

A Denotational Approach to 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.

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 Denotational Approach to Release/Acquire Concurrency. 1, 1 (April 2024), 80 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 $\llbracket - \rrbracket$ is for it to be *adequate* w.r.t. a given operational semantics: plugging program terms M and N with equal denotations—i.e. $\llbracket M \rrbracket = \llbracket N \rrbracket$ —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 $\llbracket M \rrbracket \leq \llbracket N \rrbracket$ implies that all behaviors exhibited by $\Xi[M]$ under the operational semantics are also exhibited by $\Xi[N]$.

For shared-memory concurrent programming, Brookes’s seminal work [13] defined a denotational semantics, where the denotation $\llbracket M \rrbracket$ 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.

Authors’ addresses: 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2024/4-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

50 Brookes [12] used the insights behind this semantics to develop a semantic model for separation
 51 logic, and Turon and Wand [48] used them to design a separation logic for refinement. Addition-
 52 ally, Xu et al. [50] used traces as a foundation for the Rely/Guarantee approach for verification of
 53 concurrent programs, and Liang et al., Liang et al. [36, 37] used a trace-based program logic for
 54 refinement.

55 A *memory model* decides what outcomes are possible from the execution of a program. Brookes
 56 established the adequacy of the trace-based denotational semantics w.r.t. the operational semantics
 57 of the strongest model, known as *sequential consistency* (SC), where every memory access happens
 58 instantaneously and immediately affects all concurrent threads. However, SC is too strong to model
 59 real-world shared memory, whether it be of modern hardware, such as x86-TSO [42, 46] and ARM,
 60 or of programming languages such as C/C++ and Java [4, 39]. These runtimes follow *weak memory*
 61 *models* that allow performant implementations, but admit more behaviors than SC.

62 Do weak memory models admit adequate Brookes-style denotational semantics? This question
 63 has been answered affirmatively once, by Jagadeesan et al. [25], who closely followed Brookes
 64 to define denotational semantics for x86-TSO. Other weak memory models, in particular, models
 65 of *programming languages*, and *non-multi-copy-atomic* models, where writes can be observed by
 66 different threads in different orders, have so far been out of reach of Brookes’s totally ordered
 67 traces, and were only captured by much more sophisticated models based on *partial orders* [15, 19,
 68 24, 26, 29, 43].

69 In this paper we target the Release/Acquire memory model (RA, for short). This model, obtained
 70 by restricting the C/C++11 memory model to Release/Acquire atomics, is a well-studied fundamen-
 71 tal memory model weaker than x86-TSO, which, roughly speaking, ensures “causal consistency”
 72 together with “per-location-SC” and “RMW (read-modify-write) atomicity” [30, 31]. These assur-
 73 ances make RA sufficiently strong for implementing common synchronization idioms. RA allows
 74 more performant implementations than SC, since, in particular, it allows the reordering of a write
 75 followed by a read from a different location, which is commonly performed by hardware, and it is
 76 non-multi-copy-atomic, thus allowing less centralized architectures like POWER [47].

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

87 A main challenge when developing a denotational semantics lies in making it sufficiently ab-
 88 stract. While *full* abstraction is often out of reach, as a yardstick, we want our semantics to be
 89 able to justify various compiler transformations/optimizations that are known to be sound under
 90 RA [49]. Indeed, an immediate practical application of a denotational semantics is the ability to
 91 provide *local* formal justifications of program transformations, such as those performed by opti-
 92 mizing compilers. In this setting, to show that an optimization $N \rightarrow M$ is valid amounts to showing
 93 that replacing N by M anywhere in a larger program does not introduce new behaviors, which
 94 follows from $\llbracket M \rrbracket \leq \llbracket N \rrbracket$ given a directionally adequate denotation function $\llbracket - \rrbracket$.

95 To support various compiler transformations, we close our denotations under certain opera-
 96 tions, including analogs to Brookes’s stutter and mumble, but also several RA-specific operations,
 97 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, 49].

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., Dvir et al. [6, 20] later recast it atop Moggi’s monad-based approach [40] which uses 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 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 overview 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, along with observations that will support our definition of traces. In §6 we define our trace-based denotational semantics for RA, and in §7 we work up to and establish our main results. Finally, we conclude and discuss related work in §8.

Comparing to the conference version. The conference version of this paper [21] is covered here by §§1-3 and 8, roughly speaking. The rest of this paper extends the conference version. Here, definitions and theorems are formally specified and fully proven. This account 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 commutativity of rewrite rules, which is one of the main technical challenges in the development of the metatheory.

2 PRELIMINARIES

We overview previous work and our slight variations on it to facilitate the abridged discussion of our contribution (§3). Particularly, we partially present the 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 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. Program terms M and N can be composed sequentially explicitly as $M; N$ or implicitly by left-to-right evaluation in the pairing construct $\langle M, N \rangle$. They can be composed in parallel as $M \parallel N$. We assume preemptive scheduling, thus imposing no restrictions on the interleaving

148 execution steps between parallel threads. To introduce the memory-access constructs, we present
 149 the well-known *message passing* litmus test, adapted to the functional setting:

$$150 \quad (x := 1 ; y := 1) \parallel \langle y?, x? \rangle \quad (\text{MP})$$

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

157 In the functional setting, execution results in a returned value: $\ell := v$ returns the unit value $\langle \rangle$,
 158 i.e. the empty tuple; $\ell?$, and the RMW constructs such as $\text{FAA}(\ell, v)$, return the loaded value; $M ; N$
 159 returns what N returns; and $\langle M, N \rangle$, as well as $M \parallel N$, return the pair consisting of the return
 160 value of M and the return value of N . We assume left-to-right execution of pairs, so in the (MP)
 161 example $\langle y?, x? \rangle$ steps to $\langle v, x? \rangle$ for a value v that can be loaded from y , and $\langle v, x? \rangle$ steps to
 162 $\langle v, w \rangle$ for a value w that can be loaded from x . In between, the left side of the parallel composition (\parallel)
 163 can take steps.

164 We can use intermediate results in subsequent computations via *let* binding: **let** $a = M$ **in** N binds
 165 the result of M to a in N . Thus, we execute M first, and substitute the resulting value V for a in
 166 N before executing $N[a \mapsto V]$. Similarly, we deconstruct pairs by matching: **match** M **with** $\langle a, b \rangle$. N
 167 binds the components of the pair that M returns to a and b respectively in N . The first and second
 168 projections **fst** and **snd**, as well as the operation **swap** that swaps the pair constituents, are defined
 169 using **match** standardly.

170 Traditionally, weak memory models are contrasted by finding litmus test programs, such as
 171 (MP), with which one model supports a specific *observable behavior* that the other does not. Since
 172 different models feature quite different notions of internal state, and observing the memory directly
 173 is not considered feasible anyway, internal interactions are ignored. We do not consider
 174 infinite executions in this paper, so we conflate observable behaviors with *outcomes*: values that
 175 the program may evaluate to from given initial memory values. Litmus tests are traditionally de-
 176 signed with all initial memory values set to 0 in mind.

177 **Remark** (Imperative vs. Functional). *Presentations of litmus tests for weak-memory models are*
 178 *usually presented imperatively using local registers* a, b . *This is subsumed in the functional setting by*
 179 *systematically replacing registers with let-bindings.*

180 For example, we can apply this process to the imperative message passing litmus test:

182 Style	Imperative	Functional
184 Program	$x := 1 \parallel a := y$ $y := 1 \parallel b := x$	$x := 1 ; \parallel \text{let } a = y? \text{ in}$ $y := 1 \parallel \text{let } b = x? \text{ in } \langle a, b \rangle$
186 Outcome 187 of interest	An execution that ends with $a = 1 \wedge b = 0$	An evaluations that returns $\langle \rangle, \langle 1, 0 \rangle$

188 Up to standard, memory-model agnostic equivalences, this is our functional presentation (MP).
 189

190 In the strongest memory model of Sequential Consistency (SC), every value stored is imme-
 191 diately made available to every thread, and every dereference must load the latest stored value.
 192 Thus the underlying memory model uses maps from locations to values for the memory state that
 193 evolves during program execution. Given an initial state, the behavior of a program in SC depends
 194 only on the choice of interleaving of steps. In (MP) the order of the two stores and the two loads
 195 ensures that $\langle \rangle, \langle 0, 0 \rangle, \langle \rangle, \langle 0, 1 \rangle$, and $\langle \rangle, \langle 1, 1 \rangle$ are observable, but $\langle \rangle, \langle 1, 0 \rangle$ 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 $x := 1; y := 1; y?$ versus M_2 defined as $x := 1; y := x?; y?$. 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 $- \parallel x := 2$. That is, compare $M_1 \parallel x := 2$ versus $M_2 \parallel x := 2$. 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 $\llbracket M \rrbracket$, i.e. M 's denotation, were to simply map initial states to possible results according to executions of M , we could not define $\llbracket M \parallel N \rrbracket$ in terms of $\llbracket M \rrbracket$ and $\llbracket N \rrbracket$ alone, because we would have $\llbracket M_1 \rrbracket = \llbracket M_2 \rrbracket$ but also $\llbracket M_1 \parallel x := 2 \rrbracket \neq \llbracket M_2 \parallel x := 2 \rrbracket$. We conclude that $\llbracket M \rrbracket$ must contain more information on M than an ‘‘input-output’’ relation; it must account for interference by the environment.

2.2 Brookes’s Trace-based Semantics for Sequential Consistency

A prominent approach to define compositional semantics for concurrent programs is due to Brookes [13], who defined a denotational semantics for SC by taking $\llbracket M \rrbracket$ to be a set of traces of M closed under certain rewrite rules as we detail below. Brookes established a (directional) adequacy theorem: if $\llbracket M \rrbracket \supseteq \llbracket N \rrbracket$ then the transformation $M \rightarrow N$ 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, formal and informal simulation arguments which are used to justify transformations in operational semantics can be replaced by cleaner and simpler proofs based on the denotational semantics. For example, a simple argument shows that $\llbracket x := v; x := w \rrbracket \supseteq \llbracket x := w \rrbracket$ holds in Brookes’s semantics. Thanks to adequacy, this justifies Write-Write Elimination (WW-Elim) $x := v; x := w \rightarrow x := w$ in SC.

Traces in SC. In Brookes’s semantics, a program term is denoted by the 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, $\llbracket x := w \rrbracket$ includes all traces of the form $\langle \rho, \rho[x := w] \rangle$, where $\rho[x := w]$ is equal to ρ except for mapping x to w . The definition is compositional: the traces in $\llbracket x := v; x := w \rrbracket$ are obtained from sequential compositions of traces from $\llbracket x := v \rrbracket$ with traces from $\llbracket x := w \rrbracket$, obtaining all traces of the form $\langle \mu, \mu[x := v] \rangle \langle \rho, \rho[x := w] \rangle$. Such a trace relies on μ in order to guarantee $\mu[x := v]$, and then relies on ρ in order to guarantee $\rho[x := w]$. Allowing $\rho \neq \mu[x := v]$ reflects the possibility of environment interference between the two store instructions. Indeed, when denoting parallel composition $\llbracket M \parallel N \rrbracket$ we include all traces obtained by interleaving transitions from a trace from $\llbracket M \rrbracket$ with transitions from a trace from $\llbracket N \rrbracket$. 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 $\langle \xi \rangle . r$, 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 $\llbracket x? \rrbracket$ includes all traces of the form $\langle \mu, \mu \rangle . \mu(x)$. Indeed, the execution of $x?$ does not change the memory and returns the value loaded from x . In the semantics of assignment $\llbracket x := v \rrbracket$, instead of $\langle \mu, \mu[x := v] \rangle$ we have $\langle \mu, \mu[x := v] \rangle . \langle \rangle$.

246 Rewrite rules in SC. Were denotations in **Brookes**'s semantics defined to *only* include the traces
 247 explicitly mentioned above, it would not be abstract enough to justify (**WW-Elim**), which elimi-
 248 nates redundant writes. Indeed, we only saw traces with two transitions in $\llbracket x := v ; x := w \rrbracket$, but
 249 in $\llbracket x := w \rrbracket$ we saw traces with one. The semantics would still be adequate, but it would lack ab-
 250 straction. This is where **Brookes**'s second main idea comes into play, making the denotations more
 251 abstract by closing them under two operations that rewrite traces:

252 **Stutter** adds a transition of the form $\langle \mu, \mu \rangle$ anywhere in the trace. Intuitively, a program term can
 253 always guarantee what it relies on.

254 **Mumble** combines a couple of subsequent transitions of the form $\langle \mu, \rho \rangle \langle \rho, \theta \rangle$ into a single transi-
 255 tion $\langle \mu, \theta \rangle$ anywhere in the trace. Intuitively, a program term can always omit a guarantee
 256 to the environment, and rely on its own omitted guarantee instead of relying on the envi-
 257 ronment.
 258

259 Denotations in **Brookes**'s semantics are defined to be sets of traces *closed* under rewrite rules:
 260 applying a rewrite to a trace in the set results in a trace that is also in the set. For example, $\llbracket x := w \rrbracket$
 261 is the least closed set with all traces of the form $\langle \rho, \rho [x := w] \rangle$, and $\llbracket x := v ; x := w \rrbracket$ is the
 262 least closed set with all sequential compositions of traces from $\llbracket x := v \rrbracket$ with traces from $\llbracket x := w \rrbracket$.
 263

264 Closure under these rules makes traces in $\llbracket M \rrbracket$ correspond precisely to *interrupted executions*
 265 of M , which are executions of M in which the memory can arbitrarily change between steps of
 266 execution. Each transition $\langle \mu, \rho \rangle$ in a trace in $\llbracket M \rrbracket$ corresponds to multiple execution steps of M
 267 that transition μ into ρ , and each gap between transitions accounts for possible environment in-
 268 terruption. The rewrite rules maintain this correspondence: stutter corresponds to taking 0 steps,
 269 and mumble corresponds to taking $n + m$ steps instead of taking n steps and then m steps when
 270 the environment did not change the memory in between. **Brookes**'s adequacy proof is based on
 271 this precise correspondence. In particular, the single-pair traces in $\llbracket M \rrbracket$ correspond to the (unin-
 272 terrupted) executions, the “input-output” relation, of M .
 273

274 2.3 Overview of Release/Acquire Operational Semantics

275 Memory accesses in RA are more subtle in than in SC. To address this we adopt Kang et al.'s “view-
 276 based” machine [28], an operational presentation of RA proven to be equivalent to the original
 277 declarative formulation of RA [e.g. 31]. In this model, rather than the memory holding only the
 278 latest value written to every variable, the memory accumulates a set of memory update messages
 279 for each location. Each thread maintains its own *view* that captures which messages the thread can
 280 observe, and is used to constrain the messages that the thread may read and write. The messages in
 281 the memory carry views as well, which are inherited from the thread that wrote the message, and
 282 passed to any thread that reads the message. Thus views indirectly maintain a causal relationship
 283 between messages in memory throughout the evolution of the system.

284 More concretely, causality is enforced by timestamping messages, thus placing them on their
 285 location's *timeline*. A view κ associates a timestamp κ_ℓ to each location ℓ , obscuring the portion
 286 of ℓ 's timeline before κ_ℓ . The view *points to* a message at ℓ with timestamp κ_ℓ . Messages point to
 287 messages via the view they carry, and must point to themselves.

288 To capture the atomicity of RMWs, each message occupies a half-open segment $(q, t]$ on their
 289 location's timeline, where t is the message's timestamp. A message with segment $(q, t]$ *dovetails*
 290 with a message at the same location with timestamp q , if there is one. When an RMW writes it
 291 must “modify” the message from which it read by dovetailing with it.

292 We explain our notation for messages by example. Assuming of two location, x and y , we denote
 293 by $x:1@(.5, 1.7] \langle y@3.5 \rangle$ the message at location x that carries the value 1, occupies the segment
 294

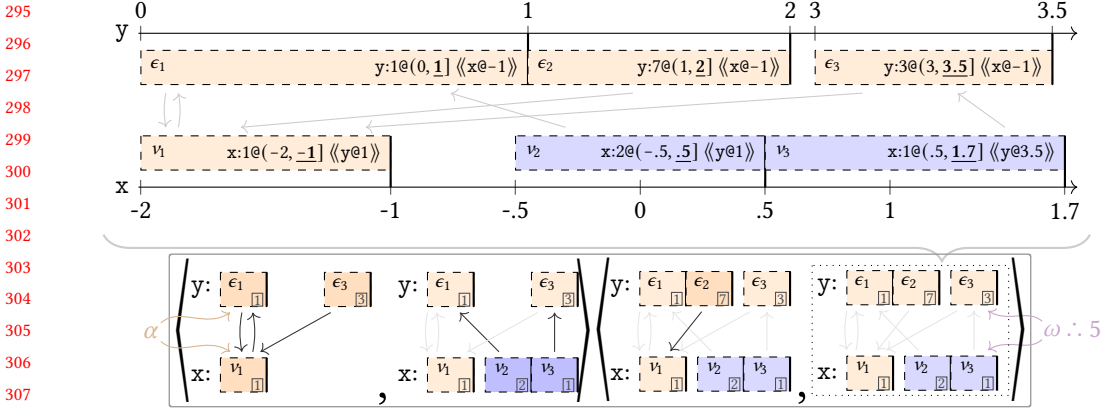


Fig. 1. Illustrations of a memory (top) and a trace (bottom), in the setting of two memory locations, x and y. **Top:** A memory holding six messages. The timelines are purposefully misaligned and not to scale to emphasize that timestamps for different locations are incomparable and that only the order between them is relevant. The graph structure that the views impose is illustrated by arrows pointing between messages. Messages that are not dovetailed are set apart, e.g. v_3 dovetails with v_2 , which does not dovetail with v_1 . **Bottom:** A trace with two transitions: $\alpha \left[\langle \mu_1, \rho_1 \rangle \langle \mu_2, \rho_2 \rangle \right] \omega . : 5$. The memory illustrated on top is ρ_2 . Messages and edges that are not part of a previous memory are highlighted. The local messages are v_2 and v_3 , and the rest are environment messages.

$(.5, 1.7]$ on x's timeline, and carries the view κ such that $\kappa_x = 1.7$ and $\kappa_y = 3.5$ (every message points to itself). An example memory is depicted at the top of Figure 1.

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 with a given message. A thread can only read from revealed messages, and when it reads, its view increases as needed to *dominate* the view of the loaded message, where 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.

Revisiting the (MP) litmus test, starting 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 depicted on the left side of Figure 2. 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. The right thread inherits this limitation through the causal relationship, so it will not be able to load 0 from x. Therefore, RA forbids the outcome $\langle \langle \rangle, \langle 1, 0 \rangle \rangle$.

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

$$(x := 1 ; y?) \parallel (y := 1 ; x?) \quad (\text{SB})$$

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 depicted on the right side of Figure 2.

We can recover the SC behavior by interspersing fences between sequenced memory accesses, which we model with FAA ($z, 0$) to a fresh location z. Thus, compare (SB) to the *store buffering with fences* litmus test:

$$(x := 1 ; \text{FAA}(z, 0) ; y?) \parallel (y := 1 ; \text{FAA}(z, 0) ; x?) \quad (\text{SB+F})$$

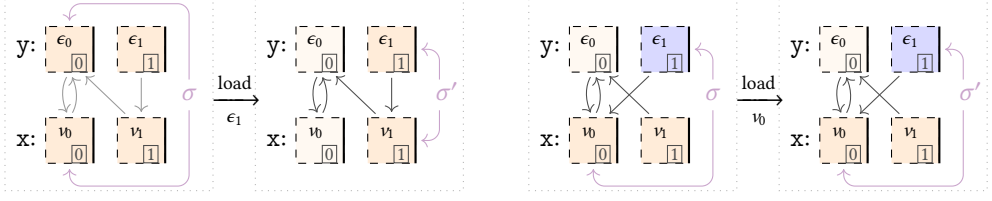


Fig. 2. Depictions of a step during an execution of a litmus test, with the view of the right thread changing from σ to σ' . The value each message carries is in its bottom-right corner. Views are illustrated implicitly in the graph structure that they impose. Obscured messages are faded. **Left:** As the right thread in (MP) loads 1 from y , it inherits the view of ϵ_1 , obscuring ν_0 . **Right:** The right thread in (SB) loading 0 from x . Storing ϵ_1 did not obscure ν_0 .

Both of the FAA ($z, 0$) instructions store messages that must dovetail with the message that they load from, and in that also inherit its view. They cannot both dovetail with the same message because their segments cannot intersect. Thus, one of them—say, the one on the right—will have to dovetail with the other. In this scenario, the view of the message that the left thread stores at z points to the message it previously stored at x . When the right thread loads the message from z it inherits this view, obscuring the initial message to x . Therefore, when it later loads from x , it must load what the left thread stored. Thus, like in SC, no execution in RA returns $\langle 0, 0 \rangle$.

3 CONTRIBUTION SUMMARY

We begin by showcasing our notion of a trace, which we adapt to RA both in the structure of the trace itself, as well as in the rewrite 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 in (§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 . Intuitively, these play a similar role here, formally grounded in analogs to the stutter and mumble rewrite rules. Seeing that the operational semantics only adds messages and never modifies them, we 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 are relied upon to be revealed in ξ 's opening memory; and a final view ω , declaring which messages are guaranteed to be revealed in ξ 's closing memory. We write the trace as $\alpha \llbracket \xi \rrbracket \omega \cdot r$. See bottom of Figure 1 for an illustrated example.

RA specific rewrite-rules. We add several more bespoke RA-specific rewrite rules to close denotations under, making the denotational semantics more abstract. For example, (WW-Elim) is also valid under RA. The reasoning we have used to justify it under SC, by showing $\llbracket x := v ; x := w \rrbracket \supseteq \llbracket x := w \rrbracket$ in Brookes's semantics, will only get us so far here. Replicating the process, the trace we end up with in $\llbracket x := v ; x := w \rrbracket$ after rewriting with mumble has two local messages, whereas traces from $\llbracket x := w \rrbracket$ only have a single local message. Roughly speaking, the equality concerning SC memories $\mu[x := v][x := w] = \mu[x := w]$ does not transfer to RA where memory, by accumulating messages, is more concrete. We resolve this by adding the absorb rewrite rule, which

replaces two dovetailed local messages with one that carries the second message’s value. Thus, in the proof for RA we follow the mumble rewrite with an absorb rewrite.

Internalized operational invariants. We further increase abstraction in our denotations by pointing out and internalizing properties of the operational semantics. Without restrictions, traces may represent behaviors that include operationally unreachable states. Forbidding such redundant traces eliminates a source of differentiation between denotations, thus increasing their abstraction.

Specifically, consider the transformation $x? ; y? \rightarrow y?$, a consequence of the RA-valid Irrelevant Read Elimination (R-Elim) $x? ; \langle \rangle \rightarrow \langle \rangle$ and structural equivalences. Consider the state S that consists of the memory at the top of Figure 1 and the view that points to v_3 and ϵ_2 . The only step $x? ; y?$ 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 $y?$ 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 ignore unreachable states in the operational semantics, we discard “unreachable” traces to refine our denotational semantics. This move allows us to justify (R-Elim): we have $\llbracket x? ; \langle \rangle \rrbracket \supseteq \llbracket \langle \rangle \rrbracket$.

3.2 Compositionality and the Monadic Presentation

One of the contributions of this work is to bridge research of weak-memory models with Moggi’s monad-based approach [40] to denotational semantics. This approach also comes with practical benefits, such as 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 stays valid through modular expansions with effects. For instance, if K is effect-free, then:

$$\llbracket \text{if } K \text{ then } M ; N \text{ else } M ; N' \rrbracket = \llbracket M ; \text{if } K \text{ then } N \text{ else } N' \rrbracket$$

So-called structural equivalences 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, in which programmable functions can take functions as input and return functions as output. Moggi’s approach supports this feature out-of-the-box in such a way that does not complicate the rest of the semantics, as the first-order fragment of the semantics need not change to include it.

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

A term is a *program* if it is *closed* (every variable occurrence is bound) and of *ground type* (all functions are applied to arguments). This definition is in line with the expectation that a program should return a concrete result that the end-user can consume. Thus, we only consider observable behaviors of programs. Transformations only need to be valid when applied within programs. Programs degenerate to closed terms in the first-order fragment.

To deal with the need to prove properties “pointwise” that abstractions bring about we use logical relations. Moggi’s toolkit provides a standard way to define these, thereby lifting properties to their higher-order counterparts.

Compositionality. In its most basic form, this key feature of denotational semantics means that a program 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 are interested in establishing a directional generalization of compositionality:

Compositionality (Thm. 7.7). *For a program context $\Xi [-]$, if $\llbracket M \rrbracket \subseteq \llbracket N \rrbracket$ then $\llbracket \Xi [M] \rrbracket \subseteq \llbracket \Xi [N] \rrbracket$.*

This is a consequence of the monadic design of the denotational semantics using monotonic operators, and is not substantially different from previous work [e.g. 20].

3.3 Relating the Denotational Semantics to the Operational Semantics

Kang et al. presentation assumes top-level parallelism, a common practice in studies of weak-memory models. This comes at the cost of the uniformity and compositionality. In particular, the denotation $\llbracket M \parallel N \rrbracket$ cannot be defined. We resolve this by extending 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, *states* that accompany executing terms consist of a memory and a view-tree. In discourse, we do not distinguish between a view-leaf and its label.

Remark. *Handling parallel composition as a first-class construct allows us to decompose Write-Read Reordering (WR-Reord) $(x := v) ; y? \rightarrow \text{fst} \langle y?, (x := v) \rangle$, a crucial reordering of memory accesses valid under RA but not under SC, into a combination of Write-Read Deorder (WR-Deord) $\langle (x := v), y? \rangle \rightarrow (x := v) \parallel y?$ together with structural transformations and laws of parallel programming:*

$$\begin{array}{c}
 \begin{array}{ccc}
 & \downarrow_{\text{Structural}} & \downarrow_{(\text{WR-Deord})} \\
 (x := v) ; y? & \rightarrow \text{snd} \langle (x := v), y? \rangle & \rightarrow \text{snd} ((x := v) \parallel y?) \\
 \downarrow_{\text{Par. Prog. Law: Symmetry}} & & \downarrow_{\text{Structural}} \quad \downarrow_{\text{Par. Prog. Law: Sequencing}} \\
 \rightarrow \text{snd} (\text{swap} (y? \parallel (x := v))) & \rightarrow \text{fst} (y? \parallel (x := v)) & \rightarrow \text{fst} \langle y?, (x := v) \rangle
 \end{array}
 \end{array}$$

This provides a separation of concerns: the components of this decomposition are supported by our semantics using independent arguments. It also sheds a light on the interesting part, as they are all valid under SC except for (WR-Deord).

Observability correspondence. We call some of our rewrite rules *abstract*, such as absorb, and others *concrete*, such as stutter and mumble. We denote the *basic denotation* of a term M by $\llbracket M \rrbracket$, which is the denotation were it defined using only the concrete rewrite rules. Traces in the basic denotations directly correspond to interrupted executions, but not so in the (regular) denotations. For example, in our analysis of (WW-Elim), by using absorb, we ended up with a trace in which only one message is added even though the program term adds two messages. Thus, the abstract rewrite rules break the direct correspondence.

Still, some indirect correspondence should remain to justify adequacy. In particular, we would like traces to correspond to observable behavior of programs. In one direction, an even stronger property holds, known as soundness:

Soundness (Thm. 7.8). *For every execution of a program M in the operational semantics of RA, there exists $\alpha \langle \mu, \rho \rangle \omega \cdot r \in \llbracket M \rrbracket$ that matches the execution: $\langle \alpha, \mu \rangle$ is the initial state, $\langle \omega, \rho \rangle$ is the final state, and r matches the value returned.*

To prove soundness, we take a trace where transitions correspond to the memory-accessing execution steps, and then use mumble to obtain a single transition.

Ignoring the final state, the correspondence holds in the other direction too:

Evaluation Lemma (Lem. 7.10). *For every program M and $\alpha \langle \mu, \rho \rangle \omega \cdot r \in \llbracket M \rrbracket$ there is an observable behavior of M with initial state $\langle \alpha, \mu \rangle$ and return value matching r .*

491 The lack of correspondence with the final state is an artifact of the concreteness-abstraction
 492 divergence between the operational and denotational semantics. Due to this divergence, it is sig-
 493 nificantly more challenging to establish this direction of the correspondence than in previous work.
 494 The challenge lies primarily in proving that abstract rewrite rules can be applied retroactively, de-
 495 ferring them to the top-level. That is, denoting closure under the abstract rewrite rules by $-\dagger$, we
 496 claim:

497 **Retroactive Closure (Lem. 7.4).** *If M is a program, then $\llbracket M \rrbracket^\dagger = \llbracket M \rrbracket$.*
 498

499 Thus, to obtain all of the traces in the (regular) denotation of a term, it is enough to close only
 500 under the concrete rewrite rules as the denotation of a program is built-up from its subterms,
 501 applying the abstract rewrite rules only at the top level.

502 The intuition that guides the proof is that the abstract rewrite rules can be percolated outwards:
 503

504 **Rewrite Commutativity (Lem. 7.1).** *Let τ and ϱ be traces such that τ can be rewritten to ϱ using
 505 both concrete and abstract rewrite rules (denoted $\tau \xrightarrow{ca} \varrho$). Then there exists a trace π , such that τ can
 506 be rewritten to π using only concrete rewrite rules (denoted $\tau \xrightarrow{c} \pi$), and π can be rewritten to ϱ using
 507 only abstract rewrite rules (denoted $\pi \xrightarrow{a} \varrho$).*
 508

509 The central result is (directional) adequacy, stating that denotational approximation corresponds
 510 to refinement of observable behaviors:
 511

512 **Adequacy (Thm. 7.9).** *If $\llbracket M \rrbracket \subseteq \llbracket N \rrbracket$, then for all program contexts $\Xi [-]$, every observable behavior
 513 of $\Xi [M]$ is an observable behavior of $\Xi [N]$.*
 514

515 In particular, $\llbracket M \rrbracket \subseteq \llbracket N \rrbracket$ implies that $N \rightarrow M$ is valid under RA, because the effect of applying
 516 it is unobservable. Adequacy follows immediately from the above results. Indeed, using soundness,
 517 an observable behavior of $\Xi [M]$ corresponds to a single-transition $\tau \in \llbracket \Xi [M] \rrbracket$; by the assumption
 518 and compositionality $\tau \in \llbracket \Xi [N] \rrbracket$; and using the evaluation lemma, τ corresponds to an observable
 519 behavior of $\Xi [N]$.

520 *Abstraction.* Brookes’s semantics is *fully abstract*, meaning that the converse to adequacy also
 521 holds: if $N \rightarrow M$ is valid under SC, then $\llbracket N \rrbracket \supseteq \llbracket M \rrbracket$. However, Brookes’s proof relies on an
 522 artificial program construct, **await**, that permits waiting for a specified memory snapshot and
 523 then step (atomically) to a second specified memory snapshot. Thus, in realistic languages, when
 524 this construct is unavailable, Brookes’s full abstraction proof does not apply.

525 Nevertheless, even without full abstraction, one can still provide evidence that an adequate
 526 semantics is abstract by ensuring that it supports known transformations.

527 To the best of our knowledge, all transformations $N \rightarrow M$ proven to be valid under RA in
 528 the existing literature are supported by our denotational semantics, i.e. $\llbracket N \rrbracket \supseteq \llbracket M \rrbracket$. Structural
 529 transformations are supported by virtue of using Moggi’s standard semantics. Our semantics also
 530 validates “algebraic laws of parallel programming”, such as sequencing $M \parallel N \rightarrow \langle M, N \rangle$ and its
 531 generalization that Hoare and van Staden [22] recognized, $(M_1 ; M_2) \parallel (N_1 ; N_2) \rightarrow (M_1 \parallel N_1) ;$
 532 $(M_2 \parallel N_2)$, which in the functional setting can take the more expressive form in which the values
 533 returned are passed on to the following computation. See Figure 3 for a partial list.

