\}$.

By order-comparing $(\omega_i)_{\epsilon.\mathtt{1c}}$ to $\epsilon.\mathtt{i}$, one also finds that $(\omega_1 \sqcup \omega_2)\,[\uparrow\epsilon] = \omega_1\,[\uparrow\epsilon] \sqcup \omega_2\,[\uparrow\epsilon]$. And so we obtain $\tau' \xrightarrow{\mathsf{Rw}}\xrightarrow{\mathsf{Di}} \tau$. □

From here on we work to prove Retroactive Closure via a logical relation. To compensate for closures nested within higher-order constructions, we use a refined notion of equality.

Egli-Milner lifting. The *trace lifting* of a relation $\sim\, \subseteq X \times Y$ is a relation $\sim\, \subseteq \mathsf{Trace}X \times \mathsf{Trace}Y$ defined $\tau \sim \tau' \coloneqq \tau.\mathtt{st} = \tau'.\mathtt{st} \wedge \tau.\mathtt{vl} \sim \tau'.\mathtt{vl}$. This in turn lifts to the Egli-Milner relation $\sim\, \subseteq \mathcal{P}(\mathsf{Trace}X) \times \mathcal{P}(\mathsf{Trace}Y)$ where $U \sim E \coloneqq \forall \tau \in U\, \exists \tau' \in E.\, \tau \sim \tau' \wedge \forall \tau' \in E\, \exists \tau \in U.\, \tau \sim \tau'$. We call this last relation the *EM-trace lifting* of $\sim\, \subseteq X \times Y$.

Logical relation. For every type A we define $\mathcal{V}^{\dagger}\{\!|A|\!\} \subseteq [\![A]\!] \times [\![A]\!]_C$ and $\mathcal{E}^{\dagger}\{\!|A|\!\} \subseteq \underline{\mathcal{A}}\,[\![A]\!] \times \underline{C}[\![A]\!]_C$ by mutual recursion. We define $\mathcal{V}^{\dagger}\{\!|A|\!\}$ standardly, reciting "related-inputs to related-outputs":

$$\mathcal{V}^{\dagger}\{\!|A \to B|\!\} \coloneqq \left\{\langle f, g\rangle \,\middle|\, \forall \langle r, s\rangle \in \mathcal{V}^{\dagger}\{\!|A|\!\}\,.\, \langle fr, gs\rangle \in \mathcal{E}^{\dagger}\{\!|B|\!\}\right\}$$

$$\mathcal{V}^{\dagger}\{\!|(A_1 * \cdots * A_n)|\!\} \coloneqq \left\{\langle\langle r_1, \ldots, r_n\rangle, \langle s_1, \ldots, s_n\rangle\rangle \,\middle|\, \forall i.\, \langle r_i, s_i\rangle \in \mathcal{V}^{\dagger}\{\!|A_i|\!\}\right\}$$

$$\mathcal{V}^{\dagger}\{\!|\{\iota_1 \ \mathbf{of}\ A_1 \mid \cdots \mid \iota_n \ \mathbf{of}\ A_n\}|\!\} \coloneqq \bigcup_i \left\{\langle \iota_i\, r, \iota_i\, s\rangle \,\middle|\, \langle r, s\rangle \in \mathcal{V}^{\dagger}\{\!|A_i|\!\}\right\}$$

The relation trivializes on ground types: $\mathcal{V}^{\dagger}\{\!|G|\!\}$ is equality. In particular for $V, W \in \cdot \vdash G$, if $\left\langle [\![V]\!]^{\mathrm{v}}, [\![W]\!]^{\mathrm{v}}_C\right\rangle \in \mathcal{V}^{\dagger}\{\!|G|\!\}$, then $V = W$ because as program values, $[\![V]\!]^{\mathrm{v}} = V$ and $[\![W]\!]^{\mathrm{v}}_C = W$.

The bespoke $\mathcal{E}^{\dagger}\{\!|A|\!\} \coloneqq \left\{\langle P, Q\rangle \,\middle|\, \langle P, Q^{\mathfrak{a}}\rangle \in \mathcal{V}^{\dagger}\{\!|A|\!\}\right\}$ uses the EM-trace lifting of $\mathcal{V}^{\dagger}\{\!|A|\!\}$ to relate abstract denotations to generating denotations by nesting $\mathfrak{a}$-closures.

In regards to open terms, for every typing context Γ we define $\mathcal{X}^{\dagger}\{\!|\Gamma|\!\} \subseteq [\![\Gamma]\!] \times [\![\Gamma]\!]_C$ by:

$$\mathcal{X}^{\dagger}\{\!|\Gamma|\!\} \coloneqq \left\{\langle \gamma, \delta\rangle \,\middle|\, \forall (a : A) \in \Gamma.\, \langle \gamma a, \delta a\rangle \in \mathcal{V}^{\dagger}\{\!|A|\!\}\right\}$$

and define $\Gamma \vDash^{\dagger} M : A$ as follows: $\forall \langle \gamma, \delta\rangle \in \mathcal{X}^{\dagger}\{\!|\Gamma|\!\}\,.\, \left\langle [\![M]\!]^{\mathrm{c}}\,\gamma, [\![M]\!]^{\mathrm{c}}_C \delta\right\rangle \in \mathcal{E}^{\dagger}\{\!|A|\!\}$. We show this semantic judgment is sound with respect to the typing relation, following some supportive lemmas.

Lemma A.1. *If* $\langle r, s\rangle \in \mathcal{V}^{\dagger}\{\!|A|\!\}$ *then* $\left\langle \mathrm{return}\, r, \mathrm{return}^{C} s\right\rangle \in \mathcal{E}^{\dagger}\{\!|A|\!\}$.

Proof. For the first half of the EM-trace lifting, let $\tau \in \mathrm{return}\, r = \left(\mathrm{return}^{C} r\right)^{\mathfrak{a}}$, where we used Rewrite Castling to reorder the rewrites. So there exists $\pi \in \mathrm{return}^{C} r$ such that $\pi \xrightarrow{\mathfrak{a}} \tau$. Obtain τ', π' from τ, π respectively by replacing their return value r with s. By construction, $\langle \tau, \tau'\rangle \in \mathcal{V}^{\dagger}\{\!|A|\!\}$. Moreover, $\pi' \in \mathrm{return}^{C} s$. By reusing the rewrite sequence, $\pi' \xrightarrow{\mathfrak{a}} \tau'$. Therefore, $\tau' \in \left(\mathrm{return}^{C} s\right)^{\mathfrak{a}}$ is a witness as required.

The same idea in reverse shows the second half of the EM-trace lifting. □

Lemma A.2. *If* $\langle P, Q\rangle \in \mathcal{E}^{\dagger}\{\!|A|\!\}$ *and* $\langle f, g\rangle \in \mathcal{V}^{\dagger}\{\!|A \to B|\!\}$ *then* $\left\langle P \gg\!= f, Q \gg\!=^{C} g\right\rangle \in \mathcal{E}^{\dagger}\{\!|B|\!\}$.

Proof. For the first half of the EM-trace lifting, let $\tau \in P \gg\!= f = \left(P \gg\!=^{C} f\right)^{\mathfrak{a}}$, where we used Lemma 8.3 to reorder the rewrites. So there exists $\pi \in P \gg\!=^{C} f$ such that $\pi \xrightarrow{\mathfrak{a}} \tau$. So there exist $\alpha\boxed{\xi}\kappa \mathrel{.\!\raisebox{1ex}{.}\!.} r \in P$ and $\sigma\boxed{\eta}\omega \mathrel{.\!\raisebox{1ex}{.}\!.} s \in fr$ where $\kappa \leq \sigma$ such that $\pi = \alpha\boxed{\xi\eta}\omega \mathrel{.\!\raisebox{1ex}{.}\!.} s$.

- By the first assumption, there exists r' such that $\langle r, r'\rangle \in \mathcal{V}^{\dagger}\{\!|A|\!\}$ and $\alpha\boxed{\xi}\kappa \mathrel{.\!\raisebox{1ex}{.}\!.} r' \in Q^{\mathfrak{a}}$.
- By the second assumption, there exists s' such that $\langle s, s'\rangle \in \mathcal{V}^{\dagger}\{\!|B|\!\}$ and $\sigma\boxed{\eta}\omega \mathrel{.\!\raisebox{1ex}{.}\!.} s' \in (gr')^{\mathfrak{a}}$.

So $\pi' \coloneqq \alpha\boxed{\xi\eta}\omega \mathrel{.\!\raisebox{1ex}{.}\!.} s' \in Q^{\mathfrak{a}} \gg\!=^{C} g^{\mathfrak{a}}$. Obtain τ' from τ by replacing its return value s by s'. By reusing the rewrite sequence, $\pi' \xrightarrow{\mathfrak{a}} \tau'$. By Deferral of Closure, $\tau' \in \left(Q^{\mathfrak{a}} \gg\!=^{C} g^{\mathfrak{a}}\right)^{\mathfrak{a}} = \left(Q \gg\!=^{C} g\right)^{\mathfrak{a}}$.

The same idea in reverse shows the second half of the EM-trace lifting. □

Lemma A.3. $\left\langle [\![\text{store}_{\ell,v}]\!], [\![\text{store}_{\ell,v}]\!]_C \right\rangle \in \mathcal{E}^{\dagger}\{\!|\mathbf{1}|\!\}$ *and* $\left\langle [\![\text{rmw}_{\ell,\Phi}]\!], [\![\text{rmw}_{\ell,\Phi}]\!]_C \right\rangle \in \mathcal{E}^{\dagger}\{\!|\mathbf{Val}|\!\}$.

Proof. Since $\mathbf{1}$ and $\mathbf{Val}$ are ground types, the sets are equal by Deferral of Closure, reasoning as in Lemma A.1. □

Lemma A.4. *If* $\langle P_i, Q_i \rangle \in \mathcal{E}^{\dagger}\{\!|A_i|\!\}$ *then* $\left\langle P_1 \mid\mid\mid P_2, Q_1 \mid\mid\mid^C Q_2 \right\rangle \in \mathcal{E}^{\dagger}\{\!|(A_1 * A_2)|\!\}$.

Proof. Similar to Lemma A.2. □

Proposition A.5. *If* $\Gamma \vdash M : A$ *then* $\Gamma \vDash^{\dagger} M : A$.

Proof. By induction on the derivation of $\Gamma \vdash M : A$. We detail some paradigmatic examples:

$$\frac{\Gamma, a : A \vDash^{\dagger} M : B}{\Gamma \vDash^{\dagger} \lambda a : A.\, M : A \to B}$$

Let $\langle \gamma, \delta \rangle \in \mathcal{X}^{\dagger}\{\!|\Gamma|\!\}$. If $\langle r, s \rangle \in \mathcal{V}^{\dagger}\{\!|A|\!\}$, then $\langle \gamma[a \mapsto r], \delta[a \mapsto s] \rangle \in \mathcal{X}^{\dagger}\{\!|\Gamma, a : A|\!\}$. By assumption, $\left\langle [\![M]\!]^{\mathsf{c}}\, \gamma[a \mapsto r], [\![M]\!]^{\mathsf{c}}_C \delta[a \mapsto s] \right\rangle \in \mathcal{E}^{\dagger}\{\!|B|\!\}$. Therefore, $\left\langle \lambda r.\, [\![M]\!]^{\mathsf{c}}\, \gamma[a \mapsto r], \lambda s.\, [\![M]\!]^{\mathsf{c}}_C \delta[a \mapsto s] \right\rangle \in \mathcal{V}^{\dagger}\{\!|A \to B|\!\}$. Applying Lemma A.1, $\left\langle [\![\lambda a.\, M]\!]^{\mathsf{c}}\, \gamma, [\![\lambda a.\, M]\!]^{\mathsf{c}}_C \delta \right\rangle \in \mathcal{E}^{\dagger}\{\!|A \to B|\!\}$.

$$\frac{\Gamma \vDash^{\dagger} M : A \qquad \Gamma \vDash^{\dagger} N : A \to B}{\Gamma \vDash^{\dagger} NM : B}$$

Let $\langle \gamma, \delta \rangle \in \mathcal{X}^{\dagger}\{\!|\Gamma|\!\}$. If $\langle f, g \rangle \in \mathcal{V}^{\dagger}\{\!|A \to B|\!\}$, then by Lemma A.2 with the first assumption, $\left\langle [\![M]\!]^{\mathsf{c}}\, \gamma \gg\!= f, [\![M]\!]^{\mathsf{c}}_C \delta \gg\!=^C g \right\rangle \in \mathcal{E}^{\dagger}\{\!|B|\!\}$. Thus $\left\langle \lambda f.\, [\![M]\!]^{\mathsf{c}}\, \gamma \gg\!= f, \lambda g.\, [\![M]\!]^{\mathsf{c}}_C \delta \gg\!=^C g \right\rangle \in \mathcal{V}^{\dagger}\{\!|(A \to B) \to B|\!\}$. So by Lemma A.2 with the second assumption, $\left\langle [\![NM]\!]^{\mathsf{c}}\, \gamma, [\![NM]\!]^{\mathsf{c}}_C \delta \right\rangle \in \mathcal{E}^{\dagger}\{\!|B|\!\}$.

The other cases follow by similar reasoning with Lemmas A.1 and A.2, where in the cases of the effects we also use the respective Lemmas A.3 and A.4. □

Proof of Retroactive Closure. Since M is a program, by Proposition A.5, $\cdot \vDash M : G$ for some ground type G. That is, $\left\langle [\![M]\!]^{\mathsf{c}}, [\![M]\!]^{\mathsf{c}}_C \right\rangle \in \mathcal{E}^{\dagger}\{\!|A|\!\}$. Since the EM-trace lifting degenerates to equality on ground types, $[\![M]\!]^{\mathsf{c}} = [\![M]\!]^{\mathsf{c}\,\mathfrak{a}}_C$. □

B Proof of Directional Compositionality

We prove Directional Compositionality via logical relation. For this, we use a refinement of the notion of set-containment.

Hoare lifting. The *trace lifting* of a relation $\sim\ \subseteq X \times Y$ is a relation $\sim\ \subseteq \mathsf{Trace}X \times \mathsf{Trace}Y$ defined $\tau \sim \tau' \coloneqq \tau.\mathtt{st} = \tau'.\mathtt{st} \wedge \tau.\mathtt{vl} \sim \tau'.\mathtt{vl}$. This in turn lifts to the Hoare relation $\sim\ \subseteq \mathcal{P}(\mathsf{Trace}X) \times \mathcal{P}(\mathsf{Trace}Y)$ where $U \sim E \coloneqq \forall \tau \in U\, \exists \tau' \in E.\, \tau \sim \tau'$. We call this last relation the *H-trace lifting* of the first relation.

Logical relation. For every type A we define $\mathcal{V}^{\circ}\{\!|A|\!\} \subseteq [\![A]\!] \times [\![A]\!]$ and $\mathcal{E}^{\circ}\{\!|A|\!\} \subseteq \underline{\mathcal{A}}\,[\![A]\!] \times \underline{\mathcal{A}}\,[\![A]\!]$ by mutual recursion. We define $\mathcal{V}^{\circ}\{\!|A|\!\}$ standardly, reciting "related-inputs to related-outputs":

$$\mathcal{V}^{\circ}\{\!|A \to B|\!\} \coloneqq \{\langle f, g \rangle \mid \forall \langle r, s \rangle \in \mathcal{V}^{\circ}\{\!|A|\!\}\,.\, \langle fr, gs \rangle \in \mathcal{E}^{\circ}\{\!|B|\!\}\}$$
$$\mathcal{V}^{\circ}\{\!|(A_1 * \cdots * A_n)|\!\} \coloneqq \{\langle \langle r_1, \ldots, r_n \rangle, \langle s_1, \ldots, s_n \rangle \rangle \mid \forall i.\, \langle r_i, s_i \rangle \in \mathcal{V}^{\circ}\{\!|A_i|\!\}\}$$
$$\mathcal{V}^{\circ}\{\!|\{\iota_1 \ \mathbf{of}\ A_1 \mid \cdots \mid \iota_n \ \mathbf{of}\ A_n\}|\!\} \coloneqq \bigcup_i \{\langle \iota_i\, r, \iota_i\, s \rangle \mid \langle r, s \rangle \in \mathcal{V}^{\circ}\{\!|A_i|\!\}\}$$

The relation trivializes on ground types: $\mathcal{V}^{\circ}\{\!|G|\!\}$ is equality. In particular for $V, W \in \cdot \vdash G$, if $\left\langle [\![V]\!]^{\mathsf{v}}, [\![W]\!]^{\mathsf{v}} \right\rangle \in \mathcal{V}^{\circ}\{\!|G|\!\}$, then $V = W$ because as program values, $[\![V]\!]^{\mathsf{v}} = V$ and $[\![W]\!]^{\mathsf{v}} = W$. We H-trace lift $\mathcal{V}^{\circ}\{\!|A|\!\}$ to obtain $\mathcal{E}^{\circ}\{\!|A|\!\}$. It too trivializes on ground types: $\mathcal{E}^{\circ}\{\!|G|\!\}$ is containment. In regards to open terms, for every typing context Γ we define $\mathcal{X}^{\circ}\{\!|\Gamma|\!\} \subseteq [\![\Gamma]\!] \times [\![\Gamma]\!]$ by:

$$\mathcal{X}^{\circ}\{\!|\Gamma|\!\} \coloneqq \{\langle \gamma, \delta \rangle \mid \forall (a : A) \in \Gamma.\, \langle \gamma a, \delta a \rangle \in \mathcal{V}^{\circ}\{\!|A|\!\}\}$$

and define $\Gamma \vDash^{\circ} M \lesssim N : A$ as follows: $\forall \langle \gamma, \delta \rangle \in \mathcal{X}^{\circ}\{\!|\Gamma|\!\}\,.\, \left\langle [\![M]\!]^{\mathsf{c}}\, \gamma, [\![N]\!]^{\mathsf{c}}\, \delta \right\rangle \in \mathcal{E}^{\circ}\{\!|A|\!\}$.

As in §A, we have the same supportive lemmas for this logical relation. The proofs are similar, though slightly simpler because there is no need for Lemma 8.5.

Lemma B.1. *If* $\langle r, s\rangle \in \mathcal{V}^{\circ}\{\!|A|\!\}$ *then* $\langle \text{return}\, r, \text{return}\, s\rangle \in \mathcal{E}^{\circ}\{\!|A|\!\}$.

Lemma B.2. *If* $\langle P, Q\rangle \in \mathcal{E}^{\circ}\{\!|A|\!\}$ *and* $\langle f, g\rangle \in \mathcal{V}^{\circ}\{\!|A \to B|\!\}$ *then* $\langle P \gg\!= f, Q \gg\!= g\rangle \in \mathcal{E}^{\circ}\{\!|B|\!\}$.

Lemma B.3. $\left\langle [\![\text{store}_{\ell,v}]\!], [\![\text{store}_{\ell,v}]\!]\right\rangle \in \mathcal{E}^{\circ}\{\!|\mathbf{1}|\!\}$ *and* $\left\langle [\![\text{rmw}_{\ell,\Phi}]\!], [\![\text{rmw}_{\ell,\Phi}]\!]\right\rangle \in \mathcal{E}^{\circ}\{\!|\mathbf{Val}|\!\}$.

Lemma B.4. *If* $\langle P_i, Q_i\rangle \in \mathcal{E}^{\circ}\{\!|A_i|\!\}$ *then* $\langle P_1 \mid\mid\mid P_2, Q_1 \mid\mid\mid Q_2\rangle \in \mathcal{E}^{\circ}\{\!|(A_1 * A_2)|\!\}$.

The judgment is closed under term contexts:

Lemma B.5. *For* $\Delta \vdash \Xi\,[\Gamma \vdash - : A] : B$, *if* $\Gamma \vDash^{\circ} M \lesssim N : A$, *then* $\Delta \vDash^{\circ} \Xi\,[M] \lesssim \Xi\,[N] : B$.

Proof. By induction on the derivation of $\Delta \vdash \Xi\,[\Gamma \vdash - : A] : B$. The metavariable case holds by assumption. The rest uses the supportive lemmas Lemmas B.1 to B.4 as in the proof of Proposition A.5. □

Proposition B.6. *If* $\underline{\mathcal{A}}\,[\![A]\!] \ni P' \subseteq P$ *and* $\langle P, Q\rangle \in \mathcal{E}^{\circ}\{\!|A|\!\}$ *then* $\langle P', Q\rangle \in \mathcal{E}^{\circ}\{\!|A|\!\}$.

Proof. Assuming a statement about all elements of P we deduce the same statement about all elements of P'. □

Lemma B.7. *For* $\Gamma \vdash M, N : A$, *if* $[\![M]\!]^{\mathsf{c}} \subseteq [\![N]\!]^{\mathsf{c}}$ *then* $\Gamma \vDash^{\circ} M \lesssim N : A$.

Proof. Let $\langle \gamma, \delta\rangle \in \mathcal{X}^{\circ}\{\!|\Gamma|\!\}$. By Lemma B.5 with N itself as the context (the degenerate case with no metavariable appearance), $\left\langle [\![N]\!]^{\mathsf{c}}\, \gamma, [\![N]\!]^{\mathsf{c}}\, \delta\right\rangle \in \mathcal{E}^{\circ}\{\!|A|\!\}$. By assumption, $[\![M]\!]^{\mathsf{c}}\, \gamma \subseteq [\![N]\!]^{\mathsf{c}}\, \gamma$. So by Proposition B.6, $\left\langle [\![M]\!]^{\mathsf{c}}\, \gamma, [\![N]\!]^{\mathsf{c}}\, \delta\right\rangle \in \mathcal{E}^{\circ}\{\!|A|\!\}$. □

Proof of Theorem 8.11. By Lemma B.7, we have $\Gamma \vDash^{\circ} M \lesssim N : A$. Thanks to Lemma B.5, we have $\cdot \vDash^{\circ} \Xi\,[M] \lesssim \Xi\,[N] : G$. That is, $\left\langle [\![\Xi\,[M]]\!]^{\mathsf{c}}, [\![\Xi\,[N]]\!]^{\mathsf{c}}\right\rangle \in \mathcal{E}^{\circ}\{\!|G|\!\}$. Since G is ground, this degenerates to $[\![\Xi\,[M]]\!]^{\mathsf{c}} \subseteq [\![\Xi\,[N]]\!]^{\mathsf{c}}$. □

C Proof of Soundness

To enable optimizations, the abstract model decouples traces far enough from the operational semantics to make it non-trivial to prove Soundness. To overcome this challenge we use a logical relation to relate the abstract model to a model which corresponds tightly to the execution steps of the operational semantics, by tracking the initial view-tree and the memory accesses individually. Formally, for a set X, an *X-run-trace* is an element of $\mathsf{VTree} \times \mathsf{Mem} \times \mathsf{Chro} \times \mathsf{View} \times X$, written $\tau = \langle T, \mu\rangle\, \boxed{\xi}\, \omega \therefore r$. We denote the set of X-run-traces by $\mathsf{OpTrace}X$.

As in traces, the run-trace's main component is its chronicle $\tau.\mathtt{ch} = \xi$, with transitions consisting of well-formed memories. Here, each transition represents a *single memory-accessing step* during the interrupted execution; i.e. those labeled by $\bullet$. We call such steps *loud*, and the other step *silent*; i.e. those labeled by $\circ$. Respectively, the run-trace is *silent* if ξ is empty, otherwise it is *loud*.

The run-trace's initial state is $\langle T, \mu\rangle$. This represents the state from the execution's initial configuration, so we require that $T \hookrightarrow \mu$. However, the environment may add messages before the term starts running, so in the loud case we only require $\mu \subseteq \xi.\mathtt{o}$.

The run-trace's final view is $\tau.\mathtt{fvw} = \omega$. The corresponding interrupted execution ends with $\dot{\omega}$. In the silent case we require $T = \dot{\omega}$ since silent steps do not change the state. As a derived notion, the run-trace's final state is $\langle \dot{\omega}, (\langle \mu, \mu\rangle\, \xi).\mathtt{c}\rangle$, so we require $\omega \hookrightarrow \langle \mu, \mu\rangle\, \xi.\mathtt{c}$.

In light of Lemma 6.12, we require moreover that $\kappa \leq \omega$ for every $\kappa \in T.\mathtt{lf}$, denote by $T \leq \omega$. Moreover, considering Lemma 6.11, we require $\forall\, \nu \in \xi.\mathtt{own}\, \exists\, \alpha \in T.\mathtt{lf}.\, \alpha \leq \nu.\mathtt{vw} \leq \omega \wedge \alpha_{\nu.\mathtt{lc}} < \nu.\mathtt{t}$.

Finally, the run-trace's return value is $\tau.\mathtt{ret} = r$. This corresponds to the program value the interrupted execution returns.

We define a monad structure $\mathcal{R}X := \left\langle \underline{\mathcal{R}}X, \text{return}^{\mathcal{R}}, \gg\!=^{\mathcal{R}} \right\rangle$:

$$\underline{\mathcal{R}}X := \mathcal{P}\,(\mathsf{OpTrace}X) \qquad \text{return}^{\mathcal{R}} r := \{\langle \dot\kappa, \mu\rangle \,\boxed{\cdot}\, \kappa \mathrel{.\!\cdot\!.} r\}$$

$$P \gg\!=^{\mathcal{R}} f := \left\{ \langle T, \mu\rangle \,\boxed{\xi\eta}\, \omega \mathrel{.\!\cdot\!.} s \in \underline{\mathcal{R}}Y \,\middle|\, \exists r, \kappa.\, \langle T, \mu\rangle \,\boxed{\xi}\, \kappa \mathrel{.\!\cdot\!.} r \in P \wedge \langle \dot\kappa, (\langle \mu, \mu\rangle\, \xi).\mathsf{c}\rangle \,\boxed{\eta}\, \omega \mathrel{.\!\cdot\!.} s \in fr \right\}$$

In the return operator, we make sure that the initial and final states are equal. In the bind operator, we make sure that the final state of the first run-trace is the initial state of the second run-trace.

Proposition C.1. *$\mathcal{R}$ is a monad.*

Next we extend $\mathcal{R}$ with shared-memory constructs.

Concurrent execution. Consider a program $M \parallel N$. Either the state has a leaf $\dot\kappa$ as its view-tree, in which case the first step it takes has to be ParInit, or it has a node as its view tree $T \frown R$, in which case the first step it takes cannot be ParInit. Either way, it then takes some steps due to steps of M and N (with ParLeft and ParRight), then finally it steps with ParFin to synchronize.

$$P_1 \,|||^{\mathcal{R}}\, P_2 := \left\{ \begin{matrix} \langle T, \mu\rangle \,\boxed{\xi}\, \omega_1 \sqcup \omega_2 \mathrel{.\!\cdot\!.} \langle r_1, r_2\rangle \in \underline{\mathcal{R}}\,(X_1 \times X_2) \;\mid\; \exists\, \xi_1, \xi_2.\, \xi \in \xi_1 \parallel \xi_2 \wedge \exists\, T_1, T_2. \\ \left(\forall i \in \{1, 2\}\,.\, \langle T_i, \mu\rangle \,\boxed{\xi_i}\, \omega_i \mathrel{.\!\cdot\!.} r_i \in P_i\right) \wedge \left(T = T_1 \frown T_2 \vee \exists\, \kappa.\, T = T_1 = T_2 = \dot\kappa\right) \end{matrix} \right\}$$

Memory access. The definitions follow the Store, ReadOnly, and RMW rules:

$$[\![\text{store}_{\ell,v}]\!]_{\mathcal{R}} := \left\{ \langle \dot\alpha, \mu\rangle \,\boxed{\langle \rho, \rho \uplus \{\ell{:}v@(q, t)\langle\!\langle \alpha[\ell{\mapsto}t]\rangle\!\rangle\}\rangle}\, \alpha[\ell{\mapsto}t] \mathrel{.\!\cdot\!.} \langle\rangle \in \underline{\mathcal{R}}\mathbf{1} \right\}$$

$$[\![\text{rmw}^{\text{RO}}_{\ell,\Phi}]\!]_{\mathcal{R}} := \left\{ \langle \dot\alpha, \mu\rangle \,\boxed{\langle \rho, \rho\rangle}\, \alpha \sqcup \kappa \mathrel{.\!\cdot\!.} v \in \underline{\mathcal{R}}\mathsf{Val} \,\middle|\, \Phi v = \bot, \ell{:}v@(-, \kappa_\ell]\langle\!\langle \kappa\rangle\!\rangle \in \rho, \alpha_\ell \le \kappa_\ell \right\}$$

$$[\![\text{rmw}^{\text{RMW}}_{\ell,\Phi}]\!]_{\mathcal{R}} := \left\{ \begin{matrix} \langle \dot\alpha, \mu\rangle \,\boxed{\langle \rho, \rho \uplus \{\ell{:}\Phi v@(\kappa_\ell, t]\langle\!\langle \omega\rangle\!\rangle\}\rangle}\, \omega \mathrel{.\!\cdot\!.} v \in \underline{\mathcal{R}}\mathsf{Val} \\ \mid\, \omega = (\alpha \sqcup \kappa)\,[\ell{\mapsto}t], \ell{:}v@(-, t]\langle\!\langle \kappa\rangle\!\rangle \in \rho \end{matrix} \right\}$$

$$[\![\text{rmw}_{\ell,\Phi}]\!]_{\mathcal{R}} := [\![\text{rmw}^{\text{RO}}_{\ell,\Phi}]\!]_{\mathcal{R}} \cup [\![\text{rmw}^{\text{RMW}}_{\ell,\Phi}]\!]_{\mathcal{R}}$$

Some of the premises of the corresponding rules appear as conditions in the set notations, while other do not appear because they hold implicitly due to the requirements on run-traces.

The importance of a run-trace's initial memory is in making sense of the initial view-tree, even if the chronicle is empty. In particular, messages unseen by the initial view-tree are redundant:

Lemma C.2. *If $\langle T, \mu'\rangle \,\boxed{\xi}\, \omega \mathrel{.\!\cdot\!.} r \in [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}$ and $T \hookrightarrow \mu \subseteq \mu'$ then $\langle T, \mu\rangle \,\boxed{\xi}\, \omega \mathrel{.\!\cdot\!.} r \in [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}$.*

Single-step soundness. To make the relationship between these denotations and the operational semantics precise, we can follow an execution backwards, adding a transition for every $\bullet$-step:

Lemma C.3. *Assume $\langle T, \mu\rangle, M \overset{e}{\rightsquigarrow}_{RA} \langle T', \mu'\rangle, M'$ and $\langle T', \mu'\rangle \,\boxed{\xi}\, \omega \mathrel{.\!\cdot\!.} r \in [\![M']\!]^{\mathsf{c}}_{\mathcal{R}}$.*

- *If $e = \circ$, then $\langle T, \mu\rangle \,\boxed{\xi}\, \omega \mathrel{.\!\cdot\!.} r \in [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}$.*
- *If $e = \bullet$, then $\langle T, \mu\rangle \,\boxed{\langle \mu, \mu'\rangle\, \xi}\, \omega \mathrel{.\!\cdot\!.} r \in [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}$.*

Proof. By induction on the derivation of $\langle T, \mu\rangle, M \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T', \mu'\rangle, M'$. Paradigmatic examples:

App Assume $\langle \dot\kappa, \mu\rangle, (\lambda a.\, M)\, V \overset{\circ}{\rightsquigarrow}_{\text{RA}} \langle \dot\kappa, \mu\rangle, M[a{\mapsto}V]$ and $\tau := \langle \dot\kappa, \mu\rangle \,\boxed{\xi}\, \omega \mathrel{.\!\cdot\!.} r \in [\![M[a{\mapsto}V]]\!]^{\mathsf{c}}_{\mathcal{R}}$. By the Substitution Lemma, $[\![M[a{\mapsto}V]]\!]^{\mathsf{c}}_{\mathcal{R}} = [\![(\lambda a.\, M)\, V]\!]^{\mathsf{c}}_{\mathcal{R}}$. So indeed $\tau \in [\![(\lambda a.\, M)\, V]\!]^{\mathsf{c}}_{\mathcal{R}}$.

ParLeft Assume $\left\langle T \frown R, \mu\right\rangle, M \parallel N \overset{\bullet}{\rightsquigarrow}_{\text{RA}} \left\langle T' \frown R, \mu'\right\rangle, M' \parallel N$ and $\left\langle T' \frown R, \mu'\right\rangle \,\boxed{\xi}\, \omega \mathrel{.\!\cdot\!.} \langle r, s\rangle \in [\![M' \parallel N]\!]^{\mathsf{c}}_{\mathcal{R}}$. So there exist ξ_1, ξ_2 such that $\xi \in \xi_1 \parallel \xi_2$, and there exist ω_1, ω_2 where $\omega =$

$\omega_1 \sqcup \omega_2$ such that $\langle T', \mu' \rangle \boxed{\xi_1} \omega_1 \therefore r \in [\![M']\!]^{\mathsf{c}}_{\mathcal{R}}$ and $\langle R, \mu' \rangle \boxed{\xi_2} \omega_2 \therefore s \in [\![N]\!]^{\mathsf{c}}_{\mathcal{R}}$. In that latter we can replace μ' with μ using Lemma C.2. By the induction hypothesis and the former, $\langle T, \mu \rangle \boxed{\langle \mu, \mu' \rangle \xi_1} \omega_1 \therefore r \in [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}$. Since $\langle \mu, \mu' \rangle \xi \in \langle \mu, \mu' \rangle \xi_1 \parallel \xi_2$, we have $\left\langle T \frown R, \mu \right\rangle \boxed{\langle \mu, \mu' \rangle \xi} \omega \therefore \langle r, s \rangle \in [\![M \parallel N]\!]^{\mathsf{c}}_{\mathcal{R}}$. ∎

We say a chronicle ξ is *gapless* if $\rho = \rho'$ whenever $\langle \mu, \rho \rangle$ is followed by $\langle \rho', \theta \rangle$ in ξ. Traces that feature gapless chronicles can be mumble-rewritten to obtain single-transition traces.

Proposition C.4. *Assume $\langle T, \mu \rangle, M \rightsquigarrow^{*}_{RA} \langle \dot{\omega}, \mu' \rangle, V$. Let $\eta \coloneqq \langle \mu, \mu \rangle \xi$. Assume η is gapless and $\eta.\mathsf{c} = \mu'$. In other words, either (i) ξ is empty and $\mu = \mu'$; or (ii) $\xi.\mathsf{o} = \mu$, $\xi.\mathsf{c} = \mu'$, and ξ is gapless. Then $\langle T, \mu \rangle \boxed{\xi} \omega \therefore [\![V]\!]^{\mathsf{v}}_{\mathcal{R}} \in [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}$.*

Proof. By induction on the number of small-steps. Case (i) applies so long as all the steps so far are silent. Case (ii) applies otherwise. ∎

Hoare run-lifting. The *run-trace lifting* of a relation $\sim \subseteq X \times Y$ is a relation $\sim \subseteq \mathsf{OpTrace}X \times \mathsf{Trace}Y$ defined $\tau \sim \tau' \coloneqq \exists T, \mu, \xi, \omega, r, s. \tau = \langle T, \mu \rangle \boxed{\xi} \omega \therefore r \wedge \tau' = \inf_\mu T \boxed{\langle \mu, \mu \rangle \xi} \omega \therefore s \wedge r \sim s$. This in turn lifts to the Hoare relation $\sim \subseteq \mathcal{P}(\mathsf{OpTrace}X) \times \mathcal{P}(\mathsf{Trace}Y)$ where $U \sim E \coloneqq \forall \tau \in U \, \exists \tau' \in E. \tau \sim \tau'$. We call this last relation the *H-run-trace lifting* of the first relation.

Logical relation. For every type A, define $\mathcal{V}^*\{\!|A|\!\} \subseteq [\![A]\!]_{\mathcal{R}} \times [\![A]\!]_{C}$ and $\mathcal{E}^*\{\!|A|\!\} \subseteq \underline{\mathcal{R}}[\![A]\!]_{\mathcal{R}} \times \underline{C}[\![A]\!]_{C}$ by mutual recursion. The definition of $\mathcal{V}^*\{\!|A|\!\}$ follows the standard "related-inputs to related-outputs" mantra, while the bespoke $\mathcal{E}^*\{\!|A|\!\}$ part transforms the view tree to its greatest lower bound using the notation $\inf_\mu T \coloneqq \inf_\mu T.\mathtt{lf}$, and adds a transition for the first memory:

$$\mathcal{V}^*\{\!|A \to B|\!\} \coloneqq \{\langle f, g \rangle \mid \forall \langle r, s \rangle \in \mathcal{V}^*\{\!|A|\!\} . \langle fr, gs \rangle \in \mathcal{E}^*\{\!|B|\!\}\}$$
$$\mathcal{V}^*\{\!|(A_1 * \cdots * A_n)|\!\} \coloneqq \{\langle \langle r_1, \ldots, r_n \rangle, \langle s_1, \ldots, s_n \rangle \rangle \mid \forall i. \langle r_i, s_i \rangle \in \mathcal{V}^*\{\!|A_i|\!\}\}$$
$$\mathcal{V}^*\{\!|\{\iota_1 \textbf{ of } A_1 \mid \cdots \mid \iota_n \textbf{ of } A_n\}|\!\} \coloneqq \textstyle\bigcup_i \{\langle \iota_i \, r, \iota_i \, s \rangle \mid \langle r, s \rangle \in \mathcal{V}^*\{\!|A_i|\!\}\}$$

The relation trivializes on ground types: $\mathcal{V}^*\{\!|G|\!\}$ is equality. In particular for $V, W \in \cdot \vdash G$, if $\langle V, W \rangle \in \mathcal{V}^*\{\!|G|\!\}$, then $V = W$ because as ground-typed values, $[\![V]\!]^{\mathsf{v}}_{\mathcal{R}} = V$ and $[\![W]\!]^{\mathsf{v}}_{C} = W$. We H-trace lift $\mathcal{V}^*\{\!|A|\!\}$ to obtain $\mathcal{E}^*\{\!|A|\!\}$.

In regards to open terms, for every typing context Γ we define $\mathcal{X}^*\{\!|\Gamma|\!\} \subseteq [\![\Gamma]\!]_{\mathcal{R}} \times [\![\Gamma]\!]_{C}$ by:

$$\mathcal{X}^*\{\!|\Gamma|\!\} \coloneqq \{\langle \gamma, \delta \rangle \mid \forall (a : A) \in \Gamma. \langle \gamma a, \delta a \rangle \in \mathcal{V}^*\{\!|A|\!\}\}$$

and define $\Gamma \vDash^* M : A$ as follows: $\forall \langle \gamma, \delta \rangle \in \mathcal{X}^*\{\!|\Gamma|\!\} . \left\langle [\![M]\!]^{\mathsf{c}}_{\mathcal{R}} \gamma, [\![M]\!]^{\mathsf{c}}_{C} \delta \right\rangle \in \mathcal{E}^*\{\!|A|\!\}$. We show this semantic judgment is sound with respect to the typing relation, following some supportive lemmas.

Lemma C.5. *If $\langle r, s \rangle \in \mathcal{V}^*\{\!|A|\!\}$ then $\left\langle \mathrm{return}^{\mathcal{R}} r, \mathrm{return}\, s \right\rangle \in \mathcal{E}^*\{\!|A|\!\}$.*

Proof. Assume $\langle r, s \rangle \in \mathcal{V}^*\{\!|A|\!\}$. W.l.o.g., let $\langle \dot{\kappa}, \mu \rangle \boxdot \kappa \therefore r \in \mathrm{return}^{\mathcal{R}} r$, where $\kappa \hookrightarrow \mu$. Note that $\kappa \boxed{\langle \mu, \mu \rangle} \kappa \therefore s \in \mathrm{return}\, s$. Trivially, $\langle \mu, \mu \rangle \cdot = \langle \mu, \mu \rangle$ and $\inf_\mu \dot{\kappa} = \kappa$. Substituting these, together with our assumption, we obtain the required precisely:

$$\forall \langle \dot{\kappa}, \mu \rangle \boxdot \kappa \therefore r \in \mathrm{return}^{\mathcal{R}} r \, \exists s. \inf_\mu \dot{\kappa} \boxed{\langle \mu, \mu \rangle \cdot} \kappa \therefore s \in \mathrm{return}\, s \wedge \langle r, s \rangle \in \mathcal{V}^*\{\!|A|\!\}$$ ∎

Lemma C.6. *If $\langle P, Q \rangle \in \mathcal{E}^*\{\!|A|\!\}$ and $\langle f, g \rangle \in \mathcal{V}^*\{\!|A \to B|\!\}$ then $\left\langle P \gg\!=^{\mathcal{R}} f, Q \gg\!= g \right\rangle \in \mathcal{E}^*\{\!|B|\!\}$.*

Proof. Assume $\langle P, Q \rangle \in \mathcal{E}^*\{\!|A|\!\}$ and $\langle f, g \rangle \in \mathcal{V}^*\{\!|A \to B|\!\}$. Let $\langle T, \mu \rangle \boxed{\xi \eta} \omega \therefore r' \in P \gg\!=^{\mathcal{R}} f$. So there exist r and κ such that $\langle T, \mu \rangle \boxed{\xi} \kappa \therefore r \in P$ and $\langle \dot{\kappa}, (\langle \mu, \mu \rangle \xi).\mathsf{c} \rangle \boxed{\eta} \omega \therefore r' \in fr$.

By the first assumption there exists an s such that $\inf_\mu T \boxed{\langle\mu,\mu\rangle\ \xi} \kappa \therefore s \in Q$ and $\langle r, s\rangle \in \mathcal{V}^*\{\!|A|\!\}$. Using the second assumption we find that $\langle fr, gs\rangle \in \mathcal{E}^*\{\!|B|\!\}$. In particular, there exists an s' such that $\kappa \boxed{\langle((\langle\mu,\mu\rangle\ \xi).\mathsf{c}, (\langle\mu,\mu\rangle\ \xi).\mathsf{c}\rangle\ \eta} \omega \therefore s' \in gs$ and $\langle r', s'\rangle \in \mathcal{V}^*\{\!|B|\!\}$. So we have

$$\inf_\mu T \boxed{\langle\mu,\mu\rangle\ \xi\ \langle((\langle\mu,\mu\rangle\ \xi).\mathsf{c}, (\langle\mu,\mu\rangle\ \xi).\mathsf{c}\rangle\ \eta} \omega \therefore s' \in Q \rangle\!\!\rangle\!\!= g$$

By using mumble, we have the required $\inf_\mu T \boxed{\langle\mu,\mu\rangle\ \xi\eta} \omega \therefore s' \in Q \rangle\!\!\rangle\!\!= g$. □

Lemma C.7. $\left\langle [\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{R}}, [\![\mathrm{store}_{\ell,v}]\!]_{C}\right\rangle \in \mathcal{E}^*\{\!|\mathbf{1}|\!\}$ *and* $\left\langle [\![\mathrm{rmw}_{\ell,\Phi}]\!]_{\mathcal{R}}, [\![\mathrm{rmw}_{\ell,\Phi}]\!]_{C}\right\rangle \in \mathcal{E}^*\{\!|\mathbf{Val}|\!\}$.

Proof. Using stutter to compensate for the additional transition from the initial memory, and rewind to compensate for the view not necessarily already pointing to the loaded message. □

Lemma C.8. *If* $\langle P_i, Q_i\rangle \in \mathcal{E}^*\{\!|A_i|\!\}$ *then* $\langle P_1 \mid\mid\mid P_2, Q_1 \mid\mid\mid Q_2\rangle \in \mathcal{E}^*\{\!|(A_1 * A_2)|\!\}$.

Proof. Assume $\langle P_i, Q_i\rangle \in \mathcal{E}^*\{\!|A_i|\!\}$, and let $\tau \in P_1 \mid\mid\mid P_2$. We proceed by case analysis depending on whether the initial view tree is a leaf:

Leaf. W.l.o.g., $\tau = \langle\dot{\kappa},\mu\rangle \boxed{\xi} \omega_1 \sqcup \omega_2 \therefore \langle r_1, r_2\rangle$, where $\langle\dot{\kappa},\mu\rangle \boxed{\xi_i} \omega_i \therefore r_i \in P_i$ and $\xi \in \xi_1 \parallel \xi_2$. So there exist s_i such that $\kappa \boxed{\langle\mu,\mu\rangle\ \xi_i} \omega_i \therefore s_i \in Q_i$ and $\langle r_i, s_i\rangle \in \mathcal{V}^*\{\!|A_i|\!\}$. By definition, $\kappa \boxed{\langle\mu,\mu\rangle\ \langle\mu,\mu\rangle\ \xi} \omega_1 \sqcup \omega_2 \therefore \langle s_1, s_2\rangle \in Q_1 \mid\mid\mid^{\mathcal{G}} Q_2$. Using mumble we have $\kappa \boxed{\langle\mu,\mu\rangle\ \xi} \omega_1 \sqcup \omega_2 \therefore \langle s_1, s_2\rangle \in Q_1 \mid\mid\mid Q_2$. Since $\langle\langle r_1, r_2\rangle, \langle s_1, s_2\rangle\rangle \in \mathcal{V}^*\{\!|(A_1 * A_2)|\!\}$, we are done.

Node. W.l.o.g., $\tau = \left\langle T_1 \frown T_2, \mu\right\rangle \boxed{\xi} \omega_1 \sqcup \omega_2 \therefore \langle r_1, r_2\rangle$, where $\langle T_i, \mu\rangle \boxed{\xi_i} \omega_i \therefore r_i \in P_i$ and $\xi \in \xi_1 \parallel \xi_2$. So there exist s_i such that $\inf_\mu T_i \boxed{\langle\mu,\mu\rangle\ \xi_i} \omega_i \therefore s_i \in Q_i$ and $\langle r_i, s_i\rangle \in \mathcal{V}^*\{\!|A_i|\!\}$. Rudimentarily, $\inf_\mu \left\{\inf_\mu T_1, \inf_\mu T_2\right\} = \inf_\mu \left(T_1 \frown T_2\right)$, so $\inf_\mu \left(T_1 \frown T_2\right) \boxed{\langle\mu,\mu\rangle\ \langle\mu,\mu\rangle\ \xi} \omega_1 \sqcup \omega_2 \therefore \langle s_1, s_2\rangle \in Q_1 \mid\mid\mid^{\mathcal{G}} Q_2$. The rest is like before. □

Proposition C.9. *If* $\Gamma \vdash M : A$ *then* $\Gamma \vDash^* M : A$.