534 4 LANGUAGE AND TYPING

535 We consider a standard extension of Moggi’s [40] computational lambda calculus with products
 536 and variants (labeled sums) further extending it with shared-memory constructs. We parameterize
 537 our language, which we call λ_{RA} , by its globally available locations, the values we store in and
 538

540 Laws of Parallel Programming			
541	Symmetry	$M \parallel N$	\rightarrow swap ($N \parallel M$)
542	Generalized Sequencing		
543		$(\text{let } a = M_1 \text{ in } M_2) \parallel (\text{let } b = N_1 \text{ in } N_2)$	\rightarrow match $M_1 \parallel N_1$ with $\langle a, b \rangle. M_2 \parallel N_2$
544			
545	Eliminations		
546	Irrelevant Read	$\ell? ; \langle \rangle$	\rightarrow $\langle \rangle$
547	Write-Write	$\ell := v ; \ell := w$	$\xrightarrow{\text{Ab}}$ $\ell := w$
548	Write-Read	$\ell := v ; \ell?$	\rightarrow $\ell := v ; v$
549	Write-FAA	$\ell := v ; \text{FAA}(\ell, w)$	$\xrightarrow{\text{Ab}}$ $\ell := (v + w) ; v$
550	Read-Write	$\text{let } a = \ell? \text{ in } \ell := (a + v) ; a$	\rightarrow $\text{FAA}(\ell, v)$
551	Read-Read	$\langle \ell?, \ell? \rangle$	\rightarrow $\text{let } a = \ell? \text{ in } \langle a, a \rangle$
552	Read-FAA	$\langle \ell?, \text{FAA}(\ell, v) \rangle$	\rightarrow $\text{let } a = \text{FAA}(\ell, v) \text{ in } \langle a, a \rangle$
553	FAA-Read	$\langle \text{FAA}(\ell, v), \ell? \rangle$	\rightarrow $\text{let } a = \text{FAA}(\ell, v) \text{ in } \langle a, a + v \rangle$
554	FAA-FAA	$\langle \text{FAA}(\ell, v), \text{FAA}(\ell, w) \rangle$	$\xrightarrow{\text{Ab}}$ $\text{let } a = \text{FAA}(\ell, v + w) \text{ in } \langle a, a + v \rangle$
555			
556	Others		
557	Irrelevant Read Introduction	$\langle \rangle$	\rightarrow $\ell? ; \langle \rangle$
558	Read to FAA	$\ell?$	$\xrightarrow{\text{Di}}$ $\text{FAA}(\ell, 0)$
559	Write-Read Deorder	$\langle (\ell := v), \ell'? \rangle$	$\xrightarrow{\text{Ti}}$ $(\ell := v) \parallel \ell'? \quad (\ell \neq \ell')$
560	Write-Read Reorder	$(\ell := v) ; \ell'?$	$\xrightarrow{\text{Ti}}$ $\text{fst } \langle \ell', (\ell := v) \rangle \quad (\ell \neq \ell')$
561			
562			
563			

564 **Fig. 3.** A selective list of supported non-structural transformations. Along with Symmetry, the denotational
 565 semantics supports all symmetric-monoidal laws with the binary operator (\parallel) and the unit $\langle \rangle$. Similar trans-
 566 formations, replacing FAA with other RMWs, are supported too. The abstract rewrite rules used to validate
 567 a transformation is mentioned, if there is one.

568 retrieve from these locations, and the primitives we use to atomically mutate these values through
 569 a unified *read-modify-write* construct.

570
 571 *Locations and Storable Values.* We fix two finite sets of (*shared memory*) *locations* Loc , ranged
 572 over by ℓ, ℓ' ; and (*storable*) *values* Val , ranged over by v, w, u . For example, we may take Loc and
 573 Val to be all 64-bit sequences. In concrete examples, we will use concrete names such as x, y, z
 574 for distinct locations, and numbers for values. For simplicity, we don't include primitives (such as
 575 addition) explicitly, since they require standard minor changes.

576
 577 *Read-modify-write (RMW).* These constructs atomically read a value from memory and possibly
 578 modify it to some other computed value. Typical languages include the following constructs, which
 579 are efficiently compiled to hardware: **Compare-and-Swap**: modify when stored value match a pa-
 580 rameter; **Fetch-and-Add**: increase by a parameter; and **Exchange**: always modify to a parameter.
 581 For convenience, we include a single RMW construct that expresses all such operations, *as well*
 582 *as standard loads*. This generalization, especially bringing together loads with RMW operations, is
 583 non-standard, but makes our development more uniform.

584 Formally, a *modifier* is a partial function $\Phi : \text{Val} \rightarrow \text{Val}$, which represents an RMW operation
 585 that reads a value v from memory; and if Φ is defined on v , atomically writes $\Phi(v)$ instead. For
 586 supporting parameters, an *n-ary modifier* is a partial function $\varphi_- : \text{Val}^n \times \text{Val} \rightarrow \text{Val}$. Our lan-
 587 guage requires a family RMW, indexed by the natural numbers, consisting of sets RMW_n of *n*-ary
 588

589 modifiers which we call *primitive* modifiers. For example, the primitive modifiers for common
 590 operations, which have efficient implementations on hardware, are as follows, where \perp means
 591 ‘undefined’:

592 **Load** $\text{load}(v) := \perp$ **Compare-and-Swap** $\text{cas}_{\langle w,u \rangle}(v) := \text{if } v = w \text{ then } u \text{ else } \perp$

593 **Exchange** $\text{xchg}_{\langle w \rangle}(v) := w$ **Fetch-and-Add** $\text{faa}_{\langle w \rangle}(v) := v + w$

594 We omitted load’s 0-parameters ($\langle \rangle$); faa requires a semantic addition operator on values (+).
 595

596 *Syntax.* Given parameters Loc, Val, and RMW, Figure 4(a) presents λ_{RA} ’s syntax. The types are
 597 standard, comprising tuple, sum and function types. We draw constructor names for variants from
 598 a countably infinite set $\iota \in \text{Lab}$. Assuming Lab contains Loc and Val, we identify them with sum
 599 types **Val** and **Loc** whose constructors are the locations and values, each labeling the empty tuple
 600 type.

601 The language’s terms are also standard other than the highlighted shared-memory constructs.
 602 We draw program variables from a countably infinite set $a, b, c \in \text{PVar}$. For simplicity, we assume
 603 labels and variables are distinct: $\text{PVar} \cap \text{Lab} = \emptyset$. Tuple and variant constructors are standard, and
 604 we require the total sum type to disambiguate each variant constructor, which we omit when this
 605 type can be inferred. The pattern matching constructs for tuples and variants are standard, with
 606 binding variables occurrences in each pattern. In the tuple case, we require the variables in the pat-
 607 tern to be distinct. Function abstraction and application are standard, and we annotate the bound
 608 variable with its type, again omitting the annotation when we can infer it. The operational seman-
 609 tics, defined in §5, follows a call-by-value evaluation strategy, and except for parallel composition
 610 $M \parallel N$, evaluation follows a left-to-right convention.

611 We index the RMW construct with a primitive modifier $\varphi \in \text{RMW}$, and its first argument is a
 612 location from which to read and possibly modify, followed by a tuple supplying the parameters.
 613

616	$A, B ::=$	type	$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A}$	$\frac{\Gamma, a : A \vdash M : B}{\Gamma \vdash \lambda a : A. M : A \rightarrow B}$
617	$A \rightarrow B$	function		
618	$ (A_1 * \dots * A_n)$	tuple/product	$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : A}$	$\frac{\Gamma \vdash N : A \rightarrow B}{\Gamma \vdash NM : B}$
619	$ \{ \iota_1 \text{ of } A_1$	variant/sum		
620	$ \dots \iota_n \text{ of } A_n \}$			$\frac{\forall i. \Gamma \vdash M_i : A_i}{\Gamma \vdash \langle M_1, \dots, M_n \rangle : (A_1 * \dots * A_n)}$
621	$M, N ::=$	term		
622	a	variable/identifier		
623	$ \lambda a : A. M$	abstraction		$\frac{\Gamma \vdash M : (A_1 * \dots * A_n)}{\Gamma, a_1 : A_1, \dots, a_n : A_n \vdash N : A}$
624	$ MN$	application		$\frac{\Gamma \vdash \text{match } M \text{ with } \langle a_1, \dots, a_n \rangle. N : A}{\Gamma \vdash M : A_i \quad A = \{ \iota_1 \text{ of } A_1 \dots \iota_n \text{ of } A_n \}}$
625	$ \langle M_1, \dots, M_n \rangle$	tuple		$\frac{\Gamma \vdash A. \iota_i M : A}{\Gamma \vdash M : \{ \iota_1 \text{ of } A_1 \dots \iota_n \text{ of } A_n \}}$
626	$ A. \iota M$	variant		$\frac{\forall i. \Gamma, a_i : A_i \vdash N_i : A}{\Gamma \vdash \text{match } M \text{ with } \{ \iota_1 a_1. N_1 \dots \iota_n a_n. N_n \} : A}$
627	$ \text{match } M \text{ with}$	pattern matching		
628	$\langle a_1, \dots, a_n \rangle. N$	on tuples		
629	$ \text{match } M \text{ with}$	on variants		
630	$\{ \iota_1 a_1. N_1 \dots \iota_n a_n. N_n \}$			
631		shared-state concurrency		$\frac{\varphi \in \text{RMW}_n \quad \Gamma \vdash M : \text{Loc} \quad \Gamma \vdash N : \text{Val}^n}{\Gamma \vdash \text{rmw}_\varphi(M; N) : \text{Val}}$
632	$ \text{rmw}_\varphi(M; N)$	read-modify-write		
633	$ M := N$	write	$\frac{\Gamma \vdash M : \text{Loc} \quad \Gamma \vdash N : \text{Val}}{\Gamma \vdash M := N : 1}$	$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash M \parallel N : (A * B)}$
634	$ M \parallel N$	parallel composition		

635
636
637 **Fig. 4.** The λ_{RA} -calculus: (a) syntax and (b) typing rules.

The term $\mathbf{rmw}_\varphi(M; N)$ executes by evaluating M to a location ℓ , then evaluating N to a tuple of values $\vec{w} = \langle w_1, \dots, w_{\varphi.\text{ar}} \rangle$. Then, atomically, reading a value v from ℓ and writing $\varphi_{\vec{w}}v$ if it's defined. Regardless of whether the write occurred, the read value is returned.

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

$$M? := \mathbf{rmw}_{\text{load}}(M; \langle \rangle) \quad \text{CAS}(M, N, K) := \mathbf{rmw}_{\text{cas}}(M; \langle N, K \rangle)$$

$$\text{XCHG}(M, N) := \mathbf{rmw}_{\text{xchg}}(M; \langle N \rangle) \quad \text{FAA}(M, N) := \mathbf{rmw}_{\text{faa}}(M; \langle N \rangle)$$

Assignment $M := N$ 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 $\langle \rangle$. Parallel composition interleaves the execution of both its threads, and evaluates to the pair of results, one for each thread.

We do not include loops in this language, which we leave to future work. While important, recursion 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, λ_{RA} is expressive enough for us to discuss interesting examples and transformations.

We use standard syntactic sugar:

Let-binding	$\mathbf{let} \ a = M \ \mathbf{in} \ N := \mathbf{match} \ \langle M \rangle \ \mathbf{with} \ \langle a \rangle. N$	Sequencing	$M ; N := \mathbf{let} \ _ = M \ \mathbf{in} \ N$
Value tuples	$\mathbf{Val}^n := (\mathbf{Val} * \dots * \mathbf{Val})$	Unit	$1 := ()$
Enum. type	$\{t_1 \mid \dots \mid t_n\} := \{t_1 \ \mathbf{of} \ 1 \mid \dots \mid t_n \ \mathbf{of} \ 1\}$	Enum. label	$A.i := A.i \ \langle \rangle$

Remark. *RMWs, unlike assignments, must modify an existing message in memory. Thus, assignment $M := N$ is not equivalent to an XCHG that discards its read value, i.e. $\text{XCHG}(M, N) ; \langle \rangle$.*

Type system. We present the type system in Figure 4(b). 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* if $\cdot \vdash M : A$. 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:

$$G ::= (G_1 * \dots * G_n) \mid \{t_1 \ \mathbf{of} \ G_1 \mid \dots \mid t_n \ \mathbf{of} \ G_n\} \quad (\text{Ground types})$$

5 OPERATIONAL SEMANTICS FOR RELEASE/ACQUIRE CONCURRENCY

We start this section with 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. Our denotational semantics also accounts for both known and novel semantic invariants on the memories that can evolve when executing a well-typed program (§5.3).

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 order in which threads will read values from memory. We use rational numbers \mathbb{Q} as timestamps, ranged over by t, q, p , though any choice of an infinite, dense, and totally-ordered set suffices.

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

693 *Messages.* A *message* v is a tuple in $\text{Msg} := \text{Loc} \times \text{Val} \times \mathbb{Q} \times \text{View}$, written $v = \ell:v@(q, \kappa_\ell) \langle\langle \kappa \rangle\rangle$,
 694 where $q < \kappa_\ell$. Here, ℓ is the location of the message, v is the value of the message, q is the *initial*
 695 timestamp of the message, and κ is the view of the message. We say this message *dovetails* another
 696 message if its timestamp is q .

697 We use projection-notation for components of v : $v.l.c := \ell$, $v.v.l := v$, $v.i := q$, and $v.v.w := \kappa$.
 698 The (*final*) *timestamp* of the message is $v.t := \kappa_\ell$. In concrete examples, we reduce duplication
 699 by eliding the timestamp from the view, e.g. $y:0@(0.5, \underline{4,2}] \langle\langle x@1 \rangle\rangle$. The message's two timestamps
 700 delimit the *segment* of the message: the interval $v.\text{seg} := (v.i, v.t]$.

701 We range over messages using ν, ϵ, β . We extend notation from messages to sets of messages by
 702 direct image: for example, given a set μ of messages, define $\mu.\text{seg} := \{v.\text{seg} \mid v \in \mu\}$.

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

705 *Example 5.1.* The memory illustrated at the top of Figure 7(a) could have resulted from a program
 706 execution starting with the memory $\{v_1, \epsilon_1\}$. A program may add messages out of the timeline
 707 order (ϵ_3 before ϵ_2); dovetail messages ($v_2.t = v_3.i$); or leave gaps between messages ($v_1.t < v_2.i$).
 708 Message views need not increase along the timeline ($\epsilon_2.t \leq \epsilon_3.t$ yet $\epsilon_2.v.w \not\leq \epsilon_3.v.w$).

709 *View trees.* Kang et al.'s [28] original presentation of the view-based semantics studies top-level
 710 parallelism, requiring flat thread-view mappings. Since we allow nesting of parallel composition
 711 anywhere in the program, we use a nested view-tree mapping. Formally, a *view-tree* is a binary
 712 tree with view-labeled leaves. We denote the set of view-trees by VTree , ranged over by T, R, H .
 713 We denote: by $\dot{\kappa}$ the leaf with label κ ; by $T \widehat{\ } R$ the tree whose immediate left and right subtrees
 714 are T and R ; and by $T.lf$ the set of labels of leaves of a view-tree T . We lift the order \leq from views
 715 to view-trees leaf-wise: $\dot{\kappa} \leq \dot{\sigma}$ when $\kappa \leq \sigma$, and $T \widehat{\ } R \leq T' \widehat{\ } R'$ when $T \leq T'$ and $R \leq R'$.

716 *Operational semantics.* Figure 5 presents the operational semantics for λ_{RA} . A *configuration*
 717 $\langle T, \mu \rangle, M$ consists of a view-tree T capturing the view of all active threads; the current memory μ ;
 718 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
 719 (atomic) steps between configurations. The label e , distinguishing the memory-accessing steps (\bullet)
 720 from the rest (\circ), is used as a proof tool (Appendix B.3) and can be otherwise ignored. We denote
 721 $\rightsquigarrow_{\text{RA}} := \overset{\bullet}{\rightsquigarrow}_{\text{RA}} \cup \overset{\circ}{\rightsquigarrow}_{\text{RA}}$.

722 *Sequential CBV constructs.* We omit the standard CBV transitions, demonstrating the standard
 723 congruence and β -reduction steps for function application: APP, APPLEFT, APPRIGHT. *Values* are:

$$724 V, W ::= \langle V_1, \dots, V_n \rangle \mid A.l V \mid \lambda a : A. M \quad (\text{Values})$$

725 The congruence steps include the assignment and RMW construct, and use any view-trees T, T'
 726 and memories μ, μ' which they maintain between the inductive hypothesis and conclusion. The
 727 β -reduction steps use any view-leaf $\dot{\kappa}$ and memory μ which they maintain across the transition.

$$\begin{array}{c}
 \text{APP} \\
 \hline
 \langle \dot{\kappa}, \mu \rangle, (\lambda a : A. M) V \overset{\circ}{\rightsquigarrow}_{\text{RA}} \langle \dot{\kappa}, \mu \rangle, M[a \mapsto V] \\
 \\
 \begin{array}{cc}
 \text{APPLEFT} & \text{APPRIGHT} \\
 \frac{\langle T, \mu \rangle, M \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T', \mu' \rangle, M'}{\langle T, \mu \rangle, MN \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T', \mu' \rangle, M'N} & \frac{\langle T, \mu \rangle, N \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T', \mu' \rangle, N'}{\langle T, \mu \rangle, VN \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T', \mu' \rangle, VN'} \\
 \text{PARINIT} & \text{PARFIN} \\
 \frac{}{\langle \dot{\kappa}, \mu \rangle, M \parallel N \overset{\circ}{\rightsquigarrow}_{\text{RA}} \langle \dot{\kappa} \widehat{\ } \dot{\kappa}, \mu \rangle, M \parallel N} & \frac{\omega = \kappa \sqcup \sigma}{\langle \dot{\kappa} \widehat{\ } \dot{\sigma}, \mu \rangle, V \parallel W \overset{\circ}{\rightsquigarrow}_{\text{RA}} \langle \dot{\omega}, \mu \rangle, \langle V, W \rangle} \\
 \text{PARLEFT} & \text{PARRIGHT} \\
 \frac{\langle T, \mu \rangle, M \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T', \mu' \rangle, M'}{\langle T \widehat{\ } R, \mu \rangle, M \parallel N \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T' \widehat{\ } R, \mu' \rangle, M' \parallel N} & \frac{\langle R, \mu \rangle, N \overset{e}{\rightsquigarrow}_{\text{RA}} \langle R', \mu' \rangle, N'}{\langle T \widehat{\ } R, \mu \rangle, M \parallel N \overset{e}{\rightsquigarrow}_{\text{RA}} \langle T \widehat{\ } R', \mu' \rangle, M \parallel N'} \\
 \\
 \text{STORE} \\
 \frac{\alpha_\ell < t \quad (q, t] \cap \bigcup \mu_\ell.\text{seg} = \emptyset \quad \omega = \alpha [\ell \mapsto t]}{\langle \dot{\alpha}, \mu \rangle, \ell := v \overset{\bullet}{\rightsquigarrow}_{\text{RA}} \langle \dot{\omega}, \mu \uplus \{ \ell : v @ (q, \omega_\ell] \langle \omega \rangle \} \rangle, \langle \rangle} \\
 \\
 \text{READONLY} \\
 \frac{\ell : v @ (_ , \kappa_\ell] \langle \kappa \rangle \in \mu \quad \alpha_\ell \leq \kappa_\ell \quad \varphi_{\vec{w}} v = \perp \quad \omega = \alpha \sqcup \kappa}{\langle \dot{\alpha}, \mu \rangle, \text{rmw}_\varphi (\ell; \vec{w}) \overset{\bullet}{\rightsquigarrow}_{\text{RA}} \langle \dot{\omega}, \mu \rangle, v} \\
 \\
 \text{RMW} \\
 \frac{\ell : v @ (q, \kappa_\ell] \langle \kappa \rangle \in \mu \quad \alpha_\ell \leq \kappa_\ell \quad \varphi_{\vec{w}} v \neq \perp \quad (\kappa_\ell, t] \cap \bigcup \mu_\ell.\text{seg} = \emptyset \quad \omega = (\alpha \sqcup \kappa) [\ell \mapsto t]}{\langle \dot{\alpha}, \mu \rangle, \text{rmw}_\varphi (\ell; \vec{w}) \overset{\bullet}{\rightsquigarrow}_{\text{RA}} \langle \dot{\omega}, \mu \uplus \{ \ell : \varphi_{\vec{w}} v @ (\kappa_\ell, \omega_\ell] \langle \omega \rangle \} \rangle, v}
 \end{array}
 \end{array}$$

 Fig. 5. The operational semantics of λ_{RA} (selected rules).

Substitution. A (program) substitution Θ is a partial function from program variables to closed values, which extends to the identity on all other variables, and then to terms by recursively applying to subprograms, removing bound variables from the substitution's domain, e.g.:

$$\Theta(\text{match } M \text{ with } \langle a_1, \dots, a_m \rangle. N) := \text{match } \Theta M \text{ with } \langle a_1, \dots, a_m \rangle. \Theta|_{\notin \{a_1, \dots, a_m\}} N$$

where $\Theta|_{\notin X}$ is obtained by removing X from Θ 's domain. We write $M[V_1/a_1 \dots V_n/a_n]$ for the application of the substitution that maps $a_i \mapsto V_i$ on M .

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 evaluation 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. We show an example execution from a birds-eye view:

$$\begin{aligned}
 \langle \mu_0, \dot{\alpha} \rangle, M ; (N_1 \parallel N_2) \rightsquigarrow_{\text{RA}}^* \langle \mu_1, \dot{\alpha}' \rangle, N_1 \parallel N_2 \rightsquigarrow_{\text{RA}} \langle \mu_1, \dot{\alpha}' \widehat{\ } \dot{\alpha}' \rangle, N_1 \parallel N_2 \\
 \rightsquigarrow_{\text{RA}}^* \langle \rho, \dot{\omega}_1 \widehat{\ } \dot{\omega}_2 \rangle, V_1 \parallel V_2 \rightsquigarrow_{\text{RA}} \langle \rho, \omega_1 \sqcup \omega_2 \rangle, \langle V_1, V_2 \rangle
 \end{aligned}$$

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 $(q, t]$ 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 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 with the message from which the RMW read, still avoiding any existing segment in this location. This dovetailing is only possible if we read from a message with no dovetailing succeeding message. In particular, a message can only be picked once to justify the RMW rule during an execution.

Initial memory and configuration. 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 configuration state is one with an initial memory and a view-leaf mapping each location to the timestamp of the unique message in memory of that location.

Evaluation. We're interested in the behaviors closed terms exhibit when run to completion. Let the Kleene star $(*)$ denote the reflexive-transitive closure of a relation. A configuration $\langle T, \mu \rangle, M$ evaluates to a value V , written $\langle T, \mu \rangle, M \Downarrow_{\text{RA}} V$, when $\langle T, \mu \rangle, M \rightsquigarrow_{\text{RA}}^* \langle R, \rho \rangle, V$ for some state $\langle R, \rho \rangle$. We write $\langle T, \mu \rangle, M \Downarrow_{\text{RA}} V$ when there is no such $\langle R, \rho \rangle$. In the next examples, we write $M \Downarrow_{\text{RA}} V$ when M may evaluate to V from every initial state, and $M \not\Downarrow_{\text{RA}} V$ when it cannot evaluate to V from any initial state.

Example 5.2. We can give a more precise account of the litmus tests (SB) and (MP) from §2:

$$\begin{aligned} & x := 0; y := 0; ((x := 1; y?) \parallel (y := 1; x?)) \Downarrow_{\text{RA}} \langle 0, 0 \rangle \\ & x := 0; y := 0; ((x := 1; y := 1) \parallel \langle y?, x? \rangle) \Downarrow_{\text{RA}} \langle \langle \rangle, \langle 1, 0 \rangle \rangle \end{aligned}$$

5.2 Non-deterministic View Forwarding

It is technically convenient to extend $\rightsquigarrow_{\text{RA}}$ with an additional step that non-deterministically advances the view of a thread. 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. 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.

Formally, we obtain $\rightsquigarrow_{\text{RA}_{\leq}}$ by adding the ADV rule in Figure 6 to the rules presented in Figure 5, after replacing RA with RA_{\leq} and removing the labels. The ADV step advances the thread's view like the READONLY rule without changing the term component of the configuration. A-priori, the resulting system may exhibit more behaviors since STORE and RMW steps will append messages with further advanced views. However, advancing views within messages only further constrains possible behaviors.

$$\text{ADV} \frac{\ell : v @ (q, \kappa_{\ell}] \langle \langle \kappa \rangle \rangle \in \mu \quad \alpha_{\ell} \leq \kappa_{\ell} \quad \omega = \alpha \sqcup \kappa}{\langle \dot{\alpha}, \mu \rangle, M \rightsquigarrow_{\text{RA}_{\leq}} \langle \dot{\omega}, \mu \rangle, M}$$

Fig. 6. The ADV rule.

We formalize this intuition using a simulation argument. One direction is straightforward: every execution in RA is an execution in RA_{\leq} that doesn't use ADV, and so RA_{\leq} exhibits every behavior RA does. For the converse, we define a binary relation between configuration states \succeq such that $\langle T, \mu \rangle \succeq \langle R, \rho \rangle$ when the following hold.

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

The relation \succeq is a weak simulation:

PROPOSITION 5.3. *If $\langle T, \mu \rangle \succeq \langle R, \rho \rangle$ 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 $\langle T', \mu' \rangle \succeq \langle R', \rho' \rangle$.*

Like \rightsquigarrow_{RA} , so does $\rightsquigarrow_{RA_{\leq}}$ yield an evaluation semantics $\Downarrow_{RA_{\leq}}$. By **Proposition 5.3**, they coincide:

COROLLARY 5.4. *For every 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 .

5.3 Semantic Invariants

Our denotational model uses semantic invariants that initial states possess and RA_{\leq} steps preserve. During our presentation of the invariants we give intuitive explanations for why they hold. These are formally grounded in **Theorem 5.15** and **Proposition 5.18** below.

Basic memory invariants. A memory μ is *scattered* when the segments of messages in the same location are pairwise disjoint: $\forall \ell \in \text{Loc} \forall v, \epsilon \in \mu_{\ell}. v.\text{seg} \cap \epsilon.\text{seg} = \emptyset \implies v = \epsilon$. Initial memories are scattered and execution steps preserve this property since added messages can only occupy unused segments.

Example 5.5. This memory is scattered, which we visualise using the scattering condition:

$$\left. \begin{array}{l} \{ y:1@(-1, \underline{0}] \ll x@5 \rangle, y:3@(0, \underline{7}] \ll x@8 \rangle \\ \{ x:0@(-1, \underline{0}] \ll y@0 \rangle, x:2@(4, \underline{5}] \ll y@7 \rangle \end{array} \right\} \quad \begin{array}{l} y: \left[\begin{array}{|c|c|} \hline \epsilon_1 & 1@(-1, \llbracket 0; x@5 \rrbracket) \\ \hline \epsilon_2 & 3@(0, \llbracket 7; x@8 \rrbracket) \\ \hline \end{array} \right] \\ x: \left[\begin{array}{|c|c|} \hline \nu_1 & 0@(-1, \llbracket 0; y@0 \rrbracket) \\ \hline \nu_2 & 2@(4, \llbracket 5; y@7 \rrbracket) \\ \hline \end{array} \right] \end{array}$$

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

Example 5.6. **Example 5.5**'s memory is not connected: ϵ_2 doesn't point to any message in x . This memory is connected (left), with its timestamp orders (middle) and points-to relations (right):

$$\left. \begin{array}{l} \{ y:2@(-1, \underline{5}] \ll x@0 \rangle, y:4@(0, \underline{7}] \ll x@0 \rangle \\ \{ x:1@(-1, \underline{0}] \ll y@0 \rangle, x:3@(4, \underline{5}] \ll y@7 \rangle \end{array} \right\} \quad \begin{array}{l} y: \left[\begin{array}{|c|c|} \hline \epsilon_1 & 2 \\ \hline \epsilon_2 & 4 \\ \hline \end{array} \right] \\ x: \left[\begin{array}{|c|c|} \hline \nu_1 & 1 \\ \hline \nu_2 & 3 \\ \hline \end{array} \right] \end{array} \quad \begin{array}{c} \nu_1 \longleftarrow x \longleftarrow \epsilon_2 \\ \downarrow y \qquad \uparrow y \\ \epsilon_1 \longrightarrow x \longrightarrow \nu_2 \end{array}$$

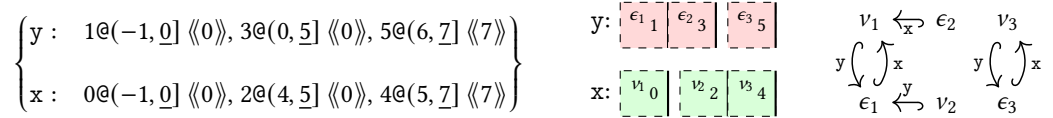
Initial memories are connected, and execution steps preserve memory connectedness, assuming that all thread views point to the current memory: when a thread adds a message to memory, it uses its own view with an advanced timestamp for the message's location, maintaining connectedness.

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 $\mu.\text{gph} := \langle \mu, (\succ) \setminus \text{id}_\mu \rangle$, 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, $\kappa \succ \epsilon$, and it dominates ϵ 's view, $\kappa \geq \epsilon.\text{vw}$. 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.: $\forall \ell \in \text{Loc} \exists \epsilon \in \mu_\ell. \kappa \hookrightarrow \epsilon$. We say that a message points downward into a memory, writing $v \hookrightarrow \mu$, when its view does: $v.\text{vw} \hookrightarrow \mu$. We say that a memory μ is *causally connected*, when it is connected, and every message within it points downwards into it: $\forall v \in \mu. v \hookrightarrow \mu$.

To further conserve space in the following examples, we omit locations from messages, instead tagging the row in the set. For example, by $5\text{@}(6, \underline{7}] \llbracket 7 \rrbracket$ in the y row we mean $y:5\text{@}(6, \underline{7}] \llbracket x\text{@}7 \rrbracket$.

Example 5.7. The memory from [Example 5.6](#) is not causally connected because $\epsilon_1 \succ v_2$ while nonetheless $\epsilon_1.\text{vw}_y = 0 \not\geq 7 = v_2.\text{vw}_y$. The following memory is causally connected:



Initial memories are causally connected, and execution steps preserve this together with view-trees labeled solely by downward-pointing views. In showing this, particularly when observing steps that load a message, the following fact helps; pointing downwards is a stronger condition than may first appear:

LEMMA 5.8. *Assume μ is causally connected. Then $\kappa \hookrightarrow \mu$ iff $\kappa = \sqcup \{ \epsilon \in \mu \mid \kappa \succ \epsilon \}.\text{vw}$.*

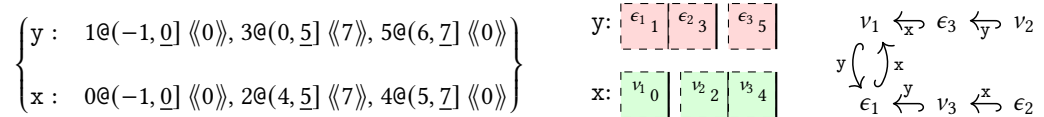
Paths in a causally connected memory's graph descend down its timelines:

PROPOSITION 5.9. *Let μ be a scattered, downward pointing memory, with a path $v \succ^* \epsilon$ in $\mu.\text{gph}$.*

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

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

Example 5.10. The memory from [Example 5.7](#) is not well-formed: its minimal messages are v_1, ϵ_1 , but v_3 and ϵ_3 are on a cycle. The following memory is:



A hypothetical language that allows atomic initialization of cyclic pointers in RA memory locations might include a construct for simultaneous mutation and will violate this invariant, but

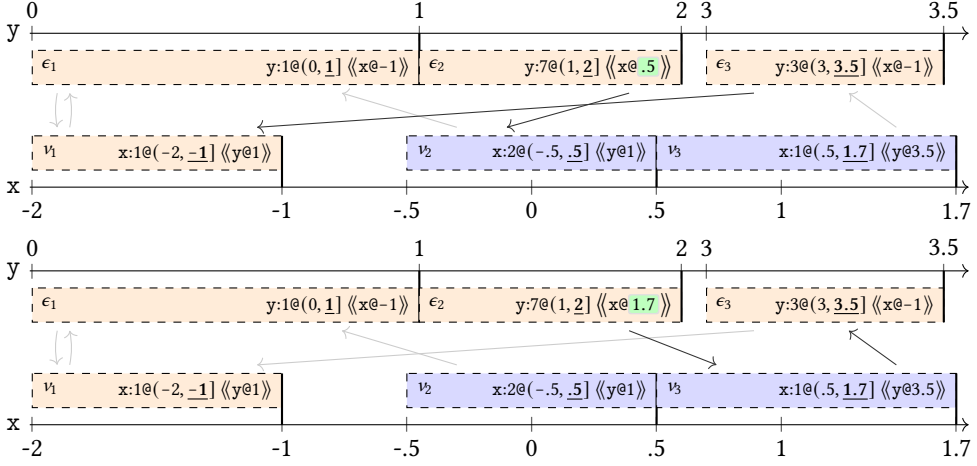


Fig. 7. Two variations on the memory illustrated in Figure 1. **Top:** This memory is well-formed. It demonstrates that the views of messages along a timeline do not have to be ordered: ϵ_2 appears earlier than ϵ_3 on y 's timeline but points to a later message on x 's timeline. **Bottom:** This memory is not well-formed because it contains an ascending path, in contradiction to Proposition 5.9. Intuitively, no thread could have written ϵ_2 because the view that ϵ_2 carries indicates that the thread would have already “known” about v_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.

λ_{RA} maintains it: 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 5.11. *Let μ be a well-formed memory, and $\ell \in \text{Loc}$.*

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

We denote the set of well-formed memories by Mem . Figure 7 gives a positive example (top) and a negative example (bottom).

View-tree invariants. Like memories, view-trees also satisfy certain 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 *points to/downward into* memory μ , and write $T \triangleright \mu$ and $T \hookrightarrow \mu$ when $\kappa \triangleright \mu$ and $\kappa \hookrightarrow \mu$ for every $\kappa \in T$. If 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. We define 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. A view-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 follow the congruence rules from the operational

981 semantics, which we demonstrate with the λ -L and λ -R rules:

$$\begin{array}{c}
 \text{982} \\
 \text{983} \\
 \text{984} \\
 \text{985}
 \end{array}
 \begin{array}{c}
 \text{LEAF} \\
 \hline
 \kappa \Vdash M
 \end{array}
 \qquad
 \begin{array}{c}
 \text{NODE} \\
 \frac{T \Vdash M \quad R \Vdash N}{T \widehat{\ } R \Vdash M \parallel N}
 \end{array}
 \qquad
 \begin{array}{c}
 \lambda\text{-L} \\
 \frac{T \widehat{\ } R \Vdash M}{T \widehat{\ } R \Vdash MN}
 \end{array}
 \qquad
 \begin{array}{c}
 \lambda\text{-R} \\
 \frac{T \widehat{\ } R \Vdash N}{T \widehat{\ } R \Vdash VN}
 \end{array}$$

986 *Example 5.12.* For $M = (\lambda a. N_1 \parallel N_2) (M_1 \parallel M_2)$, we have $\kappa \Vdash M$ and $\kappa_1 \widehat{\ } \kappa_2 \Vdash M$. Intuitively,
 987 the evaluation context in M is $(\lambda a. N_1 \parallel N_2) [-]$, and the active component—where reduction takes
 988 place—is $(M_1 \parallel M_2)$. The execution of $N_1 \parallel N_2$ is suspended under the λ -abstraction, so we asso-
 989 ciate no views with its threads. The view-node is well-formed for the active component by NODE.

990 For $N = (\lambda a. N_1 \parallel N_2) V$, the view node is *not* well-formed: $\kappa_1 \widehat{\ } \kappa_2 \not\Vdash N$. The evaluation context
 991 is empty, and the active (single) thread is $(\lambda a. N_1 \parallel N_2) V$: the next execution step has to be λ -APP.
 992 Only a view-leaf is well-formed for such a program.
 993

994 When a tree is well-formed for a program, each view in T corresponds to an M -subterm by
 995 following the derivation of $T \Vdash M$ up to the leaves:
 996

997 **PROPOSITION 5.13.** *Every tree well-formedness judgement $T \Vdash M$ has a unique derivation, the*
 998 *leaves of which induce an injective mapping from T 's view-leaves to M 's subterms.*

999 *Example 5.14.* The leaves of the unique derivation tree for $\kappa_1 \widehat{\ } \kappa_2 \Vdash (\lambda a. N_1 \parallel N_2) (M_1 \parallel M_2)$
 1000 are the instances $\kappa_i \Vdash M_i$ of the LEAF rule.
 1001

1002 *Execution invariants.* Collecting the invariants, a configuration $\langle T, \mu \rangle, M$ is *well-formed of type*
 1003 *A* when: its state $\langle T, \mu \rangle$ is well-formed; its term is (necessarily uniquely) well-typed $\cdot \vdash M : A$; and
 1004 its view-tree is well-formed for its term: $T \Vdash M$. The steps of RA_{\leq} preserve well-formedness:
 1005

1006 **THEOREM 5.15 (PRESERVATION).** *If $\langle T, \mu \rangle, M \rightsquigarrow_{\text{RA}_{\leq}} \langle R, \rho \rangle, N$ and $\langle T, \mu \rangle, M$ is a well-formed con-*
 1007 *figuration of type A, then $\langle R, \rho \rangle, N$ is a well-formed configuration of type A.*
 1008

1009 From now on, we only consider steps between well-formed configurations.

1010 Execution steps maintain some relationships between initial and final states. First, the timestamp
 1011 of a new message lies between some thread's initial and final views:
 1012

1013 **LEMMA 5.16.** *Assume $\langle T, \mu \rangle, M \rightsquigarrow_{\text{RA}_{\leq}} \langle R, \rho \rangle, N$ changed the memory, i.e. $\rho \neq \mu$. Then: the trees*
 1014 *have the same shape; $T \leq R$; and there is a message v such that $\rho = \mu \uplus \{v\}$. Moreover, there are*
 1015 *view-leaves α in T and ω in R in corresponding positions, such that $\alpha \leq v.v\omega \leq \omega$ and $\alpha_{v.lc} < v.t$.*
 1016

1017 The view-tree structure changes during PARINIT and PARFIN, so they cannot always be com-
 1018 pared leaf-to-leaf as in **Lemma 5.16**. However, the sets of views that label each tree still maintain
 1019 the *Egli-Milner order* induced by the view order:
 1020

1020 **LEMMA 5.17 (EGLI-MILNER FOR VIEW-LEAVES).** *Assume $\langle T, \mu \rangle, M \rightsquigarrow_{\text{RA}_{\leq}}^* \langle R, \rho \rangle, N$.*

- 1021 • For every $\alpha \in T.lf$, there exists a leaf $\omega \in R.lf$, such that $\alpha \leq \omega$.
- 1022 • For every $\omega \in R.lf$, there exists a leaf $\alpha \in T.lf$, such that $\alpha \leq \omega$.

1023 We combine **Lemmas 5.16** and **5.17** to obtain the following execution invariant:
 1024

1025 **PROPOSITION 5.18 (VIEWS DELIMIT EXECUTION).** *Assume $\langle T, \mu \rangle, M \rightsquigarrow_{\text{RA}_{\leq}}^* \langle R, \rho \rangle, N$. Assume that*
 1026 *α is dominated by every view in $T.lf$, and that ω dominates every view in $R.lf$. Then $\alpha \leq \omega$; and for*
 1027 *every added message $v \in \rho \setminus \mu$, both $\alpha \leq v.v\omega \leq \omega$ and $\alpha_{v.lc} < v.t$.*
 1028
 1029

1030 *Interrupted executions.* To analyze program behavior under concurrent contexts, we have to take
 1031 into account all possible ways in which the environment can interfere during the execution. An
 1032 *interrupted execution* $\langle T, \mu \rangle, M \rightsquigarrow_{RA_{\subseteq}}^* \cdots \rightsquigarrow_{RA_{\subseteq}}^* \langle R, \rho \rangle, V$ from a closed term M to a value V is a
 1033 sequence of executions of the form

$$\begin{aligned}
 1034 \quad \langle T, \mu \rangle, M &= \langle T_1, \mu_1 \rangle, M_1 \rightsquigarrow_{RA_{\subseteq}}^* \langle T_2, \rho_1 \rangle, M_2 \\
 1035 &\quad \langle T_2, \mu_2 \rangle, M_2 \rightsquigarrow_{RA_{\subseteq}}^* \langle T_3, \rho_2 \rangle, M_3 \\
 1036 &\quad \vdots \\
 1037 &\quad \vdots \\
 1038 &\quad \vdots \\
 1039 &\quad \langle T_n, \mu_n \rangle, M_n \rightsquigarrow_{RA_{\subseteq}}^* \langle T_{n+1}, \rho_n \rangle, M_{n+1} = \langle R, \rho \rangle, V
 \end{aligned}$$

1040 where $\rho_j \subseteq \mu_{j+1}$ for every $1 \leq j \leq n-1$. Between the executions in the sequence, the configuration
 1041 only changes by adding *environment messages*, the messages in $\mu_{j+1} \setminus \rho_j$, to the memory—the only
 1042 interference the environment can cause. We also have $\mu_i \subseteq \rho_i$, and we call the messages in $\rho_j \setminus \mu_j$
 1043 *local messages*. [Proposition 5.18](#) extends to interrupted executions in a straightforward manner,
 1044 replacing $\rightsquigarrow_{RA_{\subseteq}}^*$ with $\rightsquigarrow_{RA_{\subseteq}}^* \cdots \rightsquigarrow_{RA_{\subseteq}}^*$ and replacing *added* messages with *local* messages.
 1045

1046 6 DENOTATIONAL SEMANTICS

1047 [Brookes](#)-style traces ([§6.1](#)) adapted to describe behavior under RA, and [Moggi](#)'s monadic approach
 1048 ([§6.2](#)) to denotational semantics, combine to give our semantic framework ([§6.3](#)). We build upon
 1049 this framework in three stages.
 1050

1051 First we define the generating denotational semantics ([§6.4](#)). The monad structure underlying
 1052 this semantics does not satisfy the monad laws, and so does not fully conform to the monadic
 1053 approach. Still, it is useful in forming a base for the next stage, as a metatheoretic tool, as well as
 1054 a means to simpler calculations.

1055 Thus, we define the concrete denotational semantics ([§6.5](#)). Here we do have a monad, but the
 1056 denotational semantics follows the operational semantics too closely to be as abstract as we would
 1057 like, evident in program transformations that it does not support. This semantics is useful as an
 1058 intermediate step, and plays a central role in our proof of the adequacy theorem.

1059 Finally, we define the abstract denotational semantics ([§6.6](#)). This is the semantics we were aim-
 1060 ing for: adequate and abstract enough to justify transformations of interest.

1061 6.1 Trace-based Semantics

1062 Traces are a semantic counterpart to interrupted executions. Their core component is a sequence
 1063 of memory-transitions, summarizing which messages the behavior they describe relies on and
 1064 guarantees. A (*memory*)-*transition* is pair $\langle \mu, \rho \rangle$ of memories, such that $\mu \subseteq \rho$.

1065 We capture the evolving assumptions and guarantees about memory messages by a *chronicle*:
 1066 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
 1067 j . When ξ is non-empty, we denote its *opening* and *closing* memories by $\xi.o := \mu_1$ and $\xi.c :=$
 1068 ρ_n . Its *local messages* are the ones added within transitions: $\xi.\text{own} := \bigcup_{i \in \{1, \dots, n\}} (\rho_i \setminus \mu_i)$, and its
 1069 *environment messages* are the others. Let Chro be the set of chronicles, ranged over by ξ, η .

1070 In the operational semantics, some messages are obscured from any particular thread due to
 1071 its view. The trace captures only an *initial view* that declares which messages may be relied on to
 1072 be available at the beginning, and a *final view* that declares which messages are guaranteed to be
 1073 available at the end. Together, these are the *delimiting views*.

1074 Finally, a trace includes a semantic representation of the returned value [e.g. [6](#)] Given a set
 1075 representing semantic return values X , an X -*pre-trace* is an element of $\text{View} \times \text{Chro} \times \text{View} \times X$,
 1076 written $\alpha \llbracket \xi \rrbracket \omega \cdot r$, whose chronicle component is non-empty. We range over pre-traces with τ, π, ϱ ,
 1077

and use $\tau.\text{ivw}$ (initial view), $\tau.\text{ch}$ (chronicle), $\tau.\text{fvw}$ (final view), $\tau.\text{ret}$ (returned value) to retrieve the components of a pre-trace $\tau = \alpha \boxed{\xi} \omega \cdot r$ in order.

Such an X -pre-trace τ is an X -trace when each transition in ξ consists of well-formed memories; the initial view precedes the final views, each pointing downwards into the opening and closing memories respectively: $\xi.\text{o} \leftrightarrow \alpha \leq \omega \leftrightarrow \xi.\text{c}$; and the view and segment of every local message are bound by the delimiting views, i.e.: $\forall v \in \xi.\text{own}.\ \alpha \leq v.\text{vw} \leq \omega \wedge \alpha_{v.\text{lc}} < v.\text{t}$. These conditions reflect well-formedness and the invariants from [Theorem 5.15](#) and [Proposition 5.18](#). We denote the set of X -traces by $\text{Trace}X$. The bottom of [Figure 1](#) depicts an example trace.

6.2 Monad-based Semantics

We recap Moggi's [40] approach to interpret a CBV calculus like λ_{RA} using a monad. A *monad structure* $\mathcal{T} = (\underline{\mathcal{T}}, \text{return}^{\mathcal{T}}, (\!|=^{\mathcal{T}}))$ consists of three components: a set-level function $\underline{\mathcal{T}}$; a set-indexed function $\text{return}^{\mathcal{T}}$; and a two-argument set-indexed function $(\!|=^{\mathcal{T}})$. 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 $\text{return}_X^{\mathcal{T}} : X \rightarrow \underline{\mathcal{T}}X$, the *unit*, represent the program fragment that returns its input without any observable side-effects. The two-argument functions $(\!|=_{X,Y}^{\mathcal{T}}) : (\underline{\mathcal{T}}X) \times (X \rightarrow \underline{\mathcal{T}}Y) \rightarrow \underline{\mathcal{T}}Y$, the *monadic bind*, represent the sequencing $P \!|=_{X,Y}^{\mathcal{T}} f$ of an X -returning program P with an Y -returning program f that depends on the result of P .

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 construct has a corresponding semantic construct, and the interpretation proceeds structurally over the structure of types, context 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:

$$\begin{aligned} \llbracket (A_1 * \dots * A_n) \rrbracket &:= \llbracket A_1 \rrbracket \times \dots \times \llbracket A_n \rrbracket & \llbracket A \rightarrow B \rrbracket &:= \llbracket A \rrbracket \rightarrow \underline{\mathcal{T}} \llbracket B \rrbracket \\ \llbracket \{t_1 \text{ of } A_1 \mid \dots \mid t_n \text{ of } A_n\} \rrbracket &:= (\{t_1\} \times \llbracket A_1 \rrbracket) \cup \dots \cup (\{t_n\} \times \llbracket A_n \rrbracket) & \llbracket \Gamma \rrbracket &:= \prod_{(a:A) \in \Gamma} \llbracket A \rrbracket \end{aligned}$$

In particular, denotations of ground types $\llbracket G \rrbracket$ do not depend on the monad structure. For example, $\llbracket \text{Val} \rrbracket$ is in a natural bijection with the set of (storable) values Val , and we will identify them.

Value semantics. Every value $\Gamma \vdash V : A$ denotes a function $\llbracket V \rrbracket^{\vee} : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$, taking as argument a *semantic environment* $\gamma \in \llbracket \Gamma \rrbracket$ supplying a semantic value to each variable in context:

$$\begin{aligned} \llbracket b \rrbracket^{\vee} (\gamma_a)_{(a:A) \in \Gamma} &:= \gamma_b & \llbracket \langle V_1, \dots, V_n \rangle \rrbracket^{\vee} \gamma &:= (\llbracket V_1 \rrbracket^{\vee} \gamma, \dots, \llbracket V_n \rrbracket^{\vee} \gamma) \\ \llbracket A.t \rrbracket^{\vee} \gamma &:= (t, \llbracket V \rrbracket^{\vee} \gamma) & \llbracket \lambda b : B. M \rrbracket^{\vee} (\gamma_a)_{(a:A) \in \Gamma} &:= \lambda \gamma_b. \llbracket M \rrbracket^{\vee} (\gamma_a)_{(a:A) \in \Gamma, b:B} \end{aligned}$$

Closed values $\cdot \vdash V : A$ denote functions from the singleton $\llbracket \cdot \rrbracket := \{()\}$ to $\llbracket A \rrbracket$, so we write $\llbracket V \rrbracket^{\vee}$ for $\llbracket V \rrbracket^{\vee} ()$. The semantics of closed ground values do not use the monad structure.

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

$$\begin{aligned} \llbracket MN \rrbracket^{\text{c}} \gamma &:= \llbracket M \rrbracket^{\text{c}} \gamma \!|= \lambda g. \llbracket N \rrbracket^{\text{c}} \gamma \!|= \lambda a. g(a) \\ \llbracket \langle M_1, \dots, M_n \rangle \rrbracket^{\text{c}} \gamma &:= \llbracket M_1 \rrbracket^{\text{c}} \gamma \!|= \lambda a_1. \dots \!|= \llbracket M_n \rrbracket^{\text{c}} \gamma \!|= \lambda a_n. \text{return}(a_1, \dots, a_n) \end{aligned}$$

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 $V := \langle \langle 1, 2 \rangle, 3 \rangle$ has

Table 1. Summary of all rewrite rules.

1177

1178

1179

1180

1181

1182

1183

1184

1185

1186

1187

1188

1189

1190

1191

1192

1193

1194

1195

1196

1197

1198

1199

1200

1201

1202

1203

1204

1205

1206

1207

1208

1209

1210

1211

1212

1213

1214

1215

1216

1217

1218

1219

1220

1221

1222

1223

1224

1225

g	Loosen	$\alpha \left[\xi \left(\eta \overline{\cup} \{ \epsilon \} \right) \right] \omega$	$\xrightarrow{\text{Ls}}$	$\alpha \left[\xi \left(\eta \overline{\cup} \{ v \} \right) \right] \omega$	$v \leq_{\text{vw}} \epsilon$
	Expel	$\alpha \left[\xi \left(\eta \overline{\cup} \{ \epsilon_i^{v,i} \} \right) \right] \omega$	$\xrightarrow{\text{Ex}}$	$\alpha \left[\xi \left(\eta \overline{\cup} \{ v, \epsilon \} \right) \right] \omega$	$v \prec \epsilon$
	Condense	$\alpha \left[\xi \left(\eta \overline{\cup} \{ v, \epsilon \} \right) \right] \omega$	$\xrightarrow{\text{Cn}}$	$\left(\alpha \left[\xi \left(\eta \overline{\cup} \{ v \} \right) \right] \omega \right) [\uparrow \epsilon]$	$v \preceq \epsilon$
c	Stutter	$\alpha \left[\xi \eta \right] \omega$	$\xrightarrow{\text{St}}$	$\alpha \left[\xi \langle \mu, \mu \rangle \eta \right] \omega$	Rewind $\kappa \left[\xi \right] \omega \xrightarrow{\text{Rw}}$ $\alpha \left[\xi \right] \omega$ $\alpha \leq \kappa$
	Mumble	$\alpha \left[\xi \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta \right] \omega$	$\xrightarrow{\text{Mu}}$	$\alpha \left[\xi \langle \mu, \theta \rangle \eta \right] \omega$	Forward $\alpha \left[\xi \right] \kappa \xrightarrow{\text{Fw}}$ $\alpha \left[\xi \right] \omega$ $\kappa \leq \omega$
a	Tighten	$\alpha \left[\xi \langle \mu, \rho \cup \{ v \} \rangle \eta \overline{\cup} \{ v \} \right] \omega$	$\xrightarrow{\text{Tt}}$	$\alpha \left[\xi \langle \mu, \rho \cup \{ \epsilon \} \rangle \eta \overline{\cup} \{ \epsilon \} \right] \omega$	$v \leq_{\text{vw}} \epsilon$
	Absorb	$\alpha \left[\xi \langle \mu, \rho \cup \{ v, \epsilon \} \rangle \eta \overline{\cup} \{ v, \epsilon \} \right] \omega$	$\xrightarrow{\text{Ab}}$	$\alpha \left[\xi \langle \mu, \rho \cup \{ \epsilon_i^{v,i} \} \rangle \eta \overline{\cup} \{ \epsilon_i^{v,i} \} \right] \omega$	$v \prec \epsilon$
	Dilute	$\left(\alpha \left[\xi \langle \mu, \rho \cup \{ v \} \rangle \eta \overline{\cup} \{ v \} \right] \omega \right) [\uparrow \epsilon]$	$\xrightarrow{\text{Di}}$	$\alpha \left[\xi \langle \mu, \rho \cup \{ v, \epsilon \} \rangle \eta \overline{\cup} \{ v, \epsilon \} \right] \omega$	$v \preceq \epsilon$

a function ϕ is *pointwise \star -closed* when $\phi = \phi^\star$. We say that a function ϕ between subsets of traces is *\star -closed* when its restriction to \star -closed subsets is pointwise closed.

Table 1 summarizes all the rewrite rules we will use. This compact figure packs many side conditions and new notation, which we explain as we present the rules. When presenting a rewrite rule we omit the return value, because they all maintain it.

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

$$\text{return}_{\mathcal{T}X}^r := \{ \kappa \langle \mu, \mu \rangle \kappa \mid r \in \text{Trace}X \}^\star$$

$$P \gg_{\mathcal{T}X, Y}^f := \{ \alpha \left[\xi \eta \right] \omega \mid s \in \text{Trace}Y \mid \alpha \left[\xi \right] \kappa \mid r \in P, \kappa \leq \sigma, \sigma \eta \omega \mid s \in fr \}^\star$$

Trace concurrent execution. (\parallel) 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 of views pointing downward into a well-formed memory μ by $-\prec\rightarrow\mu := \{ \kappa \in \text{View} \mid \kappa \prec\rightarrow\mu \}$. 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 $\lambda \ell$. $\min_{\mu} \ell$. Consider a non-empty $U \subseteq -\prec\rightarrow\mu$. Since $-\prec\rightarrow\mu$ is finite and closed under \sqcup , the set U has a least upper bound $\sup_{\mu} U := \sqcup U$. It also has a greatest lower bound $\inf_{\mu} U := \sqcup \{ \kappa \in \text{View} \mid \sqcap U \geq \kappa \prec\rightarrow\mu \}$, noting $\sqcap U$ might not point downward into μ .

Example 6.2. Let μ be the memory from Example 5.10. For $\alpha_1 := \langle \langle x@5; y@7 \rangle \rangle$ and $\alpha_2 := \langle \langle x@7; y@5 \rangle \rangle$, we have $\alpha_1 \prec\rightarrow\mu$ and $\alpha_2 \prec\rightarrow\mu$, but $\alpha_1 \sqcap \alpha_2 = \langle \langle x@5; y@5 \rangle \rangle \not\prec\rightarrow\mu$. Here, $\inf_{\mu} \{ \alpha_1, \alpha_2 \} = \langle \langle x@0; y@0 \rangle \rangle$.

Denoting by $\xi_1 \parallel \xi_2$ the set of all the interleavings of ξ_1 and ξ_2 that form chronicles, we define:

$$P_1 \parallel P_2 := \left\{ \begin{array}{l} \inf_{\xi, \circ} \{ \alpha_1, \alpha_2 \} \left[\xi \right] \sup_{\xi, \circ} \{ \omega_1, \omega_2 \} \mid \langle r_1, r_2 \rangle \in \text{Trace}(X_1 \times X_2) \}^\star \\ \mid \xi \in (\xi_1 \parallel \xi_2) \wedge \forall i \in \{1, 2\}. \alpha_i \left[\xi_i \right] \omega_i \mid r_i \in P_i \end{array} \right.$$

Trace memory access. Mirroring the operational semantics, we interpret:

$$\llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{T}} := \{ \kappa \left[\langle \mu, \mu \cup \{ \ell: v @ (q, t) \} \rangle \langle \kappa \left[\ell \mapsto t \right] \rangle \right] \kappa \left[\ell \mapsto t \right] \mid \langle \cdot \rangle \in \text{Trace}1 \}^\star$$

$$\llbracket \text{rmw}_{\ell, \Phi} \rrbracket_{\mathcal{T}} := \llbracket \text{rmw}_{\ell, \Phi}^{\text{RO}} \rrbracket_{\mathcal{T}} \cup \llbracket \text{rmw}_{\ell, \Phi}^{\text{RMW}} \rrbracket_{\mathcal{T}} \text{ where:}$$

$$\llbracket \text{rmw}_{\ell, \Phi}^{\text{RO}} \rrbracket_{\mathcal{T}} := \{ \kappa \left[\langle \mu, \mu \rangle \right] \kappa \mid v.v1 \in \text{TraceVal} \mid \Phi(v.v1) = \perp \wedge \kappa \mapsto v \in \mu_{\ell} \}^\star$$

$$\llbracket \text{rmw}_{\ell, \Phi}^{\text{RMW}} \rrbracket_{\mathcal{T}} := \left\{ \kappa \langle \overline{\mu, \mu \uplus \{\epsilon\}} \rangle \kappa [\ell \mapsto t] \dotsc \nu. \text{v}1 \in \text{TraceVal} \mid \epsilon = \ell : \Phi(\nu. \text{v}1) @(\nu. \text{t}, t) \llbracket \kappa [\ell \mapsto t] \rrbracket, \kappa \rightsquigarrow \nu \in \mu_{\ell} \right\}^{\star}$$

Requiring the resulting pre-traces to form traces ensures the constraints on their timestamps and segment hold. Assignment adds a new message. The RMW interpretation adds a new message depending on the modifier's result. Loading is restricted to messages already pointed-to. This restricted definition relies on the bind ($\gg=$) to advance the view when concatenating traces.

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

PROPOSITION 6.3. *Let $P_i, Q_i \in \underline{\mathcal{G}}X_i$ and $f, g : X_1 \rightarrow \underline{\mathcal{G}}X_2$. If $P_i \subseteq Q_i$ and $\forall r \in X_1. fr \subseteq gr$ then:*

$$P_1 \gg= f \subseteq Q_1 \gg= g \quad P_1 \text{ ||| } P_2 \subseteq Q_1 \text{ ||| } Q_2$$

PROOF. The $(-)^{\star}$ operator is monotonic by virtue of being a closure operator. Thus, it is sufficient to show the containment for the operators as defined before taking the \star -closure, which follows straightforwardly from the set-definitions, where traces are obtained from traces in the operands. \square

6.4 Generating Denotations

In the degenerate case of the \emptyset -monad structure, which we call the *null model* and denote by \mathcal{N} , neither identity axiom hold, as evidenced by $\text{return } r \gg= \text{return } r \neq \text{return } r$, where only on the left the traces have two transitions. As merely a monad structure, the induced denotational semantics is insufficiently abstract. For example, this inequation implies $\llbracket \langle \rangle ; \langle \rangle \rrbracket^{\mathcal{N}} \neq \llbracket \langle \rangle \rrbracket^{\mathcal{N}}$ – this model fails to satisfy even the most basic semantic equivalences. Still, we will find that we can use less abstract models as stepping stones to more abstract ones.

We identify a set of rewrite rules $\mathfrak{g} := \{\text{Ls}, \text{Ex}, \text{Cn}\}$ under which the operations of \mathcal{N} are closed. That is, return is pointwise closed under \mathfrak{g} ; if f is pointwise \mathfrak{g} -closed, then $\gg= f$ is \mathfrak{g} -closed; and similarly for the effect operations. We explain how the \mathfrak{g} -rewrite maintain this proposition as we present them. For now, let the *generating model* be the \mathfrak{g} -monad structure, which we denote by \mathcal{G} . So we have:

PROPOSITION 6.4. *For all $P_i \in \underline{\mathcal{G}}X_i$ and $f : X_1 \rightarrow \underline{\mathcal{G}}X_2$:*

$$P_1 \gg=^{\mathcal{N}} f = P_1 \gg=^{\mathcal{G}} f \quad P_1 \text{ ||| }^{\mathcal{N}} P_2 = P_1 \text{ ||| }^{\mathcal{G}} P_2$$

Moreover, $\text{return}^{\mathcal{N}} = \text{return}^{\mathcal{G}}$, $\llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{N}} = \llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{G}}$, and $\llbracket \text{rmw}_{\ell, \Phi} \rrbracket_{\mathcal{N}} = \llbracket \text{rmw}_{\ell, \Phi} \rrbracket_{\mathcal{G}}$.

This means that we can calculate in \mathcal{G} quite concretely; we need not worry about traces that are obtained from the set-definitions after some arbitrarily long chain of rewrites. So the connections we establish later between \mathcal{G} and the more abstract monad structures (§7.1) become easier to use.

Remark. *The difference between denotations in \mathcal{G} and in \mathcal{N} lies in the higher-order fragment. For example, return values of traces in $\llbracket \lambda f : 1 \rightarrow 1. f \langle \rangle \rrbracket_{\mathcal{T}}^{\mathcal{C}}$ are functions that take as argument elements in $\llbracket 1 \rrbracket \rightarrow \underline{\mathcal{T}} \llbracket 1 \rrbracket$. In particular, the denotation depends on $\underline{\mathcal{T}}$.*

In presenting the \mathfrak{g} -rewrite rules below, we provide operational intuition by drawing explicit connections with interrupted executions. However, this intuition should be taken with a grain of salt: the abstract model (§6.6) uses these rules as well, where traces do not correspond to interrupted executions as they do here.

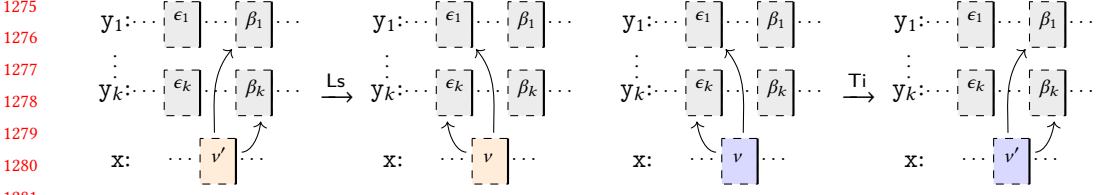


Fig. 8. Schematic depictions of the loosen (left) and tighten (right) rewrite rules, focusing on a particular memory snapshot within the trace, in the setting of $k+1$ locations. For every i , β_i and ϵ_i may dovetail, coincide, or be separated. **Left:** The environment message v' is “loosened” to v . **Right:** The local message v is “tightened” to v' .

Loosen. When a program relies on a message from the environment, it relies on the message’s view being small enough, to not obstruct the behavior that follows. In addition, it relies on the message’s timestamp, which is part of the view, to be big enough for it not to be obscured when needed. The rule is depicted on the left of Figure 8.