Proof. By induction on the derivation of $\Gamma \vdash M : A$. We detail some paradigmatic examples:

$$\frac{\Gamma, a : A \vDash^* M : B}{\Gamma \vDash^* \lambda a : A.\, M : A \to B}$$

Let $\langle\gamma,\delta\rangle \in \mathcal{X}^*\{\!|\Gamma|\!\}$. We have $[\![\lambda a.\, M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma = \mathrm{return}^{\mathcal{R}} \lambda r.\, [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma[a{\mapsto}r]$ and $[\![\lambda a.\, M]\!]^{\mathsf{c}}_{C}\delta = \mathrm{return}\, \lambda s.\, [\![M]\!]^{\mathsf{c}}_{C}\delta[a{\mapsto}s]$ by definition. By Lemma C.5 and the definition of $\mathcal{V}^*\{\!|A \to B|\!\}$, it suffices to show that if $\langle r, s\rangle \in \mathcal{V}^*\{\!|A|\!\}$, then $\left\langle [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma[a{\mapsto}r], [\![M]\!]^{\mathsf{c}}_{C}\delta[a{\mapsto}s]\right\rangle \in \mathcal{E}^*\{\!|B|\!\}$, which is implied by the induction hypothesis.

$$\frac{\Gamma \vDash^* M : A \qquad \Gamma \vDash^* N : A \to B}{\Gamma \vDash^* NM : B}$$

Let $\langle\gamma,\delta\rangle \in \mathcal{X}^*\{\!|\Gamma|\!\}$. By definition, $[\![NM]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma = [\![N]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma \rangle\!\!\rangle\!\!=^{\mathcal{R}} \lambda f.\, [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma \rangle\!\!\rangle\!\!=^{\mathcal{R}} f$, and $[\![NM]\!]^{\mathsf{c}}_{C}\delta = [\![N]\!]^{\mathsf{c}}_{C}\delta \rangle\!\!\rangle\!\!= \lambda g.\, [\![M]\!]^{\mathsf{c}}_{C}\delta \rangle\!\!\rangle\!\!= g$. By the first induction hypothesis, $\left\langle [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma, [\![M]\!]^{\mathsf{c}}_{C}\delta\right\rangle \in \mathcal{E}^*\{\!|A|\!\}$. So by Lemma C.6, if $\langle f, g\rangle \in \mathcal{V}^*\{\!|A \to B|\!\}$ then $\left\langle [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma \rangle\!\!\rangle\!\!=^{\mathcal{R}} f, [\![M]\!]^{\mathsf{c}}_{C}\delta \rangle\!\!\rangle\!\!= g\right\rangle \in \mathcal{E}^*\{\!|B|\!\}$. But this is exactly the definition of $\left\langle \lambda f.\, [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma \rangle\!\!\rangle\!\!=^{\mathcal{R}} f, \lambda g.\, [\![M]\!]^{\mathsf{c}}_{C}\delta \rangle\!\!\rangle\!\!= g\right\rangle \in \mathcal{V}^*\{\!|(A \to B) \to B|\!\}$.

By the second induction hypothesis, $\left\langle [\![N]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma, [\![N]\!]^{\mathsf{c}}_{C}\delta\right\rangle \in \mathcal{E}^*\{\!|A \to B|\!\}$. Using Lemma C.6 again, we have $\left\langle [\![N]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma \rangle\!\!\rangle\!\!=^{\mathcal{R}} \lambda f.\, [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}\gamma \rangle\!\!\rangle\!\!=^{\mathcal{R}} f, [\![N]\!]^{\mathsf{c}}_{C}\delta \rangle\!\!\rangle\!\!= \lambda g.\, [\![M]\!]^{\mathsf{c}}_{C}\delta \rangle\!\!\rangle\!\!= g\right\rangle \in \mathcal{E}^*\{\!|B|\!\}$, as required.

The other cases follow by similar reasoning with Lemmas C.5 and C.6, where in the cases of the effects we also use the respective Lemmas C.7 and C.8. □

The proof of soundness concludes by using Propositions C.4 and C.9:

Proof of Soundness. We have $\langle T, \mu\rangle \boxed{\xi} \omega \therefore V \in [\![M]\!]^{\mathsf{c}}_{\mathcal{R}}$ by Proposition C.4 since M is of ground type. Therefore, Proposition C.9 implies that $\inf_\mu T \boxed{\langle\mu,\mu\rangle\ \xi} \omega \therefore V \in [\![M]\!]^{\mathsf{c}}_{C}$. Thanks to the extra conclusions of Proposition C.4, $\inf_\mu T \boxed{\langle\mu,\mu'\rangle} \omega \therefore V \in [\![M]\!]^{\mathsf{c}}_{C}$ by iteratively mumble-rewriting. □

D Proof of Adequacy

The proof of adequacy starts with the Fundamental Lemma, stating that C-traces correspond to interrupted executions, generalizing the Concrete Traces Lemma. The main reason behind this fact is simple: $\mathfrak{c}$-rewrites preserve this correspondence. That is:

Lemma D.1. *If $M : \tau : V$ and $\tau \xrightarrow{\mathsf{x}} \pi$ for $\mathsf{x} \in \mathfrak{c}$, then $M : \pi : V$.*

Proof. We split to the different $\mathsf{x} \in \mathfrak{c}$ cases:

St Add a transition that doesn't change the configuration.
Mu Meld adjacent transitions with equal configurations at the boundary.
Fw Append an Adv step to the final transition.
Rw Prepend an Adv step to the initial transition. ◻

Logical relation. We mutually define, indexed over type A, sets $\mathcal{V}\{\!|A|\!\}$ of closed values of type A and sets $\mathcal{E}\{\!|A|\!\}$ of closed terms of type A:

$$\mathcal{V}\{\!|A \to B|\!\} \coloneqq \{\lambda a.\, M \mid \forall V \in \mathcal{V}\{\!|A|\!\}\,.\, M[a{\mapsto}V] \in \mathcal{E}\{\!|B|\!\}\}$$
$$\mathcal{V}\{\!|(A_1 * \cdots * A_n)|\!\} \coloneqq \{\langle V_1, \ldots, V_n\rangle \mid \forall i.\, V_i \in \mathcal{V}\{\!|A_i|\!\}\}$$
$$\mathcal{V}\{\!|\{\iota_1 \ \mathbf{of}\ A_1 \,\mathbf{|}\, \cdots \,\mathbf{|}\, \iota_n \ \mathbf{of}\ A_n\}|\!\} \coloneqq \bigcup_i \{\iota_i\, V \mid V \in \mathcal{V}\{\!|A_i|\!\}\}$$
$$\mathcal{E}\{\!|A|\!\} \coloneqq \left\{M \in \cdot \vdash A \;\middle|\; \forall \tau \in [\![M]\!]^{\mathsf{c}}_{C} \exists V \in \mathcal{V}\{\!|A|\!\}\,.\, M : \tau : V\right\}$$

In regards to open terms, for every typing context Γ we define

$$\mathcal{X}\{\!|\Gamma|\!\} \coloneqq \{\Theta \in \mathsf{Sub}_{\Gamma} \mid \forall (a : A) \in \Gamma.\, \Theta_a \in \mathcal{V}\{\!|A|\!\}\}$$

and define $\Gamma \vDash M : A$ for $\Gamma \vdash M : A$ as $\forall \Theta \in \mathcal{X}\{\!|\Gamma|\!\}\,.\, \Theta M \in \mathcal{E}\{\!|A|\!\}$.

Theorem D.2 (Fundamental Lemma). *If $\Gamma \vdash M : A$, then $\Gamma \vDash M : A$.*

We devote lemmas to inductive cases of the Fundamental Lemma's proof.

Lemma D.3. *If $\tau \coloneqq \alpha \boxed{\xi} \omega \therefore \langle\rangle \in [\![\mathrm{store}_{\ell,v}]\!]_C$, then $\ell \coloneqq v : \tau : \langle\rangle$.*

Proof. W.l.o.g. $\tau \in [\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{G}}$, because the general case then follows from Lemma D.1. Thus, the interrupted execution is just a single Store step. Indeed, the states $\langle \dot{\alpha}, \xi.\mathsf{o}\rangle$ and $\langle \dot{\omega}, \xi.\mathsf{c}\rangle$ match those in Store's conclusion. The conditions of Store are met thanks to τ being a trace, e.g. the segment of the stored message being unoccupied due to $\xi.\mathsf{c}$ being well-formed. ◻

Lemma D.4. *If $\tau \coloneqq \alpha \boxed{\xi} \omega \therefore v \in [\![\mathrm{rmw}_{\ell,\varphi_{\vec{w}}}]\!]_C$, then $\mathbf{rmw}_{\varphi}\,(\ell; \vec{w}) : \tau : v$.*

Proof. W.l.o.g. $\tau \in [\![\mathrm{rmw}_{\ell,\varphi_{\vec{w}}}]\!]_{\mathcal{G}}$, because the general case then follows from Lemma D.1. Thus, the interrupted execution is a single ReadOnly step (if $\tau \in [\![\mathrm{rmw}^{\mathrm{RO}}_{\ell,\varphi_{\vec{w}}}]\!]_{\mathcal{G}}$) or a single RMW step (if $\tau \in [\![\mathrm{rmw}^{\mathrm{RMW}}_{\ell,\varphi_{\vec{w}}}]\!]_{\mathcal{G}}$), in which the initial view points to the loaded message. ◻

Lemma D.5. *If $\xi \in \xi_1 \parallel \xi_2$ and $M_i : \alpha_i \boxed{\xi_i} \omega_i \therefore r_i : V_i$, then*

$$M_1 \parallel M_2 : \inf_{\xi.\mathsf{o}} \{\alpha_1, \alpha_2\} \boxed{\xi} \sup_{\xi.\mathsf{c}} \{\omega_1, \omega_2\} \therefore \langle r_1, r_2\rangle : \langle V_1, V_2\rangle$$

Proof. We obtain the required interrupted execution by interleaving the interrupted executions following the interleaving that generated ξ from ξ_1 and ξ_2 with the following modifications:

- prepending Adv—lifted using ParLeft/ParRight—to the first transition taken by each side;
- prepending ParInit to the first transition;
- appending ParFin to the last transition (since $\sup_{\xi.\mathsf{c}} \{\omega_1, \omega_2\} = \omega_1 \sqcup \omega_2$). ◻

Proof of the Fundamental Lemma. By induction on the typing derivation $\Gamma \vdash M : A$.

$$\frac{(a : A) \in \Gamma}{\Gamma \vDash a : A}$$

Let $\Theta \in \mathsf{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{\!|A|\!\}$. To show $\Theta a = \Theta_a \in \mathcal{E}\{\!|A|\!\}$, let $\tau \in [\![\Theta_a]\!]^{\mathsf{c}}_C = \mathrm{return}^{C}[\![\Theta_a]\!]^{\mathsf{v}}_C$. Suffice it we show $\Theta_a : \tau : \Theta_a$. Using Lemma D.1, we restrict to the case of $\tau \in \mathrm{return}^{\mathcal{G}}[\![\Theta_a]\!]^{\mathsf{v}}_C$. So τ is of the form $\kappa \boxed{\langle \mu, \mu \rangle} \kappa \therefore [\![\Theta_a]\!]^{\mathsf{v}}_C$. The required interrupted execution is obtained by taking no steps in its only transition.

$$\frac{\Gamma, a : A \vDash M : B}{\Gamma \vDash \lambda a : A. M : A \to B}$$

Let $\Theta \in \mathsf{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{\!|A|\!\}$. Denote by K the term $\Theta\,(\lambda a : A. M) = \lambda a : A.\ \Theta|_{\notin\{a\}} M$. To show that $K \in \mathcal{E}\{\!|A|\!\}$, let $\tau \in [\![K]\!]^{\mathsf{c}}_C = \mathrm{return}^{C}[\![K]\!]^{\mathsf{v}}_C$. Like the previous case, we can show $K : \tau : K$ using Lemma D.1. This is sufficient, because $K \in \mathcal{V}\{\!|A \to B|\!\}$. Indeed, for $V \in \mathcal{V}\{\!|A|\!\}$, denoting by $\Theta[V/a]$ the substitution equal to Θ except at V which it maps to a, by the induction hypothesis we have $\left(\Theta|_{\notin\{a\}} M\right)[a \mapsto V] = \Theta[V/a]M \in \mathcal{E}\{\!|B|\!\}$.

$$\frac{\Gamma \vDash M : A \qquad \Gamma \vDash N : A \to B}{\Gamma \vDash NM : B}$$

Let $\Theta \in \mathsf{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{\!|A|\!\}$. To show that $\Theta(NM) = (\Theta N)\,(\Theta M) \in \mathcal{E}\{\!|A|\!\}$ holds, let $\tau \in [\![(\Theta N)\,(\Theta M)]\!]^{\mathsf{c}}_C = [\![\Theta N]\!]^{\mathsf{c}}_C \mathbin{\rangle\!\!\rangle\!=^{C}} \lambda f. [\![\Theta M]\!]^{\mathsf{c}}_C \mathbin{\rangle\!\!\rangle\!=^{C}} f$. Unfolding:

$$\exists\, \tau_1 \coloneqq \alpha_1 \boxed{\xi_1} \omega_1 \therefore f \in [\![\Theta N]\!]^{\mathsf{c}}_C, \tau_2 \coloneqq \alpha_2 \boxed{\xi_2} \omega_2 \therefore r \in [\![\Theta M]\!]^{\mathsf{c}}_C, \tau_3 \coloneqq \alpha_3 \boxed{\xi_3} \omega_3 \therefore s \in fr.$$
$$\omega_1 \le \alpha_2 \wedge \omega_2 \le \alpha_3 \wedge \tau = \alpha_1 \boxed{\xi_1 \xi_2 \xi_3} \omega_3 \therefore s \in [\![(\Theta N)\,(\Theta M)]\!]^{\mathsf{c}}_C$$

By the induction hypotheses, there exists $\lambda a : A. K \in \mathcal{V}\{\!|A \to B|\!\}$ such that $\Theta N : \tau_1 : \lambda a : A. K$, and there exists $V \in \mathcal{V}\{\!|A|\!\}$ such that $\Theta M : \tau_2 : V$. So $K[a \mapsto V] \in \mathcal{E}\{\!|B|\!\}$, and using the Substitution Lemma: $fr = [\![\lambda a : A. K]\!]^{\mathsf{v}}_C [\![V]\!]^{\mathsf{v}}_C = [\![K[a \mapsto V]]\!]^{\mathsf{c}}_C$. Therefore, there exists $W \in \mathcal{V}\{\!|B|\!\}$ such that $K[a \mapsto V] : \tau_3 : W$. We transform to sequence the interrupted executions into one that corresponds to τ as follows: we lift the one corresponding to τ_1 using AppLeft to the context $[-]\,(\Theta M)$, we lift the one corresponding to τ_2 using AppRight to the context $(\lambda a : A. K)\,[-]$, and we prepend App to the one corresponding to τ_3. By using Adv to compensate for the difference in delimiting views, we get $(\Theta N)\,(\Theta M) : \tau : W$.

$$\frac{\Gamma \vDash M : \mathbf{Loc} \qquad \Gamma \vDash N : \mathbf{Val}}{\Gamma \vDash M \coloneqq N : \mathbf{1}} \qquad \frac{\varphi \in \mathsf{RMW}_n \qquad \Gamma \vDash M : \mathbf{Loc} \qquad \Gamma \vDash N : \mathbf{Val}^n}{\Gamma \vDash \mathbf{rmw}_\varphi\,(M; N) : \mathbf{Val}}$$

Binds unfold like in the case above. The rest is handled using Lemma D.3 and Lemma D.4 respectively.

$$\frac{\Gamma \vDash M_1 : A_1 \qquad \Gamma \vDash M_2 : A_2}{\Gamma \vDash M_1 \parallel M_2 : (A_1 * A_2)}$$

Let $\Theta \in \mathsf{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{\!|A|\!\}$. Thanks to Lemma D.1, to show $\Theta(M_1 \parallel M_2) = \Theta M_1 \parallel \Theta M_2 \in \mathcal{E}\{\!|A_1 * A_2|\!\}$, it is sufficient to consider $\tau \in [\![\Theta M_1]\!]^{\mathsf{c}}_C \mathbin{|||^{\mathcal{G}}} [\![\Theta M_2]\!]^{\mathsf{c}}_C$. Unfolding the concurrent construct, there exist $\tau_i \coloneqq \alpha_i \boxed{\xi_i} \omega_i \therefore r_i \in [\![\Theta M_i]\!]^{\mathsf{c}}_C$ and $\xi \in \xi_1 \parallel \xi_2$ such that $\tau \coloneqq \inf_{\xi.\mathsf{o}} \{\alpha_1, \alpha_2\} \boxed{\xi} \sup_{\xi.\mathsf{c}} \{\omega_1, \omega_2\} \therefore \langle r_1, r_2 \rangle$. By induction hypotheses, there exist $V_i \in \mathcal{V}\{\!|A_i|\!\}$ such that $\Theta M_i : \tau_i : V_i$. So $\langle V_1, V_2 \rangle \in \mathcal{V}\{\!|A_1 * A_2|\!\}$, and by Lemma D.5, $\Theta M_1 \parallel \Theta M_2 : \tau : \langle V_1, V_2 \rangle$.

The other cases are treated similarly. □

We obtain Concrete Traces Lemma by specifying the Fundamental Lemma to ground types. To prove the Evaluation Lemma, we observe that $\mathfrak{a}$-rewrites preserve evaluation:

Lemma D.6. *For* $\mathsf{x} \in \mathfrak{a}$, *if* $\pi \xrightarrow{\mathsf{x}} \tau$ *and* $\langle \pi.\dot{\mathtt{i}}\mathtt{vw}, \pi.\mathtt{ch.o} \rangle, M \Downarrow \pi.\mathtt{vl}$, *then* $\langle \tau.\dot{\mathtt{i}}\mathtt{vw}, \tau.\mathtt{ch.o} \rangle, M \Downarrow \tau.\mathtt{vl}$.

Proof. We have $\tau.\mathtt{vl} = \pi.\mathtt{vl}$ because the closure rules all preserve the return value. If $\mathsf{x} = \mathsf{Ti}$ or $\mathsf{x} = \mathsf{Ab}$, then $\tau.\mathtt{ivw} = \pi.\mathtt{ivw}$ and $\tau.\mathtt{ch.o} = \tau.\mathtt{ch.o}$, so the claim holds trivially. Only $\mathsf{x} = \mathsf{Di}$ remains, where $\pi.\mathtt{ivw} = \tau.\mathtt{ivw}\,[\uparrow\epsilon]$ and $\pi.\mathtt{ch.o} = \tau.\mathtt{ch.o}\,[\uparrow\epsilon]$ for some message ϵ. We obtain the required execution underlying $\langle \tau.\dot{\mathtt{i}}\mathtt{vw}, \tau.\mathtt{ch.o} \rangle, M \Downarrow \tau.\mathtt{vl}$ from the one that underlies $\langle \pi.\dot{\mathtt{i}}\mathtt{vw}, \pi.\mathtt{ch.o} \rangle, M \Downarrow$

π.vl by replacing the timestamp ϵ.i with ϵ.t everywhere. A straightforward simulation argument that justifies this. □

Proof of the Evaluation Lemma. Denote $\tau \coloneqq \alpha \boxed{\langle\mu,\rho\rangle} \omega \therefore r \in [\![M]\!]^{\mathfrak{c}}$. By Retroactive Closure, $[\![M]\!]^{\mathfrak{c}} = [\![M]\!]_C^{\mathfrak{c}\,\mathfrak{a}}$. So there exists $\pi \in [\![M]\!]_C^{\mathfrak{c}}$ such that $\pi \xrightarrow{\mathfrak{a}} \tau$. Proceed by induction on the number of $\mathfrak{a}$-rewrites. If none, $\tau = \pi \in [\![M]\!]_C^{\mathfrak{c}}$, so by the Fundamental Lemma, $M : \tau : V$ for some V. Since M is of ground type, so is $V = [\![V]\!]_C^{\mathsf{v}} = r$, and thus $\langle\dot\alpha,\mu\rangle, M \rightsquigarrow^*_{\mathrm{RA}_\le} \langle\dot\omega,\rho\rangle, r$, so $\langle\dot\alpha,\mu\rangle, M \Downarrow r$.

Otherwise, we have $\pi \xrightarrow{\mathfrak{a}} \tau' \xrightarrow{\mathsf{x}} \tau$ where $\mathsf{x} \in \mathfrak{a}$ and $\langle \tau'.\mathtt{ivw}, \tau'.\mathtt{ch.o}\rangle, M \Downarrow \tau'.\mathtt{vl}$ by the induction hypothesis. We replace τ' with τ using Lemma D.6, as required. □

E Validating Transformations

In the following we prove selected results from Table 3. We explicitly mention the use of $\mathfrak{a}$-rewrites, but often leave uses of $\mathfrak{c}$-rewrites implicit. For convenience, we denote $\alpha \boxed{\xi\eta} \omega \therefore s \coloneqq (\alpha \boxed{\xi} \kappa \therefore r) \ggg\!= (\sigma \boxed{\eta} \omega \therefore s)$, and we say this trace resulted from binding the first with the second.

Proposition E.1. *If $\Gamma \vdash M_1 : A_1$; $\Gamma \vdash N_1 : B_1$; $\Gamma, a : A' \vdash M_2 : A_2$; and $\Gamma, b : B' \vdash N_2 : B_2$:*

$$[\![(\mathbf{let}\, a = M_1\, \mathbf{in}\, M_2) \parallel (\mathbf{let}\, b = N_1\, \mathbf{in}\, N_2)]\!]^{\mathfrak{c}} \supseteq [\![\mathbf{match}\, M_1 \parallel N_1\, \mathbf{with}\, \langle a,b\rangle.\, M_2 \parallel N_2]\!]^{\mathfrak{c}}$$

(Formally, in the right denotation we use $\Gamma, a : A', b : B' \vdash M_2 : A_2$ and $\Gamma, a : A', b : B' \vdash N_2 : B_2$.)

Proof. Let $\gamma \in [\![\Gamma]\!]$ and instantiate the denotations with this context, denoting the resulting sets P and Q. Thus we require $P \supseteq Q$.

Let $\varrho \in Q$. By Deferral of Closure, ϱ is in the $\mathfrak{ca}$-closure of:

$$Q' \coloneqq [\![M_1]\!]^{\mathfrak{c}}\gamma \mid\mid\mid^{\mathcal{G}} [\![N_1]\!]^{\mathfrak{c}}\gamma \ggg\!=^{\mathcal{G}} \lambda\langle\gamma_a,\gamma_b\rangle.\ \ [\![M_2]\!]^{\mathfrak{c}}(\gamma_c)_{(c:C)\in\Gamma,a:A'} \mid\mid\mid^{\mathcal{G}}\ \ [\![N_2]\!]^{\mathfrak{c}}(\gamma_c)_{(c:C)\in\Gamma,b:B'}$$

So there exists $\varrho' \in Q'$ that $\mathfrak{ca}$-rewrites to ϱ. This ϱ' results from binding two traces. On the left, $\inf_{\xi.\mathrm{o}}\{\alpha_1,\kappa_1\} \boxed{\xi} \omega_1 \sqcup \sigma_1 \therefore \langle r_1, s_1\rangle$, where:

$$\tau_1 \coloneqq \alpha_1 \boxed{\xi_1} \omega_1 \therefore r_1 \in [\![M_1]\!]^{\mathfrak{c}}\gamma;\quad \pi_1 \coloneqq \kappa_1 \boxed{\eta_1} \sigma_1 \therefore s_1 \in [\![N_1]\!]^{\mathfrak{c}}\gamma;\quad \xi \in \xi_1 \parallel \eta_1$$

On the right, $\inf_{\eta.\mathrm{o}}\{\alpha_2,\kappa_2\} \boxed{\eta} \omega_2 \sqcup \sigma_2 \therefore \langle r_2, s_2\rangle$, where, setting $\gamma_a \coloneqq r_1$ and $\gamma_b \coloneqq s_1$:

$$\tau_2 \coloneqq \alpha_2 \boxed{\xi_2} \omega_2 \therefore r_2 \in [\![M_2]\!]^{\mathfrak{c}}(\gamma_c)_{(c:C)\in\Gamma,a:A'};\quad \pi_2 \coloneqq \kappa_2 \boxed{\eta_2} \sigma_2 \therefore s_2 \in [\![N_2]\!]^{\mathfrak{c}}(\gamma_c)_{(c:C)\in\Gamma,b:B'};\quad \eta \in \xi_2 \parallel \eta_2$$

The binding implies that $\omega_1 \sqcup \sigma_1 \le \inf_{\eta.\mathrm{o}}\{\alpha_2,\kappa_2\}$. In particular, $\omega_1 \le \alpha_2$ and $\sigma_1 \le \kappa_2$. Therefore, $\tau_1 \ggg\!= \tau_2 = \alpha_1 \boxed{\xi_1\xi_2} \omega_2 \therefore r_2 \in [\![\mathbf{let}\, a = M_1\, \mathbf{in}\, M_2]\!]^{\mathfrak{c}}$ and $\pi_1 \ggg\!= \pi_2 = \kappa_1 \boxed{\eta_1\eta_2} \sigma_2 \therefore s_2 \in [\![\mathbf{let}\, b = N_1\, \mathbf{in}\, N_2]\!]^{\mathfrak{c}}$. Since $\xi\eta \in \xi_1\xi_2 \parallel \eta_1\eta_2$ and $(\xi\eta).\mathrm{o} = \xi.\mathrm{o}$, we obtain ϱ' by interleaving these. Therefore, $\varrho' \in P$. Since P is $\mathfrak{ca}$-closed, $\varrho \in P$. □

In the rest of this section we show results of the form $[\![M]\!]^{\mathfrak{c}} \supseteq [\![N]\!]_{\mathcal{G}}^{\mathfrak{c}}$. Each entails $[\![M]\!]^{\mathfrak{c}} \supseteq [\![N]\!]^{\mathfrak{c}}$ by Deferral of Closure, thus justifying $M \twoheadrightarrow N$ in Table 3.