Define the loosen (Ls) rewrite rule:

$$\text{Assuming } v \leq_{vw} \epsilon, \alpha \left[\xi(\eta \sqcup \{\epsilon\}) \right] \omega \xrightarrow{\text{Ls}} \alpha \left[\xi(\eta \sqcup \{v\}) \right] \omega \quad (\text{Loosen})$$

Here, we use the partial order on messages $v \leq_{vw} \epsilon$ defined by requiring that they may only differ in their timestamps for other memory locations for which v ’s must precede ϵ ’s: $v.lc = \epsilon.lc$, $v.vl = \epsilon.vl$, $v.seg = \epsilon.seg$, and $v.vw \leq \epsilon.vw$. If the source in (Loosen) is a trace, then the target is a trace iff either η is empty or $v \hookrightarrow (\eta \sqcup \{v\}).o$.

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 bother with a formal proof, since the abstract model §6.6 violates this simplifying assumption anyway.

Given an interrupted execution, we can replace an environment message ϵ with a message $v \leq_{vw} \epsilon$ 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 $\omega := \alpha \sqcup \epsilon.vw$. In the new interrupted execution, we instead use the ADV rule to compensate for the earlier view in v , 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 $\epsilon.lc = v.lc$ hasn’t changed during the ADV steps. The RMW-modifier still fails in the new execution because v 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 v , and it inherits the current view: ω . Steps via other rules remain the same.

The \mathcal{N} operations are Ls-closed since the inclusion of a trace never relies on the view of an environment message other than its value, segment, and it being dominated by another view: $\epsilon.vw \leq \kappa$. Since $v \leq_{vw} \epsilon$, the value and segment agree and $v.vw \leq \epsilon.vw \leq \kappa$, and so the target trace will appear in the result of the operation.

Expel. The rewrite expel (Ex) replaces an environment message with two dovetailing messages that occupy the same segment and have the same view, the latter message also having the same

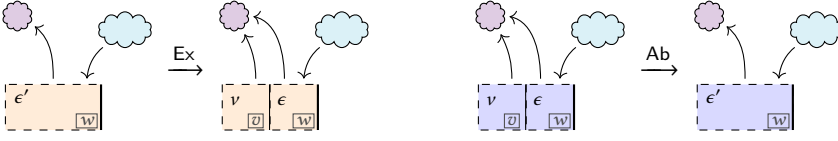


Fig. 9. Schematic depictions of the expel (left) and absorb (right) rewrite rules, that focus on the segment of the dovetailed messages together with all pointers into and out of them, within a particular memory snapshot. The *circular* cloud represents the subset of the memory that the messages in focus are pointing to, showing their views are the same. The *elliptical* cloud represents views—possibly including the initial and final view, as well as other messages—that point to each of the dovetailing messages. Thus, no view may point to v . A condition that is not depicted is that all the messages must appear in the same places in the chronicle. **Left:** The environment message v is “expelled” from the message ϵ' , which becomes ϵ . **Right:** The local message v is “absorbed” into the message ϵ , which becomes ϵ' .

value, as depicted on the left of Figure 9. This ensures that the value is available at the same timestamp with the same carried view, and that no more of the timeline is occupied. Formally:

$$\text{Assuming } v \prec \epsilon, \alpha \left[\xi(\eta \bar{\cup} \{\epsilon [i \mapsto v.i]\}) \right] \omega \xrightarrow{\text{Ex}} \alpha \left[\xi(\eta \bar{\cup} \{v, \epsilon\}) \right] \omega \quad (\text{Expel})$$

Here, $v \prec \epsilon$ is the *monotone dovetailing* relation, i.e., the two messages dovetail: $v.lc = \epsilon.lc$, $v.t = \epsilon.i$; and moreover their views compare: $v.vw \leq \epsilon.vw$. The final condition relaxes the rule as depicted in Figure 9 where the $v.vw = \epsilon.vw$ was required. This makes no difference to the model, because the relaxed version is obtained by applying *loosen* after the strict version, to obtain the required target.

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 $v \hookrightarrow (\eta \bar{\cup} \{v, \epsilon\}).o$.

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

The operations of \mathcal{N} are Ex-closed since they never rely on the absence of messages, only for the availability of segments, which is preserved by this rule.

Condense. In the condense (Cn) rewrite rule, the source behavior may include an environment message ϵ dovetailing some prior message v that carries the same value and view. The target behavior removes ϵ , and modifies v to a message v' that occupies the same segment as the two messages combined, as depicted on the left of Figure 10.

To formally capture how the views in the trace change in this rule, we define *pulling* a view κ along a message ϵ to be the view in which, if the timestamp at $\epsilon.lc$ is the initial timestamp of ϵ , then we update the timestamp to be the final timestamp (depicted on the right):

$$\kappa [\uparrow \epsilon] := \begin{cases} \kappa_{\epsilon.lc} = \epsilon.i : & \kappa [\epsilon.lc \mapsto \epsilon.t] \\ \text{otherwise:} & \kappa \end{cases} \quad \begin{array}{ccc} \kappa_{\epsilon.lc} & \kappa [\uparrow \epsilon]_{\epsilon.lc} & \\ \downarrow & \downarrow & \\ & (\epsilon.i, \epsilon.t) & \end{array}$$

We extend the pulling operation to messages, memories, chronicles, (pre-)traces, and view trees, by pulling the view associated with these objects. In particular, our representation of messages means that pulling a dovetailing message preceding ϵ along ϵ merges them into one contiguous message.

The rewrite rule, formally:

$$\text{Assuming } v \preceq \epsilon, \alpha \left[\xi(\eta \bar{\cup} \{v, \epsilon\}) \right] \omega \xrightarrow{\text{Cn}} \left(\alpha \left[\xi(\eta \bar{\cup} \{v\}) \right] \omega \right) [\uparrow \epsilon] \quad (\text{Condense})$$



Fig. 10. Schematic depictions of the condense (left) and dilute (right) rewrite rules, in the style of Figure 9. A condition that is not depicted is that v and v' must appear in the same places in the chronicle, and ϵ may not appear before them. The views that point to v' in the source can point either to v or to ϵ in the target. **Left:** The message v turns into v' by “condensing” the environment message ϵ . **Right:** The message v' turns into v by “diluting” out the local message ϵ .

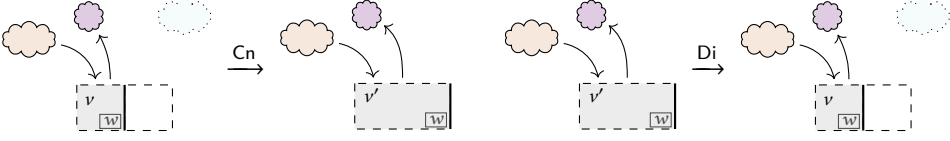


Fig. 11. Schematic depictions of the condense (left) and dilute (right) rewrite rules as in Figure 10, focusing this time on a memory without ϵ . **Left:** Since ϵ is to appear as an environment message in the chronicle, it can appear since the opening memory, not appear even in the closing memory, or 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.

Here we use the monotone *repetitive* dovetailing relation $v \dashv \epsilon$ where the two messages dovetail monotonically: $v \prec \epsilon$, and have the same value: $v.v1 = \epsilon.v1$. As was the case for *expel*, relaxing the condition that the views must be equal as depicted more strictly in Figure 10 is admissible, this time by applying *loosen* before the strict version, to obtain the required source.

The decomposition of the chronicle in the rule determines where ϵ first appears, but v can first appear earlier. This situation is depicted in Figure 11.

Unlike *loosen* and *expel*, when η is empty the target may differ from the source even though ϵ , nor any other message, is removed. This is due to *Cn* pulling views along ϵ , whether ϵ is there or not. So when η is empty the target differs from the source iff there is a message at $\epsilon.i = v.t$. In this case, assuming the source is a trace, for the target to be a trace $\epsilon.seg$ must be available, otherwise there will be a memory that is not scattered. If $\epsilon.seg$ 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 6.5. $\forall \epsilon \in \text{Msg} \forall \kappa, \sigma \in \text{View}. \kappa_{\epsilon.lc}, \sigma_{\epsilon.lc} \notin \epsilon.seg \setminus \epsilon.t \implies \kappa \leq \sigma \implies \kappa[\uparrow\epsilon] \leq \sigma[\uparrow\epsilon]$.

To summarize, if the source in (*Condense*) is a trace, then the target is a trace iff either η is non-empty, $v.t \notin \xi.c.t$, or $\epsilon.seg \cap \bigcup \xi.c.seg = \emptyset$.

If we have an interrupted execution with two messages v and ϵ as in *condense*, we will also have an interrupted execution without the environment message ϵ , and with v' instead of v . In the new interrupted execution, v' is used whenever either v or message ϵ were used in the original.

The operations of \mathcal{N} are *Cn*-closed. This 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 with ϵ in the source, it dovetails with v' in the target. Thus, if a message was added due to an RMW in the source, the condition to dovetail with a message that holds the loaded value is still met in the target. There are also new considerations involving the rewrite affecting the entire trace rather than just one or two messages. For instance, to show that $(\gg)=$ 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 is

subtle because the delimiting views change, and thus the condition imposed on binding the traces changes from $\kappa \leq \sigma$ to $\kappa [\uparrow\epsilon] \leq \sigma [\uparrow\epsilon]$. The condition still holds due to [Lemma 6.5](#) since neither κ nor σ point into the *interior* of ϵ .seg, because no message has a timestamp there. This insight resolves similar subtleties for the other \mathcal{N} -constructs.

6.5 Concrete Denotations

Brookes [13] pioneered two rewrite rules to make denotations abstract and support desired program transformations: stuttering and mumbling. To define our next model, we adapt these to our setting, as well as add two additional ones: $\mathfrak{c} := \{\text{St}, \text{Mu}, \text{Fw}, \text{Rw}\}$. We combine notations of rewrite-rule sets, e.g. $\mathfrak{gc} := \mathfrak{g} \cup \mathfrak{c}$. Thus, we denote by \mathcal{M} the \mathfrak{gc} -monad structure. We call this model the *concrete model* because, like the generating model, it still maintains a close correspondence to the operational semantics RA_{\leq} . However, \mathcal{M} is a monad, a crucial element in the proof of the adequacy theorem ([Appendix B.4](#)).

PROPOSITION 6.6. \mathcal{M} is a monad.

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

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

Note that for the target in (Stutter) to form a trace, provided that its source is a trace, we need to further require that μ is a well-formed memory, and that α points to μ (which may not be the case 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 previous, and is contained in subsequent, memories. This insertion does not change the initial or the final configurations of the interrupted execution.

As a concrete (contrived) example, stutter is used for validating the transformation $\langle \rangle ; \langle \rangle \rightarrow \langle \rangle ; \langle \rangle ; \langle \rangle$. Indeed, it is not true that $\llbracket \langle \rangle ; \langle \rangle \rrbracket_{\mathcal{G}}^{\mathfrak{c}} \supseteq \llbracket \langle \rangle ; \langle \rangle ; \langle \rangle \rrbracket_{\mathcal{G}}^{\mathfrak{c}}$, but we do have $\llbracket \langle \rangle ; \langle \rangle \rrbracket_{\mathcal{G}}^{\mathfrak{c} \{ \text{St} \}} \supseteq \llbracket \langle \rangle ; \langle \rangle ; \langle \rangle \rrbracket_{\mathcal{G}}^{\mathfrak{c}}$.

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:

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

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 $\dots \langle T, \mu \rangle, M \rightsquigarrow_{\text{RA}_{\leq}}^* \langle R, \rho \rangle, N \quad \langle R, \rho \rangle, N \rightsquigarrow_{\text{RA}_{\leq}}^* \langle H, \theta \rangle, K \dots$ that is compatible with the source trace, then we clearly have a shorter interrupted execution $\dots \langle T, \mu \rangle, M \rightsquigarrow_{\text{RA}_{\leq}}^* \langle H, \theta \rangle, K \dots$ that is compatible with the target trace.

As a concrete example, mumble is used for validating the transformation $\langle \rangle ; M \rightarrow M$. Indeed, it is not true that $\llbracket \langle \rangle ; M \rrbracket_{\mathcal{G}}^{\mathfrak{c}} \supseteq \llbracket M \rrbracket_{\mathcal{G}}^{\mathfrak{c}}$, because the traces on the left have an additional transition. However, $\llbracket \langle \rangle ; M \rrbracket_{\mathcal{G}}^{\mathfrak{c} \{ \text{Mu} \}} \supseteq \llbracket M \rrbracket_{\mathcal{G}}^{\mathfrak{c}}$ is true, because we can always pick the memory in the transition from $\llbracket \langle \rangle \rrbracket_{\mathcal{G}}^{\mathfrak{c}}$ to match that of $\llbracket M \rrbracket_{\mathcal{G}}^{\mathfrak{c}}$, and then get rid of the transition by using mumble.

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:

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

1470



Fig. 12. Schematic depictions of the rewind and forward rewrite rule, focusing on a single location, where the initial/final view points to v before and points to ϵ after. The messages v and ϵ may coincide, dovetail, or be separated. **Left:** The initial view α is “rewound” to α' . **Right:** The final view ω is “forwarded” to ω' .

The rule is also depicted in Figure 12. Note that for the target in (Forward) to form a trace (rather than a pre-trace), provided that its source is a trace, we need to further require that $\omega \hookrightarrow \xi.c$.

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

As a concrete example, St and Fw are used in validating the transformation $M \rightarrow M; \langle \rangle$. We can use St to compensate for the transition added by $\llbracket \langle \rangle \rrbracket_{\mathcal{G}}^c$. However, this is insufficient on its own, because not only is there an extra transition, it is also possible that the final view from $\llbracket M \rrbracket_{\mathcal{G}}^c$ and the initial view from $\llbracket \langle \rangle \rrbracket_{\mathcal{G}}^c$ used in the binding are different. To compensate for that we use Fw.

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:

$$\text{Assuming } \alpha \leq \kappa, \kappa \llbracket \xi \rrbracket \omega \xrightarrow{\text{Rw}} \alpha \llbracket \xi \rrbracket \omega \quad (\text{Rewind})$$

The rule is also depicted in Figure 12. Note that for the target in (Rewind) to form a trace (rather than a pre-trace), provided that its source is a trace, we need to further require that $\alpha \hookrightarrow \xi.o$.

We can also understand rewind using interrupted executions, similarly to how 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, rewind and stutter are used in validating the transformation $M \rightarrow \langle \rangle; M$.

6.6 Abstract Denotations

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

PROPOSITION 6.7. \mathcal{A} is a monad.

By including the additional rewrite rules of \mathfrak{a} we give up the strictly operational interpretation that we have assumed when presenting the previous rules. This 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 rewrite rule, we present a program transformations whose validation uses that particular rule, though other gc-rewrites are often required as well.

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 v , its view serves to guarantee that loading it would not obscure any message within a certain portion of the memory. Therefore, replacing v by ϵ that only differs in its view, where $v \leq_{vw} \epsilon$, as depicted on the right of Figure 8, 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.

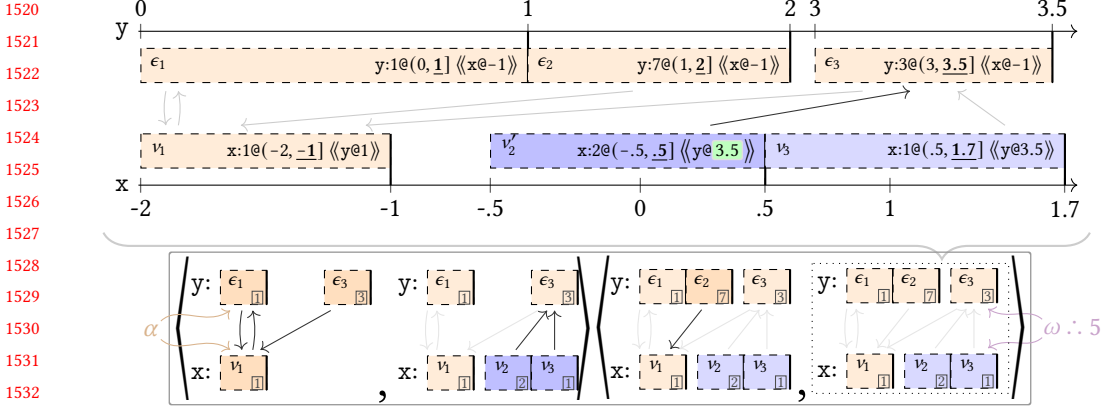


Fig. 13. A possible result from rewriting the trace from Figure 1 using tighten. Since v_2 is local in the trace from Figure 1, tighten can advance its view to point to ϵ_3 instead of ϵ_1 . The same replacement is applied throughout the trace’s sequence, not just the closing memory.

This is the effect of the tighten (Ti) rewrite rule. Formally:

$$\text{Assuming } v \leq_{vw} \epsilon, \alpha \left[\xi \langle \mu, \rho \uplus \{v\} \rangle \eta \bar{\uplus} \{v\} \right] \omega \xrightarrow{\text{Ti}} \alpha \left[\xi \langle \mu, \rho \uplus \{\epsilon\} \rangle \eta \bar{\uplus} \{\epsilon\} \right] \omega \quad (\text{Tighten})$$

See Figure 13 for a concrete example.

As a concrete benefit of tighten, consider the RA-valid (but SC-invalid) write-read-reordering transformation $\ell := v; \text{let } a = \ell' \text{ in } a \rightarrow \text{let } a = \ell' \text{ in } \ell := v; a$, where $\ell \neq \ell'$. On the right, the added message carries the view of the thread after it is increased by the view of the loaded message, but, on the left, the added message carries the initial view of the thread. By applying tighten to traces of the left, we compensate for this difference.

Absorb. Applying absorb (Ab) removes a local message v and decreases the initial timestamp of a dovetailing local message ϵ with the same view, such that the resulting ϵ' covers the segment of v . This is depicted on the right of Figure 9. 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 v 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 $\epsilon_1^t := \epsilon [i \mapsto t]$:

$$\text{Assuming } v \prec \epsilon, \alpha \left[\xi \langle \mu, \rho \uplus \{v, \epsilon\} \rangle \eta \bar{\uplus} \{v, \epsilon\} \right] \omega \xrightarrow{\text{Ab}} \alpha \left[\xi \langle \mu, \rho \uplus \{\epsilon_1^{v.i}\} \rangle \eta \bar{\uplus} \{\epsilon_1^{v.i}\} \right] \omega \quad (\text{Absorb})$$

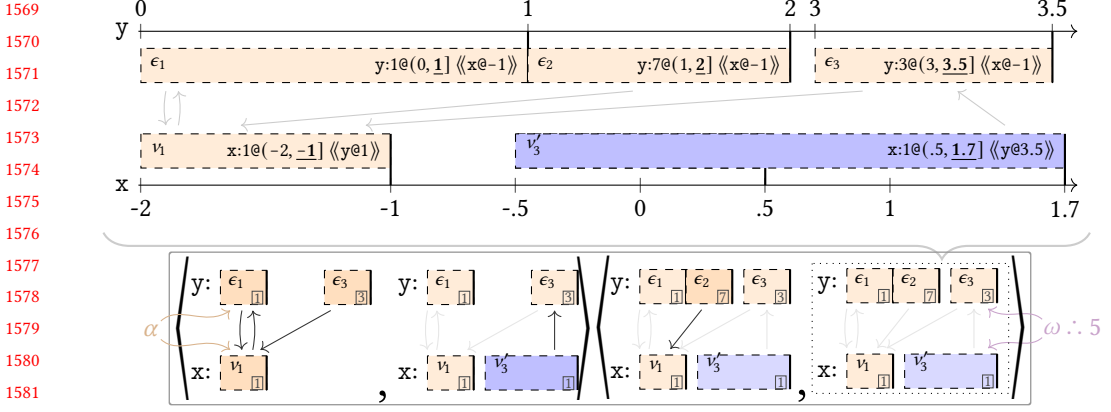
See Figure 14 for a concrete example.

As in *expel*, we relax the condition that the views must be equal, which tighten can compensate for.

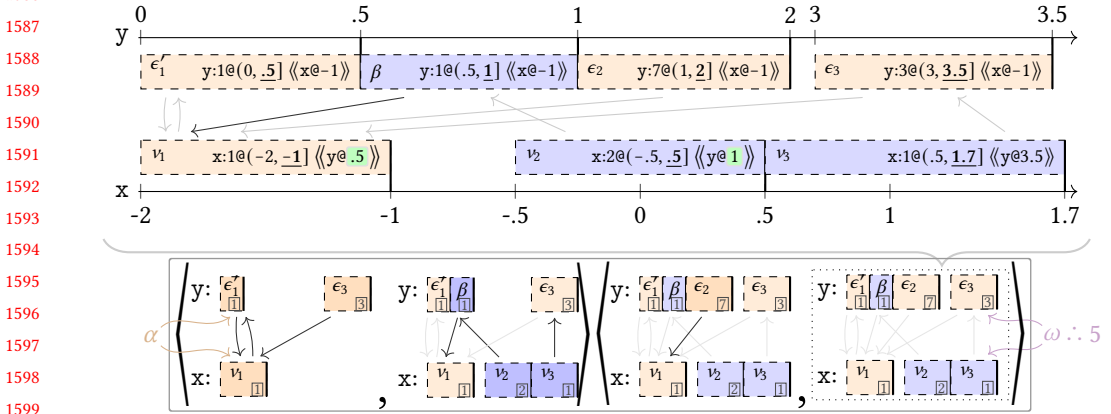
The transformation $\ell := w; \ell := v \rightarrow \ell := v$ is a concrete example where this rule is useful, in which we use absorb to compensate for the extra message. Specifically, if the local message on the right is β , we pick some t from the interior of $\beta.\text{seg}$, a trace with a local message due to $\ell := w$ that has the segment $(\beta.i, t]$ and a trace with a local message due to $\ell := u$ that has the segment $(t, \beta.\tau]$. After binding, we use *mumble* to combine the transitions, then absorb to replace these two messages with β .

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

$$\text{Assuming } v \prec \epsilon, \left(\alpha \left[\xi \langle \mu, \rho \uplus \{v\} \rangle \eta \bar{\uplus} \{v\} \right] \omega \right) [\uparrow \epsilon] \xrightarrow{\text{Di}} \alpha \left[\xi \langle \mu, \rho \uplus \{v, \epsilon\} \rangle \eta \bar{\uplus} \{v, \epsilon\} \right] \omega \quad (\text{Dilute})$$



1583 **Fig. 14.** A possible result from rewriting of the trace from Figure 13 using absorb. The dovetailed messages v_2 and v_3 are local in the trace from Figure 1, added within the same transition, so by rewriting by absorb they can be replaced by v_3 obtained by stretching v_3 's segment to cover v_2 's segment.



1601 **Fig. 15.** 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 with it and carrying the same value. The message ϵ_2 , that used to dovetail with ϵ_1 , now dovetails with β .

1602 See Figure 15 for a concrete example.

1603 We restrict to the case that $v.vw = \epsilon.vw$ when explaining the rule. The rest can be seen as a
1604 formal extension which is admissible in the presence of tighten, much like we had with condense
1605 and loosen.

1606 Unpacking this definition, we first note that, although we are focusing on the case where the
1607 source and target are traces, the pre-trace expression $\tau' := \alpha \left[\xi \langle \mu, \rho \uplus \{v\} \rangle \eta \bar{\mu} \uplus \{v\} \right] \omega$ within the
1608 source may not be a trace itself. In particular, there could be views in τ' that point to ϵ even though
1609 there is no message there until the pulling ($-$) $[\uparrow\epsilon]$ takes effect, after which they will point to v $[\uparrow\epsilon]$.
1610 There could also be views that point to v in τ' , which too point to v $[\uparrow\epsilon]$ in the source. Therefore,
1611 views that point to v $[\uparrow\epsilon]$ in the source could point to either v or ϵ in the target—the latter being a
1612 pointer moving. That is, in terms of the memory graph's structure, we think of v $[\uparrow\epsilon]$ and v being
1613

the same vertex labeled differently in the memories before and after the rewrite respectively, with some pointers moved to the newly added ϵ -labeled vertex. This is depicted on the right of Figure 10.

Another tricky thing about this rule is that ν could appear first before ϵ , and could be either a local message or an environment message. This is depicted on the right of Figure 11.

Justifying this rewrite, if the program relies on $\nu.v1$ being available at $\nu[\uparrow\epsilon].t$, it can instead rely on it being available with $\nu.t$ with a view that will impose the same restrictions on the program once it loads the message and inherits the view; all this, so long as the remainder of the segment remains unoccupied until it guarantees the same value and the same view there (with ϵ). Similarly, if the program guarantees the value with $\nu[\uparrow\epsilon]$, it can guarantee it with ν instead, so long as only ϵ can occupy the remaining segment.

As a concrete example of this rule in use, consider the transformation $\ell? \rightarrow \text{FAA}(\ell, 0)$. A trace from the target has a message ϵ added to dovetail with an existing message ν . So there is a matching trace in the source without that added message. By closure under \mathfrak{g} , we can Cn -rewrite the trace, pulling ν by ϵ (the Cn -rewrite is defined even when pulling by a message that is not there). Then, we can apply dilute to add ϵ .

7 METATHEORY

The difference between the different monads from §6 are due to the abstraction afforded to them by the rewrite rules under which they are closed. 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 monads using properties of the rewrites rules and their interactions (§7.1).

Then, focusing on the denotational semantics of interest, over \mathcal{A} , we prove (directional) compositionality (§7.2) and soundness (§7.3). These results, interesting in their own right, are also stepping stones towards (directional) adequacy (§7.4). Finally, we exhibit the sufficient abstraction of the denotational semantics with various transformations it supports (§7.5).

7.1 Commutativity of Rewrites

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 adding 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 rewrite $\tau \xrightarrow{\text{St}} \pi \xrightarrow{\text{Fw}} \varrho$, it may not be possible to find any π' such that $\tau \xrightarrow{\text{Fw}} \pi' \xrightarrow{\text{St}} \varrho$.

At other times, commuting in this way is guaranteed to be possible. As a relatively simple example, an application of loosen can always be made before one of stutter rather than after it. Even if the message that loosen acts on happens to appear in the transition that stutter adds to the chronicle in the sequence $\tau \xrightarrow{\text{St}} \pi \xrightarrow{\text{Ls}} \varrho$, in the alternative sequence $\tau \xrightarrow{\text{Ls}} \pi' \xrightarrow{\text{St}} \varrho$ the transition added already includes the “loosened” message. It is important to check that the pre-trace π' is in fact a trace. Since τ is a trace, then—other than the trivial case in which π' is τ itself—we only need to check that the “loosened” message points downwards into the memories in which it appears. This we infer from the fact that every memory in π' appears in ϱ , which is itself a trace.

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 the metatheory, and it is an immediate consequence of the following lemma. We write $x \stackrel{\times}{\Leftarrow} y$ when $\xrightarrow{x} \xrightarrow{y} \subseteq \xrightarrow{y} \xrightarrow{x}$, where \xrightarrow{x} and \xrightarrow{y} are restricted to traces.

LEMMA 7.1 (REWRITE COMMUTATIVITY). *If $x \in \mathfrak{a}$ and $y \in \mathfrak{gc}$, or $x \in \mathfrak{ca}$ and $y \in \mathfrak{g}$, then $x \stackrel{\times}{\Leftarrow} y$.*

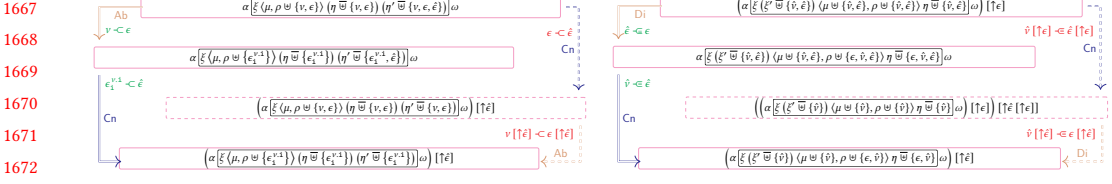


Fig. 16. Two cases from the proof of [Rewrite Commutativity](#) shown side-by-side, in which “active” messages overlap.

PROOF. The proof proceeds by case analysis on x and y , each encapsulated in diagram(s) such as the two in [Figure 16](#). Each diagram shows the assumed rewrite sequence $\tau \xrightarrow{x} \pi \xrightarrow{y} \varrho$ on the left, with the conditions that are known because they were required for the rewrites to be applicable; and the deduced sequence $\tau \xrightarrow{x} \pi' \xrightarrow{y} \varrho$ on the right, with the conditions that need to hold for the rewrites to be applicable. The conditions are enough to show that the rewrite 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 rewrite rules.

The cases in [Figure 16](#) are among the more interesting cases in which the activities of x and y overlap. The left diagram shows a sub-case of $\text{Ab} \Leftrightarrow \text{Cn}$ in which the absorbing message (ϵ) also serves as the condensing message. On the right, a sub-case of $\text{Di} \Leftrightarrow \text{Cn}$ 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 $(-\uparrow[\hat{\epsilon}])[\uparrow\epsilon[\hat{\epsilon}]] = (-\uparrow[\hat{\epsilon}])[\uparrow\epsilon[\uparrow\hat{\epsilon}]]$.

The entire collection of diagrams is in [Appendix D](#). \square

Remark. When defining the rewrite rules, we could have restricted $v \prec \epsilon$ (and similarly $v \prec \hat{\epsilon}$) to messages with equal views: $v.v\omega = \epsilon.v\omega$, resulting in equivalent closures. For example, to apply the restricted version of absorb, one first applies tighten, which is also an α -rewrite, to make the views equal. In fact, we used this slightly simpler presentation in the abridged version of this paper [21]. However, we would have to make [Rewrite Commutativity](#) less granular.

For example, we would not have $\text{Di} \Leftrightarrow \text{Ls}$. Here, it may be the case that we “dilute” an environment message and then “loosen” it. After commuting, if we only have the restricted version of dilute, we then need to “tighten” the new local message to recover the resulting trace from the original rewrite sequence.

As a corollary to [Rewrite Commutativity](#), we can commute ca -rewrites out of the \mathcal{G} -operators:

LEMMA 7.2 (DEFERRAL OF CLOSURE). Let $\mathfrak{c} \subseteq \star \subseteq \text{ca}$. For all $P_i \in \underline{\mathcal{G}}X_i$ and $f : X_1 \rightarrow \underline{\mathcal{G}}X_2$:

$$\left(P_1^\star \gg^{\mathcal{G}} f^\star \right)^\star = \left(P_1 \gg^{\mathcal{G}} f \right)^\star \quad \left(P_1^\star \parallel^{\mathcal{G}} P_2^\star \right)^\star = \left(P_1 \parallel^{\mathcal{G}} P_2 \right)^\star$$

PROOF. In the proof we rely on the fact that every closure in \mathfrak{a} is mirrored in \mathfrak{g} . For example, instead of rewriting some trace $\tau \in P_1$ by Ab and then “binding” it with a trace $\pi \in f(\tau.v1)$, we can instead mirror its effect by Ex -rewriting π to make its messages match τ ’s, bind *those* together, and then use Ab after the bind.

The detailed proof is in [Appendix B.1](#). \square

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

1716 *Example 7.3.* The associativity laws for \mathcal{M} and \mathcal{A} are implied by the one for \mathcal{G} . To show this
1717 for \mathcal{M} , we specialize [Deferral of Closure](#) to $\star = \mathfrak{c}$, and restrict to $P \in \underline{\mathcal{M}}X$, $f : X \rightarrow \underline{\mathcal{M}}Y$, and
1718 $g : Y \rightarrow \underline{\mathcal{M}}Z$, obtaining:

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

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

1720 The same can be repeated for \mathcal{A} by specializing to $\star = \mathfrak{ca}$.
1721

1722 When calculating denotations of terms, we can use [Deferral of Closure](#) to similarly delay taking
1723 the closure. For programs specifically, we can delay all the way through, only taking the closure at
1724 the top level. Relating \mathcal{M} to \mathcal{A} in this way is a key step in our proof of adequacy. Thus, we state:
1725

1726 LEMMA 7.4 (RETROACTIVE CLOSURE). *If M is a program, then $\llbracket M \rrbracket^{\mathfrak{c}} = \llbracket M \rrbracket_{\mathcal{M}}^{\mathfrak{c} \mathfrak{a}}$.*
1727

1728 The proof is in [Appendix B.3](#).
1729

1730 7.2 Compositionality

1731 To state compositionality, and later adequacy, we need a few technical concepts involving captur-
1732 ing and capture-avoiding substitution in λ_{RA} and its semantics. We extend λ_{RA} with well-typed
1733 second-order *metavariables*: these are binding-aware identifiers $\Gamma \vdash \mathbf{M} : A$. Metavariables repre-
1734 sent “holes” into which we can slot well-typed terms $\Gamma \vdash M : A$, in an operation called *metavariable*
1735 *substitution*. When such a metavariable appears in a term, it is accompanied by an explicit value
1736 substitution governing which values to substitute when we slot a term into it. Metavariable sub-
1737 stitution captures the variables of which the metavariables are aware.
1738

1739 *Example 7.5.* Consider the following metavariable that is aware of a context with two variables:
1740 $a : \text{Loc}, b : \text{Val} \vdash \mathbf{M} : 1$. The term $\vdash \mathbf{M}[a \mapsto x, b \mapsto 42] : 1$ contains this metavariable and no other
1741 variables. Metasubstituting the open term $a : \text{Loc}, b : \text{Val} \vdash a := b$ for \mathbf{M} yields $\vdash x := 42 : 1$.
1742

1743 This treatment of metavariables and their substitution is tedious but standard given the binding
1744 structure in the syntax. A (*term*) *context* $\Delta \vdash \Xi [\Gamma \vdash - : A] : B$ is a term of type B with variables
1745 from Δ and one meta-variable $\Gamma \vdash - : A$ of type A that assumes a binding context Γ . It is a *program*
1746 *context* if Δ is empty and $B = G$ is ground.
1747

1748 The recursive definition of a term’s denotation only uses the denotations of its subterms, so the
1749 semantics is automatically compositional. Abbreviating $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$ into $\Gamma \vdash M, N : A$:
1750

1751 PROPOSITION 7.6 (COMPOSITIONALITY). *Let $\Delta \vdash \Xi [\Gamma \vdash - : A] : B$ be a term context and assume*
1752 $\Gamma \vdash M, N : A$. *If $\llbracket M \rrbracket^{\mathfrak{c}} = \llbracket N \rrbracket^{\mathfrak{c}}$ then $\llbracket \Xi [M] \rrbracket^{\mathfrak{c}} = \llbracket \Xi [N] \rrbracket^{\mathfrak{c}}$.*
1753

1754 However, we are interested in a directional version of this, dealing not only with set equality
1755 but also with set inclusion. Simply replacing $=$ with \subseteq in [Proposition 7.6](#) results in a false claim.
1756 This is because the language is higher-order, so only a “nested” form of containment holds, which
1757 degenerates to containment when restricted to programs:
1758

1759 THEOREM 7.7 (DIRECTIONAL COMPOSITIONALITY). *Let $\cdot \vdash \Xi [\Gamma \vdash - : A] : G$ be a program context*
1760 *and assume $\Gamma \vdash M, N : A$. If $\llbracket M \rrbracket^{\mathfrak{c}} \subseteq \llbracket N \rrbracket^{\mathfrak{c}}$ then $\llbracket \Xi [M] \rrbracket^{\mathfrak{c}} \subseteq \llbracket \Xi [N] \rrbracket^{\mathfrak{c}}$.*
1761

1762 The proof is in [Appendix B.2](#).
1763

7.3 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 7.8 (SOUNDNESS). *For a program M , if $\langle T, \mu \rangle, M \Downarrow V$, then $\inf_{\mu} T \langle \overline{\mu}, \mu' \rangle \omega \cdot V \in \llbracket M \rrbracket^c$ for some μ' and ω .*

The proof is in [Appendix B.3](#).

7.4 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 $\Gamma \vdash M \sqsubseteq N : A$, or $M \sqsubseteq N$ for short, if $\langle \dot{\alpha}, \mu \rangle, \Xi [M] \Downarrow V \implies \langle \dot{\alpha}, \mu \rangle, \Xi [N] \Downarrow V$ for every program context $\cdot \vdash \Xi [\Gamma \vdash - : A] : G$, initial configuration state $\langle \dot{\alpha}, \mu \rangle$, and value V .

THEOREM 7.9 (DIRECTIONAL ADEQUACY). *If $\llbracket M \rrbracket^c \subseteq \llbracket N \rrbracket^c$ then $M \sqsubseteq N$.*

The proof begins by examining the tight correspondence between traces in denotations over \mathcal{M} and interrupted executions. Formally, we write $M : \tau : V$ when M *executes through* τ to V : there is an interrupted execution from M to V such that $\tau.v\downarrow = \llbracket V \rrbracket_{\mathcal{M}}^v$, which starts with the view-leaf labeled by $\tau.i\downarrow v$, passes exactly through the memory transitions of $\tau.ch$, and ends with the view-leaf labeled by $\tau.f\downarrow v$. By the [Fundamental Lemma](#), the statement and proof of which we relegate to [Appendix B.4](#), if $\tau \in \llbracket M \rrbracket_{\mathcal{M}}^c$ then there exists an appropriate value V such that $M : \tau : V$.

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

LEMMA 7.10 (EVALUATION LEMMA). *For a program M , if $\alpha \langle \overline{\mu}, \rho \rangle \omega \cdot r \in \llbracket M \rrbracket^c$ then $\langle \dot{\alpha}, \mu \rangle, M \Downarrow r$.*

PROOF. By [Retroactive Closure](#) the trace is obtained by \mathfrak{a} -rewriting a trace in the \mathcal{M} denotation. We proceed by induction on the length of this sequence. In the base case, we use the [Fundamental Lemma](#) which, for a single-transition trace degenerates to an uninterrupted execution. For the step, we observe that \mathfrak{a} -rewrites preserve evaluation. We leave the details to [Appendix B.4](#). \square

The converse of the [Evaluation Lemma](#) we already have as a special case of [Soundness](#). These two, together with [Directional Compositionality](#), give us [Directional Adequacy](#):

PROOF OF DIRECTIONAL ADEQUACY. Assume $\llbracket M \rrbracket^c \subseteq \llbracket N \rrbracket^c$. 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](#), $\tau \in \llbracket \Xi [M] \rrbracket^c$ for some τ of the form $\alpha \langle \overline{\mu}, - \rangle \cdot \cdot V$. By [Directional Compositionality](#) and the assumption, $\tau \in \llbracket \Xi [N] \rrbracket^c$. By the [Evaluation Lemma](#), $\langle \dot{\alpha}, \mu \rangle, \Xi [N] \Downarrow V$. \square

7.5 Validating Transformations

Using [Directional Adequacy](#), we can validate $M \rightarrow N$ in our model by showing that $\llbracket M \rrbracket^c \supseteq \llbracket N \rrbracket^c$. This already justifies structural transformations by virtue of using standard denotational semantics, as mentioned in [§6.2](#). For others, thanks to [Deferral of Closure](#) and closure preserving containment, we can use the \mathcal{G} operators instead of the \mathcal{A} operators, making calculations simpler.

[Figure 3](#) lists various transformations that we support in this way. The interested reader can find a more general collection in [Appendix C \(Table 2\)](#) and proofs thereof.

1814 The listed memory-access transformations are stated in ground terms, but imply more general
 1815 variants. For example, Write-Write Elimination is stated as $\ell := w; \ell := v \rightarrow \ell := v$, from which we can
 1816 deduce e.g., $\lambda a : \text{Loc}. a := w; a := v \rightarrow \lambda a : \text{Loc}. a := v$. This is a consequence of using the standard
 1817 semantics: structural transformations include any pure computations that result in the same value,
 1818 and in particular, we can replace the locations and (storable) values with pure computations that
 1819 result in them, or program variables of the same type.

1820
 1821 All told, we claim that our adequate denotational semantics is sufficiently abstract. This sup-
 1822 ports the case that Moggi’s semantic toolkit can successfully scale to handle the intricacies of RA
 1823 concurrency by adapting Brookes’s traces.

1824 8 RELATED WORK AND CONCLUDING REMARKS

1826 Our work follows the approach of Brookes [13] and its extension to higher-order functions us-
 1827 ing monads by Benton et al. [6]. Brookes developed a denotational semantics for shared memory
 1828 concurrency under standard sequentially consistency [35], and established full abstraction w.r.t. a
 1829 language that has a global atomic `await` instruction that locks the entire memory. The concepts
 1830 behind this approach had been used in multiple related developments, e.g. [12, 36, 37, 48]. We hope
 1831 that our work that targets RA will pave the way for similar continuations.

1832 Jagadeesan et al. [25] adapted Brookes’s semantics to the x86-TSO memory model [42]. They
 1833 showed that for x86-TSO it suffices to include the final store buffer at the end of the trace and add
 1834 two additional simple closure rules that emulate non-deterministic propagation of writes from
 1835 store buffers to memory, and identify observably equivalent store buffers. The x86-TSO model,
 1836 however, is much closer to sequential consistency than RA, which we study in this paper. In partic-
 1837 ular, unlike RA, x86-TSO is “multi-copy-atomic” (writes by one thread are made globally visible to
 1838 all other threads at the same time) and successful RMW operations are immediately globally visible.
 1839 Additionally, the parallel composition construct in Jagadeesan et al. [25] is rather strong: threads
 1840 are forked and joined only when the store buffers are empty. Being non-multi-copy-atomic, RA
 1841 requires a more delicate notion of traces and closure rules, but it has more natural meta-theoretic
 1842 properties, which one would expect from a programming language concurrency model: sequenc-
 1843 ing, a.k.a. thread-inlining, is unsound under x86-TSO [see 25, 33] but sound under RA (see Figure 3).

1844 Burckhardt et al. [14] developed a denotational semantics for hardware weak memory mod-
 1845 els (including x86-TSO) following an alternative approach. They represent sequential code blocks
 1846 by sequences of operations that the code performs, and close them under certain rewrite rules (re-
 1847 orderings and eliminations) that characterize the memory model. This approach does not validate
 1848 important optimizations, such as Read-Read Elimination. Moreover, unlike x86-TSO, RA cannot
 1849 be characterized by rewrite operations on SC traces [33].

1850 Dodds et al. [19] developed a fully abstract denotational semantics for RA, extended with fences
 1851 and non-atomic accesses. Their semantics is based on RA’s *declarative* (a.k.a. axiomatic) formu-
 1852 lation as acyclicity criteria on execution graphs. Roughly speaking, their denotation of code blocks
 1853 (that they assume to be sequential) quantifies over all possible context execution graphs and cal-
 1854 culates for each context the “happens-before” relation between context actions that is induced by
 1855 the block. They further use a finite approximation of these histories to atomically validate refine-
 1856 ment in a model checker. While we target RA as well, there are two crucial differences between
 1857 our work and Dodds et al. [19]. First, we employ Brookes-style totally ordered traces and use
 1858 interleaving-based operational presentation of RA. Second, and more importantly, we strive for
 1859 a compositional semantics where denotations of compound programs are defined as functions
 1860 of denotations of their constituents, which is not the case for Dodds et al. [19]. Their model can
 1861 nonetheless validate transformations by checking them locally without access to the full program.

1862

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 [45] 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. 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.

1896

1897

1898

1899

1900

1901

1902

1903

1904

1905

1906

1907

1908

1909

1910

1911

1911

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](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

- 1912 [4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In
 1913 *POPL*. ACM. <https://doi.org/10.1145/1926385.1926394>
- 1914 [5] Nick Benton, Martin Hofmann, and Vivek Nigam. 2014. Abstract effects and proof-relevant logical relations. In *POPL*.
 1915 ACM.
- 1916 [6] Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-dependent transformations for concurrent programs.
 1917 In *PPDP*. ACM. <https://doi.org/10.1145/2967973.2968602>
- 1918 [7] Nick Benton, John Hughes, and Eugenio Moggi. 2000. Monads and Effects. In *APPSEM*.
- 1919 [8] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based
 1920 program transformations with dynamic allocation. In *PPDP*. ACM.
- 1921 [9] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational semantics for effect-based
 1922 program transformations: higher-order store. In *PPDP*. ACM.
- 1923 [10] Nick Benton and Benjamin Leperchey. 2005. Relational Reasoning in a Nominal Semantics for Storage. In *TLCA*.
 1924 Springer.
- 1925 [11] Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *CSL (LIPIcs, Vol. 16)*.
 1926 Schloss Dagstuhl - Leibniz-Zentrum für Informatik. <https://doi.org/10.4230/LIPIcs.CSL.2012.107>
- 1927 [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>
- 1928 [13] Stephen D. Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Inf. Comput.* 127, 2 (1996).
 1929 <https://doi.org/10.1006/inco.1996.0056>
- 1930 [14] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. 2010. Verifying Local Transformations on Relaxed Mem-
 1931 ory Models. In *CC (LNCS, Vol. 6011)*. Springer. https://doi.org/10.1007/978-3-642-11970-5_7
- 1932 [15] Simon Castellan. 2016. Weak memory models using event structures. In *JFLA*. Saint-Malo, France. <https://hal.inria.fr/hal-01333582>
- 1933 [16] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *CGO*.
 1934 ACM. <https://doi.org/10.1145/2854038.2854051>
- 1935 [17] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular data-race-freedom guarantees in the
 1936 promising semantics. In *PLDI*. ACM. <https://doi.org/10.1145/3453483.3454082>
- 1937 [18] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential reasoning for optimizing
 1938 compilers under weak memory concurrency. In *PLDI*. ACM. <https://doi.org/10.1145/3519939.3523718>
- 1939 [19] Mike Dodds, Mark Batty, and Alexey Gotsman. 2018. Compositional Verification of Compiler Optimisations on Re-
 1940 laxed Memory. In *ESOP (LNCS, Vol. 10801)*. Springer. https://doi.org/10.1007/978-3-319-89884-1_36
- 1941 [20] Yotam Dvir, Ohad Kammar, and Ori Lahav. 2022. An Algebraic Theory for Shared-State Concurrency. In *APLAS*
 1942 (*LNCS, Vol. 13658*). Springer. https://doi.org/10.1007/978-3-031-21037-2_1
- 1943 [21] Yotam Dvir, Ohad Kammar, and Ori Lahav. 2024. A Denotational Approach to Release/Acquire Concurrency. In *ESOP*
 1944 (*LNCS*). Springer, 121–149. https://doi.org/10.1007/978-3-031-57267-8_5
- 1945 [22] Tony Hoare and Stephan van Staden. 2014. The laws of programming unify process calculi. *Sci. Comput. Program.* 85
 1946 (2014). <https://doi.org/10.1016/j.scico.2013.08.012>
- 1947 [23] Martin Hofmann. 2008. Correctness of effect-based program transformations. In *Formal Logical Methods for System*
 1948 *Security and Correctness*. IOS Press.
- 1949 [24] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory.
 1950 *Proc. ACM Program. Lang.* 4, OOPSLA (2020). <https://doi.org/10.1145/3428262>
- 1951 [25] Radha Jagadeesan, Gustavo Petri, and James Riely. 2012. Brookes Is Relaxed, Almost!. In *FOSSACS (LNCS, Vol. 7213)*.
 1952 Springer. https://doi.org/10.1007/978-3-642-28729-9_12
- 1953 [26] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon:
 1954 compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022).
 1955 <https://doi.org/10.1145/3498716>
- 1956 [27] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak
 1957 Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Pro-*
 1958 *gramming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für
 1959 Informatik. <https://doi.org/10.4230/LIPIcs.ECOOP.2017.17>
- 1960 [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>

- 1961 [31] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*. ACM. <https://doi.org/10.1145/2837614.2837643>
- 1962 [32] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making weak memory
- 1963 models fair. *Proc. ACM Program. Lang.* 5, OOPSLA (2021). <https://doi.org/10.1145/3485475>
- 1964 [33] Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM*
- 1965 (*LNCS*, Vol. 9995). https://doi.org/10.1007/978-3-319-48989-6_29
- 1966 [34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency
- 1967 in C/C++11. In *PLDI*. ACM. <https://doi.org/10.1145/3062341.3062352>
- 1968 [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>
- 1969 [36] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program
- 1970 transformations. In *POPL*. ACM. <https://doi.org/10.1145/2103656.2103711>
- 1971 [37] Hongjin Liang, Xinyu Feng, and Ming Fu. 2014. Rely-Guarantee-Based Simulation for Compositional Verification of
- 1972 Concurrent Program Transformations. *ACM Trans. Program. Lang. Syst.* 36, 1 (2014). <https://doi.org/10.1145/2576235>
- 1973 [38] Kenji Maillard, Cătălin Hrițcu, Exequiel Rivas, and Antoine Van Muylder. 2019. The next 700 Relational Program
- 1974 Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (Dec. 2019). <https://doi.org/10.1145/3371072>
- 1975 [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>
- 1976 [40] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991). [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4)
- 1977 [41] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimi-
- 1978 sations in the C11/C++11 memory model. In *PLDI*. ACM. <https://doi.org/10.1145/2491956.2491967>
- 1979 [42] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs (LNCS*,
- 1980 *Vol. 5674)*. Springer. https://doi.org/10.1007/978-3-642-03359-9_27
- 1981 [43] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed
- 1982 Dependencies in Weak Memory Concurrency. In *ESOP (LNCS, Vol. 12075)*. Springer. https://doi.org/10.1007/978-3-030-44914-8_22
- 1983 [44] Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *FOSSACS*. Springer Berlin
- 1984 Heidelberg.
- 1985 [45] Daniel Poetzl and Daniel Kroening. 2016. Formalizing and Checking Thread Refinement for Data-Race-Free Execution
- 1986 Models. In *TACAS (LNCS, Vol. 9636)*. Springer. https://doi.org/10.1007/978-3-662-49674-9_30
- 1987 [46] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM
- 1988 concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL
- 1989 (2018). <https://doi.org/10.1145/3158107>
- 1990 [47] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multi-
- 1991 processors. In *PLDI*. ACM. <https://doi.org/10.1145/1993498.1993520>
- 1992 [48] Aaron Joseph Turon and Mitchell Wand. 2011. A separation logic for refining concurrent objects. In *POPL*. ACM. <https://doi.org/10.1145/1926385.1926415>
- 1993 [49] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Com-
- 1994 mon 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>
- 1995 [50] Qiwen Xu, Willem P. de Roever, and Jifeng He. 1997. The Rely-Guarantee Method for Verifying Shared Variable
- 1996 Concurrent Programs. *Formal Aspects Comput.* 9, 2 (1997). <https://doi.org/10.1007/BF01211617>
- 1997
- 1998
- 1999

2000 A OPERATIONAL SEMANTICS PROOFS

2001 A.1 Properties of Memories

2002 PROOF OF PROPOSITION 5.3. By induction on the step. In ADV steps, we retain the simulation. In

2003 the remaining cases, we take the same RA step, which retains the simulation. In the STORE case,

2004 we store the unique corresponding message that is permissible according to the rule. Other than

2005 the timestamp, the view is determined by the current view tree. In the READONLY case, we load

2006 the corresponding message according to the bijection given by the weak simulation relation. The

2007 RMW case is a combination of both of the above. The other cases retains the simulation as they

2008 propagate the state by induction or without change. \square

2009

PROOF OF LEMMA 5.8. Let $\ell' \in \text{Loc}$. Then $\{\epsilon \in \mu_{\ell'} \mid \kappa \rightsquigarrow \epsilon\}.\text{vw} = \{\kappa\}$, and so

$$\kappa_{\ell'} = \left(\bigsqcup \{\epsilon \in \mu_{\ell'} \mid \kappa \rightsquigarrow \epsilon\}.\text{vw} \right)_{\ell'} \leq \left(\bigsqcup \{\epsilon \in \mu \mid \kappa \rightsquigarrow \epsilon\}.\text{vw} \right)_{\ell'}$$

Therefore, $\kappa \leq \bigsqcup \{\epsilon \in \mu \mid \kappa \rightsquigarrow \epsilon\}.\text{vw}$.

Conversely, since $\kappa \hookrightarrow \mu$, if $\kappa \rightsquigarrow \epsilon \in \mu$ then $\kappa \geq \epsilon.\text{vw}$. Thus $\kappa \geq \bigsqcup \{\epsilon \in \mu \mid \kappa \rightsquigarrow \epsilon\}.\text{vw}$. \square

PROOF OF PROPOSITION 5.9. Item 1 follows from the fact that the memory is downwards pointing in the case of a single-edge path. This extends to a path of any length by induction. The other items are direct consequences of the first. \square

PROOF OF PROPOSITION 5.11(1). Since μ is connected, there exists $\epsilon \in \mu_{\ell}$ such that $v \rightsquigarrow \epsilon$. By Proposition 5.9, $(\min \mu_{\ell}).\text{t} \geq \epsilon.\text{t}$. By minimality $(\min \mu_{\ell}).\text{t} = \epsilon.\text{t}$, and since μ is scattered, $\epsilon = \min \mu_{\ell}$. Thus v is on a cycle (with ϵ). Since μ is built-up, v is minimal. \square

PROOF OF PROPOSITION 5.11(2). Since μ is built-up, $\min \mu_{\ell}$ appears in a cycle in $\mu.\text{gph}$, and thus in a cycle of the supergraph $\rho.\text{gph}$. Since ρ is built-up, $\min \mu_{\ell}$ is minimal in ρ . \square

A.2 Properties of View Trees

Only leaves are well-formed for values, being a special case of a single-threaded program:

LEMMA A.1. *If $T \Vdash V$ then T is a leaf, i.e. $\exists \kappa \in \text{View}. T = \dot{\kappa}$.*

PROOF. The LEAF rule is the only rule in which the term in the conclusion can be a value. \square

PROOF OF PROPOSITION 5.13. Let ∇_1 and ∇_2 be derivations of $T \Vdash M$. Induction on ∇_1 shows that $\nabla_1 = \nabla_2$. In every case there is only one applicable derivation rule, and the premises are determined by the conclusion. To illustrate this point, consider the case of $T \widehat{R} \Vdash VN$. The application of λ -L is ruled-out since $T \widehat{R} \Vdash V$ contradicts Lemma A.1.

Since the derivation is unique, an mapping as claimed can be constructed by deconstructing the term following the derivation. The mapping is injective because the subterms and subtrees in NODE, the only rule with more than one premise, are disjoint. \square

A.3 Properties of the Operational Semantics

PROOF OF THEOREM 5.15. Preservation of the type is standard. The well-formedness of the state we show by induction on the step derivation.

The view-tree after the step is well-formed if it is a leaf by the LEAF rule. Otherwise, we use a corresponding well-formedness rule. For example, for the APPLLEFT case we use the λ -L rule and the induction hypothesis; for the PARINIT case we use the NODE rule, and satisfy the premises using the LEAF rule.

That the view-tree points downwards into the memory, and that the memory is well-formed, are both immediate from the induction hypothesis when there is one. The PARINIT case is also immediate since neither the memory nor the set of views changes. The memory also doesn't change in the PARFIN case, into which the view-leaf after the step points downward since it is the pointwise maximum of views that do.

For the cases of memory-accessing steps, we also use this fact that pointwise maximum preserves pointing downwards for those steps that load a message. The steps that add a message change the timestamp 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 succeed themselves. The memory remains well-formed after adding such a message since the condition on the segment ensures that

the memory remains scattered; the message is not minimal because the timestamp is chosen to increase; the message is not on a cycle because no other message points to it; and, as we have just established, its view points downwards, and the views of the other messages point to the same messages so they too point downwards as they have before the step. \square

PROOF OF LEMMA 5.16. By induction on the step derivation. The congruence cases are all immediate from the induction hypothesis. Of the others, only STORE and RMW add a message, in which cases the premises ensure the claim holds. \square

PROOF OF LEMMA 5.17. This property extends from a single step inductively. For a single step, we proceed by induction on the step derivation. The congruence cases and those that do not change the view-tree are all immediate from the induction hypothesis. In the memory-accessing steps, the claim follows from their premises. The cases that change the tree structure, PARINIT and PARFIN, are trivial to check. \square

PROOF OF PROPOSITION 5.18. That $\alpha \leq \omega$ follows from Lemma 5.17. The rest follows by induction on the number of steps. Indeed, combining our assumption with Lemma 5.16, when a message is added there exist $\alpha' \in T.lf$ and $\omega' \in R.lf$ such that, $\alpha \leq \alpha' \leq v.vw \leq \omega' \leq \omega'$ and $\alpha_{v.lc} \leq \alpha'_{v.lc} < v.t$. \square

B METATHEORY PROOFS

B.1 Proofs for Commutativity

The diagrams for the proof of Rewrite Commutativity are in Appendix D. Below are proofs of other claims from §7.1.

PROOF OF DEFERRAL OF CLOSURE. Since $(-)^*$ is a closure operator, it is monotonic, so the \supseteq containment follows from the monotonicity of $(\gg^{\mathcal{G}})$ and $(\|\|\mathcal{G})$ (Proposition 6.3). Moreover, for the \subseteq containment, suffice it we show that $P_1^* \gg^{\mathcal{G}} f^* \subseteq (P_1 \gg^{\mathcal{G}} f)^*$ and $P_1^* \|\|\mathcal{G} P_2^* \subseteq (P_1 \|\|\mathcal{G} P_2)^*$.

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 $n_1, n_2 \in \mathbb{N}$, $P_1^{n_1} \gg^{\mathcal{G}} f^{n_2} \subseteq (P_1 \gg^{\mathcal{G}} f)^*$ and $P_1^{n_1} \|\|\mathcal{G} P_2^{n_2} \subseteq (P_1 \|\|\mathcal{G} P_2)^*$. We show this by induction on $n_1 + n_2$, where the base case $P_1 \gg^{\mathcal{G}} f \subseteq (P_1 \gg^{\mathcal{G}} f)^*$ and $P_1 \|\|\mathcal{G} P_2 \subseteq (P_1 \|\|\mathcal{G} P_2)^*$ holds since $(-)^*$ 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 $(\|\|\mathcal{G})$, since we find that proving the claim for $(\gg^{\mathcal{G}})$ to be similar and somewhat easier.

Let $\tau \in P_1^{n_1} \|\|\mathcal{G} P_2^{n_2}$. So $\tau = \inf_{\xi.o} \{\alpha_1, \alpha_2\} [\xi] \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$ where $\tau_i := \alpha_i [\xi_i] \omega_i \cdot r_i \in P_i^{n_i}$ and $\xi \in \xi_1 \|\|\mathcal{G} \xi_2$. Assume w.l.o.g. that $n_1 = n'_1 + 1$. So there is some $\tau'_1 \in P_1^{n'_1}$ and $x \in \star$ such that $\tau'_1 \xrightarrow{x} \tau_1$. By case analysis on x , we show that there exists $\tau' \in P_1^{n'_1} \|\|\mathcal{G} P_2^{n_2}$ such that $\tau' \star$ -rewrites to τ . By the induction hypothesis $\tau' \in (P_1 \|\|\mathcal{G} P_2)^*$, and so $\tau \in (P_1 \|\|\mathcal{G} P_2)^*$.

For the $x \in \star \cap c$ cases, we construct τ' from τ'_1 and τ_2 . The procedure depends on x :

Rw. So $\tau'_1 = \alpha'_1 [\xi'_1] \omega_1 \cdot r_1$ where $\alpha_1 \leq \alpha'_1$. We take

$$\tau' := \inf_{\xi.o} \{\alpha'_1, \alpha_2\} [\xi] \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$$

Since $\inf_{\xi.o} \{\alpha_1, \alpha_2\} \leq \inf_{\xi.o} \{\alpha'_1, \alpha_2\}$, we have $\tau' \xrightarrow{\text{Rw}} \tau$.

2108 Fw. Similar to Rw.

2109 St. So $\tau'_1 = \alpha_1 \boxed{\eta_1 \eta'_1} \omega_1 \cdot r_1$ 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
 2110 $\xi = \eta \langle \mu, \mu \rangle \eta'$, where η includes the transitions from η_1 and η' includes the transitions
 2111 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
 2112 particular, $\eta \eta' \in \eta_1 \eta'_1 \parallel \eta_2 \eta'_2 = \eta_1 \eta'_1 \parallel \xi_2$. Denoting $\xi' := \eta \eta'$, we take

$$2113 \tau' := \inf_{\xi'.o} \{ \alpha_1, \alpha_2 \} \boxed{\xi'} \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$$

2115 We have $\xi.o \subseteq \xi'.o$, so $\inf_{\xi.o} \{ \alpha_1, \alpha_2 \} \leq \inf_{\xi'.o} \{ \alpha_1, \alpha_2 \}$. So $\tau' \xrightarrow{\text{Rw}} \tau''$, where

$$2116 \tau'' := \inf_{\xi.o} \{ \alpha_1, \alpha_2 \} \boxed{\xi} \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$$

2117 Since $\xi = \eta \langle \mu, \mu \rangle \eta'$, we have $\tau'' \xrightarrow{\text{St}} \tau$.

2118 Mu. So $\tau'_1 = \alpha_1 \boxed{\eta_1 \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta'_1} \omega_1 \cdot r_1$ where $\xi_1 = \eta_1 \langle \mu, \theta \rangle \eta'_1$. Since $\xi \in \xi_1 \parallel \xi_2$, there exist
 2119 η, η' such that $\xi = \eta \langle \mu, \theta \rangle \eta'$, where η includes the transitions from η_1 and η' includes the
 2120 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 $\eta' \in$
 2121 $\eta'_1 \parallel \eta'_2$. In particular, $\eta \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta' \in \eta_1 \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta'_1 \parallel \eta_2 \eta'_2 = \eta_1 \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta'_1 \parallel \xi_2$.
 2122 Denoting $\xi' := \eta \langle \mu, \rho \rangle \langle \rho, \theta \rangle \eta'$, we take

$$2123 \tau' := \inf_{\xi'.o} \{ \alpha_1, \alpha_2 \} \boxed{\xi'} \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$$

2124 Since $\xi = \eta \langle \mu, \theta \rangle \eta'$, and $\xi'.o = \xi.o$ and $\eta'.o = \eta.o$, we have $\tau' \xrightarrow{\text{Mu}} \tau$.