Proposition E.2. $[\![\langle\rangle]\!]^{\mathfrak{c}} \supseteq [\![\ell?\,;\langle\rangle]\!]_{\mathcal{G}}^{\mathfrak{c}}$.

Proof. Let $\tau \in [\![\ell?\,;\langle\rangle]\!]_{\mathcal{G}}^{\mathfrak{c}}$. Unfolding definitions:

$$[\![\ell?\,;\langle\rangle]\!]_{\mathcal{G}}^{\mathfrak{c}} \coloneqq [\![\mathrm{rmw}_{\ell,\lambda\_.\bot}]\!]_{\mathcal{G}} \ggg\!=^{\mathcal{G}} \lambda\_.\,\mathrm{return}^{\mathcal{G}}\langle\rangle = \left\{\alpha \boxed{\langle\mu,\mu\rangle\,\langle\rho,\rho\rangle} \omega \therefore \langle\rangle \in \mathsf{Trace1} \,\middle|\, \exists v \in \mu_\ell.\, \alpha \rightarrowtail v\right\}$$

Therefore, we have the form $\tau = \alpha \boxed{\langle\mu,\mu\rangle\,\langle\rho,\rho\rangle} \omega \therefore \langle\rangle$. From $\alpha \boxed{\langle\mu,\mu\rangle} \alpha \therefore \langle\rangle \in [\![\langle\rangle]\!]_{\mathcal{G}}^{\mathfrak{c}}$, we obtain $\tau \in [\![\langle\rangle]\!]^{\mathfrak{c}}$ by stuttering (St) and forwarding (Fw). □

Proposition E.3. $[\![\ell?\,;\langle\rangle]\!]^{\mathfrak{c}} \supseteq [\![\langle\rangle]\!]_{\mathcal{G}}^{\mathfrak{c}}$.

Proof. Let $\kappa \boxed{\langle\mu,\mu\rangle} \kappa \therefore \langle\rangle \in [\![\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}}$. Since the trace is well-formed, $\kappa \hookrightarrow \mu$. In particular, there exists a message $\nu \in \mu$ such that $\kappa \rightarrowtail \nu$. Therefore, $\kappa \boxed{\langle\mu,\mu\rangle} \kappa \therefore \nu.\mathtt{vl} \in [\![\ell?]\!]^{\mathsf{c}}_{\mathcal{G}}$, and so:

$$(\kappa \boxed{\langle\mu,\mu\rangle} \kappa \therefore \nu.\mathtt{vl}) \rangle\!\!\rangle\!= (\kappa \boxed{\langle\mu,\mu\rangle} \kappa \therefore \langle\rangle) = \kappa \boxed{\langle\mu,\mu\rangle\,\langle\mu,\mu\rangle} \kappa \therefore \langle\rangle \in [\![\ell?\,;\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}}$$

We obtain $\kappa \boxed{\langle\mu,\mu\rangle} \kappa \therefore \langle\rangle \in [\![\ell?\,;\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}}$ by mumbling (Mu). ☐

Proposition E.4. $[\![\ell \coloneqq v]\!]^{\mathsf{c}} \supseteq [\![\mathrm{XCHG}\,(\ell, v)\,;\langle\rangle]\!]^{\mathsf{c}}_{\mathcal{G}}$.

Proof. By taking the traces in $[\![\ell \coloneqq v]\!]^{\mathsf{c}}_{\mathcal{G}}$ in which the newly added message dovetails after the previous message in memory by choosing the initial timestamp appropriately. ☐

Proposition E.5. *Assuming* $\ell \neq \ell'$, $\quad [\![\langle \ell \coloneqq v, \ell'?\rangle]\!]^{\mathsf{c}} \supseteq [\![\ell \coloneqq v \parallel \ell'?]\!]^{\mathsf{c}}_{\mathcal{G}}$.

Proof. The elements of $[\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{G}} \,|||^{\mathcal{G}}\, [\![\mathrm{rmw}_{\ell',\lambda\_.\bot}]\!]_{\mathcal{G}}$ are formed by interleaving a store

$$\kappa \boxed{\langle\mu, \mu \uplus \{\ell{:}v@(q,t]\langle\!\langle\kappa[\ell{\mapsto}t]\rangle\!\rangle\}\rangle} \kappa[\ell{\mapsto}t] \therefore \langle\rangle \in [\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{G}}$$

with a load $\sigma \boxed{\langle\rho,\rho\rangle} \sigma \therefore w \in [\![\mathrm{rmw}_{\ell',\lambda\_.\bot}]\!]_{\mathcal{G}}$. Depending on the order, this results in one of:

$$\inf_\mu\{\kappa,\sigma\} \boxed{\langle\mu, \mu \uplus \{\ell{:}v@(q,t]\langle\!\langle\kappa[\ell{\mapsto}t]\rangle\!\rangle\}\rangle\,\langle\rho,\rho\rangle} \kappa[\ell{\mapsto}t] \sqcup \sigma \therefore \langle\langle\rangle, w\rangle \tag{WR}$$

$$\inf_\rho\{\kappa,\sigma\} \boxed{\langle\rho,\rho\rangle\,\langle\mu, \mu \uplus \{\ell{:}v@(q,t]\langle\!\langle\kappa[\ell{\mapsto}t]\rangle\!\rangle\}\rangle} \kappa[\ell{\mapsto}t] \sqcup \sigma \therefore \langle\langle\rangle, w\rangle \tag{RW}$$

We prove separately that these interleavings are in $[\![\langle \ell \coloneqq v, \ell'?\rangle]\!]^{\mathsf{c}}$.

- (WR): Denoting $\theta \coloneqq (\rho \setminus \{\ell{:}v@(q,t]\langle\!\langle\kappa[\ell{\mapsto}t]\rangle\!\rangle\}) \uplus \{\ell{:}v@(q,t]\langle\!\langle\alpha[\ell{\mapsto}t]\rangle\!\rangle\}$ where $\alpha \coloneqq \inf_\mu\{\kappa,\sigma\}$:

$$\alpha \boxed{\langle\mu, \mu \uplus \{\ell{:}v@(q,t]\langle\!\langle\alpha[\ell{\mapsto}t]\rangle\!\rangle\}\rangle} \alpha[\ell{\mapsto}t] \therefore \langle\rangle \in [\![\mathrm{store}_{\ell,v}]\!]_{\mathcal{G}}$$
$$\alpha[\ell{\mapsto}t] \sqcup \sigma \boxed{\langle\theta,\theta\rangle} \alpha[\ell{\mapsto}t] \sqcup \sigma \therefore w \in [\![\mathrm{rmw}_{\ell',\lambda\_.\bot}]\!]_{\mathcal{G}}$$

  By forwarding (Fw) after binding we obtain:

$$\alpha \boxed{\langle\mu, \mu \uplus \{\ell{:}v@(q,t]\langle\!\langle\alpha[\ell{\mapsto}t]\rangle\!\rangle\}\rangle\,\langle\theta,\theta\rangle} \kappa[\ell{\mapsto}t] \sqcup \sigma \therefore \langle\langle\rangle, w\rangle \in [\![\langle \ell \coloneqq v, \ell'?\rangle]\!]^{\mathsf{c}}$$

  All that remains is to tighten (Ti) $\ell{:}v@(q,t]\langle\!\langle\alpha[\ell{\mapsto}t]\rangle\!\rangle$ to $\ell{:}v@(q,t]\langle\!\langle\kappa[\ell{\mapsto}t]\rangle\!\rangle$.
- (RW): Using the result for (WR), with $\theta \coloneqq \mu \uplus \{\ell{:}v@(q,t]\langle\!\langle\kappa[\ell{\mapsto}t]\rangle\!\rangle\}$:

$$\inf_\mu\{\kappa,\sigma\} \boxed{\langle\mu,\theta\rangle\,\langle\theta,\theta\rangle} \kappa[\ell{\mapsto}t] \sqcup \sigma \therefore \langle\langle\rangle, w\rangle \in [\![\langle \ell \coloneqq v, \ell'?\rangle]\!]^{\mathsf{c}}$$

  We can rewind (Rw) $\inf_\mu\{\kappa,\sigma\}$ to $\inf_\rho\{\kappa,\sigma\}$, since $\rho \subseteq \mu$. By mumbling (Mu) and stuttering (St), we are done. ☐

Proposition E.6. *Assuming* $\varphi_{\vec{v}} \leq \psi_{\vec{w}}$, $\quad \left[\!\left[\mathbf{rmw}_\varphi(\ell;\vec{v})\right]\!\right]^{\mathsf{c}} \supseteq \left[\!\left[\mathbf{rmw}_\psi(\ell;\vec{w})\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$.

Proof. Let $\tau \in \left[\!\left[\mathbf{rmw}_\psi(\ell;\vec{w})\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$, resulting from loading a value u from a message ν. If $\varphi_{\vec{v}}u = \psi_{\vec{w}}u$, then obviously $\tau \in \left[\!\left[\mathbf{rmw}_\varphi(\ell;\vec{v})\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$. Otherwise, by assumption $\varphi_{\vec{v}}u = \bot$ and $\psi_{\vec{w}}u = u$. So we have $\tau = \kappa \boxed{\langle\mu, \mu \uplus \{\epsilon\}\rangle} \kappa[\ell{\mapsto}t] \therefore u$, with $\epsilon \coloneqq \ell{:}u@(\kappa_\ell, t]\langle\!\langle\kappa[\ell{\mapsto}t]\rangle\!\rangle$, where $\nu \in \mu_\ell$ and $\nu.\mathtt{t} = \kappa_\ell$. In the left denotation, by loading $\nu[\uparrow\epsilon]$ we have $\kappa[\uparrow\epsilon] \boxed{\langle\mu[\uparrow\epsilon], \mu[\uparrow\epsilon]\rangle} \kappa[\uparrow\epsilon] \therefore u = \left(\kappa \boxed{\langle\mu,\mu\rangle} \kappa[\ell{\mapsto}t] \therefore u\right)[\uparrow\epsilon]$. By diluting (Di) we obtain τ. ☐

Proposition E.7. *Assuming* $\forall v' \in \mathsf{Val}.\ \zeta_{\vec{u}}v' = \left(\psi_{\vec{w}v'} \circ^{\mathrm{id}} \varphi_{\vec{v}}\right) v'$,

$$\left[\!\left[\mathbf{let}\ a = \mathbf{rmw}_\varphi(\ell;\vec{v})\ \mathbf{in}\ \left\langle a, \mathbf{rmw}_\psi(\ell;\vec{w}a)\right\rangle\right]\!\right]^{\mathsf{c}} \supseteq \left[\!\left[\mathbf{let}\ a = \mathbf{rmw}_\zeta(\ell;\vec{u})\ \mathbf{in}\ \left\langle a, \varphi^{\mathrm{id}}_{\vec{v}}a\right\rangle\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$$

Proof. Let $\pi \in \left[\!\left[\textbf{let } a = \textbf{rmw}_{\zeta}\ (\ell;\vec{u})\ \textbf{in}\ \left\langle a, \varphi^{\mathrm{id}}_{\vec{v}} a\right\rangle\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$. So a $\tau' \coloneqq \alpha\boxed{\langle\mu,\rho\rangle}\omega .\!\cdot\!. v' \in \left[\!\left[\textbf{rmw}_{\zeta}\ (\ell;\vec{u})\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$ exists due to loading $\nu \in \mu_\ell$ with $\nu.\mathtt{vl} = v'$, such that $\tau \coloneqq \alpha\boxed{\langle\mu,\rho\rangle}\omega \therefore \left\langle v', \varphi^{\mathrm{id}}_{\vec{v}} v'\right\rangle \xrightarrow{\mathsf{St}}\xrightarrow{\mathsf{Fw}} \pi$.

RO. If $\tau' \in \left[\!\left[\mathrm{rmw}^{\mathrm{RO}}_{\ell,\zeta_{\vec{u}}}\right]\!\right]_{\mathcal{G}}$, then we have $\tau = \kappa\boxed{\langle\mu,\mu\rangle}\kappa \therefore \left\langle v', \varphi^{\mathrm{id}}_{\vec{v}} v'\right\rangle$ where $\zeta_{\vec{u}} v' = \bot$ and $\nu.\mathtt{t} = \kappa_\ell$. By assumption, $\varphi_{\vec{v}} v' = \bot$, so $\varphi^{\mathrm{id}}_{\vec{v}} v' = v'$; and $\psi_{\vec{w}v'} v' = \bot$, so by loading ν in both RMWs we can obtain τ in the left denotation.

RMW. If $\tau' \in \left[\!\left[\mathrm{rmw}^{\mathrm{RMW}}_{\ell,\zeta_{\vec{u}}}\right]\!\right]_{\mathcal{G}}$, then we have $\tau = \kappa\boxed{\langle\mu,\mu \uplus \{\ell{:}u'@(\kappa_\ell,t]\langle\!\langle\kappa[\ell\mapsto t]\rangle\!\rangle\}\rangle}\kappa[\ell\mapsto t] \therefore \left\langle v', \varphi^{\mathrm{id}}_{\vec{v}} v'\right\rangle$ where $\zeta_{\vec{u}} v' = u'$ and $\nu.\mathtt{t} = \kappa_\ell$. The $\varphi_{\vec{v}} v' = \bot$ case is similar to before, where again τ is found in the left denotation by loading ν in both RMWs, with the difference that here $\psi_{\vec{w}v'} v' = u'$, to the second RMW also writes the message $\ell{:}u'@(\kappa_\ell,t]\langle\!\langle\kappa[\ell\mapsto t]\rangle\!\rangle$.
The $\varphi_{\vec{v}} v' = w'$ case remains, in which $\varphi^{\mathrm{id}}_{\vec{v}} v' = w'$. In the sub-case that $\psi_{\vec{v}} w' = \bot$ we have $w' = u'$, and we find τ in the left denotation by loading ν and writing $\ell{:}u'@(\kappa_\ell,t]\langle\!\langle\kappa[\ell\mapsto t]\rangle\!\rangle$ in the first RMW, which the second RMW loads.
In the sub-case where $\psi_{\vec{v}} w' = u'$, the first RMW writes $\ell{:}w'@(\kappa_\ell, \frac{\kappa_\ell+t}{2}]\left\langle\!\left\langle\kappa[\ell\mapsto\frac{\kappa_\ell+t}{2}]\right\rangle\!\right\rangle$ instead. For the second RMW we take a trace with initial view $\kappa[\ell\mapsto\frac{\kappa_\ell+t}{2}]$, enabling its loading of this new message and writing $\ell{:}u'@(\frac{\kappa_\ell+t}{2},t]\langle\!\langle\kappa[\ell\mapsto t]\rangle\!\rangle$. To find τ in the left denotation we have the latter message absorb (Ab) the former message.

Either way, τ is in $\left[\!\left[\textbf{let } a = \textbf{rmw}_{\varphi}\ (\ell;\vec{v})\ \textbf{in}\ \langle a, \textbf{rmw}_{\psi}\ (\ell;\vec{w}a)\rangle\right]\!\right]^{\mathsf{c}}$, and therefore so is π. □

Corollary E.8. *Assuming* $\zeta_{\vec{u}} = \psi_{\vec{w}} \circ^{\mathrm{id}} \varphi_{\vec{v}}$,

$$\left[\!\left[\left\langle\textbf{rmw}_{\varphi}\ (\ell;\vec{v}), \textbf{rmw}_{\psi}\ (\ell;\vec{w})\right\rangle\right]\!\right]^{\mathsf{c}} \supseteq \left[\!\left[\textbf{let } a = \textbf{rmw}_{\zeta}\ (\ell;\vec{u})\ \textbf{in}\ \left\langle a, \varphi^{\mathrm{id}}_{\vec{v}} a\right\rangle\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$$

Proof. Using a special case of Proposition E.7, where ψ is independent of its final parameter. □

Proposition E.9. $[\![\ell \coloneqq v\, ; \textbf{rmw}_{\varphi}\ (\ell;\vec{w})]\!]^{\mathsf{c}} \supseteq \left[\!\left[\ell \coloneqq \varphi^{\mathrm{id}}_{\vec{w}} v\, ; v\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$.

Proof. Same as the RMW case in the proof of Proposition E.7, except the initial timestamp does not have to equal the timestamp of the loaded message. □

Proposition E.10. $[\![\ell \coloneqq w\, ; \ell \coloneqq v]\!]^{\mathsf{c}} \supseteq [\![\ell \coloneqq v]\!]^{\mathsf{c}}_{\mathcal{G}}$.

Proof. Replace the second assignment on the left using Proposition E.4, and follow with Proposition E.9. □

Proposition E.11. *Assuming* $\mathrm{dom}\,\psi_{\vec{w}} \supseteq \mathrm{dom}\,\varphi_{\vec{u}}$,

$$\left[\!\left[\textbf{let } a = \textbf{rmw}_{\varphi}\ (\ell;\vec{u})\ \textbf{in match } \psi_{\vec{w}} a\ \textbf{with}\ \{\iota_\bot\, \_.a \mid \iota_\top\, v.\ell \coloneqq v\, ; a\}\right]\!\right]^{\mathsf{c}} \supseteq \left[\!\left[\textbf{rmw}_{\psi}\ (\ell;\vec{w})\right]\!\right]^{\mathsf{c}}_{\mathcal{G}}$$

Proof. Let $\tau \in \left[\!\left[\textbf{rmw}_{\psi}\ (\ell;\vec{w})\right]\!\right]^{\mathsf{c}}_{\mathcal{G}} = \left[\!\left[\mathrm{rmw}_{\ell,\psi_{\vec{w}}}\right]\!\right]_{\mathcal{G}} = \left[\!\left[\mathrm{rmw}^{\mathrm{RO}}_{\ell,\psi_{\vec{w}}}\right]\!\right]_{\mathcal{G}} \cup \left[\!\left[\mathrm{rmw}^{\mathrm{RMW}}_{\ell,\psi_{\vec{w}}}\right]\!\right]_{\mathcal{G}}$.

RO. If $\tau \in \left[\!\left[\mathrm{rmw}^{\mathrm{RO}}_{\ell,\psi_{\vec{w}}}\right]\!\right]_{\mathcal{G}}$, then we have $\tau = \kappa\boxed{\langle\mu,\mu\rangle}\kappa \therefore \nu.\mathtt{vl}$ where $\psi_{\vec{w}}\,(\nu.\mathtt{vl}) = \bot$, $\nu \in \mu_\ell$, and $\nu.\mathtt{t} = \kappa_\ell$. Structurally, we have

$$[\![\textbf{match}\ (\psi_{\vec{w}})\ \nu.\mathtt{vl}\ \textbf{with}\ \{\iota_\bot\, \_.\nu.\mathtt{vl} \mid \iota_\top\, v.\ell \coloneqq v\, ; \nu.\mathtt{vl}\}]\!]^{\mathsf{c}} = \mathrm{return}\,\nu.\mathtt{vl}$$

By assumption, $\varphi_{\vec{u}}\,(\nu.\mathtt{vl}) = \bot$. Loading the same message ν, we have $\tau \in [\![\textbf{rmw}_{\varphi}\ (\ell;\vec{u})]\!]^{\mathsf{c}}$. We obtain the desired trace from binding it with $\kappa\boxed{\langle\mu,\mu\rangle}\kappa \therefore \nu.\mathtt{vl} \in \mathrm{return}\,\nu.\mathtt{vl}$.

Table 4. Components filling roles in the definition of closure rules, using the notations of Table 2.

rule	source		target	
	’er	’ee	’er	’ee
loosen		ϵ		ν
expel	$\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}$		ϵ	ν
condense	ν	ϵ	$\nu\,[\uparrow\epsilon]$	
stutter				$\langle\mu,\mu\rangle$
mumble	$\langle\mu,\rho\rangle$	$\langle\rho,\theta\rangle$	$\langle\mu,\theta\rangle$	
tighten		ν		ϵ
absorb	ϵ	ν	$\epsilon_{\mathtt{i}}^{\nu.\mathtt{i}}$	
dilute	$\nu\,[\uparrow\epsilon]$		ν	e

RMW. If $\tau \in \left[\!\left[\mathrm{rmw}^{\mathrm{RMW}}_{\ell,\psi_{\vec{w}}}\right]\!\right]_{\mathcal{G}}$, then we have $\tau = \kappa\boxed{\langle\mu,\mu\uplus\{\ell{:}v@(\nu.\mathtt{t},t]\langle\!\langle\kappa[\ell\mapsto t]\rangle\!\rangle\}\rangle}\kappa[\ell\mapsto t] \therefore \nu.\mathtt{vl}$ where $\psi_{\vec{w}}\,(\nu.\mathtt{vl}) = v$, $\nu \in \mu_\ell$, and $\nu.\mathtt{t} = \kappa_\ell$. Structurally, we have

$$[\![\mathbf{match}\ \psi_{\vec{w}}\,(\nu.\mathtt{vl})\ \mathbf{with}\ \{\iota_\bot\,\_.\nu.\mathtt{vl} \mid \iota_\top\, v.\ell \coloneqq v\,;\nu.\mathtt{vl}\}]\!]^{\mathsf{c}} = [\![\ell \coloneqq v\,;\nu.\mathtt{vl}]\!]^{\mathsf{c}}$$

Loading the same message ν, we proceed depending on $\varphi_{\vec{u}}\,(\nu.\mathtt{vl})$.
$\varphi_{\vec{u}}\,(\nu.\mathtt{vl}) = \bot$. We can bind $\kappa\boxed{\langle\mu,\mu\rangle}\kappa \therefore \nu.\mathtt{vl} \in \left[\!\left[\mathbf{rmw}_\varphi\,(\ell;\vec{u})\right]\!\right]^{\mathsf{c}}$, with $\tau \in [\![\ell \coloneqq v\,;\nu.\mathtt{vl}]\!]^{\mathsf{c}}$.
$\varphi_{\vec{u}}\,(\nu.\mathtt{vl}) \neq \bot$. Then we have $\kappa\boxed{\langle\mu,\rho\rangle}\kappa[\ell\mapsto\frac{\kappa_\ell+t}{2}] \therefore \nu.\mathtt{vl} \in \left[\!\left[\mathbf{rmw}_\varphi\,(\ell;\vec{u})\right]\!\right]^{\mathsf{c}}$, where $\rho \coloneqq \mu\uplus\left\{\ell{:}\varphi_{\vec{u}}\,(\nu.\mathtt{vl})\,@(\nu.\mathtt{t},\frac{\kappa_\ell+t}{2}]\left\langle\!\!\left\langle\kappa[\ell\mapsto\frac{\kappa_\ell+t}{2}]\right\rangle\!\!\right\rangle\right\}$. We can bind it with

$$\kappa[\ell\mapsto\tfrac{\kappa_\ell+t}{2}]\boxed{\left\langle\rho,\rho\uplus\left\{\ell{:}v@(\tfrac{\kappa_\ell+t}{2},t]\langle\!\langle\kappa[\ell\mapsto t]\rangle\!\rangle\right\}\right\rangle}\kappa[\ell\mapsto t] \therefore \nu.\mathtt{vl} \in [\![\ell \coloneqq v\,;\nu.\mathtt{vl}]\!]^{\mathsf{c}}$$

where at the end we absorb (Ab) the first message into the second. □

F Proof of Rewrite Castling

In this section we prove Rewrite Castling. We make a few observations to help us navigate the elaborate case-split that makes up the proof.

Active roles in closure rules. Intuitively, the $\mathfrak{g}$- and $\mathfrak{a}$-rules have an object message that is acted upon, and sometimes a subject message that partakes in the action. For example, in the absorb rule there is a message, which we call the absorb’ee, that is being “absorbed” into another, which we call the absorb’er. We think of the absorb’er as a message that changed, rather than two different messages. In stutter and mumble the active components are transitions rather than messages.

Table 4 lists which components of the source and target of each closure rule fill the subject and object roles. We use these roles to distinguish scenarios within each castling case in the proof of Rewrite Castling. The roles for forward and rewind are omitted because there is no need to analyze different scenarios within the cases involving them in the proof.

Conditions for closure validity. As we introduced the closure rules in §7, we noted conditions that imply that the target of a rewrite is a trace, assuming the source is. We summarize these below:

Lemma F.1. *For* $\mathsf{x} \in \mathfrak{gc}$*, assume* τ *is a trace and* $\tau \xrightarrow{\mathsf{x}} \pi$*. Then, using the notations of Table 2:*

- *If* $\mathsf{x} = \mathsf{Mu}$*, then* $\pi \in \mathsf{Trace}$.
- *If* $\mathsf{x} = \mathsf{Ls}$*, then* $\pi \in \mathsf{Trace}$ *iff* $\mathsf{Ls}^{\checkmark}(\nu,\eta)$*: either* η *is empty, or* $\nu \hookrightarrow \left(\eta \mathbin{\overline{\uplus}} \{\nu\}\right).\mathtt{o}$.
- *If* $\mathsf{x} = \mathsf{Ex}$*, then* $\pi \in \mathsf{Trace}$ *iff* $\mathsf{Ex}^{\checkmark}(\nu,\eta)$*: either* η *is empty, or* $\nu \hookrightarrow \left(\eta \mathbin{\overline{\uplus}} \{\nu\}\right).\mathtt{o}$.
- *If* $\mathsf{x} = \mathsf{Cn}$*, then* $\pi \in \mathsf{Trace}$ *iff* $\mathsf{Cn}^{\checkmark}(\epsilon,\xi)$*: either* ξ *is empty,* $\epsilon.\mathtt{i} \notin \xi.\mathtt{c}.\mathtt{t}$*, or* $\epsilon.\mathtt{seg} \cap \bigcup \xi.\mathtt{c}.\mathtt{seg} = \emptyset$.

Table 5. Diagrams for different scenarios of each case of $\mathsf{x} \rightleftarrows \mathsf{y}$.

y\x	St	Mu	Fw	Rw	Ti	Ab	Di
Ls	1, 2	7, 8	13	14	19, 20	26, 27	33, 34, 35
Ex	3, 4	9, 10	15	16	21, 22	28, 29	36, 37, 38
Cn	5, 6	11, 12	17	18	23, 24, 25	30, 31, 32	39, 40, 41, 42
St					43, 44	45, 46	47, 48
Mu					49, 50, 51, 52	53, 54, 55, 56	57, 58, 59, 60
Fw					61	63	65
Rw					62	64	66

- *If* $\mathsf{x} = \mathsf{St}$, *then* $\pi \in \mathsf{Trace}$ *iff* $\mathsf{St}^{\checkmark}(\alpha, \mu)\colon \alpha \rightarrowtail \mu \in \mathsf{Mem}$.
- *If* $\mathsf{x} = \mathsf{Fw}$, *then* $\pi \in \mathsf{Trace}$ *iff* $\mathsf{Fw}^{\checkmark}(\omega, \xi)\colon \omega \hookrightarrow \xi.\mathsf{c}$.
- *If* $\mathsf{x} = \mathsf{Rw}$, *then* $\pi \in \mathsf{Trace}$ *iff* $\mathsf{Rw}^{\checkmark}(\alpha, \xi)\colon \alpha \hookrightarrow \xi.\mathsf{o}$.

In each case in the proof, the rewrite sequence after castling includes a new pre-trace. We must show that this is a trace for the sequence to be valid, because Rewrite Castling concerns the restriction of the closure rules to traces. Lemma F.1 is the workhorse that powers this task.

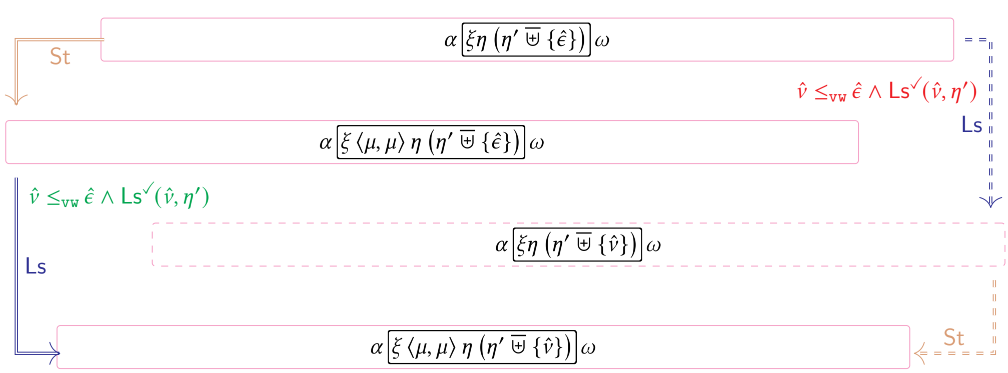
Proof of Rewrite Castling. Diagrams attached below depict how rewrites castle in different scenarios. We summarize the reasoning involved below. Use Table 5 to navigate through the cases.

- For cases of $\mathsf{St} \rightleftarrows \mathsf{y}$ where $\mathsf{y} \in \mathfrak{g}$ (1, 2, 3, 4, 5, 6), the required condition is about the same chronicle as the assumed condition, except for possibly a removed transition. This means that its opening memory is an extension of the original (result of adding messages), and the closing memory is a reduction of the original (result of removing messages). The condition of pointing downwards into a memory is stable under extensions, and the condition of non-intersection is stable under reductions. Cases of $\mathsf{Mu} \rightleftarrows \mathsf{y}$ where $\mathsf{y} \in \mathfrak{g}$ (7, 8, 9, 10, 11, 12) are simpler because the opening and closing memory remain the same.
- The cases of $\mathsf{Fw} \rightleftarrows \mathsf{y}$ and $\mathsf{Rw} \rightleftarrows \mathsf{y}$ where $\mathsf{y} \in \mathfrak{g}$ (13, 14, 15, 16, 17, 18) are trivial because the required condition remains the same.
- For cases of $\mathsf{Ti} \rightleftarrows \mathsf{y}$ where $\mathsf{y} \in \mathfrak{g}$, the required condition in the cases of $\mathsf{y} \in \{\mathsf{Ls}, \mathsf{Ex}\}$ (19, 20, 21, 22) holds because pointing downwards into a memory is stable under "loosening" a message within the memory ($\nu \leq \epsilon$). The remaining $\mathsf{y} = \mathsf{Cn}$ case (23, 24, 25) holds because the difference between the required condition and the original keeps the occupied timestamps the same, and the $\mathrel{-\!\!\in}$ relation is stable under "loosening" the first argument.
- For cases of $\mathsf{Ab} \rightleftarrows \mathsf{y}$ where $\mathsf{y} \in \mathfrak{g}$, the required condition in the cases of $\mathsf{y} \in \{\mathsf{Ls}, \mathsf{Ex}\}$ (26, 27, 28, 29) holds because pointing downwards into a memory is stable under changing a message's initial timestamp and adding a message within the memory. The remaining $\mathsf{y} = \mathsf{Cn}$ case (30, 31, 32) holds because the difference between the required condition and the original keeps the occupied timestamps the same, and the $\mathrel{-\!\!\in}$ relation is stable under changing the initial timestamp of the first argument.
- Cases of $\mathsf{Di} \rightleftarrows \mathsf{y}$ where $\mathsf{y} \in \mathfrak{g}$ hold thanks to Lemma 7.6 when $\mathsf{y} \in \{\mathsf{Ls}, \mathsf{Ex}\}$ (33, 34, 35, 36, 37, 38). The remaining $\mathsf{y} = \mathsf{Cn}$ case (39, 40, 41, 42) is the most complicated. First, we note that $(-\,[\uparrow\epsilon])\,[\uparrow\hat{\epsilon}\,[\uparrow\epsilon]] = (-\,[\uparrow\hat{\epsilon}])\,[\uparrow\epsilon\,[\uparrow\hat{\epsilon}]]$, which means that the pre-trace to be "diluted" is of the correct shape. The rewrite is valid because $\mathrel{-\!\!\in}$ is stable under changing the timestamp of the second argument. In particular, it is stable under pulling both arguments along the same

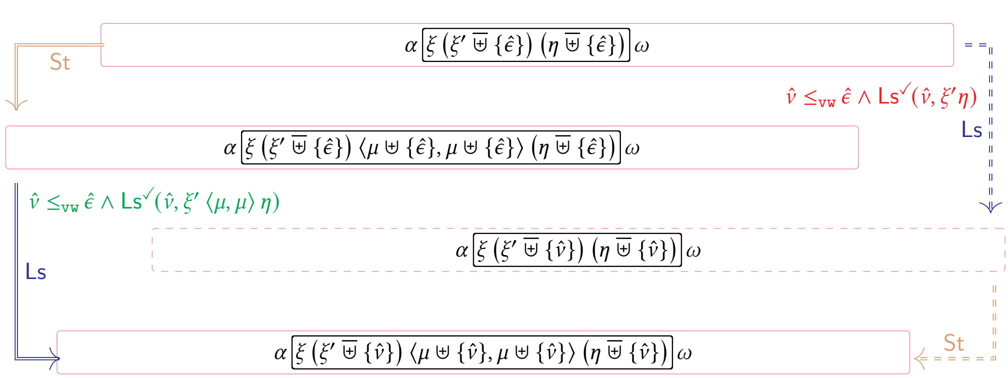
message which does not intersect the arguments' segments. Finally, the condition for the pre-trace to be a trace is satisfied because the message is being pulled along a message that was either removed or known to appear later in the chronicle (as a local message); either way, the segment is free. There are also the cases where they overlap this way or that, in which we use the trivial dovetailing geometry of $\multimap\!\in$.

- For cases of $\mathsf{x} \rightleftarrows \mathsf{St}$ where $\mathsf{x} \in \mathfrak{a}$, the required condition in the cases of $\mathsf{x} \in \{\mathsf{Ti}, \mathsf{Ab}\}$ (43, 44, 45, 46) holds because there remains a message at each timestamp where there was a message originally, and the initial view remains the same. The remaining $\mathsf{y} = \mathsf{Di}$ case (47, 48) holds because pointing-to is stable under pulling along a message; and if the initial view pointed to the dilute'ee then after pulling it, it will point to the dilute'er pulled along the dilute'ee.
- The cases of $\mathsf{x} \rightleftarrows \mathsf{Mu}$ where $\mathsf{x} \in \mathfrak{a}$ (49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60) do not require special considerations regarding conditions.
- For cases of $\mathsf{x} \rightleftarrows \mathsf{y}$ where $\mathsf{x} \in \mathfrak{a}$ and $\mathsf{y} \in \{\mathsf{Fw}, \mathsf{Rw}\}$, the required condition in the cases of $\mathsf{x} \in \{\mathsf{Ti}, \mathsf{Ab}\}$ (61, 62, 63, 64) holds because pointing downwards into a memory is stable under "loosening" a message within the memory ($\nu \leq \epsilon$). The case of $\mathsf{x} \in \mathsf{Di}$ (65, 66) hold thanks to Lemma 7.6, and the fact that pointing downward into a memory is stable under pulling along the same message; and if the delimiting view pointed to the dilute'ee then after pulling it, it will point to the dilute'er pulled along the dilute'ee. □

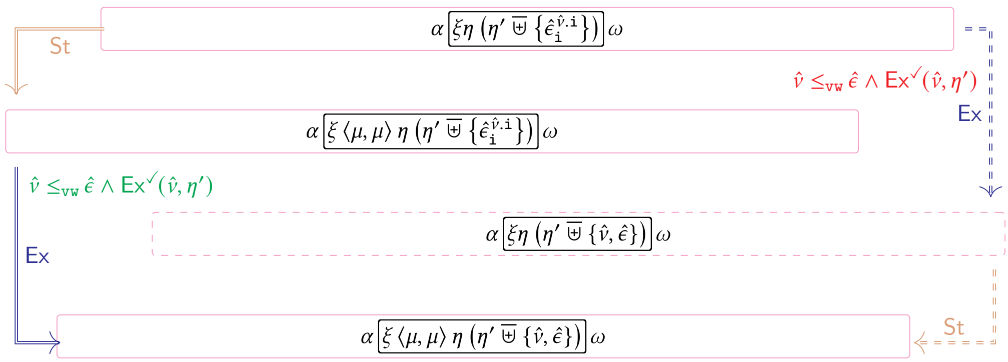
The rest of the manuscript is the collection of figures for the proof above.



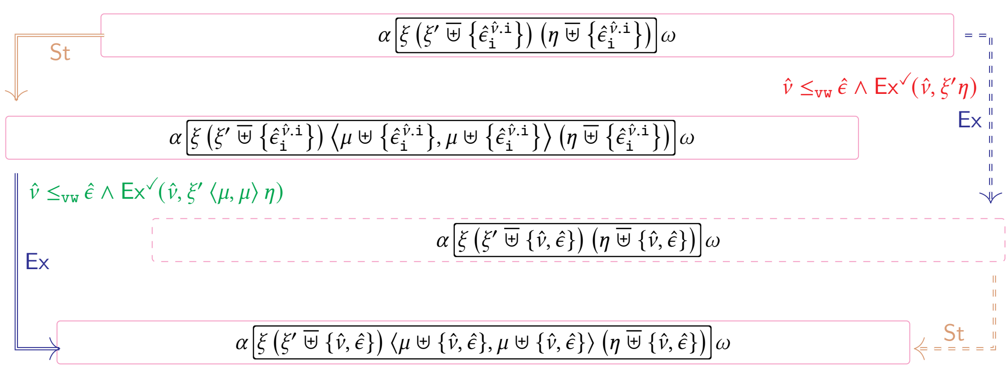
1. The $\mathsf{St} \rightleftarrows \mathsf{Ls}$ case when the loosen'ee does not appear across the stutter'ee.



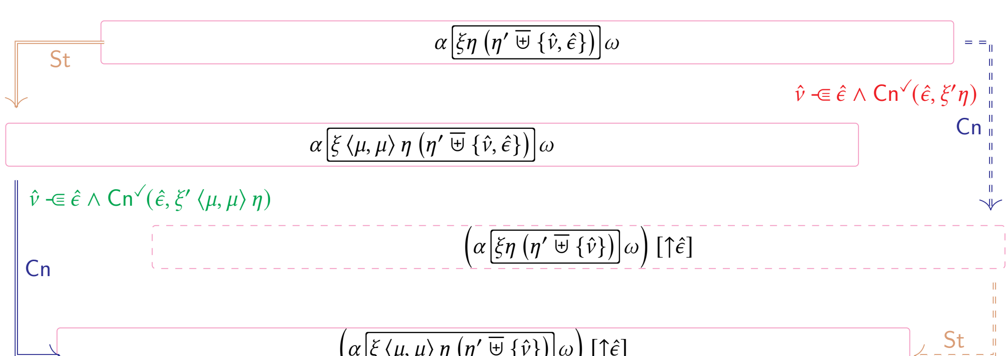
2. The $\mathsf{St} \rightleftarrows \mathsf{Ls}$ case when the loosen'ee appears across the stutter'ee.



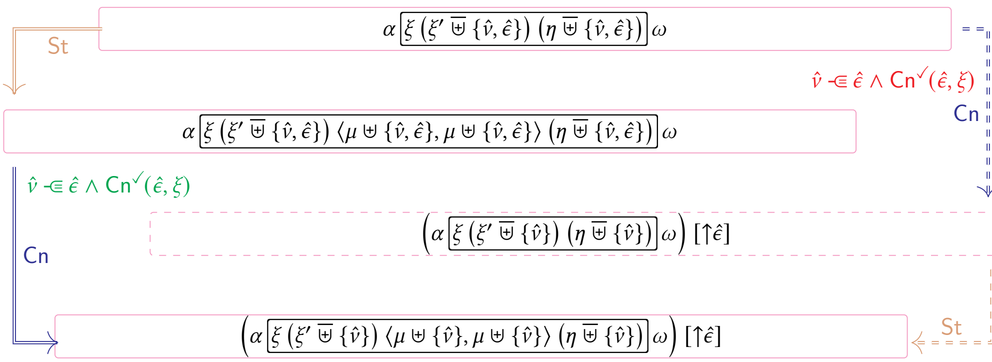
3. The St ⇄ Ex case when the expel'ee does not appear across the stutter'ee.



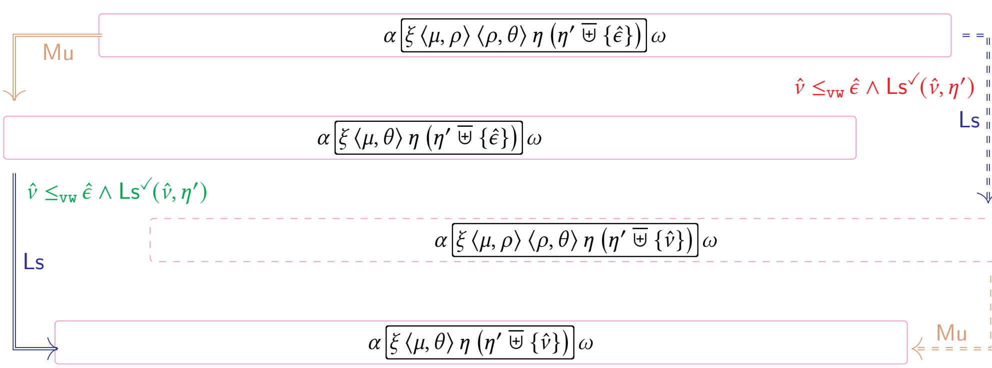
4. The St ⇄ Ex case when the expel'ee appears across the stutter'ee.



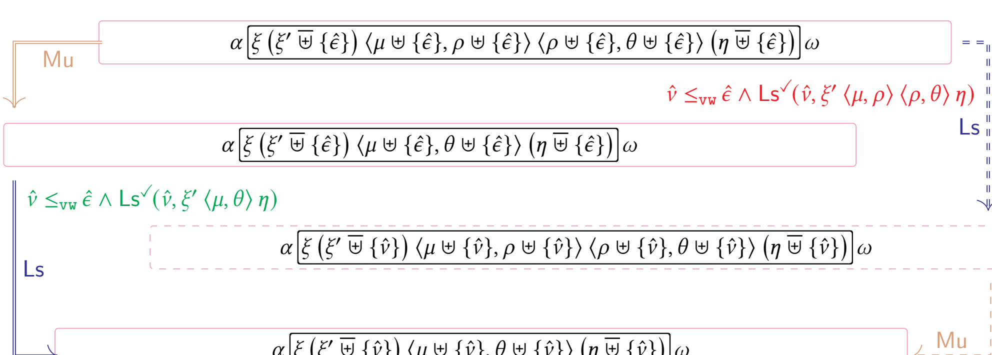
5. The St ⇄ Cn case when the condense'ee does not appear across the stutter'ee.



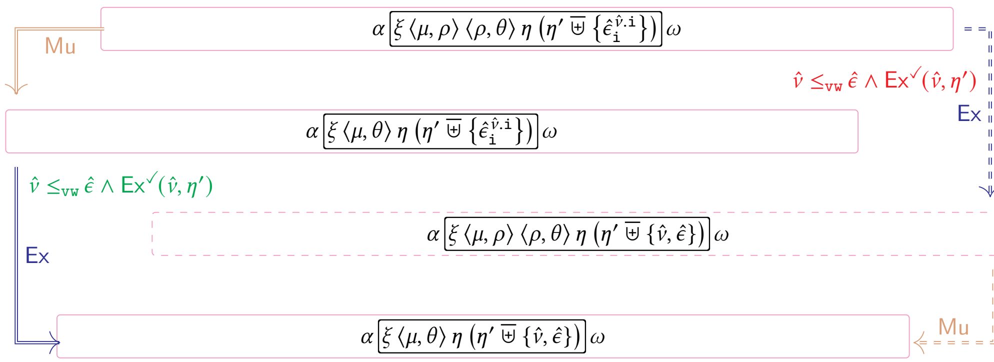
6. The St ⇄ Cn case when the condense'ee appears across the stutter'ee.



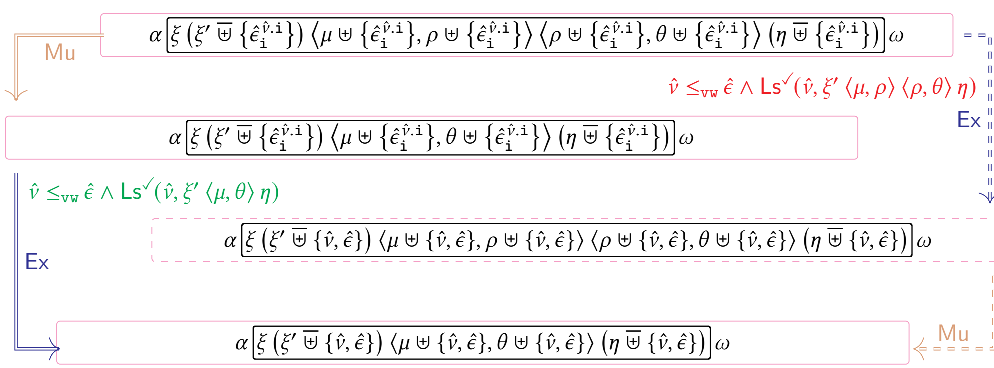
7. The Mu ⇄ Ls case when the loosen'ee does not appear across the mumble'er.