2125 For the $\times \in \star \cap \mathbf{a}$ cases, we construct τ' from τ'_1 and a τ'_2 defined such that $\tau_2 \xrightarrow{y} \tau'_2$ for some $y \in \mathfrak{g}$.
 2126 By iterating **Rewrite Commutativity** n_2 times to commute \xrightarrow{y} through the \star -rewrite sequence that
 2127 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 \times :

2128 Ti. So $\tau'_1 = \alpha_1 \boxed{\eta_1 \langle \mu, \rho \uplus \{v\} \rangle \eta'_1 \bar{\uplus} \{v\}} \omega_1 \cdot r_1$ where $\xi_1 = \eta_1 \langle \mu, \rho \uplus \{\epsilon\} \rangle \eta'_1 \bar{\uplus} \{\epsilon\}$ and $v \leq_{\text{vw}} \epsilon$. Since
 2129 $\xi \in \xi_1 \parallel \xi_2$, there are $\eta, \eta', \eta_2, \eta'_2$ such that $\xi = \eta \langle \mu, \rho \uplus \{\epsilon\} \rangle (\eta' \bar{\uplus} \{\epsilon\})$, $\xi_2 = \eta_2 (\eta'_2 \bar{\uplus} \{\epsilon\})$,
 2130 $\eta \in \eta_1 \parallel \eta_2$ and $\eta' \bar{\uplus} \{\epsilon\} \in \eta'_1 \bar{\uplus} \{\epsilon\} \parallel \eta'_2 \bar{\uplus} \{\epsilon\}$. Taking the same order of interleaving,
 2131 $\eta' \bar{\uplus} \{v\} \in \eta'_1 \bar{\uplus} \{v\} \parallel \eta'_2 \bar{\uplus} \{v\}$. Therefore, we have $\xi' \in \xi_1 \parallel \xi_2$, where

$$2132 \xi' := \eta \langle \mu, \rho \uplus \{v\} \rangle (\eta' \bar{\uplus} \{v\}), \quad \xi'_1 := \eta_1 \langle \mu, \rho \uplus \{v\} \rangle \eta'_1 \bar{\uplus} \{v\}, \quad \text{and} \quad \xi'_2 := \eta_2 (\eta'_2 \bar{\uplus} \{v\})$$

2133 Define $\tau'_2 := \alpha_2 \boxed{\xi'_2} \omega_2 \cdot r_2$. Since $\tau_2 \xrightarrow{\text{Ls}} \tau'_2$, indeed $\tau'_2 \in P_2^{n_2}$. We take

$$2134 \tau' := \inf_{\xi'.o} \{ \alpha_1, \alpha_2 \} \boxed{\xi'} \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$$

2135 Since $\xi.o = \xi'.o$, we have $\tau' \xrightarrow{\text{Ti}} \tau$.

2136 Ab. Similar to Ti, using $\tau_2 \xrightarrow{\text{Ex}} \tau'_2$.

2137 Di. So $\tau'_1 = \left(\alpha_1 \boxed{\eta_1 \langle \mu, \rho \uplus \{v\} \rangle \eta'_1 \bar{\uplus} \{v\}} \omega_1 \cdot r \right) [\uparrow \epsilon]$, where $\xi = \eta_1 \langle \mu, \rho \uplus \{v, \epsilon\} \rangle \eta'_1 \bar{\uplus} \{v, \epsilon\}$. The
 2138 reasoning in this case proceeds similarly, using $\tau_2 \xrightarrow{\text{Cn}} \tau'_2$ and interleaving τ'_1 with τ'_2 to take

$$2139 \tau' := \inf_{\xi'.o} \{ \alpha_1 [\uparrow \epsilon], \alpha_2 [\uparrow \epsilon] \} \boxed{\xi'} \omega_1 [\uparrow \epsilon] \sqcup \omega_2 [\uparrow \epsilon] \cdot \langle r_1, r_2 \rangle$$

2140 We have $\xi.o = \xi'.o$ again too. Moreover, ξ is the chronicle of a trace, and ϵ appears in it.
 2141 So no view that appears in the trace can point into the interior of ϵ 's segment. Otherwise,
 2142 since view must point to timestamps of messages, we would have a memory that is not
 2143 scattered.

2144 We show $\inf_{\xi.o} \{ \alpha_1, \alpha_2 \} [\uparrow \epsilon] \leq \inf_{\xi'.o} \{ \alpha_1 [\uparrow \epsilon], \alpha_2 [\uparrow \epsilon] \}$. Indeed, for $\kappa \hookrightarrow \xi.o$, assume $\kappa \leq$
 2145 α_i . Therefore, $\kappa [\uparrow \epsilon] \leq \alpha_i [\uparrow \epsilon]$, and so $\kappa [\uparrow \epsilon] \leq \inf_{\xi'.o} \{ \alpha_1 [\uparrow \epsilon], \alpha_2 [\uparrow \epsilon] \}$. Thus in particular
 2146 for $\kappa = \inf_{\xi.o} \{ \alpha_1, \alpha_2 \}$.

2157 By order-comparing $(\omega_i)_{\epsilon,1c}$ to $\epsilon.i$, one also finds that $(\omega_1 \sqcup \omega_2) [\uparrow\epsilon] = \omega_1 [\uparrow\epsilon] \sqcup \omega_2 [\uparrow\epsilon]$.
 2158 And so we obtain $\tau' \xrightarrow{\text{Rw, Di}} \tau$. □

2159
 2160 From here on we work to prove **Retroactive Closure** via logical relation. To compensate for
 2161 the rewrite closure being taken at different stages of higher-order constructions, we use a refined
 2162 notion of equality.

2163 *Egli-Milner lifting.* The *trace lifting* of a relation $\sim \subseteq X \times Y$ is a relation $\sim \subseteq \text{Trace}X \times \text{Trace}Y$
 2164 defined $\tau \sim \tau' := \tau.\text{st} = \tau'.\text{st} \wedge \tau.v1 \sim \tau'.v1$. This in turn lifts to the Egli-Milner relation
 2165 $\sim \subseteq \mathcal{P}(\text{Trace}X) \times \mathcal{P}(\text{Trace}Y)$ where $U \sim E := \forall \tau \in U \exists \tau' \in E. \tau \sim \tau' \wedge \forall \tau' \in E \exists \tau \in U. \tau \sim \tau'$.
 2166 We call this last relation the *EM-trace lifting* of the first relation. We use the same notation for the
 2167 relations because we will always be able to infer which relation is meant by the objects related.

2169 *Logical relation.* For every type A we define $\mathcal{V}^\dagger\{A\} \subseteq \llbracket A \rrbracket \times \llbracket A \rrbracket_{\mathcal{M}}$ and $C^\dagger\{A\} \subseteq \mathcal{A} \llbracket A \rrbracket \times$
 2170 $\mathcal{M} \llbracket A \rrbracket_{\mathcal{M}}$ by mutual recursion. The definition of $\mathcal{V}^\dagger\{A\}$ follows the standard “related-inputs to
 2171 related-outputs” mantra:

$$\begin{aligned} 2172 \mathcal{V}^\dagger\{A \rightarrow B\} &:= \{ \langle f, g \rangle \mid \forall \langle r, s \rangle \in \mathcal{V}^\dagger\{A\}. \langle fr, gs \rangle \in C^\dagger\{B\} \} \\ 2173 \mathcal{V}^\dagger\{(A_1 * \dots * A_n)\} &:= \{ \langle \langle r_1, \dots, r_n \rangle, \langle s_1, \dots, s_n \rangle \rangle \mid \forall i. \langle r_i, s_i \rangle \in \mathcal{V}^\dagger\{A_i\} \} \\ 2174 \mathcal{V}^\dagger\{\{t_1 \text{ of } A_1 \mid \dots \mid t_n \text{ of } A_n\}\} &:= \bigcup_i \{ \langle t_i r, t_i s \rangle \mid \langle r, s \rangle \in \mathcal{V}^\dagger\{A_i\} \} \end{aligned}$$

2175
 2176 The relation trivializes on ground types: $\mathcal{V}^\dagger\{G\}$ is equality. In particular for $V, W \in \cdot \vdash G$, if
 2177 $\langle \llbracket V \rrbracket^v, \llbracket W \rrbracket^v_{\mathcal{M}} \rangle \in \mathcal{V}^\dagger\{G\}$, then $V = W$ because as program values, $\llbracket V \rrbracket^v = V$ and $\llbracket W \rrbracket^v_{\mathcal{M}} = W$.

2178 The bespoke $C^\dagger\{A\} := \{ \langle P, Q \rangle \mid \langle P, Q^a \rangle \in \mathcal{V}^\dagger\{A\} \}$ uses the EM-trace lifting of $\mathcal{V}^\dagger\{A\}$ to relate
 2179 abstract denotations to generating denotations by nesting α -closures.

2180 In regards to open terms, for every typing context Γ we define $\mathcal{X}^\dagger\{\Gamma\} \subseteq \llbracket \Gamma \rrbracket \times \llbracket \Gamma \rrbracket_{\mathcal{M}}$ by:

$$2181 \mathcal{X}^\dagger\{\Gamma\} := \{ \langle \gamma, \delta \rangle \mid \forall (a : A) \in \Gamma. \langle \gamma a, \delta a \rangle \in \mathcal{V}^\dagger\{A\} \}$$

2182 and define $\Gamma \vDash^\dagger M : A$ as follows: $\forall \langle \gamma, \delta \rangle \in \mathcal{X}^\dagger\{\Gamma\}. \langle \llbracket M \rrbracket^c \gamma, \llbracket M \rrbracket^c_{\mathcal{M}} \delta \rangle \in C^\dagger\{A\}$. We show this
 2183 semantic judgment is sound with respect to the typing relation, following some supportive lemmas.

2184 **LEMMA B.1.** *If $\langle r, s \rangle \in \mathcal{V}^\dagger\{A\}$ then $\langle \text{return } r, \text{return}^{\mathcal{M}} s \rangle \in C^\dagger\{A\}$.*

2185 **PROOF.** For the first half of the EM-trace lifting, let $\tau \in \text{return } r = (\text{return}^{\mathcal{M}} r)^a$, where we
 2186 used **Rewrite Commutativity** to reorder the rewrites. So there exists $\pi \in \text{return}^{\mathcal{M}} r$ such that
 2187 $\pi \xrightarrow{a} \tau$. Obtain τ', π' from τ, π respectively by replacing their return value r with s . By construction,
 2188 $\langle \tau, \tau' \rangle \in \mathcal{V}^\dagger\{A\}$. Moreover, $\pi' \in \text{return}^{\mathcal{M}} s$. By reusing the rewrite sequence, $\pi' \xrightarrow{a} \tau'$. Therefore,
 2189 $\tau' \in (\text{return}^{\mathcal{M}} s)^a$ is a witness as required.

2190 The same idea in reverse shows the second half of the EM-trace lifting. □

2191 **LEMMA B.2.** *If $\langle P, Q \rangle \in C^\dagger\{A\}$ and $\langle f, g \rangle \in \mathcal{V}^\dagger\{A \rightarrow B\}$ then $\langle P \gg^{\mathcal{M}} f, Q \gg^{\mathcal{M}} g \rangle \in C^\dagger\{B\}$.*

2192 **PROOF.** For the first half of the EM-trace lifting, let $\tau \in P \gg^{\mathcal{M}} f = (P \gg^{\mathcal{M}} f)^a$, where we used
 2193 **Lemma 7.1** to reorder the rewrites. So there exists $\pi \in P \gg^{\mathcal{M}} f$ such that $\pi \xrightarrow{a} \tau$. So there exist
 2194 $\alpha \llbracket \xi \rrbracket \kappa \cdot r \in P$ and $\sigma \llbracket \eta \rrbracket \omega \cdot s \in fr$ where $\kappa \leq \sigma$ such that $\pi = \alpha \llbracket \xi \rrbracket \eta \omega \cdot s$.

- 2195 • By the first assumption, there exists r' such that $\langle r, r' \rangle \in \mathcal{V}^\dagger\{A\}$ and $\alpha \llbracket \xi \rrbracket \kappa \cdot r' \in Q^a$.
- 2196 • By the second assumption, there exists s' such that $\langle s, s' \rangle \in \mathcal{V}^\dagger\{B\}$ and $\sigma \llbracket \eta \rrbracket \omega \cdot s' \in (gr')^a$.

2200

So $\pi' := \alpha \left[\frac{\xi}{\eta} \right] \omega . \cdot . s' \in Q^a \gg^M g^a$. Obtain τ' from τ by replacing its return value s by s' . By reusing the rewrite sequence, $\pi' \xrightarrow{a} \tau'$. By **Deferral of Closure**, $\tau' \in \left(Q^a \gg^M g^a \right)^a = \left(Q \gg^M g \right)^a$.

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

LEMMA B.3. $\langle \llbracket \text{store}_{\ell, v} \rrbracket, \llbracket \text{store}_{\ell, v} \rrbracket_M \rangle \in C^\dagger \{1\}$ and $\langle \llbracket \text{rmw}_{\ell, \varphi} \rrbracket, \llbracket \text{rmw}_{\ell, \varphi} \rrbracket_M \rangle \in C^\dagger \{\text{Val}\}$.

PROOF. Since **1** and **Val** are ground types, the sets are equal by **Deferral of Closure**, reasoning as in **Lemma B.1**. \square

LEMMA B.4. If $\langle P_i, Q_i \rangle \in C^\dagger \{A_i\}$ then $\langle P_1 \parallel P_2, Q_1 \parallel Q_2 \rangle \in C^\dagger \{(A_1 * A_2)\}$.

PROOF. Similar to **Lemma B.2**. \square

PROPOSITION B.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:

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

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, $\langle \llbracket M \rrbracket^c \gamma [a \mapsto r], \llbracket M \rrbracket_M^c \delta [a \mapsto s] \rangle \in C^\dagger \{B\}$. Therefore, $\langle \lambda r. \llbracket M \rrbracket^c \gamma [a \mapsto r], \lambda s. \llbracket M \rrbracket_M^c \delta [a \mapsto s] \rangle \in \mathcal{V}^\dagger \{A \rightarrow B\}$. Applying **Lemma B.1**, $\langle \llbracket \lambda a. M \rrbracket^c \gamma, \llbracket \lambda a. M \rrbracket_M^c \delta \rangle \in C^\dagger \{A \rightarrow B\}$.

$$\frac{\Gamma \vDash^\dagger M : A \quad \Gamma \vDash^\dagger N : A \rightarrow B}{\Gamma \vDash^\dagger NM : B}$$

Let $\langle \gamma, \delta \rangle \in \mathcal{X}^\dagger \{\Gamma\}$. If $\langle f, g \rangle \in \mathcal{V}^\dagger \{A \rightarrow B\}$, then by **Lemma B.2** with the first assumption, $\langle \llbracket M \rrbracket^c \gamma \gg f, \llbracket M \rrbracket_M^c \delta \gg^M g \rangle \in C^\dagger \{B\}$.

Thus $\langle \lambda f. \llbracket M \rrbracket^c \gamma \gg f, \lambda g. \llbracket M \rrbracket_M^c \delta \gg^M g \rangle \in \mathcal{V}^\dagger \{(A \rightarrow B) \rightarrow B\}$.

So by **Lemma B.2** with the second assumption, $\langle \llbracket NM \rrbracket^c \gamma, \llbracket NM \rrbracket_M^c \delta \rangle \in C^\dagger \{B\}$.

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

PROOF OF **RETROACTIVE CLOSURE**. Since M is a program, by **Proposition B.5**, $\cdot \vDash M : G$ for some ground type G . That is, $\langle \llbracket M \rrbracket^c, \llbracket M \rrbracket_M^c \rangle \in C^\dagger \{A\}$. Since the EM-trace lifting degenerates to equality on ground types, $\llbracket M \rrbracket^c = \llbracket M \rrbracket_M^c$. \square

B.2 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 $\sim \subseteq \text{Trace}X \times \text{Trace}Y$ defined $\tau \sim \tau' := \tau.\text{st} = \tau'.\text{st} \wedge \tau.\text{v1} \sim \tau'.\text{v1}$. This in turn lifts to the Hoare relation $\sim \subseteq \mathcal{P}(\text{Trace}X) \times \mathcal{P}(\text{Trace}Y)$ where $U \sim E := \forall \tau \in U \exists \tau' \in E. \tau \sim \tau'$. We call this last relation the *H-trace lifting* of the first relation.

Logical relation. For every type A we define $\mathcal{V}^\circ \{A\} \subseteq \llbracket A \rrbracket \times \llbracket A \rrbracket$ and $C^\circ \{A\} \subseteq \mathcal{A} \llbracket A \rrbracket \times \mathcal{A} \llbracket A \rrbracket$ by mutual recursion. The definition of $\mathcal{V}^\circ \{A\}$ follows the standard “related-inputs to related-outputs” mantra:

$$\mathcal{V}^\circ \{A \rightarrow B\} := \{ \langle f, g \rangle \mid \forall \langle r, s \rangle \in \mathcal{V}^\circ \{A\}. \langle fr, gs \rangle \in C^\circ \{B\} \}$$

$$\mathcal{V}^\circ \{(A_1 * \dots * A_n)\} := \{ \langle \langle r_1, \dots, r_n \rangle, \langle s_1, \dots, s_n \rangle \rangle \mid \forall i. \langle r_i, s_i \rangle \in \mathcal{V}^\circ \{A_i\} \}$$

$$\mathcal{V}^\circ \{ \{ \iota_1 \text{ of } A_1 \mid \dots \mid \iota_n \text{ of } A_n \} \} := \bigcup_i \{ \langle \iota_i r, \iota_i s \rangle \mid \langle r, s \rangle \in \mathcal{V}^\circ \{A_i\} \}$$

The relation trivializes on ground types: $\mathcal{V}^\circ \{G\}$ is equality. In particular for $V, W \in \cdot \vdash G$, if $\langle \llbracket V \rrbracket^v, \llbracket W \rrbracket^v \rangle \in \mathcal{V}^\circ \{G\}$, then $V = W$ because as program values, $\llbracket V \rrbracket^v = V$ and $\llbracket W \rrbracket^v = W$. We

H-trace lift $\mathcal{V}^\circ\{A\}$ to obtain $C^\circ\{A\}$. It too trivializes on ground types: $C^\circ\{G\}$ is containment. In regards to open terms, for every typing context Γ we define $X^\circ\{\Gamma\} \subseteq [\Gamma] \times [\Gamma]$ by:

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

and define $\Gamma \vDash M \lesssim N : A$ as follows: $\forall \langle \gamma, \delta \rangle \in X^\circ\{\Gamma\}. \langle \llbracket M \rrbracket^c \gamma, \llbracket N \rrbracket^c \delta \rangle \in C^\circ\{A\}$.

As in [Appendix B.1](#), we have the same supportive lemmas for this logical relation. The proofs are similar, though slightly simpler because there is no need for [Lemma 7.2](#).

LEMMA B.6. *If $\langle r, s \rangle \in \mathcal{V}^\circ\{A\}$ then $\langle \text{return } r, \text{return } s \rangle \in C^\circ\{A\}$.*

LEMMA B.7. *If $\langle P, Q \rangle \in C^\circ\{A\}$ and $\langle f, g \rangle \in \mathcal{V}^\circ\{A \rightarrow B\}$ then $\langle P \gg f, Q \gg g \rangle \in C^\circ\{B\}$.*

LEMMA B.8. *$\langle \llbracket \text{store}_{\ell, v} \rrbracket, \llbracket \text{store}_{\ell, v} \rrbracket \rangle \in C^\circ\{1\}$ and $\langle \llbracket \text{rmw}_{\ell, \varphi} \rrbracket, \llbracket \text{rmw}_{\ell, \varphi} \rrbracket \rangle \in C^\circ\{\text{Val}\}$.*

LEMMA B.9. *If $\langle P_i, Q_i \rangle \in C^\circ\{A_i\}$ then $\langle P_1 \parallel P_2, Q_1 \parallel Q_2 \rangle \in C^\circ\{(A_1 * A_2)\}$.*

The judgment is closed under term contexts:

LEMMA B.10. *For $\Delta \vdash \Xi [\Gamma \vdash - : A] : B$, if $\Gamma \vDash M \lesssim N : A$, then $\Delta \vDash \Xi [M] \lesssim \Xi [N] : B$.*

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.6 to B.9](#) as in the proof of [Proposition B.5](#). \square

PROPOSITION B.11. *If $\mathcal{A} [A] \ni P' \subseteq P$ and $\langle P, Q \rangle \in C^\circ\{A\}$ then $\langle P', Q \rangle \in C^\circ\{A\}$.*

PROOF. Assuming a statement about all elements of P we deduce the same statement about all elements of P' . \square

LEMMA B.12. *For $M, N \in \Gamma \vdash A$, if $\llbracket M \rrbracket^c \subseteq \llbracket N \rrbracket^c$ then $\Gamma \vDash M \lesssim N : A$.*

PROOF. Let $\langle \gamma, \delta \rangle \in X^\circ\{\Gamma\}$. By [Lemma B.10](#) with N itself as the context (the degenerate case with no metavariable appearance), $\langle \llbracket N \rrbracket^c \gamma, \llbracket N \rrbracket^c \delta \rangle \in C^\circ\{A\}$. By assumption, $\llbracket M \rrbracket^c \gamma \subseteq \llbracket N \rrbracket^c \gamma$. So by [Proposition B.11](#), $\langle \llbracket M \rrbracket^c \gamma, \llbracket N \rrbracket^c \delta \rangle \in C^\circ\{A\}$. \square

PROOF OF [THEOREM 7.7](#). By [Lemma B.12](#), $\Gamma \vDash M \lesssim N : A$. By [Lemma B.10](#), $\cdot \vDash \Xi [M] \lesssim \Xi [N] : G$. That is, $\langle \llbracket \Xi [M] \rrbracket^c, \llbracket \Xi [N] \rrbracket^c \rangle \in C^\circ\{G\}$. Since G is ground, this degenerates to $\llbracket \Xi [M] \rrbracket^c \subseteq \llbracket \Xi [N] \rrbracket^c$. \square

B.3 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 $\text{VTree} \times \text{Mem} \times \text{Chro} \times \text{View} \times X$, written $\tau = \langle T, \mu \rangle \boxed{\xi} \omega \therefore r$, where:

- The run-trace's chronicle is $\tau.\text{ch} = \xi$. Each transition consists of well-formed memories, and 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 program even starts running, so in the loud case we only require $\mu \subseteq \xi.\circ$.

- 2304 • The run-trace's final view is $\tau.\text{fvw} = \omega$. The corresponding interrupted execution ends with
- 2305 $\dot{\omega}$. In the silent case we require $T = \dot{\omega}$ since silent steps do not change the state.
- 2306 • The run-trace's final state is $\langle \dot{\omega}, (\langle \mu, \mu \rangle \xi).c \rangle$, so we require $\omega \hookrightarrow \langle \mu, \mu \rangle \xi.c$. In light of
- 2307 [Lemma 5.17](#), we require moreover that $\kappa \leq \omega$ for every $\kappa \in T$. If, denote by $T \leq \omega$.
- 2308 • Considering [Lemma 5.16](#), we require $\forall v \in \xi.\text{own} \exists \alpha \in T. \text{lf}. \alpha \leq v.\text{vw} \leq \omega \wedge \alpha_{v.\text{lc}} < v.\text{t}$.
- 2309 • The run-trace's return value is $\tau.\text{ret} = r$. This corresponds to the program value the inter-
- 2310 rupted execution returns.

2311 We denote the set of X -run-traces by $\text{OpTrace}X$.

2312 We define a monad structure $\mathcal{R}X := \langle \underline{\mathcal{R}}X, \text{return}^{\mathcal{R}}, \gg^{\mathcal{R}} \rangle$:

$$2314 \quad \underline{\mathcal{R}}X := \mathcal{P}(\text{OpTrace}X) \quad \text{return}^{\mathcal{R}}r := \{ \langle \dot{\kappa}, \mu \rangle \sqsupset \kappa \cdot r \}$$

$$2316 \quad P \gg^{\mathcal{R}} f := \{ \langle T, \mu \rangle \boxed{\xi\eta} \omega \cdot s \in \underline{\mathcal{R}}Y \mid \exists r, \kappa. \langle T, \mu \rangle \boxed{\xi} \kappa \cdot r \in P \wedge \langle \dot{\kappa}, (\langle \mu, \mu \rangle \xi).c \rangle \boxed{\eta} \omega \cdot s \in fr \}$$

2317 In the return operator, we make sure that the initial and final states are equal. In the bind operator,

2318 we make sure that the final state of the first run-trace is the initial state of the second run-trace.

2320 **PROPOSITION B.13.** \mathcal{R} is a monad.

2321 Next we extend \mathcal{R} with shared-memory constructs.

2323 *Concurrent execution.* Consider a program $M \parallel N$. Either the state has a leaf $\dot{\kappa}$ as its view-tree,

2324 in which case the first step it takes has to be `PARINIT`, or it has a node as its view tree $T \widehat{\text{---}} R$, in

2325 which case the first step it takes cannot be `PARINIT`. Either way, it then takes some steps due to

2326 steps of M and N (with `PARLEFT` and `PARRIGHT`), then finally it steps with `PARFIN` to synchronize.

$$2328 \quad P_1 \parallel \parallel^{\mathcal{R}} P_2 := \left\{ \langle T, \mu \rangle \boxed{\xi} \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle \in \underline{\mathcal{R}}(X_1 \times X_2) \mid \exists \xi_1, \xi_2. \xi \in \xi_1 \parallel \xi_2 \wedge \exists T_1, T_2. \right. \\ \left. \left(\forall i \in \{1, 2\}. \langle T_i, \mu \rangle \boxed{\xi_i} \omega_i \cdot r_i \in P_i \right) \wedge \left(T = T_1 \widehat{\text{---}} T_2 \vee \exists \kappa. T = T_1 = T_2 = \dot{\kappa} \right) \right\}$$

2331 *Memory access.* The definitions follow the `STORE`, `READONLY`, and `RMW` rules:

$$2332 \quad \llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{R}} := \{ \langle \dot{\alpha}, \mu \rangle \boxed{\rho, \rho \uplus \{ \ell: v @ (q, t) \langle \langle \alpha \rangle [\ell \mapsto t] \rangle \}} \alpha [\ell \mapsto t] \cdot \langle \rangle \in \underline{\mathcal{R}}1 \}$$

$$2334 \quad \llbracket \text{rmw}_{\ell, \Phi}^{\text{RO}} \rrbracket_{\mathcal{R}} := \{ \langle \dot{\alpha}, \mu \rangle \boxed{\rho, \rho} \alpha \sqcup \kappa \cdot v \in \underline{\mathcal{R}}\text{Val} \mid \Phi v = \perp, \ell: v @ (-, \kappa_{\ell}) \langle \langle \kappa \rangle \rangle \in \rho, \alpha_{\ell} \leq \kappa_{\ell} \}$$

$$2336 \quad \llbracket \text{rmw}_{\ell, \Phi}^{\text{RMW}} \rrbracket_{\mathcal{R}} := \left\{ \langle \dot{\alpha}, \mu \rangle \boxed{\rho, \rho \uplus \{ \ell: \Phi v @ (\kappa_{\ell}, t) \langle \langle \omega \rangle \rangle \}} \omega \cdot v \in \underline{\mathcal{R}}\text{Val} \right\} \\ \left\{ \mid \omega = (\alpha \sqcup \kappa) [\ell \mapsto t], \ell: v @ (-, t) \langle \langle \kappa \rangle \rangle \in \rho \right\}$$

$$2338 \quad \llbracket \text{rmw}_{\ell, \Phi} \rrbracket_{\mathcal{R}} := \llbracket \text{rmw}_{\ell, \Phi}^{\text{RO}} \rrbracket_{\mathcal{R}} \cup \llbracket \text{rmw}_{\ell, \Phi}^{\text{RMW}} \rrbracket_{\mathcal{R}}$$

2339 Some of the premises of the corresponding rules appear as conditions in the set notations, while

2340 other do not appear because they hold implicitly due to the requirements on run-traces.

2341 The importance of a run-trace's initial memory is in making sense of the initial view-tree, even

2342 if the chronicle is empty. In particular, messages unseen by the initial view-tree are redundant:

2344 **LEMMA B.14.** If $\langle T, \mu' \rangle \boxed{\xi} \omega \cdot r \in \llbracket M \rrbracket_{\mathcal{R}}^c$ and $T \hookrightarrow \mu \subseteq \mu'$ then $\langle T, \mu \rangle \boxed{\xi} \omega \cdot r \in \llbracket M \rrbracket_{\mathcal{R}}^c$.

2346 *Single-step soundness.* To make the relationship between these denotations and the operational

2347 semantics precise, we can follow an execution backwards, adding a transition for every \bullet -step:

2349 **LEMMA B.15.** Assume $\langle T, \mu \rangle, M \xrightarrow{e}_{\text{RA}} \langle T', \mu' \rangle, M'$ and $\langle T', \mu' \rangle \boxed{\xi} \omega \cdot r \in \llbracket M' \rrbracket_{\mathcal{R}}^c$.

- 2350 • If $e = \circ$, then $\langle T, \mu \rangle \boxed{\xi} \omega \cdot r \in \llbracket M \rrbracket_{\mathcal{R}}^c$.
- 2351 • If $e = \bullet$, then $\langle T, \mu \rangle \boxed{\langle \mu, \mu' \rangle \xi} \omega \cdot r \in \llbracket M \rrbracket_{\mathcal{R}}^c$.

2353 **PROOF.** By induction on the derivation of $\langle T, \mu \rangle, M \overset{e}{\rightsquigarrow}_{RA} \langle T', \mu' \rangle, M'$. Paradigmatic examples
 2354 follow:

2355 **APP** Assume $\langle \dot{\kappa}, \mu \rangle, (\lambda a. M) V \overset{\circ}{\rightsquigarrow}_{RA} \langle \dot{\kappa}, \mu \rangle, M[a \mapsto V]$ and $\tau := \langle \dot{\kappa}, \mu \rangle \boxed{\xi} \omega \cdot r \in \llbracket M[a \mapsto V] \rrbracket_{\mathcal{R}}^c$.
 2356 By the **Substitution Lemma**, $\llbracket M[a \mapsto V] \rrbracket_{\mathcal{R}}^c = \llbracket (\lambda a. M) V \rrbracket_{\mathcal{R}}^c$. So indeed $\tau \in \llbracket (\lambda a. M) V \rrbracket_{\mathcal{R}}^c$.

2357 **PARLEFT** Assume $\langle T \widehat{\ } R, \mu \rangle, M \parallel N \overset{\bullet}{\rightsquigarrow}_{RA} \langle T' \widehat{\ } R, \mu' \rangle, M' \parallel N$ and $\langle T' \widehat{\ } R, \mu' \rangle \boxed{\xi} \omega \cdot \langle r, s \rangle \in$
 2358 $\llbracket M' \parallel N \rrbracket_{\mathcal{R}}^c$. So there exist ξ_1, ξ_2 such that $\xi \in \xi_1 \parallel \xi_2$, and there exist ω_1, ω_2 where
 2359 $\omega = \omega_1 \sqcup \omega_2$ such that $\langle T', \mu' \rangle \boxed{\xi_1} \omega_1 \cdot r \in \llbracket M' \rrbracket_{\mathcal{R}}^c$ and $\langle R, \mu' \rangle \boxed{\xi_2} \omega_2 \cdot s \in \llbracket N \rrbracket_{\mathcal{R}}^c$. In
 2360 that latter we can replace μ' with μ using **Lemma B.14**. By the induction hypothesis and
 2361 the former, $\langle T, \mu \rangle \boxed{\langle \mu, \mu' \rangle \xi_1} \omega_1 \cdot r \in \llbracket M \rrbracket_{\mathcal{R}}^c$. Since $\langle \mu, \mu' \rangle \xi \in \langle \mu, \mu' \rangle \xi_1 \parallel \xi_2$, we have
 2362 $\langle T \widehat{\ } R, \mu \rangle \boxed{\langle \mu, \mu' \rangle \xi} \omega \cdot \langle r, s \rangle \in \llbracket M \parallel N \rrbracket_{\mathcal{R}}^c$. \square

2363 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
 2364 feature gapless chronicles can be rewritten using Mu to obtain single-transition traces.

2365 **PROPOSITION B.16.** *If $\langle T, \mu \rangle, M \rightsquigarrow_{RA}^* \langle \dot{\omega}, \mu' \rangle, V$, then $\langle T, \mu \rangle \boxed{\xi} \omega \cdot \llbracket V \rrbracket_{\mathcal{R}}^v \in \llbracket M \rrbracket_{\mathcal{R}}^c$, where $\eta =$
 2366 $\langle \mu, \mu \rangle \xi$ is gapless and $\eta.c = \mu'$, i.e. either: (i) ξ is empty and $\mu = \mu'$; or (ii) $\xi.o = \mu$, $\xi.c = \mu'$, and ξ
 2367 is gapless.*

2368 **PROOF.** By induction on the number of small-steps. Case (i) applies so long as all the steps so
 2369 far are silent. Case (ii) applies otherwise. \square

2370 **Hoare run-lifting.** The *run-trace lifting* of a relation $\sim \subseteq X \times Y$ is a relation $\sim \subseteq \text{OpTrace}X \times$
 2371 $\text{Trace}Y$ defined $\tau \sim \tau' := \exists T, \mu, \xi, \omega, r, s. \tau = \langle T, \mu \rangle \boxed{\xi} \omega \cdot r \wedge \tau' = \inf_{\mu} T \boxed{\langle \mu, \mu \rangle \xi} \omega \cdot s \wedge r \sim s$. This
 2372 in turn lifts to the Hoare relation $\sim \subseteq \mathcal{P}(\text{OpTrace}X) \times \mathcal{P}(\text{Trace}Y)$ where $U \sim E := \forall \tau \in U \exists \tau' \in$
 2373 $E. \tau \sim \tau'$. We call this last relation the *H-run-trace lifting* of the first relation.

2374 **Logical relation.** For every type A we define $\mathcal{V}^*\{A\} \subseteq \llbracket A \rrbracket_{\mathcal{R}} \times \llbracket A \rrbracket$ and $C^*\{A\} \subseteq \mathcal{R}\llbracket A \rrbracket_{\mathcal{R}} \times \mathcal{A}\llbracket A \rrbracket$
 2375 by mutual recursion. The definition of $\mathcal{V}^*\{A\}$ follows the standard “related-inputs to related-out-
 2376 puts” mantra, while the bespoke $C^*\{A\}$ part transforms the view tree to its greatest lower bound
 2377 using the notation $\inf_{\mu} T := \inf_{\mu} T.lf$, and adds a transition for the first memory:

$$\mathcal{V}^*\{A \rightarrow B\} := \{ \langle f, g \rangle \mid \forall \langle r, s \rangle \in \mathcal{V}^*\{A\}. \langle fr, gs \rangle \in C^*\{B\} \}$$

$$\mathcal{V}^*\{(A_1 * \dots * A_n)\} := \{ \langle \langle r_1, \dots, r_n \rangle, \langle s_1, \dots, s_n \rangle \rangle \mid \forall i. \langle r_i, s_i \rangle \in \mathcal{V}^*\{A_i\} \}$$

$$\mathcal{V}^*\{\{t_1 \text{ of } A_1 \mid \dots \mid t_n \text{ of } A_n\}\} := \bigcup_i \{ \langle t_i r, t_i s \rangle \mid \langle r, s \rangle \in \mathcal{V}^*\{A_i\} \}$$

2378 The relation trivializes on ground types: $\mathcal{V}^*\{G\}$ is equality. In particular for $V, W \in \cdot \vdash G$, if
 2379 $\langle V, W \rangle \in \mathcal{V}^*\{G\}$, then $V = W$ because as ground-typed values, $\llbracket V \rrbracket_{\mathcal{R}}^v = V$ and $\llbracket W \rrbracket^v = W$. We H-
 2380 trace lift $\mathcal{V}^*\{A\}$ to obtain $C^*\{A\}$.

2381 In regards to open terms, for every typing context Γ we define $\mathcal{X}^*\{\Gamma\} \subseteq \llbracket \Gamma \rrbracket_{\mathcal{R}} \times \llbracket \Gamma \rrbracket$ by:

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

2382 and define $\Gamma \vDash^* M : A$ as follows: $\forall \langle \gamma, \delta \rangle \in \mathcal{X}^*\{\Gamma\}. \langle \llbracket M \rrbracket_{\mathcal{R}}^c \gamma, \llbracket M \rrbracket^c \delta \rangle \in C^*\{A\}$. We show this
 2383 semantic judgment is sound with respect to the typing relation, following some supportive lemmas.

2384 **LEMMA B.17.** *If $\langle r, s \rangle \in \mathcal{V}^*\{A\}$ then $\langle \text{return}^{\mathcal{R}} r, \text{return } s \rangle \in C^*\{A\}$.*

2385 **PROOF.** Assume $\langle r, s \rangle \in \mathcal{V}^*\{A\}$. W.l.o.g., let $\langle \dot{\kappa}, \mu \rangle \sqcap \kappa \cdot r \in \text{return}^{\mathcal{R}} r$, where $\kappa \hookrightarrow \mu$. Note that
 2386 $\kappa \boxed{\langle \mu, \mu \rangle} \kappa \cdot s \in \text{return } s$. Trivially, $\langle \mu, \mu \rangle \cdot = \langle \mu, \mu \rangle$ and $\inf_{\mu} \dot{\kappa} = \kappa$. Substituting these, together with
 2387 our assumption, we obtain the required precisely:

$$\forall \langle \dot{\kappa}, \mu \rangle \sqcap \kappa \cdot r \in \text{return}^{\mathcal{R}} r \exists s. \inf_{\mu} \dot{\kappa} \boxed{\langle \mu, \mu \rangle} \kappa \cdot s \in \text{return } s \wedge \langle r, s \rangle \in \mathcal{V}^*\{A\} \quad \square$$

LEMMA B.18. *If $\langle P, Q \rangle \in C^*\{A\}$ and $\langle f, g \rangle \in \mathcal{V}^*\{A \rightarrow B\}$ then $\langle P \gg^{\mathcal{R}} f, Q \gg^{\mathcal{R}} g \rangle \in C^*\{B\}$.*

PROOF. Assume $\langle P, Q \rangle \in C^*\{A\}$ and $\langle f, g \rangle \in \mathcal{V}^*\{A \rightarrow B\}$. Let $\langle T, \mu \rangle \boxed{\xi \eta} \omega \cdot r' \in P \gg^{\mathcal{R}} f$. So there exist r and κ such that $\langle T, \mu \rangle \boxed{\xi} \kappa \cdot r \in P$ and $\langle \dot{\kappa}, (\langle \mu, \mu \rangle \xi).c \rangle \boxed{\eta} \omega \cdot r' \in fr$.

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

$$\inf_{\mu} T \boxed{\langle \mu, \mu \rangle \xi \langle \langle \langle \mu, \mu \rangle \xi \rangle.c, (\langle \mu, \mu \rangle \xi).c \rangle \eta} \omega \cdot s' \in Q \gg^{\mathcal{R}} g$$

By using Mu, we have the required $\inf_{\mu} T \boxed{\langle \mu, \mu \rangle \xi \eta} \omega \cdot s' \in Q \gg^{\mathcal{R}} g$. \square

LEMMA B.19. $\langle \llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{R}}, \llbracket \text{store}_{\ell, v} \rrbracket \rangle \in C^*\{1\}$ and $\langle \llbracket \text{rmw}_{\ell, \varphi} \rrbracket_{\mathcal{R}}, \llbracket \text{rmw}_{\ell, \varphi} \rrbracket \rangle \in C^*\{\text{Val}\}$.

PROOF. Using St to compensate for the additional transition from the initial memory, and Rw to compensate for the view not necessarily already pointing to the loaded message. \square

LEMMA B.20. *If $\langle P_i, Q_i \rangle \in C^*\{A_i\}$ then $\langle P_1 \parallel P_2, Q_1 \parallel Q_2 \rangle \in C^*\{(A_1 * A_2)\}$.*

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

Leaf. W.l.o.g., $\tau = \langle \dot{\kappa}, \mu \rangle \boxed{\xi} \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$, where $\langle \dot{\kappa}, \mu \rangle \boxed{\xi_i} \omega_i \cdot r_i \in P_i$ and $\xi \in \xi_1 \parallel \xi_2$.

So there exist s_i such that $\kappa \boxed{\langle \mu, \mu \rangle \xi_i} \omega_i \cdot s_i \in Q_i$ and $\langle r_i, s_i \rangle \in \mathcal{V}^*\{A_i\}$. By definition, $\kappa \boxed{\langle \mu, \mu \rangle \langle \mu, \mu \rangle \xi} \omega_1 \sqcup \omega_2 \cdot \langle s_1, s_2 \rangle \in Q_1 \parallel Q_2$. Using Mu we have $\kappa \boxed{\langle \mu, \mu \rangle \xi} \omega_1 \sqcup \omega_2 \cdot \langle s_1, s_2 \rangle \in Q_1 \parallel Q_2$. 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., $\tau = \langle T_1 \widehat{T_2}, \mu \rangle \boxed{\xi} \omega_1 \sqcup \omega_2 \cdot \langle r_1, r_2 \rangle$, where $\langle T_i, \mu \rangle \boxed{\xi_i} \omega_i \cdot r_i \in P_i$ and $\xi \in \xi_1 \parallel \xi_2$.

So there exist s_i such that $\inf_{\mu} T_i \boxed{\langle \mu, \mu \rangle \xi_i} \omega_i \cdot s_i \in Q_i$ and $\langle r_i, s_i \rangle \in \mathcal{V}^*\{A_i\}$. Rudimentarily, $\inf_{\mu} \{ \inf_{\mu} T_1, \inf_{\mu} T_2 \} = \inf_{\mu} (T_1 \widehat{T_2})$, so $\inf_{\mu} (T_1 \widehat{T_2}) \boxed{\langle \mu, \mu \rangle \langle \mu, \mu \rangle \xi} \omega_1 \sqcup \omega_2 \cdot \langle s_1, s_2 \rangle \in Q_1 \parallel Q_2$. The rest is like before. \square

PROPOSITION B.21. *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:

$\Gamma, a : A \vDash^* M : B$	Let $\langle \gamma, \delta \rangle \in \mathcal{X}^*\{\Gamma\}$. We have $\llbracket \lambda a. M \rrbracket_{\mathcal{R}}^c \gamma = \text{return}^{\mathcal{R}} \lambda r. \llbracket M \rrbracket_{\mathcal{R}}^c \gamma [a \mapsto r]$ and $\llbracket \lambda a. M \rrbracket^c \delta = \text{return} \lambda s. \llbracket M \rrbracket^c \delta [a \mapsto s]$ by definition. By Lemma B.17 and the definition of $\mathcal{V}^*\{A \rightarrow B\}$, it suffices to show that if $\langle r, s \rangle \in \mathcal{V}^*\{A\}$, then $\langle \llbracket M \rrbracket_{\mathcal{R}}^c \gamma [a \mapsto r], \llbracket M \rrbracket^c \delta [a \mapsto s] \rangle \in C^*\{B\}$, which is implied by the induction hypothesis.
$\Gamma \vDash^* \lambda a : A. M : A \rightarrow B$	

$\Gamma \vDash^* M : A \quad \Gamma \vDash^* N : A \rightarrow B$	Let $\langle \gamma, \delta \rangle \in \mathcal{X}^*\{\Gamma\}$. By definition, $\llbracket NM \rrbracket_{\mathcal{R}}^c \gamma = \llbracket N \rrbracket_{\mathcal{R}}^c \gamma \gg^{\mathcal{R}} \lambda f. \llbracket M \rrbracket_{\mathcal{R}}^c \gamma \gg^{\mathcal{R}} f$, and $\llbracket NM \rrbracket^c \delta = \llbracket N \rrbracket^c \delta \gg^{\mathcal{R}} \lambda g. \llbracket M \rrbracket^c \delta \gg^{\mathcal{R}} g$. By the first induction hypothesis, $\langle \llbracket M \rrbracket_{\mathcal{R}}^c \gamma, \llbracket M \rrbracket^c \delta \rangle \in C^*\{A\}$. So by Lemma B.18, if $\langle f, g \rangle \in \mathcal{V}^*\{A \rightarrow B\}$ then $\langle \llbracket M \rrbracket_{\mathcal{R}}^c \gamma \gg^{\mathcal{R}} f, \llbracket M \rrbracket^c \delta \gg^{\mathcal{R}} g \rangle \in C^*\{B\}$. But this is exactly the definition of $\langle \lambda f. \llbracket M \rrbracket_{\mathcal{R}}^c \gamma \gg^{\mathcal{R}} f, \lambda g. \llbracket M \rrbracket^c \delta \gg^{\mathcal{R}} g \rangle \in \mathcal{V}^*\{(A \rightarrow B) \rightarrow B\}$.
$\Gamma \vDash^* NM : B$	

By the second induction hypothesis, $\langle \llbracket N \rrbracket_{\mathcal{R}}^c \gamma, \llbracket N \rrbracket^c \delta \rangle \in C^*\{A \rightarrow B\}$. Using Lemma B.18 again, we have $\langle \llbracket N \rrbracket_{\mathcal{R}}^c \gamma \gg^{\mathcal{R}} \lambda f. \llbracket M \rrbracket_{\mathcal{R}}^c \gamma \gg^{\mathcal{R}} f, \llbracket N \rrbracket^c \delta \gg^{\mathcal{R}} \lambda g. \llbracket M \rrbracket^c \delta \gg^{\mathcal{R}} g \rangle \in C^*\{B\}$, as required.

The other cases follow by similar reasoning with Lemmas B.17 and B.18, where in the cases of the effects we also use the respective Lemmas B.19 and B.20. \square

The proof of soundness concludes by using [Propositions B.16](#) and [B.21](#):

PROOF OF SOUNDNESS. We have $\langle T, \mu \rangle \llbracket \xi \rrbracket \omega .: V \in \llbracket M \rrbracket_{\mathcal{R}}^c$ by [Proposition B.16](#) since M is of ground type. Therefore, [Proposition B.21](#) implies that $\inf_{\mu} T \llbracket \langle \mu, \mu' \rangle \xi \rrbracket \omega .: V \in \llbracket M \rrbracket^c$. Thanks to the extra conclusions of [Proposition B.16](#), $\inf_{\mu} T \llbracket \langle \mu, \mu' \rangle \rrbracket \omega .: V \in \llbracket M \rrbracket^c$ by iteratively Mu-rewriting. \square

B.4 Proof of Adequacy

The proof of adequacy starts with the [Fundamental Lemma](#), stating that \mathcal{M} -traces correspond to interrupted executions. The main reason behind this fact is simple: \mathfrak{c} -rewrites preserve this correspondence. That is:

LEMMA B.22. *If $M : \tau : V$ and $\tau \xrightarrow{\times} \pi$ for $\times \in \mathfrak{c}$, then $M : \pi : V$.*

PROOF. We split to the different $\times \in \mathfrak{c}$ 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. \square

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

$$\mathcal{V}\{A \rightarrow B\} := \{\lambda a. M \mid \forall V \in \mathcal{V}\{A\}. M[a \mapsto V] \in \mathcal{C}\{B\}\}$$

$$\mathcal{V}\{(A_1 * \dots * A_n)\} := \{\langle V_1, \dots, V_n \rangle \mid \forall i. V_i \in \mathcal{V}\{A_i\}\}$$

$$\mathcal{V}\{\{ \iota_1 \text{ of } A_1 \mid \dots \mid \iota_n \text{ of } A_n \}\} := \bigcup_i \{ \iota_i V \mid V \in \mathcal{V}\{A_i\} \}$$

$$\mathcal{C}\{A\} := \{M \in \cdot \vdash A \mid \forall \tau \in \llbracket M \rrbracket_{\mathcal{M}}^c \exists V \in \mathcal{V}\{A\}. M : \tau : V\}$$

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

$$\mathcal{X}\{\Gamma\} := \{\Theta \in \text{Sub}_{\Gamma} \mid \forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{A\}\}$$

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

THEOREM B.23 (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 B.24. *If $\tau := \alpha \llbracket \xi \rrbracket \omega .: \langle \rangle \in \llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{M}}$, then $\ell := v : \tau : \langle \rangle$.*

PROOF. W.l.o.g. $\tau \in \llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{G}}$, because the general case then follows from [Lemma B.22](#).

Thus, the interrupted execution is just a single STORE step. Indeed, the states $\langle \dot{\alpha}, \xi, \circ \rangle$ and $\langle \dot{\omega}, \xi, \mathfrak{c} \rangle$ 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 $\xi.c$ being well-formed. \square

LEMMA B.25. *If $\tau := \alpha \llbracket \xi \rrbracket \omega .: v \in \llbracket \text{rmw}_{\ell, \varphi \vec{w}} \rrbracket_{\mathcal{M}}$, then $\text{rmw}_{\varphi}(\ell; \vec{w}) : \tau : v$.*

PROOF. W.l.o.g. $\tau \in \llbracket \text{rmw}_{\ell, \varphi \vec{w}} \rrbracket_{\mathcal{G}}$, because the general case then follows from [Lemma B.22](#).

Thus, the interrupted execution is a single READONLY step (if $\tau \in \llbracket \text{rmw}_{\ell, \varphi \vec{w}}^{\text{RO}} \rrbracket_{\mathcal{G}}$) or a single RMW step (if $\tau \in \llbracket \text{rmw}_{\ell, \varphi \vec{w}}^{\text{RMW}} \rrbracket_{\mathcal{G}}$), in which the initial view points to the loaded message. \square

To address concurrent execution, let $\langle T, M \rangle \rightsquigarrow^* \langle R, V \rangle$ be an interrupted execution of the form $\langle T_1, \mu_1 \rangle, M_1 \rightsquigarrow_{RA \leq}^* \langle T_2, \rho_1 \rangle, M_2 \langle T_2, \mu_2 \rangle, M_2 \rightsquigarrow_{RA \leq}^* \langle T_3, \rho_2 \rangle, M_3 \dots \langle T_n, \mu_n \rangle, M_n \rightsquigarrow_{RA \leq}^* \langle T_{n+1}, \rho_n \rangle, M_{n+1}$ where $M_1 = M, M_{n+1} = V, T_1 = T, T_{n+1} = R$, and $\xi = \langle \mu_1, \rho_1 \rangle \dots \langle \mu_n, \rho_n \rangle$.

LEMMA B.26. *If $\xi \in \xi_1 \parallel \xi_2$ and $M_i : \alpha_i \boxed{\xi_i} \omega_i \cdot r_i : V_i$, then*

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

PROOF. By assumption, $\langle \alpha_i, M_i \rangle \rightsquigarrow^* \langle \omega_i, V_i \rangle$ and $\llbracket V_i \rrbracket_{\mathcal{M}}^v = r_i$. We obtain the required

$$\langle \inf_{\xi.o} \{ \alpha_1, \alpha_2 \}, M_1 \parallel M_2 \rangle \rightsquigarrow^* \left\langle \sup_{\xi.c} \{ \omega_1, \omega_2 \}, \langle V_1, V_2 \rangle \right\rangle$$

by interleaving the interrupted executions following the interleaving that generated ξ , prepending to the first transition `PARINIT` followed by `PARLEFT/PARRIGHT`-lifted `ADV`'s, and appending to the last transition `PARFIN` (since $\sup_{\xi.c} \{ \omega_1, \omega_2 \} = \omega_1 \sqcup \omega_2$). \square

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 $\Theta \in \text{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{A\}$. To show $\Theta a = \Theta_a \in C\{A\}$, let $\tau \in \llbracket \Theta_a \rrbracket_{\mathcal{M}}^c = \text{return}^M \llbracket \Theta_a \rrbracket_{\mathcal{M}}^v$. Suffice it we show $\Theta_a : \tau : \Theta_a$. Using Lemma B.22, we restrict to the case of $\tau \in \text{return}^{\mathcal{G}} \llbracket \Theta_a \rrbracket_{\mathcal{M}}^v$. So τ is of the form $\kappa \langle \mu, \mu \rangle \kappa \cdot \llbracket \Theta_a \rrbracket_{\mathcal{M}}^v$. The required interrupted execution is obtained by taking no steps in its only transition.

$\frac{\Gamma, a : A \vDash M : B}{\Gamma \vDash \lambda a : A. M : A \rightarrow B}$ Let $\Theta \in \text{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{A\}$. Denote by K the term $\Theta(\lambda a : A. M) = \lambda a : A. \Theta_{\{\neq a\}} M$. To show that $K \in C\{A\}$, let $\tau \in \llbracket K \rrbracket_{\mathcal{M}}^c = \text{return}^M \llbracket K \rrbracket_{\mathcal{M}}^v$. Like the previous case, we can show $K : \tau : K$ using Lemma B.22. This is sufficient, because $K \in \mathcal{V}\{A \rightarrow B\}$. 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 $(\Theta_{\{\neq a\}} M)[a \mapsto V] = \Theta[V/a]M \in C\{B\}$.

$\frac{\Gamma \vDash M : A \quad \Gamma \vDash N : A \rightarrow B}{\Gamma \vDash NM : B}$ Let $\Theta \in \text{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{A\}$. To show that $\Theta(NM) = (\Theta N)(\Theta M) \in C\{A\}$ holds, let $\tau \in \llbracket (\Theta N)(\Theta M) \rrbracket_{\mathcal{M}}^c = \llbracket \Theta N \rrbracket_{\mathcal{M}}^c \gg^M \lambda f. \llbracket \Theta M \rrbracket_{\mathcal{M}}^c \gg^M f$. Unfolding:

$$\exists \tau_1 := \alpha_1 \boxed{\xi_1} \omega_1 \cdot f \in \llbracket \Theta N \rrbracket_{\mathcal{M}}^c, \tau_2 := \alpha_2 \boxed{\xi_2} \omega_2 \cdot r \in \llbracket \Theta M \rrbracket_{\mathcal{M}}^c, \tau_3 := \alpha_3 \boxed{\xi_3} \omega_3 \cdot s \in fr.$$

$$\omega_1 \leq \alpha_2 \wedge \omega_2 \leq \alpha_3 \wedge \tau = \alpha_1 \boxed{\xi_1 \xi_2 \xi_3} \omega_3 \cdot s \in \llbracket (\Theta N)(\Theta M) \rrbracket_{\mathcal{M}}^c$$

By the induction hypotheses, there exists $\lambda a : A. K \in \mathcal{V}\{A \rightarrow B\}$ 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 $K[a \mapsto V] \in C\{B\}$, and using the Substitution Lemma: $fr = \llbracket \lambda a : A. K \rrbracket_{\mathcal{M}}^v \llbracket V \rrbracket_{\mathcal{M}}^v = \llbracket K[a \mapsto V] \rrbracket_{\mathcal{M}}^c$. 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 $(\Theta N)(\Theta M) : \tau : W$.

$\frac{\Gamma \vDash M : \text{Loc} \quad \Gamma \vDash N : \text{Val}}{\Gamma \vDash M := N : 1}$

$\frac{\varphi \in \text{RMW}_n \quad \Gamma \vDash M : \text{Loc} \quad \Gamma \vDash N : \text{Val}^n}{\Gamma \vDash \text{rmw}_\varphi(M; N) : \text{Val}}$

Binds unfold like in the case above. The rest is handled

using Lemma B.24 and Lemma B.25 respectively.

$$\frac{\Gamma \vDash M_1 : A_1 \quad \Gamma \vDash M_2 : A_2}{\Gamma \vDash M_1 \parallel M_2 : (A_1 * A_2)}$$
 Let $\Theta \in \text{Sub}_\Gamma$ be such that $\forall (a : A) \in \Gamma. \Theta_a \in \mathcal{V}\{A\}$. Thanks to **Lemma B.22**, to show $\Theta(M_1 \parallel M_2) = \Theta M_1 \parallel \Theta M_2 \in \mathcal{C}\{A_1 * A_2\}$, it is sufficient to consider $\tau \in \llbracket \Theta M_1 \rrbracket_{\mathcal{M}}^c \parallel \llbracket \Theta M_2 \rrbracket_{\mathcal{M}}^c$. Unfolding the concurrent construct, there exist $\tau_i := \alpha_i \llbracket \xi_i \rrbracket \omega_i \cdot r_i \in \llbracket \Theta M_i \rrbracket_{\mathcal{M}}^c$ and $\xi \in \xi_1 \parallel \xi_2$ such that $\tau := \inf_{\xi.o} \{\alpha_1, \alpha_2\} \llbracket \xi \rrbracket \sup_{\xi.c} \{\omega_1, \omega_2\} \cdot \langle r_1, r_2 \rangle$. 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 B.26**, $\Theta M_1 \parallel \Theta M_2 : \tau : \langle V_1, V_2 \rangle$.

The other cases are treated similarly. \square

To prove the **Evaluation Lemma** we observe that α -rewrites preserve evaluation:

LEMMA B.27. For $x \in \mathbf{a}$, if $\pi \xrightarrow{x} \tau$ and $\langle \pi.\text{ivw}, \pi.\text{ch.o} \rangle, M \Downarrow \pi.\text{v1}$, then $\langle \tau.\text{ivw}, \tau.\text{ch.o} \rangle, M \Downarrow \tau.\text{v1}$.

PROOF. In any case, $\tau.\text{v1} = \pi.\text{v1}$. If $x = \text{Ti}$ or $x = \text{Ab}$, then also $\tau.\text{ivw} = \pi.\text{ivw}$ and $\tau.\text{ch.o} = \pi.\text{ch.o}$, so the claim holds trivially. Only $x = \text{Di}$ remains, where $\pi.\text{ivw} = \tau.\text{ivw} \llbracket \uparrow \epsilon \rrbracket$ and $\pi.\text{ch.o} = \tau.\text{ch.o} \llbracket \uparrow \epsilon \rrbracket$ for some message ϵ . We obtain the required execution underlying $\langle \tau.\text{ivw}, \tau.\text{ch.o} \rangle, M \Downarrow \tau.\text{v1}$ from the one that underlies $\langle \pi.\text{ivw}, \pi.\text{ch.o} \rangle, M \Downarrow \pi.\text{v1}$ by replacing the timestamp $\epsilon.i$ with $\epsilon.t$ everywhere. We elide the straightforward simulation argument that justifies this. \square

PROOF OF THE EVALUATION LEMMA. Denote $\tau := \alpha \llbracket \langle \mu, \rho \rangle \rrbracket \omega \cdot r \in \llbracket M \rrbracket^c$. By **Retroactive Closure**, $\llbracket M \rrbracket^c = \llbracket M \rrbracket_{\mathcal{M}}^c$. So there exists $\pi \in \llbracket M \rrbracket_{\mathcal{M}}^c$ such that $\pi \xrightarrow{\alpha} \tau$. Proceed by induction on the number of α -rewrites. If none, $\tau = \pi \in \llbracket M \rrbracket_{\mathcal{M}}^c$, so by the **Fundamental Lemma**, $M : \tau : V$ for some V . Since M is of ground type, so is $V = \llbracket V \rrbracket_{\mathcal{M}}^v = r$, and thus $\langle \dot{\alpha}, \mu \rangle, M \rightsquigarrow_{\text{RA} \leq}^* \langle \dot{\omega}, \rho \rangle, r$, so $\langle \dot{\alpha}, \mu \rangle, M \Downarrow r$.

Otherwise, we have $\pi \xrightarrow{\alpha} \tau' \xrightarrow{x} \tau$ where $x \in \mathbf{a}$ and $\langle \tau'.\text{ivw}, \tau'.\text{ch.o} \rangle, M \Downarrow \tau'.\text{v1}$ by the induction hypothesis. We replace τ' with τ using **Lemma B.27**, as required. \square

C VALIDATING TRANSFORMATIONS

Table 2 lists various transformations $M \rightarrow N$ that can be proved this way, 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 $\Phi, \Psi \in \text{Val} \rightarrow \text{Val}$:

- The *domain of definition* of Φ is $\text{dom } \Phi := \{v \in \text{Val} \mid \Phi v \neq \perp\}$.
- We say that Ψ is an *expansion* of Φ , denoted by $\Phi \leq \Psi$, if $\Phi v \neq \Psi v$ occurs only when $\Phi v = \perp$ 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 atomically reads and rewrites the read value.
- We denote by Φ^{id} the unique expansion of Φ that is total: $\Phi^{\text{id}} v := \text{if } \Phi v = \perp \text{ then } v \text{ else } \Phi v$. Intuitively, Φ^{id} rewrites the read value whenever Φ reads but does not write.
- We let $(\Psi \circ^{\text{id}} \Phi) v := \text{if } \Phi v = \perp \text{ then } \Psi v \text{ else } \Psi^{\text{id}}(\Phi v)$. Intuitively, $(\Psi \circ^{\text{id}} \Phi)$ composes the modification of Φ followed by the modification of Ψ , only failing if both do.

Moreover, some optimizations involving modifiers assume the language can express corresponding constructs. For example, the Write-RMW Elimination instantiated with $\varphi = \text{faa}$ requires addition (+), and the RMW-Write Elimination instantiated with $\varphi = \text{cas}$ requires branching on value comparison (**if** – = – **then** – **else** –). Under this assumption, for every primitive modifier φ and every tuple \vec{v} of length $\varphi.\text{ar}$, both $\varphi_{\vec{v}}$ and $\varphi_{\vec{v}}^{\text{id}}$ are represented by closed, pure (effect-free) terms, of type $\text{Val} \rightarrow \{\iota_{\perp} \text{ of } \perp \mid \iota_{\tau} \text{ of Val}\}$ and $\text{Val} \rightarrow \text{Val}$ respectively. These are used implicitly in **Table 2**.

In the following we prove selected results from **Table 2**. We explicitly mention the use of α -rewrites, but often leave uses of τ -rewrites implicit. For convenience, we denote $\alpha \llbracket \xi \eta \rrbracket \omega \cdot s := (\alpha \llbracket \xi \rrbracket \kappa \cdot r) \gg (\sigma \llbracket \eta \rrbracket \omega \cdot s)$, and we say this trace resulted from binding the first with the second.

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

$$\llbracket (\text{let } a = M_1 \text{ in } M_2) \parallel (\text{let } b = N_1 \text{ in } N_2) \rrbracket^c \supseteq \llbracket \text{match } M_1 \parallel N_1 \text{ with } \langle a, b \rangle. M_2 \parallel N_2 \rrbracket^c$$

Table 2. Validated transformations. Needed rules used from \mathfrak{a} appear above the symbol \rightarrow .