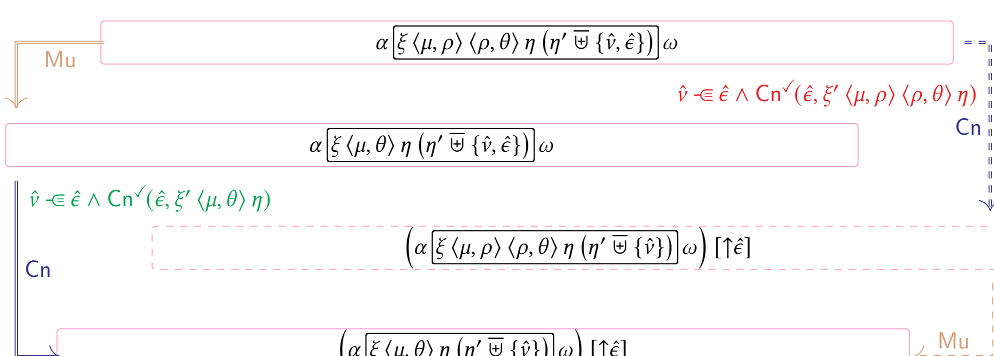
8. The Mu ⇄ Ls case when the loosen'ee appears across the mumble'er.



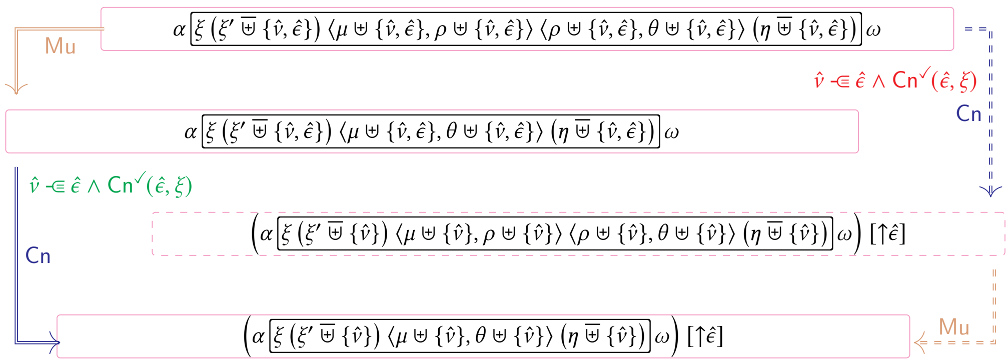
9. The Mu ⇄ Ex case when the expel'ee does not appear across the mumble'er.



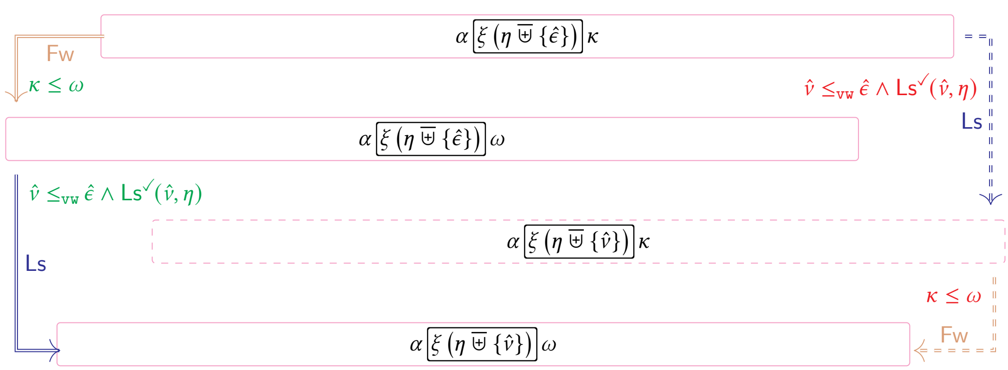
10. The Mu ⇄ Ex case when the expel'ee appears across the mumble'er.



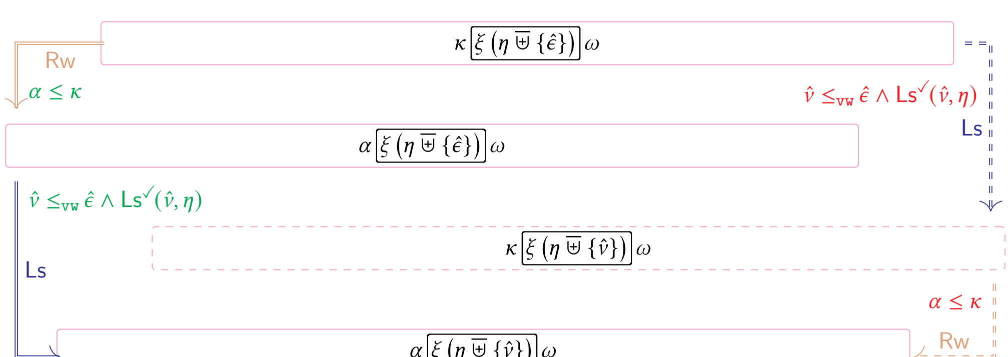
11. The Mu ⇄ Cn case when the condense'ee does not appear across the mumble'er.



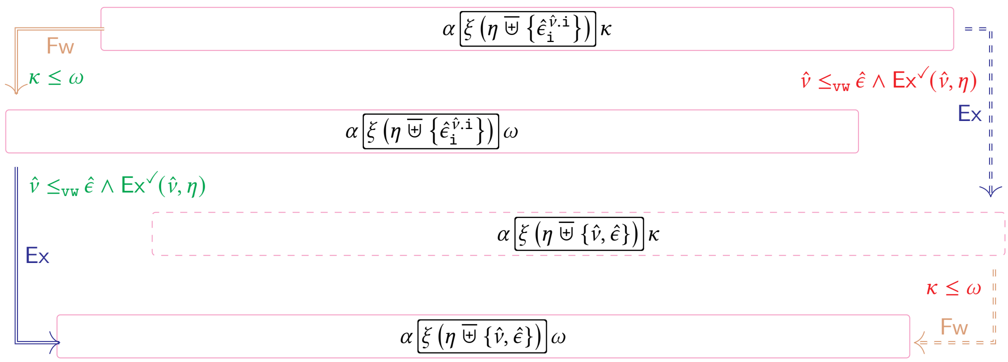
12. The Mu ⇄ Cn case when the condense'ee appears across the mumble'er.



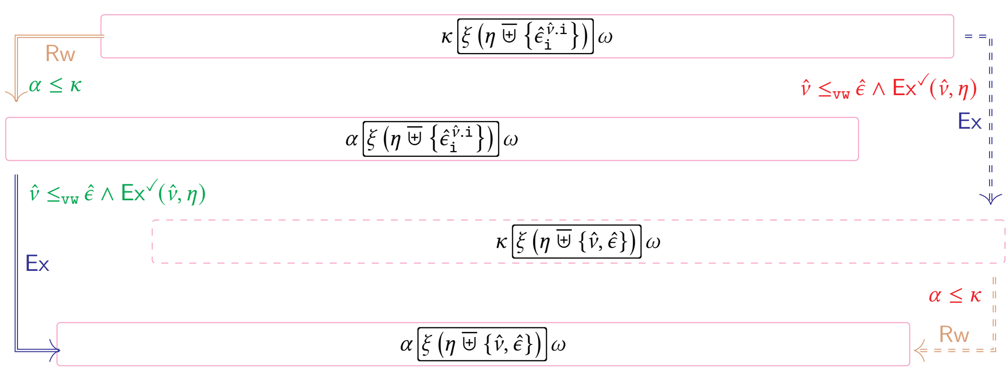
13. The Fw ⇄ Ls case.



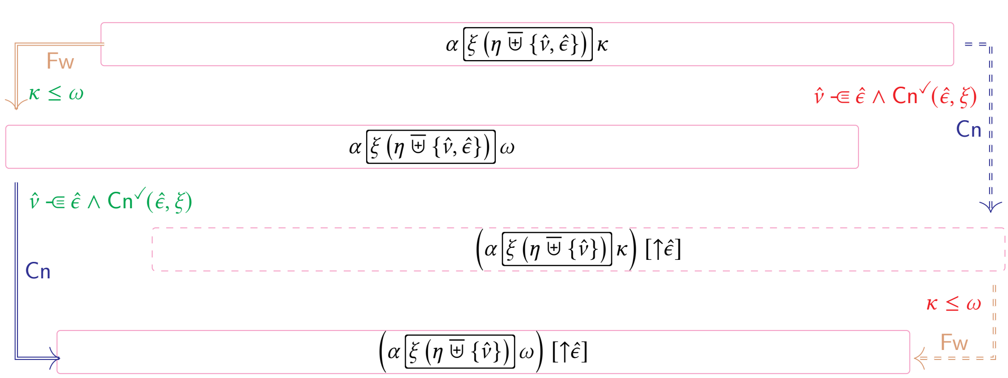
14. The Rw ⇄ Ls case.



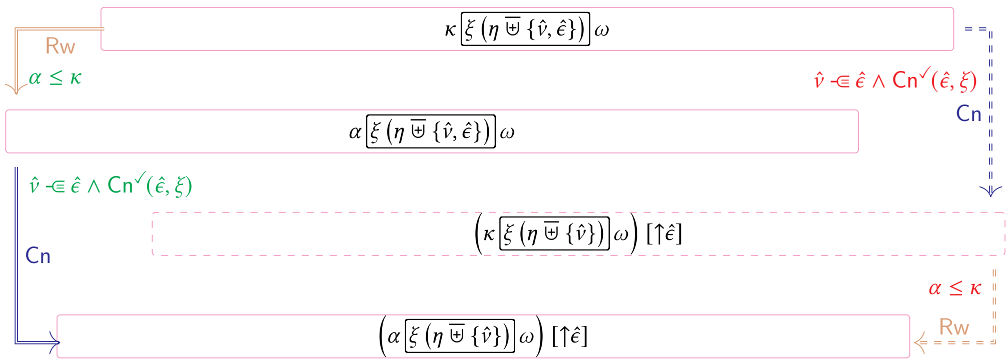
15. The Fw ⇄ Ex case.



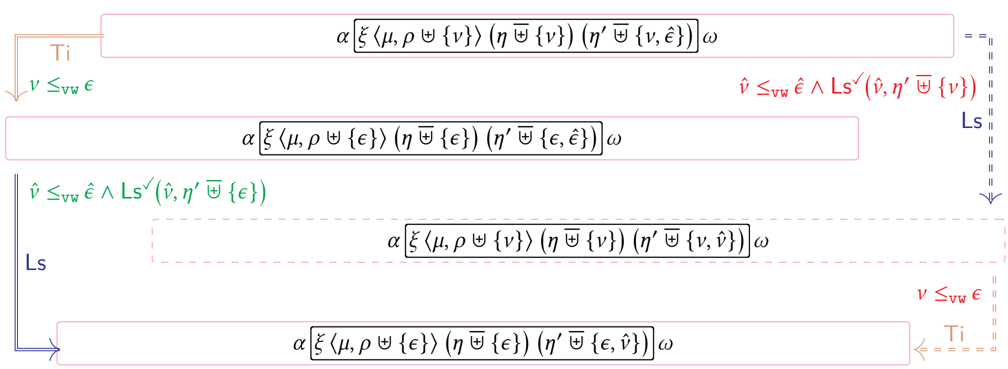
16. The Rw ⇄ Ex case.



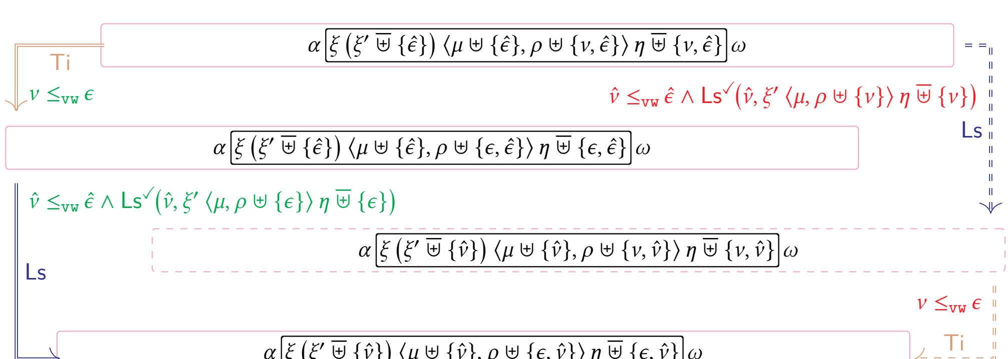
17. The Fw ⇄ Cn case.



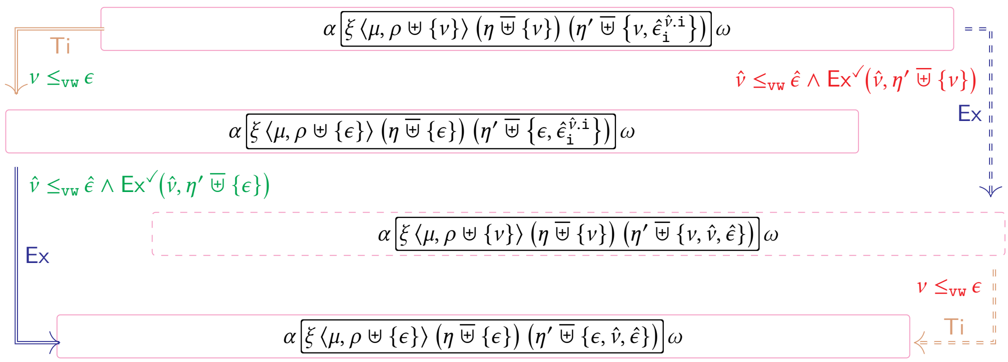
18. The Rw ⇄ Cn case.



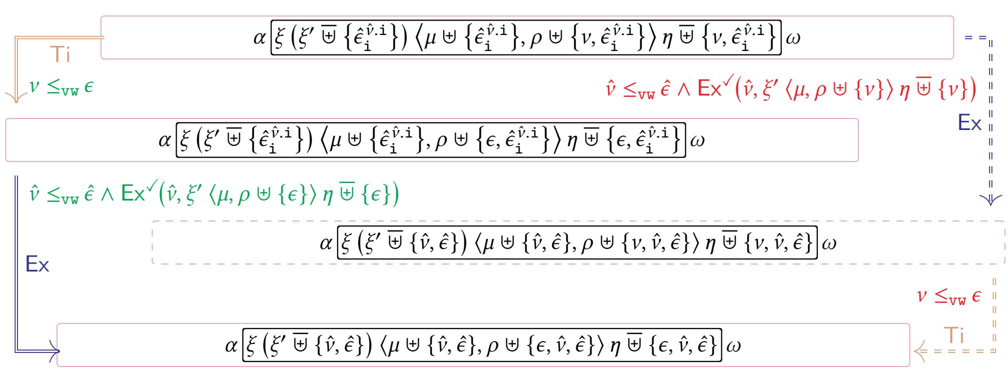
19. The Ti ⇄ Ls case when the loosen'ee appears first after the tighten'ee.



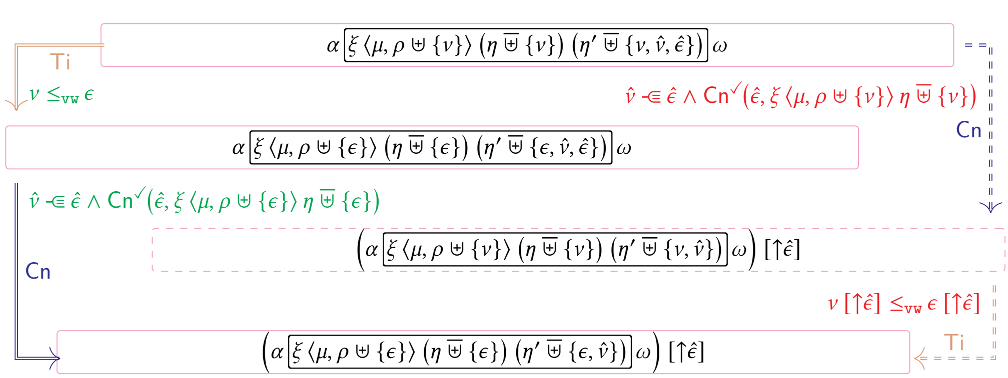
20. The Ti ⇄ Ls case when the loosen'ee appears first before the tighten'ee.



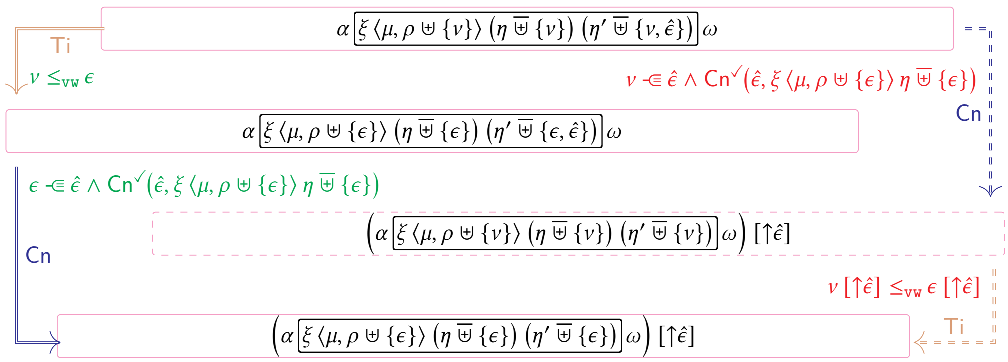
21. The Ti ⇄ Ex case when the expel'ee appears first after the tighten'ee.



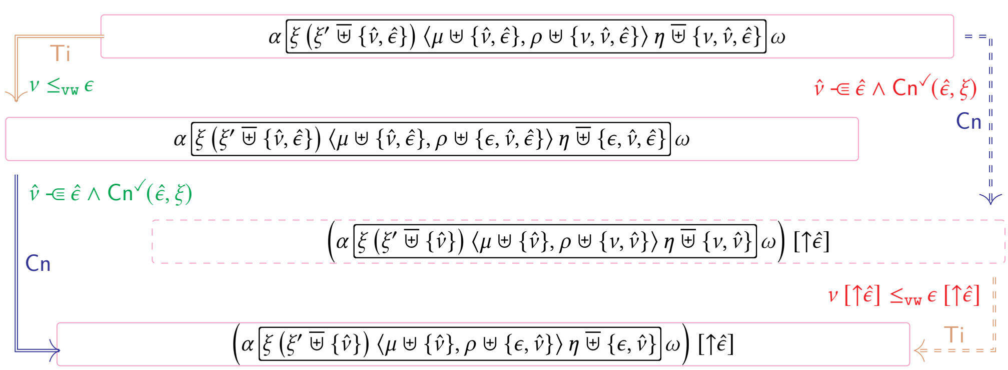
22. The Ti ⇄ Ex case when the expel'ee appears first before the tighten'ee.



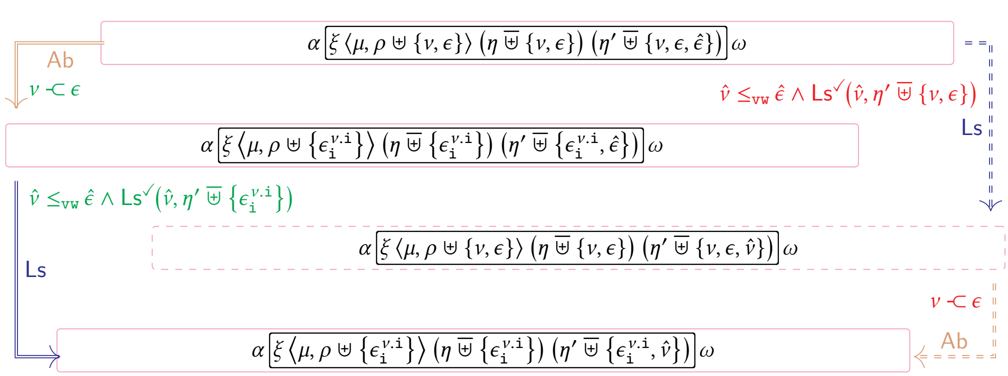
23. The Ti ⇄ Cn case when the condense'ee appears first after the tighten'ee, and the tighten'ee is not the condense'er.



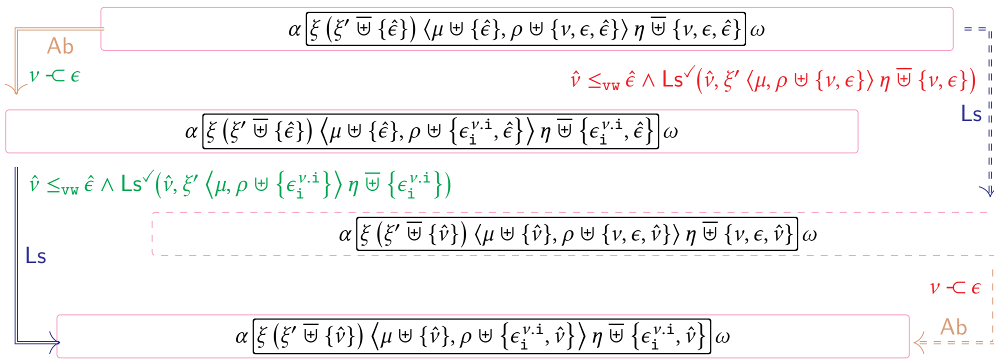
24. The Ti ⇄ Cn case when the condense'ee appears first after the tighten'ee, and the tighten'ee is the condense'er.



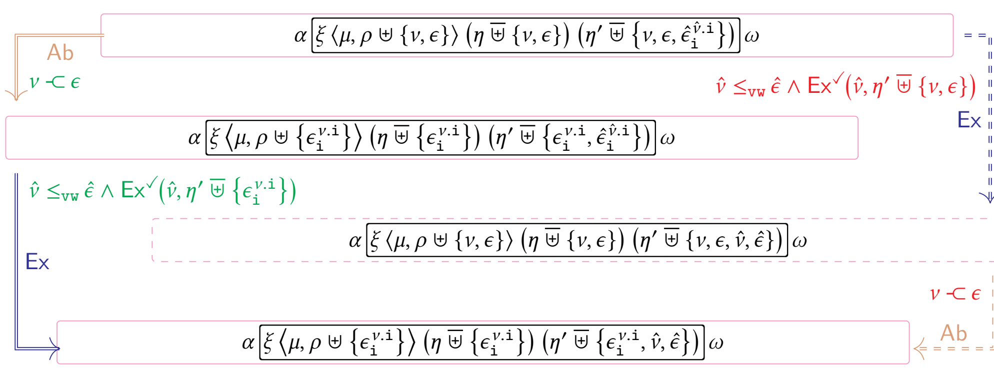
25. The Ti ⇄ Cn case when the condense'ee appears first before the tighten'ee.



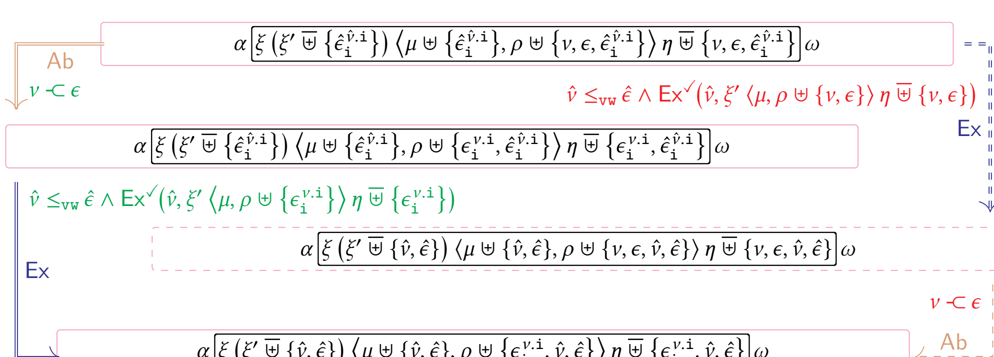
26. The Ab ⇄ Ls case when the loosen'ee appears first after the absorb'ee.



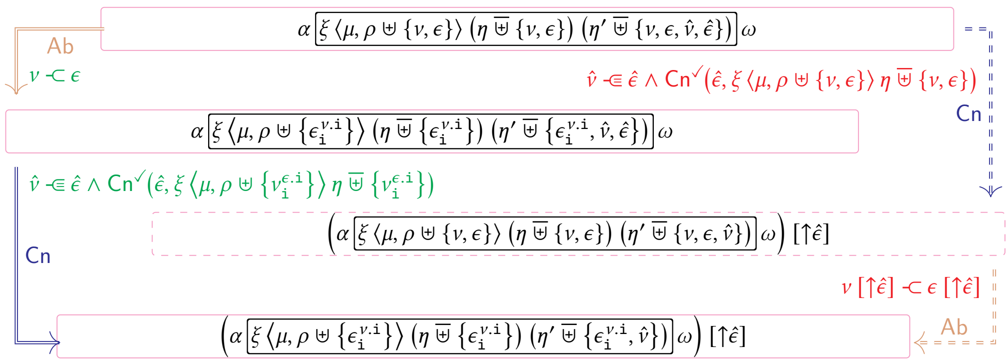
27. The Ab ⇄ Ls case when the loosen'ee appears first before the absorb'ee.



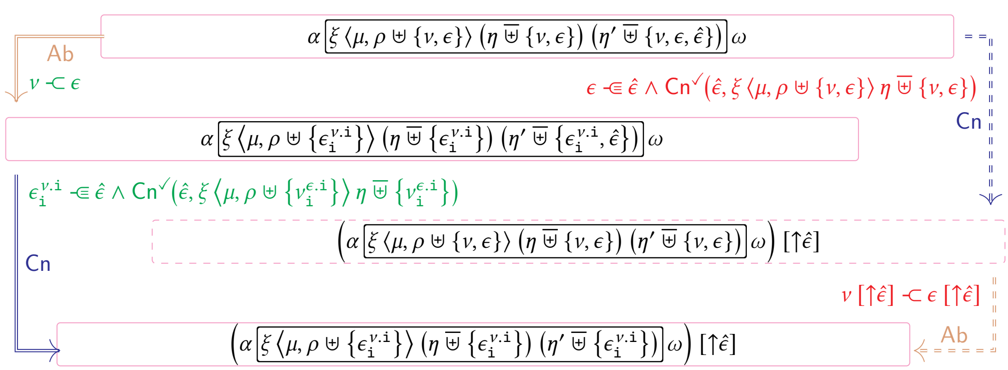
28. The Ab ⇄ Ex case when the expel'ee appears first after the absorb'ee.



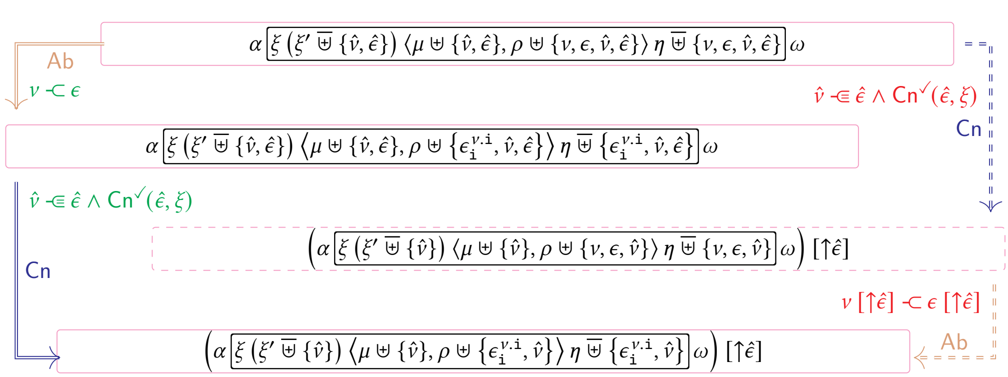
29. The Ab ⇄ Ex case when the expel'ee appears first before the absorb'ee.



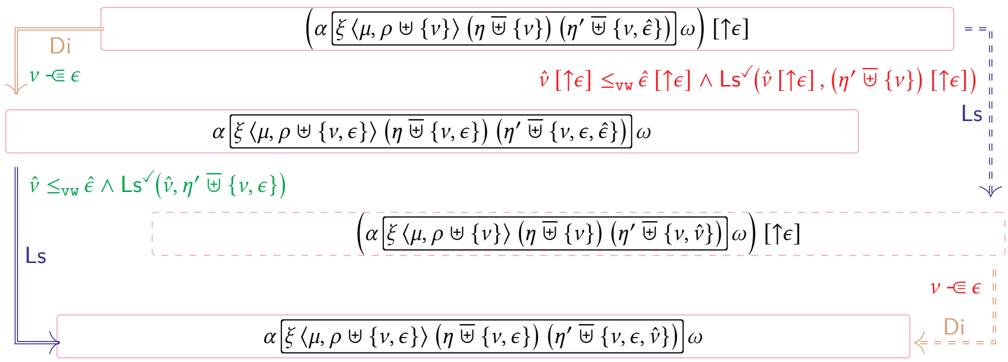
30. The Ab ⇄ Cn case when the condense'ee appears first after the absorb'er, and the absorb'er is not the condense'er.



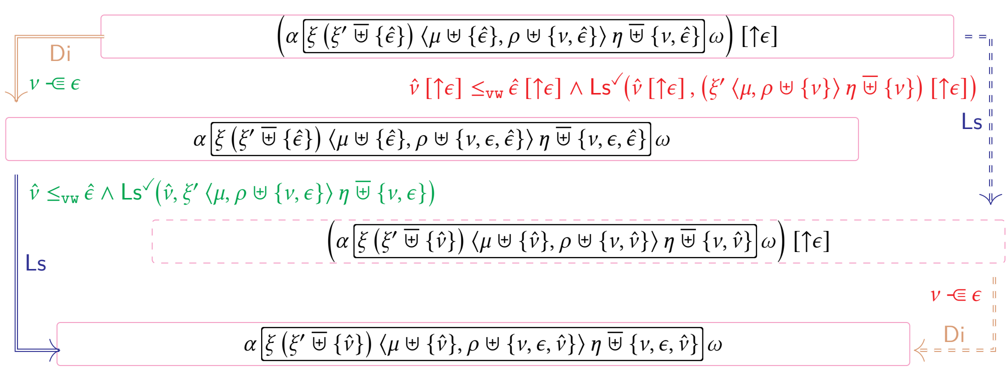
31. The Ab ⇄ Cn case when the condense'ee appears first after the absorb'er, and the absorb'er is the condense'er.



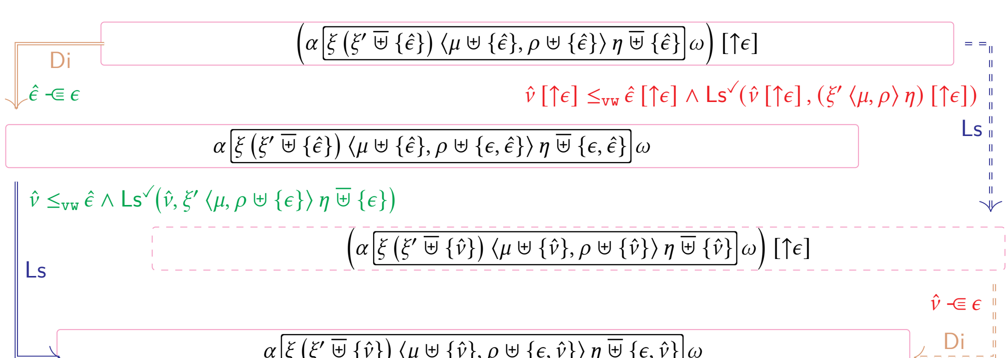
32. The Ab ⇄ Cn case when the condense'ee appears first before the absorb'er.



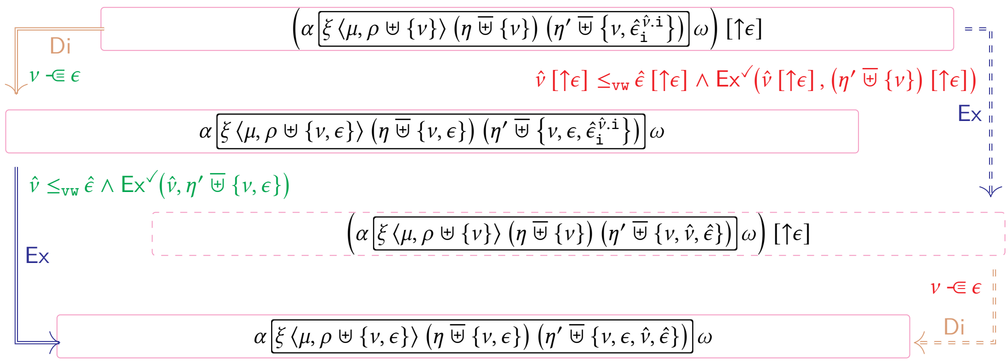
33. The Di ⇄ Ls case when the loosen'ee appears first after the dilute'ee.



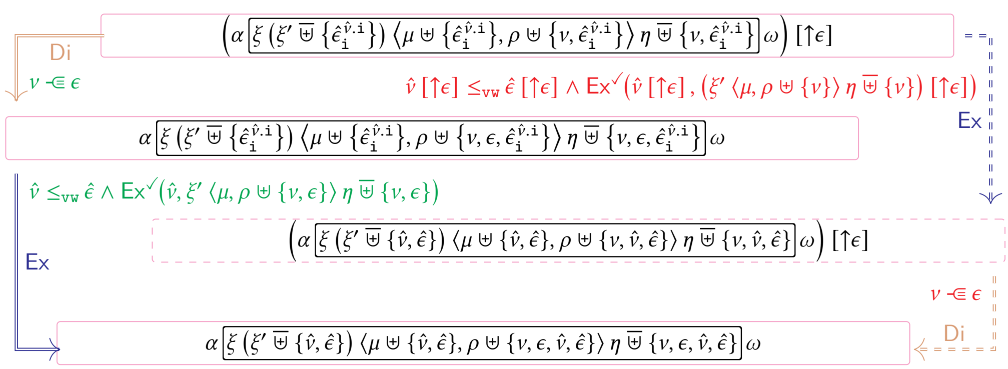
34. The Di ⇄ Ls case when the loosen'ee appears first before the dilute'ee, and the dilute'er is not the loosen'ee.



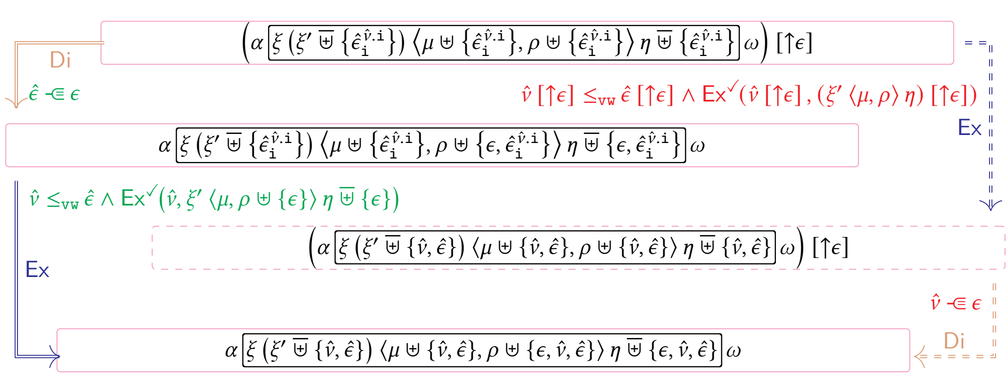
35. The Di ⇄ Ls case when the loosen'ee appears first before the dilute'ee, and the dilute'er is the loosen'ee.



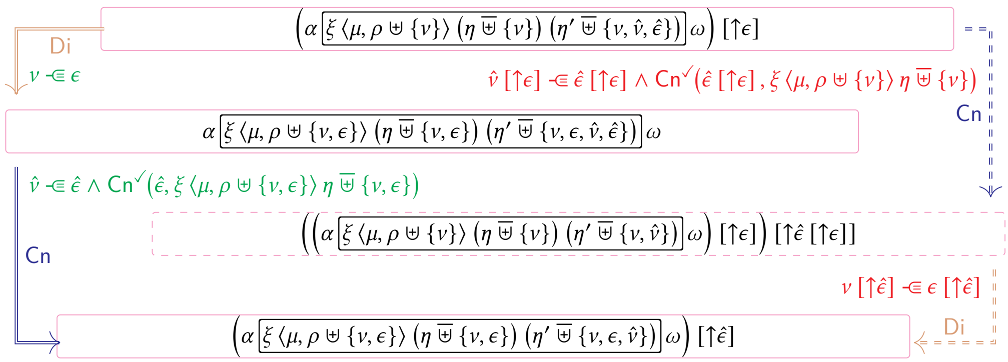
36. The Di ⇄ Ex case when the expel'er appears first after the dilute'ee.



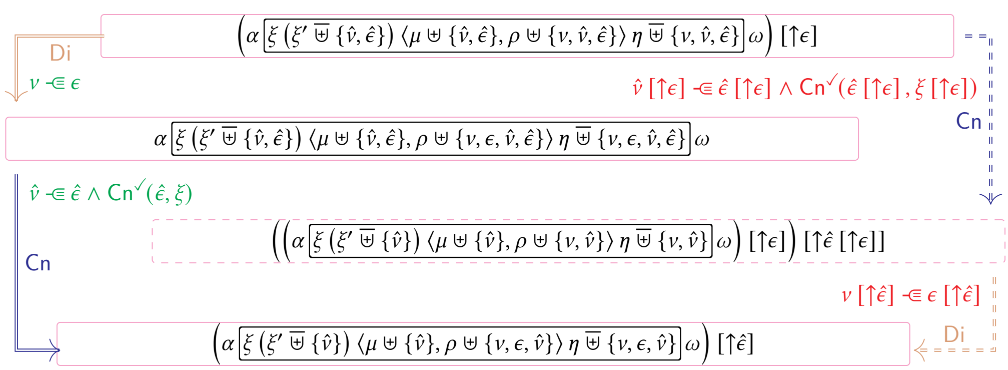
37. The Di ⇄ Ex case when the expel'er appears first before the dilute'ee, and the dilute'er is not the expel'er.



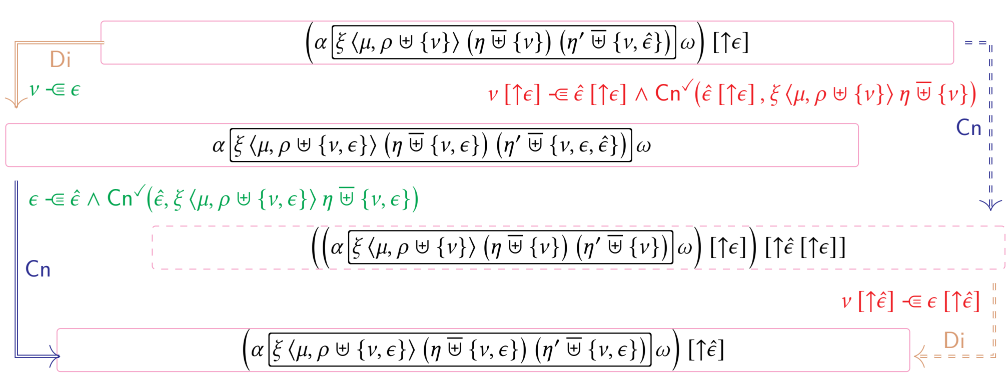
38. The Di ⇄ Ex case when the expel'er appears first before the dilute'ee, and the dilute'er is the expel'er.



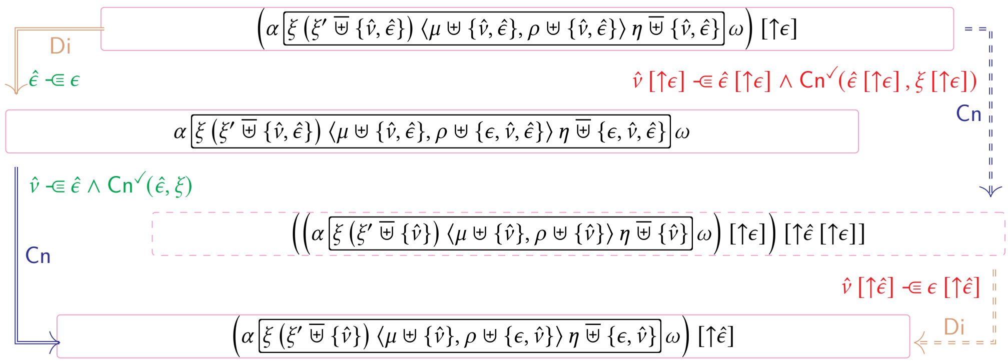
39. The Di ⇄ Cn case when the condense'ee appears first after the dilute'ee, and the dilute'ee is not the condense'er.



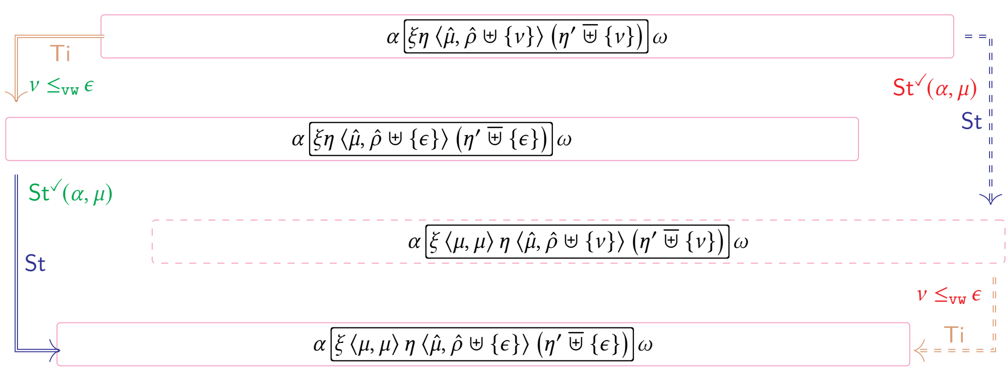
40. The Di ⇄ Cn case when the condense'ee appears first before the dilute'ee, and the dilute'er is not the condense'ee.



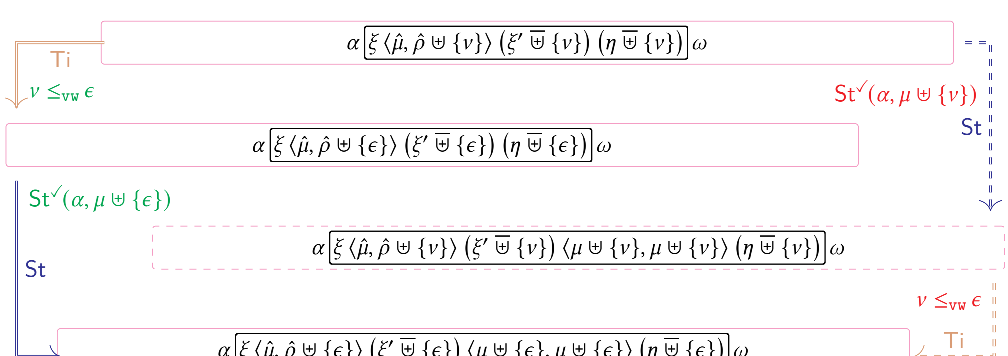
41. The Di ⇄ Cn case when the condense'ee appears first after the dilute'ee, and the dilute'ee is the condense'er.



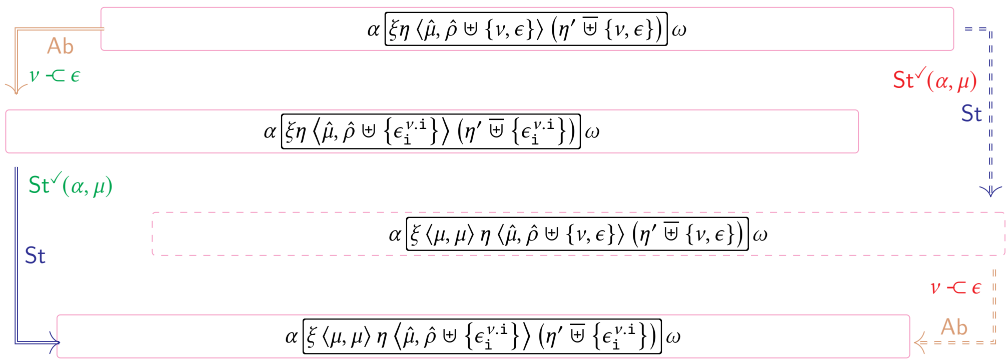
42. The Di ⇄ Cn case when the condense'ee appears first before the dilute'ee, and the dilute'er is the condense'ee.



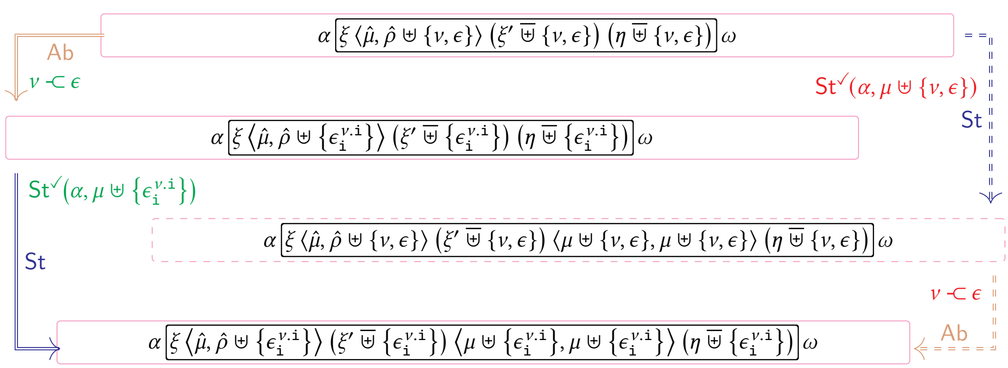
43. The Ti ⇄ St case when the tighten'ee does not appear across the stutter'ee.



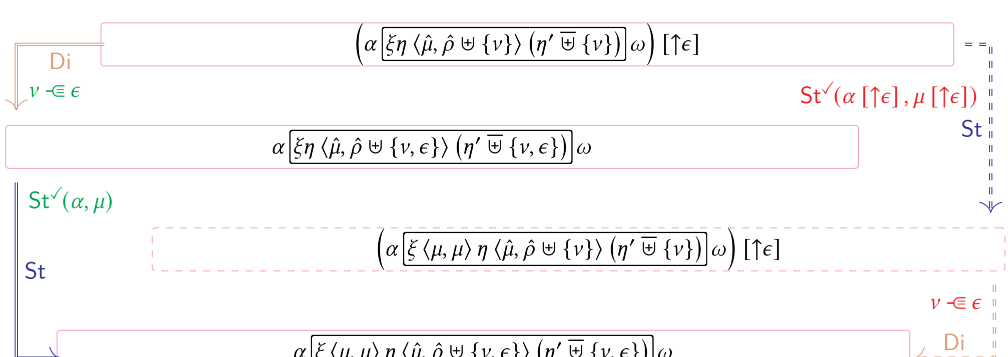
44. The Ti ⇄ St case when the tighten'ee appears across the stutter'ee.



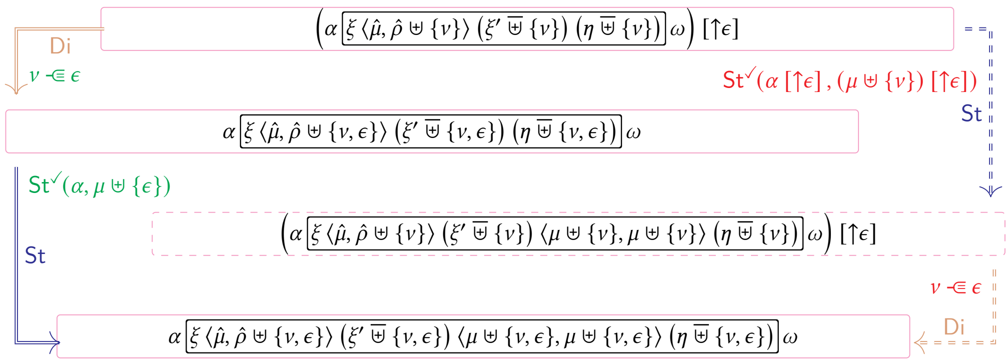
45. The Ab ⇄ St case when the absorb'ee does not appear across the stutter'ee.



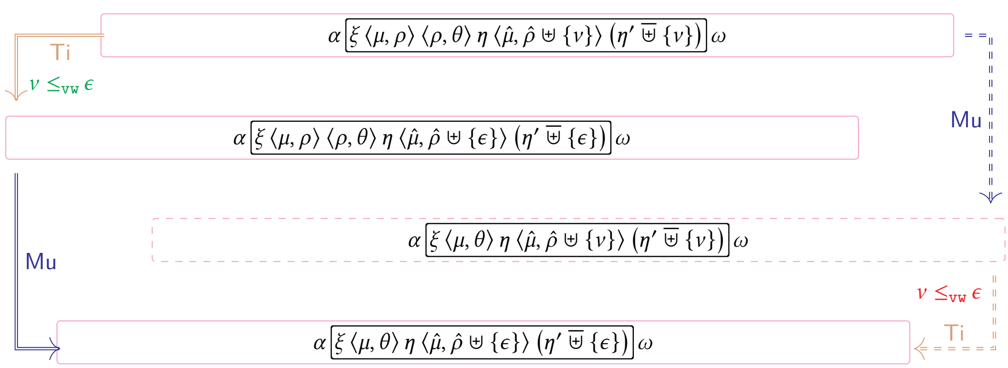
46. The Ab ⇄ St case when the absorb'ee appears across the stutter'ee.



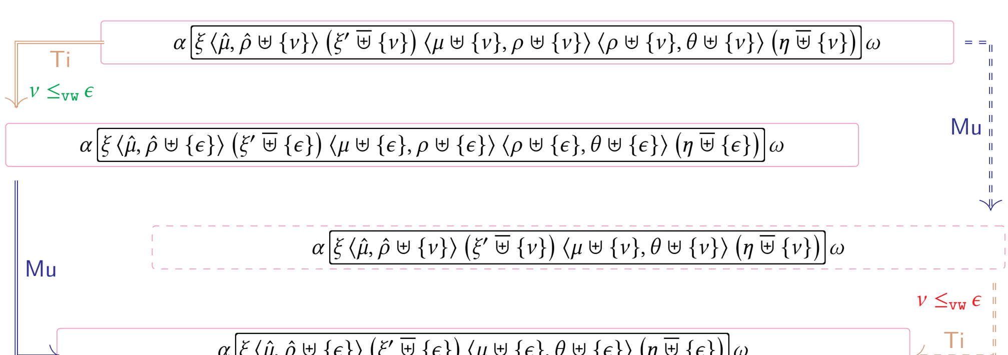
47. The Di ⇄ St case when the dilute'ee does not appear across the stutter'ee.



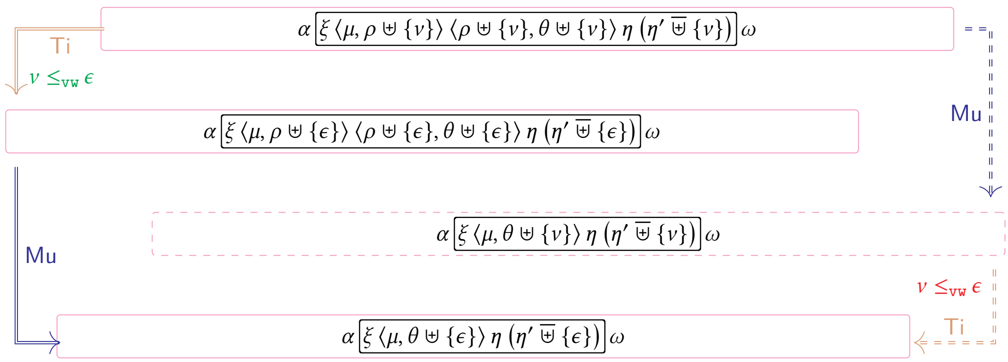
48. The Di ⇄ St case when the dilute'ee appears across the stutter'ee.



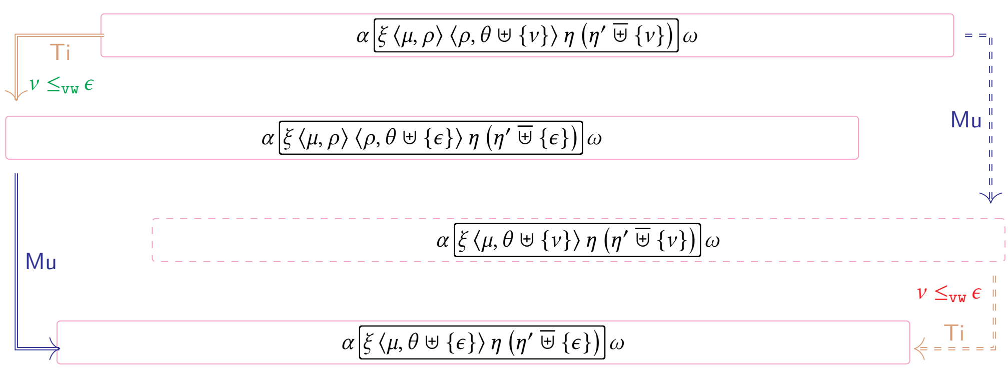
49. The Ti ⇄ Mu case when the tighten'ee appears in neither the mumble'er nor the mumble'ee.



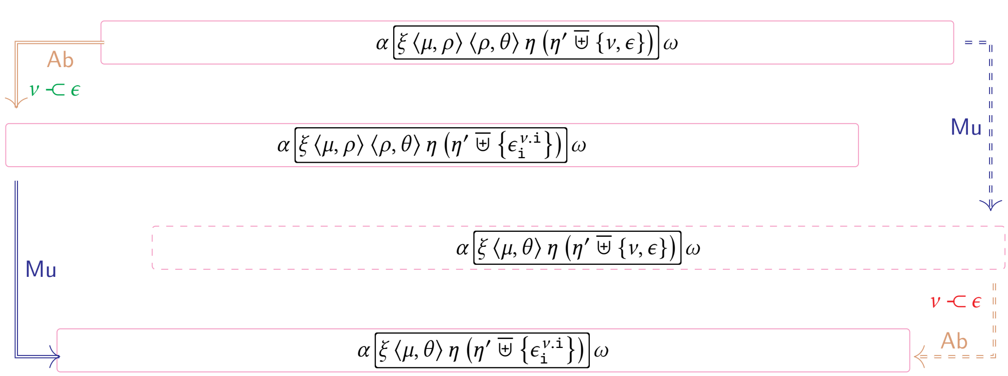
50. The Ti ⇄ Mu case when the tighten'ee appears in both the mumble'er and the mumble'ee.



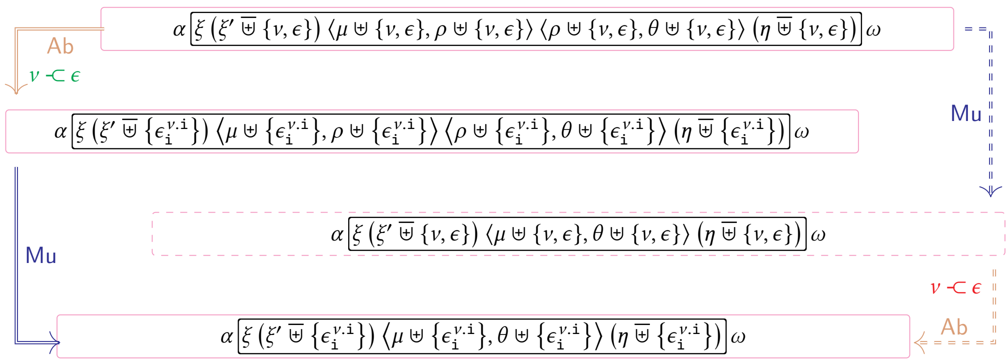
51. The Ti ⇄ Mu case when the tighten'ee appears first in the mumble'er.



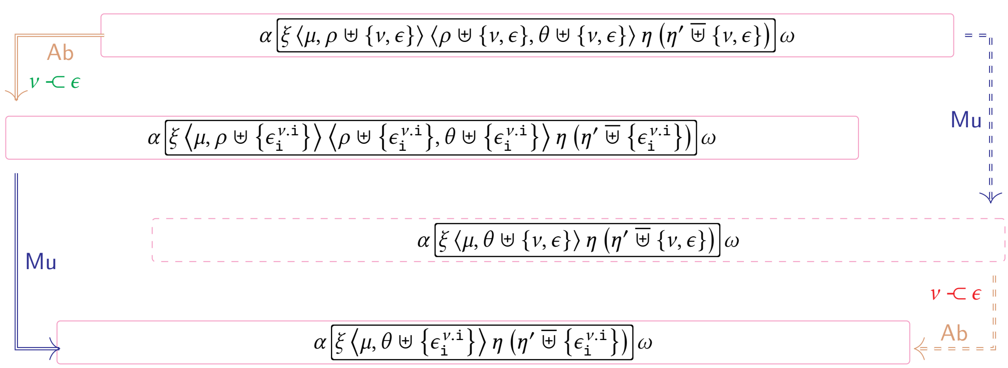
52. The Ti ⇄ Mu case when the tighten'ee appears first in the mumble'ee.



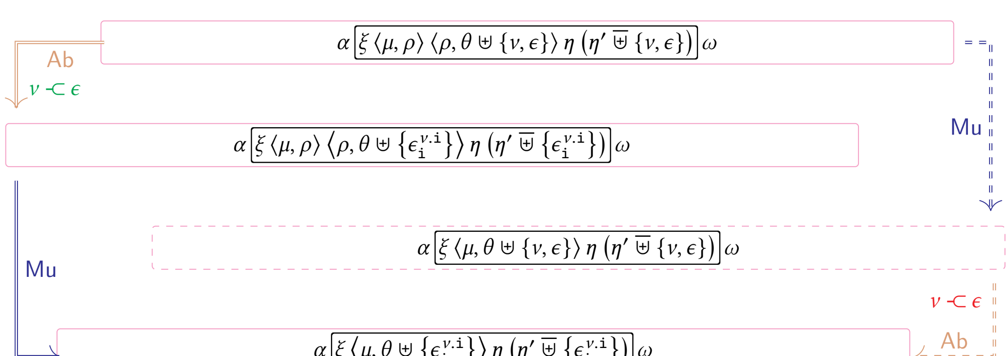
53. The Ab ⇄ Mu case when the absorb'ee appears in neither the mumble'er nor the mumble'ee.



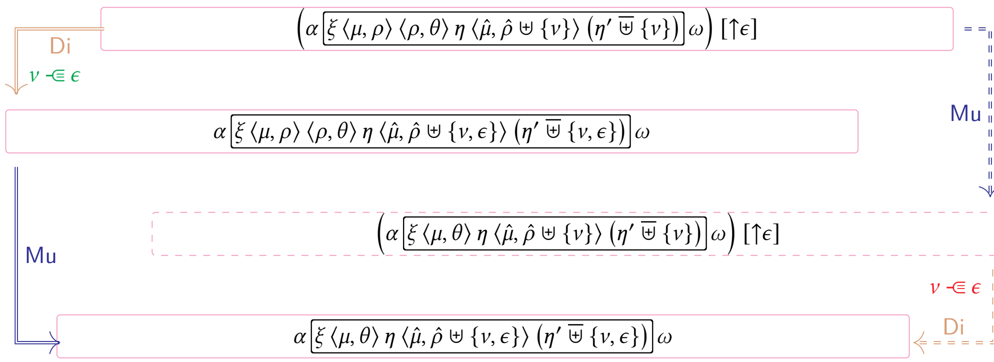
54. The Ab ⇄ Mu case when the absorb'ee appears in both the mumble'er and the mumble'ee.



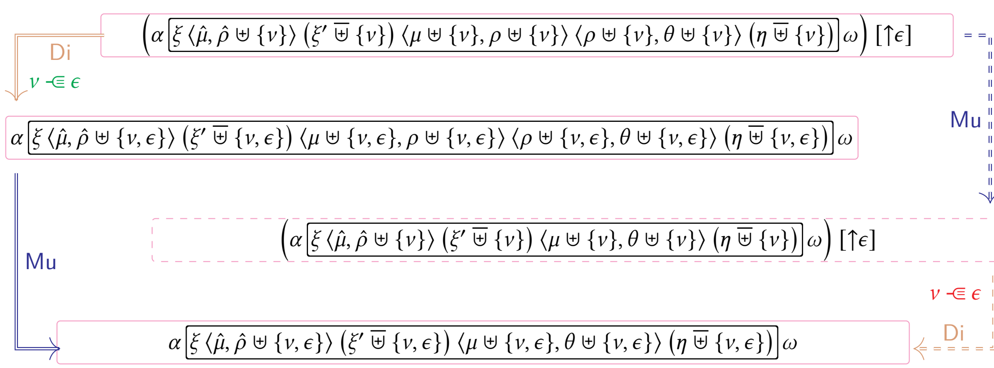
55. The Ab ⇄ Mu case when the absorb'ee appears first in the mumble'er.



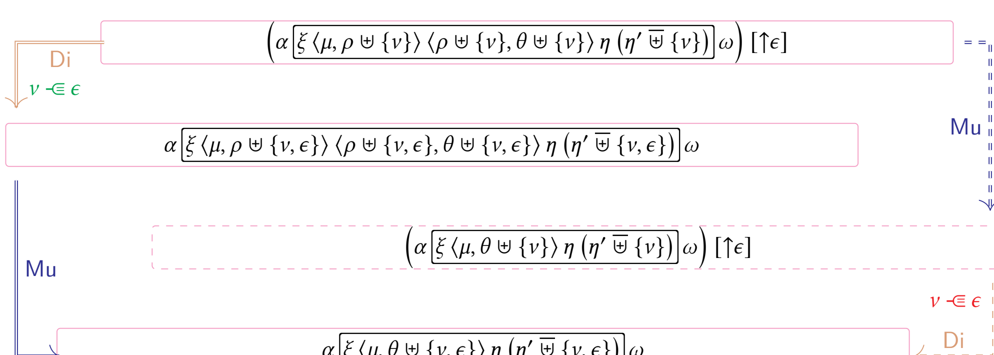
56. The Ab ⇄ Mu case when the absorb'ee appears first in the mumble'ee.



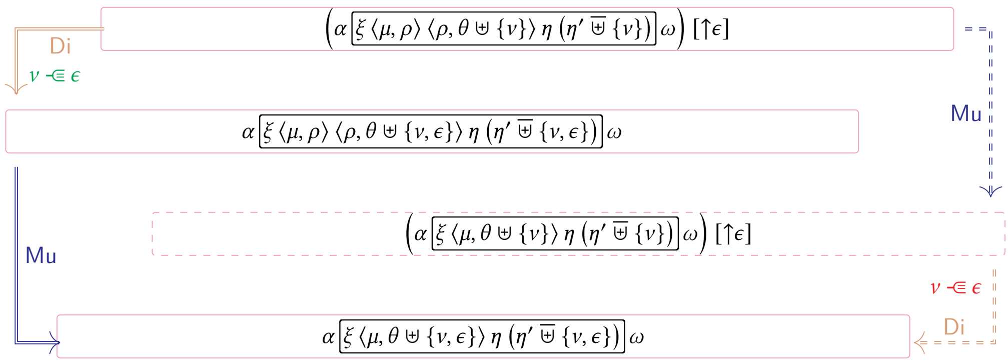
57. The Di ⇄ Mu case when the dilute'ee appears in neither the mumble'er nor the mumble'ee.



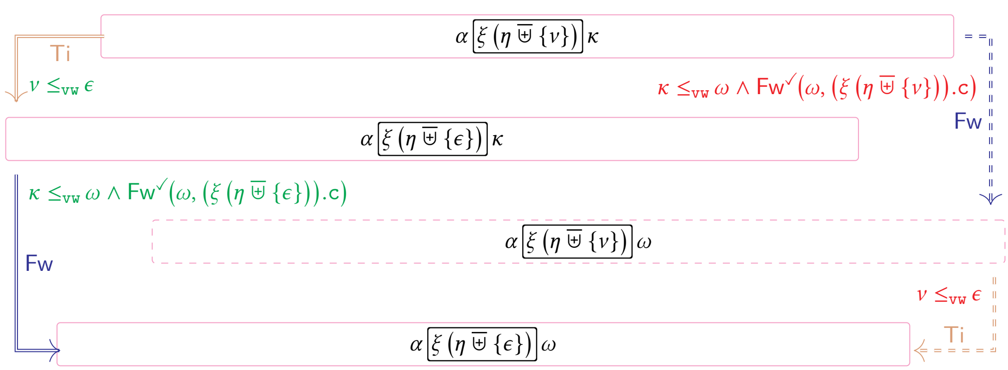
58. The Di ⇄ Mu case when the dilute'ee appears in both the mumble'er and the mumble'ee.



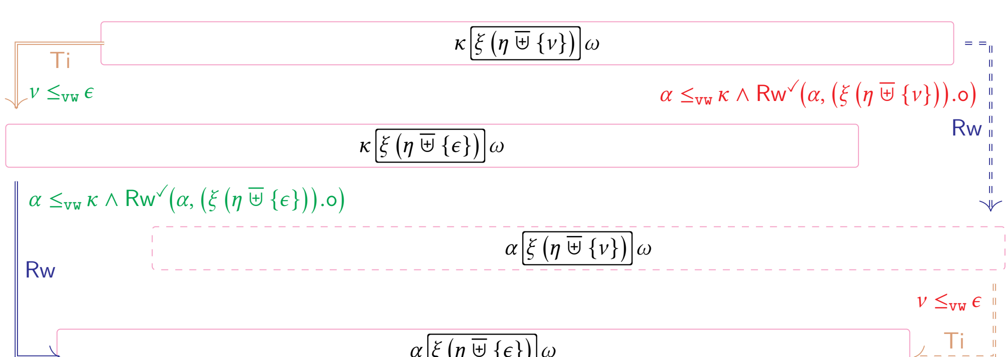
59. The Di ⇄ Mu case when the dilute'ee appears first in the mumble'er.



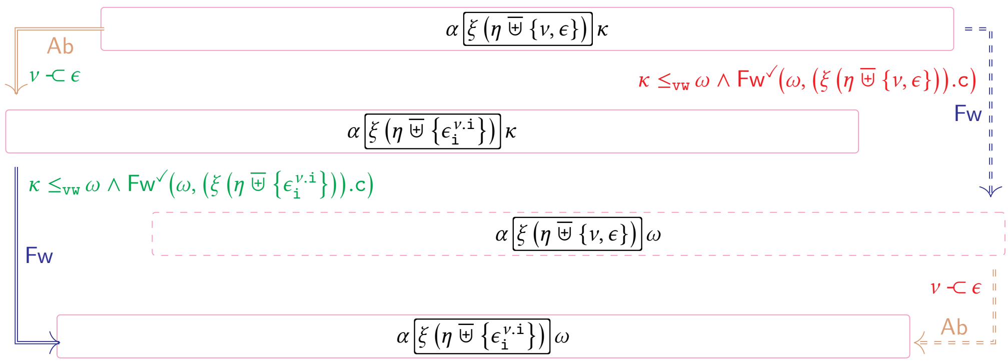
60. The Di ⇄ Mu case when the dilute'ee appears first in the mumble'ee.



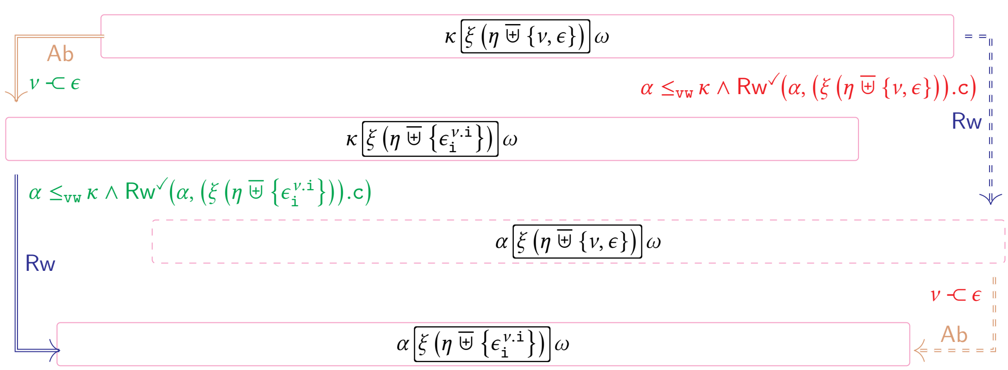
61. The Ti ⇄ Fw case.



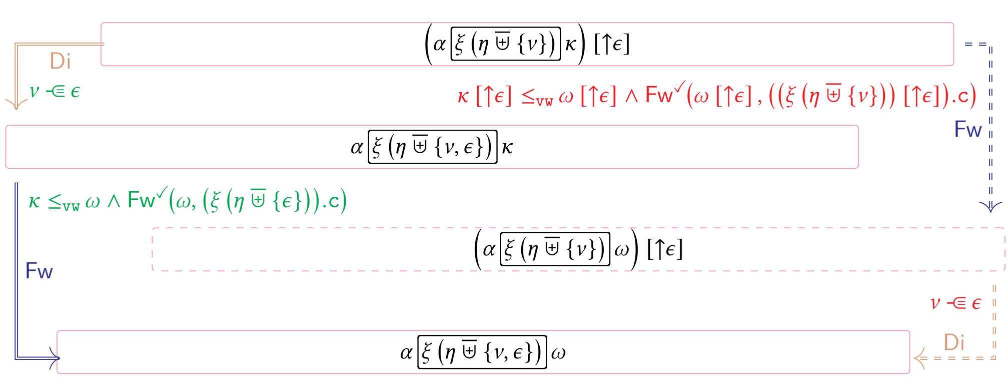
62. The Ti ⇄ Rw case.



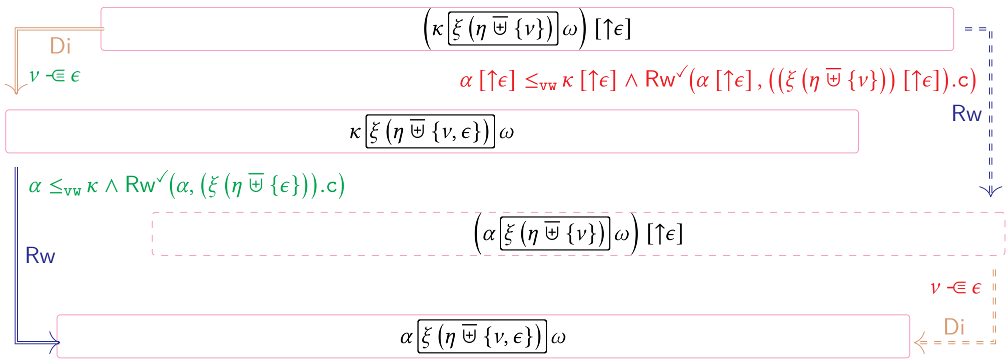
63. The Ab ⇄ Fw case.



64. The Ab ⇄ Rw case.



65. The Di ⇄ Fw case.



66. The Di $\rightleftarrows$ Rw case.