2598		
2599		
2600	Generalized Sequencing	Symmetric-Monoidal Laws, e.g.
2601	$(\text{let } a = M_1 \text{ in } M_2) \parallel (\text{let } b = N_1 \text{ in } N_2) \rightarrow$	$M \parallel N \rightarrow \text{match } N \parallel M \text{ with } \langle b, a \rangle. \langle a, b \rangle$
2602	$\text{match } M_1 \parallel N_1 \text{ with } \langle a, b \rangle. M_2 \parallel N_2$	
2603	Sequencing $M \parallel N \rightarrow \langle M, N \rangle$	Write-RMW Elimination
2604	Irrelevant Read Introduction $\langle \rangle \rightarrow \ell? ; \langle \rangle$	$\ell := v ; \text{rmw}_\varphi(\ell; \vec{w}) \xrightarrow{\text{Ab}} \ell := \varphi_{\vec{w}}^{\text{id}} v ; v$
2605	Irrelevant Read Elimination $\ell? ; \langle \rangle \rightarrow \langle \rangle$	$\ell := v ; \ell? \rightarrow \ell := v ; v$
2606	Write-Write Elimination	$\ell := v ; \text{CAS}(\ell, v, u) \xrightarrow{\text{Ab}} \ell := u ; v$
2607	$\ell := w ; \ell := v \xrightarrow{\text{Ab}} \ell := v$	$\ell := v ; \text{CAS}(\ell, w, u) \rightarrow \ell := v ; v \quad (v \neq w)$
2608	Write-Read Deorder $(\ell \neq \ell')$	$\ell := v ; \text{FAA}(\ell, w) \xrightarrow{\text{Ab}} \ell := v + w ; v$
2609	$\langle \ell := v, \ell' \rangle \xrightarrow{\text{T}_1} \ell := v \parallel \ell'?$	$\ell := v ; \text{XCHG}(\ell, w) \xrightarrow{\text{Ab}} \ell := w ; v$
2610	RMW Expansion $(\varphi_{\vec{v}} \leq \psi_{\vec{w}})$	RMW-Write Elimination $(\text{dom } \psi_{\vec{w}} \supseteq \text{dom } \varphi_{\vec{v}})$
2611	$\text{rmw}_\varphi(\ell; \vec{v}) \xrightarrow{\text{Di}} \text{rmw}_\psi(\ell; \vec{w})$	$\text{let } a = \text{rmw}_\varphi(\ell; \vec{v}) \text{ in}$
2612	$\ell? \xrightarrow{\text{Di}} \text{CAS}(\ell, v, v)$	$\text{match } (\psi_{\vec{w}}) \text{ awith}$
2613	$\text{CAS}(\ell, v, v) \xrightarrow{\text{Di}} \text{FAA}(\ell, 0)$	$\{ \iota_{\perp} _ a \mid \iota_{\top} v. \ell := v ; a \} \xrightarrow{\text{Ab}} \text{rmw}_\psi(\ell; \vec{w})$
2614		$\text{let } a = \ell? \text{ in (if } a = v$
2615	Atomic Store	$\text{then } \ell := w \text{ else } \langle \rangle) ; a \rightarrow \text{CAS}(\ell, v, w)$
2616	$\ell := v \rightarrow \text{XCHG}(\ell, v) ; \langle \rangle$	$\text{let } a = \ell? \text{ in } \ell := a + v ; a \rightarrow \text{FAA}(\ell, v)$
2617		$\text{let } a = \ell? \text{ in } \ell := v ; a \rightarrow \text{XCHG}(\ell, v)$
2618		
2619	RMW-RMW Elimination $\langle \text{rmw}_\varphi(\ell; \vec{v}), \text{rmw}_\psi(\ell; \vec{w}) \rangle \xrightarrow{\text{Ab}} \text{let } a = \text{rmw}_\zeta(\ell; \vec{u}) \text{ in } \langle a, \varphi_{\vec{v}}^{\text{id}} a \rangle$	$(\zeta_{\vec{u}} = \psi_{\vec{w}} \circ^{\text{id}} \varphi_{\vec{v}})$
2620	$\langle \ell?, \ell? \rangle \rightarrow \text{let } a = \ell? \text{ in } \langle a, a \rangle$	$\langle \text{FAA}(\ell, v), \text{FAA}(\ell, w) \rangle \rightarrow \text{let } a = \text{FAA}(\ell, v + w) \text{ in } \langle a, a + v \rangle$
2621	$\langle \ell?, \text{CAS}(\ell, v, w) \rangle \rightarrow \text{let } a = \text{CAS}(\ell, v, w) \text{ in } \langle a, a \rangle$	$\langle \text{XCHG}(\ell, w), \ell? \rangle \rightarrow \text{let } a = \text{XCHG}(\ell, w) \text{ in } \langle a, w \rangle$
2622		
2623		
2624	(Formally, in the right denotation we use $\Gamma, a : A', b : B' \vdash M_2 : A_2$ and $\Gamma, a : A', b : B' \vdash N_2 : B_2$.)	
2625	PROOF. Let $\gamma \in \llbracket \Gamma \rrbracket$ and denote the left and right sets of the required containment by P and Q	
2626	respectively. Thus we require $P \supseteq Q$.	
2627	Let $\varrho \in Q$. By Deferral of Closure , ϱ is in the ca -closure of:	
2628	$Q' := \llbracket M_1 \rrbracket^c \gamma \parallel \llbracket N_1 \rrbracket^c \gamma \gg^{\mathcal{G}} \lambda \langle \gamma_a, \gamma_b \rangle. \llbracket M_2 \rrbracket^c (\gamma_c)_{(c:C) \in \Gamma, a:A'} \parallel \llbracket N_2 \rrbracket^c (\gamma_c)_{(c:C) \in \Gamma, b:B'}$	
2629		
2630	So there exists $\varrho' \in Q'$ that ca -rewrites to ϱ . This ϱ' results from binding two traces. On the left,	
2631	$\text{inf}_{\xi, \circ} \{ \alpha_1, \kappa_1 \} \llbracket \xi \rrbracket \omega_1 \sqcup \sigma_1 \cdot \langle r_1, s_1 \rangle$, where:	
2632	$\tau_1 := \alpha_1 \llbracket \xi_1 \rrbracket \omega_1 \cdot r_1 \in \llbracket M_1 \rrbracket^c \gamma ; \pi_1 := \kappa_1 \llbracket \eta_1 \rrbracket \sigma_1 \cdot s_1 \in \llbracket N_1 \rrbracket^c \gamma ; \xi \in \xi_1 \parallel \eta_1$	
2633		
2634	On the right, $\text{inf}_{\eta, \circ} \{ \alpha_2, \kappa_2 \} \llbracket \eta \rrbracket \omega_2 \sqcup \sigma_2 \cdot \langle r_2, s_2 \rangle$, where, setting $\gamma_a := r_1$ and $\gamma_b := s_1$:	
2635	$\tau_2 := \alpha_2 \llbracket \xi_2 \rrbracket \omega_2 \cdot r_2 \in \llbracket M_2 \rrbracket^c (\gamma_c)_{(c:C) \in \Gamma, a:A'} ; \pi_2 := \kappa_2 \llbracket \eta_2 \rrbracket \sigma_2 \cdot s_2 \in \llbracket N_2 \rrbracket^c (\gamma_c)_{(c:C) \in \Gamma, b:B'} ; \eta \in \xi_2 \parallel \eta_2$	
2636		
2637	The binding implies that $\omega_1 \sqcup \sigma_1 \leq \text{inf}_{\eta, \circ} \{ \alpha_2, \kappa_2 \}$. In particular, $\omega_1 \leq \alpha_2$ and $\sigma_1 \leq \kappa_2$. Therefore,	
2638	$\tau_1 \gg^{\mathcal{G}} \tau_2 = \alpha_1 \llbracket \xi_1 \xi_2 \rrbracket \omega_2 \cdot r_2 \in \llbracket \text{let } a = M_1 \text{ in } M_2 \rrbracket^c$ and $\pi_1 \gg^{\mathcal{G}} \pi_2 = \kappa_1 \llbracket \eta_1 \eta_2 \rrbracket \sigma_2 \cdot s_2 \in \llbracket \text{let } b = N_1 \text{ in } N_2 \rrbracket^c$.	
2639	Since $\xi \eta \in \xi_1 \xi_2 \parallel \eta_1 \eta_2$ and $(\xi \eta) \cdot \circ = \xi \cdot \circ$, we obtain ϱ' by interleaving these. Therefore, $\varrho' \in P$.	
2640	Since P is ca -closed, $\varrho \in P$. \square	
2641		
2642	PROPOSITION C.2. $\llbracket \langle \rangle \rrbracket^c \supseteq \llbracket \ell? ; \langle \rangle \rrbracket_{\mathcal{G}}^c$.	
2643	PROOF. Let $\tau \in \llbracket \ell? ; \langle \rangle \rrbracket_{\mathcal{G}}^c$. Unfolding definitions:	
2644	$\llbracket \ell? ; \langle \rangle \rrbracket_{\mathcal{G}}^c := \llbracket \text{rmw}_{\ell, \lambda_{\perp}} \rrbracket_{\mathcal{G}} \gg^{\mathcal{G}} \lambda_{\perp}. \text{return}^{\mathcal{G}} \langle \rangle = \{ \alpha \langle \underline{\mu}, \underline{\mu} \rangle \langle \underline{\rho}, \underline{\rho} \rangle \omega \cdot \langle \rangle \in \text{Trace1} \mid \exists v \in \mu_{\ell}. \alpha \mapsto v \}$	
2645		
2646		

2647 Therefore, we have the form $\tau = \alpha \langle \mu, \mu \rangle \langle \rho, \rho \rangle \omega \cdot \langle \rangle$. From $\alpha \langle \mu, \mu \rangle \alpha \cdot \langle \rangle \in \llbracket \langle \rangle \rrbracket_{\mathcal{G}}^c$, we obtain
 2648 $\tau \in \llbracket \langle \rangle \rrbracket^c$ by stuttering (St) and forwarding (Fw). \square

2649 PROPOSITION C.3. $\llbracket \ell := v \rrbracket^c \supseteq \llbracket \text{XCHG}(\ell, v); \langle \rangle \rrbracket_{\mathcal{G}}^c$.

2651 PROOF. By taking the traces in $\llbracket \ell := v \rrbracket_{\mathcal{G}}^c$ in which the newly added message dovetails with the
 2652 previous message in memory by choosing the initial timestamp appropriately. \square

2654 PROPOSITION C.4. Assuming $\ell \neq \ell'$, $\llbracket \ell := v, \ell' \rrbracket^c \supseteq \llbracket \ell := v \parallel \ell' \rrbracket_{\mathcal{G}}^c$.

2656 PROOF. The elements of $\llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{G}} \parallel \llbracket \text{rmw}_{\ell', \lambda, \perp} \rrbracket_{\mathcal{G}}$ are formed by interleaving a store

$$2657 \kappa \langle \mu, \mu \uplus \{ \ell : v @ (q, t) \langle \langle \kappa [\ell \mapsto t] \rangle \rangle \} \rangle \kappa [\ell \mapsto t] \cdot \langle \rangle \in \llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{G}}$$

2659 with a load $\sigma \langle \rho, \rho \rangle \sigma \cdot \langle \rangle$. $w \in \llbracket \text{rmw}_{\ell', \lambda, \perp} \rrbracket_{\mathcal{G}}$. Depending on the order, this results in one of:

$$2660 \inf_{\mu} \{ \kappa, \sigma \} \langle \mu, \mu \uplus \{ \ell : v @ (q, t) \langle \langle \kappa [\ell \mapsto t] \rangle \rangle \} \rangle \langle \rho, \rho \rangle \kappa [\ell \mapsto t] \sqcup \sigma \cdot \langle \rangle, w \quad (\text{WR})$$

$$2662 \inf_{\rho} \{ \kappa, \sigma \} \langle \rho, \rho \rangle \langle \mu, \mu \uplus \{ \ell : v @ (q, t) \langle \langle \kappa [\ell \mapsto t] \rangle \rangle \} \rangle \kappa [\ell \mapsto t] \sqcup \sigma \cdot \langle \rangle, w \quad (\text{RW})$$

2664 We prove separately that these interleavings are in $\llbracket \langle \ell := v, \ell' \rrbracket^c$.

- 2665 • (WR): Denoting $\alpha := \inf_{\mu} \{ \kappa, \sigma \}$ and $\theta := (\rho \setminus \{ \ell : v @ (q, t) \langle \langle \kappa [\ell \mapsto t] \rangle \rangle \}) \uplus \{ \ell : v @ (q, t) \langle \langle \alpha [\ell \mapsto t] \rangle \rangle \}$:

$$2666 \alpha \langle \mu, \mu \uplus \{ \ell : v @ (q, t) \langle \langle \alpha [\ell \mapsto t] \rangle \rangle \} \rangle \alpha [\ell \mapsto t] \cdot \langle \rangle \in \llbracket \text{store}_{\ell, v} \rrbracket_{\mathcal{G}}$$

$$2668 \alpha [\ell \mapsto t] \sqcup \sigma \langle \theta, \theta \rangle \alpha [\ell \mapsto t] \sqcup \sigma \cdot \langle \rangle \in \llbracket \text{rmw}_{\ell', \lambda, \perp} \rrbracket_{\mathcal{G}}$$

2669 By forwarding (Fw) after binding we obtain:

$$2670 \alpha \langle \mu, \mu \uplus \{ \ell : v @ (q, t) \langle \langle \alpha [\ell \mapsto t] \rangle \rangle \} \rangle \langle \theta, \theta \rangle \kappa [\ell \mapsto t] \sqcup \sigma \cdot \langle \rangle, w \in \llbracket \langle \ell := v, \ell' \rrbracket^c$$

2672 All that remains is to tighten (Ti) $\ell : v @ (q, t) \langle \langle \alpha [\ell \mapsto t] \rangle \rangle$ to $\ell : v @ (q, t) \langle \langle \kappa [\ell \mapsto t] \rangle \rangle$.

- 2673 • (RW): Using the result for (WR), with $\theta := \mu \uplus \{ \ell : v @ (q, t) \langle \langle \kappa [\ell \mapsto t] \rangle \rangle \}$:

$$2675 \inf_{\mu} \{ \kappa, \sigma \} \langle \mu, \theta \rangle \langle \theta, \theta \rangle \kappa [\ell \mapsto t] \sqcup \sigma \cdot \langle \rangle, w \in \llbracket \langle \ell := v, \ell' \rrbracket^c$$

2676 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. \square

2679 PROPOSITION C.5. Assuming $\varphi_{\vec{w}} \leq \psi_{\vec{w}}$, $\llbracket \text{rmw}_{\varphi}(\ell; \vec{v}) \rrbracket^c \supseteq \llbracket \text{rmw}_{\psi}(\ell; \vec{w}) \rrbracket_{\mathcal{G}}^c$.

2681 PROOF. Let $\tau \in \llbracket \text{rmw}_{\psi}(\ell; \vec{w}) \rrbracket_{\mathcal{G}}^c$, resulting from loading a value u from a message v . If $\varphi_{\vec{v}} u =$
 2682 $\psi_{\vec{w}} u$, then obviously $\tau \in \llbracket \text{rmw}_{\varphi}(\ell; \vec{v}) \rrbracket_{\mathcal{G}}^c$. Otherwise, by assumption $\varphi_{\vec{v}} u = \perp$ and $\psi_{\vec{w}} u = u$. So
 2683 we have $\tau = \kappa \langle \mu, \mu \uplus \{ \epsilon \} \rangle \kappa [\ell \mapsto t] \cdot u$, with $\epsilon := \ell : u @ (\kappa_{\ell}, t) \langle \langle \kappa [\ell \mapsto t] \rangle \rangle$, where $v \in \mu_{\ell}$ and
 2684 $v.t = \kappa_{\ell}$. In the left denotation, by loading $v [\uparrow \epsilon]$ we have $\kappa [\uparrow \epsilon] \langle \mu [\uparrow \epsilon], \mu [\uparrow \epsilon] \rangle \kappa [\uparrow \epsilon] \cdot u =$
 2685 $(\kappa \langle \mu, \mu \rangle \kappa [\ell \mapsto t] \cdot u) [\uparrow \epsilon]$. By diluting (Di) we obtain τ . \square

2688 PROPOSITION C.6. Assuming $\forall v' \in \text{Val}. \zeta_{\vec{u}} v' = (\psi_{\vec{w}v'} \circ^{\text{id}} \varphi_{\vec{v}}) v'$,

$$2689 \llbracket \text{let } a = \text{rmw}_{\varphi}(\ell; \vec{v}) \text{ in } \langle a, \text{rmw}_{\psi}(\ell; \vec{w}a) \rangle \rrbracket^c \supseteq \llbracket \text{let } a = \text{rmw}_{\zeta}(\ell; \vec{u}) \text{ in } \langle a, \varphi_{\vec{v}}^{\text{id}} a \rangle \rrbracket_{\mathcal{G}}^c$$

2692 PROOF. Let $\pi \in \llbracket \text{let } a = \text{rmw}_{\zeta}(\ell; \vec{u}) \text{ in } \langle a, \varphi_{\vec{v}}^{\text{id}} a \rangle \rrbracket_{\mathcal{G}}^c$. So a $\tau' := \alpha \langle \mu, \rho \rangle \omega \cdot v' \in \llbracket \text{rmw}_{\zeta}(\ell; \vec{u}) \rrbracket_{\mathcal{G}}^c$

2694 exists due to loading $v \in \mu_{\ell}$ with $v.v1 = v'$, such that $\tau := \alpha \langle \mu, \rho \rangle \omega \cdot \langle v', \varphi_{\vec{v}}^{\text{id}} v' \rangle \xrightarrow{\text{St}} \xrightarrow{\text{Fw}} \pi$.

2695

2696 **RO.** If $\tau' \in \llbracket \text{rmw}_{\ell, \zeta_{\vec{u}}}^{\text{RO}} \rrbracket_{\mathcal{G}}$, then we have $\tau = \kappa \langle \mu, \mu \rangle \kappa \cdot \langle v', \varphi_{\vec{v}}^{\text{id}} v' \rangle$ where $\zeta_{\vec{u}} v' = \perp$ and $v.\mathfrak{t} = \kappa_{\ell}$.

2697 By assumption, $\varphi_{\vec{v}} v' = \perp$, so $\varphi_{\vec{v}}^{\text{id}} v' = v'$; and $\psi_{\vec{w}} v' = \perp$, so by loading v in both RMWs we
 2698 can obtain τ in the left denotation.

2699 **RMW.** If $\tau' \in \llbracket \text{rmw}_{\ell, \zeta_{\vec{u}}}^{\text{RMW}} \rrbracket_{\mathcal{G}}$, then we have $\tau = \kappa \langle \mu, \mu \uplus \{ \ell : u' @ (\kappa_{\ell}, t) \langle \kappa [\ell \mapsto t] \rangle \} \rangle \kappa [\ell \mapsto t] \cdot$
 2700

2701 $\langle v', \varphi_{\vec{v}}^{\text{id}} v' \rangle$ where $\zeta_{\vec{u}} v' = u'$ and $v.\mathfrak{t} = \kappa_{\ell}$. The $\varphi_{\vec{v}} v' = \perp$ case is similar to before, where again
 2702 τ is found in the left denotation by loading v in both RMWs, with the difference that here
 2703 $\psi_{\vec{w}} v' = u'$, to the second RMW also writes the message $\ell : u' @ (\kappa_{\ell}, t) \langle \kappa [\ell \mapsto t] \rangle$.

2704 The $\varphi_{\vec{v}} v' = w'$ case remains, in which $\varphi_{\vec{v}}^{\text{id}} v' = w'$. In the sub-case that $\psi_{\vec{v}} w' = \perp$ we have
 2705 $w' = u'$, and we find τ in the left denotation by loading v and writing $\ell : u' @ (\kappa_{\ell}, t) \langle \kappa [\ell \mapsto t] \rangle$
 2706 in the first RMW, which the second RMW loads.

2707 In the sub-case where $\psi_{\vec{v}} w' = u'$, the first RMW writes $\ell : w' @ (\kappa_{\ell}, \frac{\kappa_{\ell} + t}{2}) \langle \kappa [\ell \mapsto \frac{\kappa_{\ell} + t}{2}] \rangle$ in-
 2708 stead. For the second RMW we take a trace with initial view $\kappa [\ell \mapsto \frac{\kappa_{\ell} + t}{2}]$, enabling its
 2709 loading of this new message and writing $\ell : u' @ (\frac{\kappa_{\ell} + t}{2}, t) \langle \kappa [\ell \mapsto t] \rangle$. To find τ in the left
 2710 denotation we have the latter message absorb (Ab) the former message.

2711 Either way, τ is in $\llbracket \text{let } a = \text{rmw}_{\varphi}(\ell; \vec{v}) \text{ in } \langle a, \text{rmw}_{\psi}(\ell; \vec{w}a) \rangle \rrbracket^c$, and therefore so is π . \square

2712 **COROLLARY C.7.** Assuming $\zeta_{\vec{u}} = \psi_{\vec{w}} \circ^{\text{id}} \varphi_{\vec{v}}$,

$$2713 \llbracket \langle \text{rmw}_{\varphi}(\ell; \vec{v}), \text{rmw}_{\psi}(\ell; \vec{w}) \rangle \rrbracket^c \supseteq \llbracket \text{let } a = \text{rmw}_{\zeta}(\ell; \vec{u}) \text{ in } \langle a, \varphi_{\vec{v}}^{\text{id}} a \rangle \rrbracket_{\mathcal{G}}^c$$

2714 **PROOF.** Using a special case of [Proposition C.6](#), where ψ is independent of its final parameter. \square

$$2715 \llbracket \ell := v ; \text{rmw}_{\varphi}(\ell; \vec{w}) \rrbracket^c \supseteq \llbracket \ell := \varphi_{\vec{w}}^{\text{id}} v ; v \rrbracket_{\mathcal{G}}^c$$

2716 **PROOF.** Same as the RMW case in the proof of [Proposition C.6](#), except the initial timestamp does
 2717 not have to equal the timestamp of the loaded message. \square

$$2718 \llbracket \ell := w ; \ell := v \rrbracket^c \supseteq \llbracket \ell := v \rrbracket_{\mathcal{G}}^c$$

2719 **PROOF.** Replace the second assignment on the left using [Proposition C.3](#), and follow with [Propo-
 2720 sition C.8](#). \square

2721 **PROPOSITION C.10.** Assuming $\text{dom } \psi_{\vec{w}} \supseteq \text{dom } \varphi_{\vec{u}}$,

$$2722 \llbracket \text{let } a = \text{rmw}_{\varphi}(\ell; \vec{u}) \text{ in match } \psi_{\vec{w}} a \text{ with } \{ \iota_{\perp} _ . a \mid \iota_{\top} v . \ell := v ; a \} \rrbracket^c \supseteq \llbracket \text{rmw}_{\psi}(\ell; \vec{w}) \rrbracket_{\mathcal{G}}^c$$

$$2723 \llbracket \text{rmw}_{\psi}(\ell; \vec{w}) \rrbracket_{\mathcal{G}}^c = \llbracket \text{rmw}_{\ell, \psi_{\vec{w}}} \rrbracket_{\mathcal{G}} = \llbracket \text{rmw}_{\ell, \psi_{\vec{w}}}^{\text{RO}} \rrbracket_{\mathcal{G}} \cup \llbracket \text{rmw}_{\ell, \psi_{\vec{w}}}^{\text{RMW}} \rrbracket_{\mathcal{G}}$$

2724 **RO.** If $\tau \in \llbracket \text{rmw}_{\ell, \psi_{\vec{w}}}^{\text{RO}} \rrbracket_{\mathcal{G}}$, then we have $\tau = \kappa \langle \mu, \mu \rangle \kappa \cdot v.\mathfrak{v}1$ where $\psi_{\vec{w}}(v.\mathfrak{v}1) = \perp$, $v \in \mu_{\ell}$, and
 2725 $v.\mathfrak{t} = \kappa_{\ell}$. Structurally, we have

$$2726 \llbracket \text{match } (\psi_{\vec{w}}) v.\mathfrak{v}1 \text{ with } \{ \iota_{\perp} _ . v.\mathfrak{v}1 \mid \iota_{\top} v . \ell := v ; v.\mathfrak{v}1 \} \rrbracket^c = \text{return } v.\mathfrak{v}1$$

2727 By assumption, $\varphi_{\vec{u}}(v.\mathfrak{v}1) = \perp$. Loading the same message v , we have $\tau \in \llbracket \text{rmw}_{\varphi}(\ell; \vec{u}) \rrbracket_{\mathcal{G}}^c$.

2728 We obtain the desired trace from binding it with $\kappa \langle \mu, \mu \rangle \kappa \cdot v.\mathfrak{v}1 \in \text{return } v.\mathfrak{v}1$.

2729 **RMW.** If $\tau \in \llbracket \text{rmw}_{\ell, \psi_{\vec{w}}}^{\text{RMW}} \rrbracket_{\mathcal{G}}$, then we have $\tau = \kappa \langle \mu, \mu \uplus \{ \ell : v @ (v.\mathfrak{t}, t) \langle \kappa [\ell \mapsto t] \rangle \} \rangle \kappa [\ell \mapsto t] \cdot$
 2730 $v.\mathfrak{v}1$ where $\psi_{\vec{w}}(v.\mathfrak{v}1) = v$, $v \in \mu_{\ell}$, and $v.\mathfrak{t} = \kappa_{\ell}$. Structurally, we have

$$2731 \llbracket \text{match } \psi_{\vec{w}}(v.\mathfrak{v}1) \text{ with } \{ \iota_{\perp} _ . v.\mathfrak{v}1 \mid \iota_{\top} v . \ell := v ; v.\mathfrak{v}1 \} \rrbracket^c = \llbracket \ell := v ; v.\mathfrak{v}1 \rrbracket^c$$

2745 Loading the same message v , we proceed depending on $\varphi_{\vec{u}}(v.v\perp)$.

2746 $\varphi_{\vec{u}}(v.v\perp) = \perp$. We can bind $\kappa \langle \mu, \mu \rangle \kappa \cdot v.v\perp \in \llbracket \mathbf{rmw}_{\varphi}(\ell; \vec{u}) \rrbracket^c$, with $\tau \in \llbracket \ell := v; v.v\perp \rrbracket^c$.

2747 $\varphi_{\vec{u}}(v.v\perp) \neq \perp$. Then we have $\kappa \langle \mu, \rho \rangle \kappa [\ell \mapsto \frac{\kappa_{\ell+t}}{2}] \cdot v.v\perp \in \llbracket \mathbf{rmw}_{\varphi}(\ell; \vec{u}) \rrbracket^c$, where $\rho :=$

2748 $\mu \uplus \{ \ell : \varphi_{\vec{u}}(v.v\perp) @ (v.t, \frac{\kappa_{\ell+t}}{2}) \llbracket \kappa [\ell \mapsto \frac{\kappa_{\ell+t}}{2}] \rrbracket \}$. We can bind it with

2750 $\kappa [\ell \mapsto \frac{\kappa_{\ell+t}}{2}] \langle \rho, \rho \uplus \{ \ell : v @ (\frac{\kappa_{\ell+t}}{2}, t) \llbracket \kappa [\ell \mapsto t] \rrbracket \} \rangle \kappa [\ell \mapsto t] \cdot v.v\perp \in \llbracket \ell := v; v.v\perp \rrbracket^c$

2751 where at the end we absorb (Ab) the first message into the second. □

2753 D COMMUTATIVITY DIAGRAMS

2754 We prove [Rewrite Commutativity](#) via an elaborate case-split. The following lemma summarizing
2755 insights we noted when we introduced the rewrite rules in §6, which will be helpful during the
2756 case analysis.

2757 LEMMA D.1. For $x \in \mathbf{gc}$, assume τ is a trace and $\tau \xrightarrow{x} \pi$. Then, using the notations of [Table 1](#):

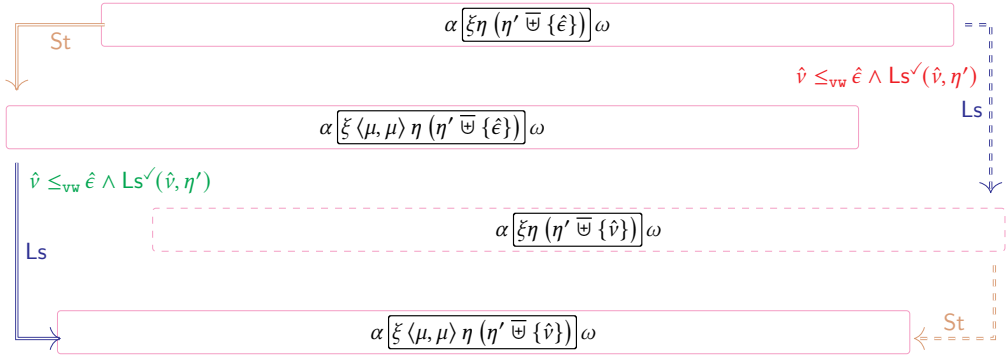
- 2759 • If $x = \mathbf{Mu}$, then $\pi \in \mathbf{Trace}$.
- 2760 • If $x = \mathbf{Ls}$, then $\pi \in \mathbf{Trace}$ iff $\mathbf{Ls}^{\vee}(v, \eta)$: either η is empty, or $v \hookrightarrow (\eta \bar{\cup} \{v\}).o$.
- 2761 • If $x = \mathbf{Ex}$, then $\pi \in \mathbf{Trace}$ iff $\mathbf{Ex}^{\vee}(v, \eta)$: either η is empty, or $v \hookrightarrow (\eta \bar{\cup} \{v\}).o$.
- 2762 • If $x = \mathbf{Cn}$, then $\pi \in \mathbf{Trace}$ iff $\mathbf{Cn}^{\vee}(\epsilon, \xi)$: either ξ is empty, $\epsilon.i \notin \xi.c.t$, or $\epsilon.\text{seg} \cup \xi.c.\text{seg} = \emptyset$.
- 2763 • If $x = \mathbf{St}$, then $\pi \in \mathbf{Trace}$ iff $\mathbf{St}^{\vee}(\alpha, \mu)$: $\alpha \mapsto \mu \in \mathbf{Mem}$.
- 2764 • If $x = \mathbf{Fw}$, then $\pi \in \mathbf{Trace}$ iff $\mathbf{Fw}^{\vee}(\omega, \xi)$: $\omega \hookrightarrow \xi.c$.
- 2765 • If $x = \mathbf{Rw}$, then $\pi \in \mathbf{Trace}$ iff $\mathbf{Rw}^{\vee}(\alpha, \xi)$: $\alpha \hookrightarrow \xi.o$.

2766 We address each case, grouping those for which we reason similarly:

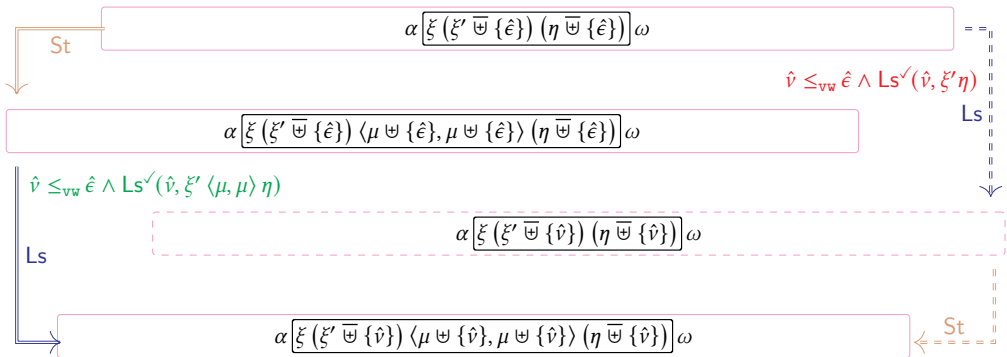
- 2767 • For cases of $\mathbf{St} \leftrightsquigarrow y$ where $y \in \mathbf{g}$ (1, 2, 3, 4, 5, 6), the required condition is about the
2768 same chronicle as the assumed condition, except for possibly a removed transition. This
2769 means that its opening memory is an extension of the original, and the closing memory is
2770 a reduction of the original. The condition of pointing downwards into a memory is stable
2771 under extensions, and the condition of non-intersection is stable under reductions. Cases
2772 of $\mathbf{Mu} \leftrightsquigarrow y$ where $y \in \mathbf{g}$ (7, 8, 9, 10, 11, 12) are simpler because the opening and closing
2773 memory remain the same.
- 2774 • The cases of $\mathbf{Fw} \leftrightsquigarrow y$ and $\mathbf{Rw} \leftrightsquigarrow y$ where $y \in \mathbf{g}$ (13, 14, 15, 16, 17, 18) are trivial because the
2775 required condition remains the same.
- 2776 • For cases of $\mathbf{Ti} \leftrightsquigarrow y$ where $y \in \mathbf{g}$, the required condition in the cases of $y \in \{\mathbf{Ls}, \mathbf{Ex}\}$ (19,
2777 20, 21, 22) holds because pointing downwards into a memory is stable under “loosening”
2778 a message within the memory ($v \leq \epsilon$). The remaining $y = \mathbf{Cn}$ case (23, 24, 25) holds be-
2779 cause the difference between the required condition and the original keeps the occupied
2780 timestamps the same, and the \leq relation is stable under “loosening” the first argument.
- 2781 • For cases of $\mathbf{Ab} \leftrightsquigarrow y$ where $y \in \mathbf{g}$, the required condition in the cases of $y \in \{\mathbf{Ls}, \mathbf{Ex}\}$
2782 (26, 27, 28, 29) holds because pointing downwards into a memory is stable under changing
2783 a message’s initial timestamp and adding a message within the memory. The remaining
2784 $y = \mathbf{Cn}$ case (30, 31, 32) holds because the difference between the required condition and
2785 the original keeps the occupied timestamps the same, and the \leq relation is stable under
2786 changing the initial timestamp of the first argument.
- 2787 • Cases of $\mathbf{Di} \leftrightsquigarrow y$ where $y \in \mathbf{g}$ hold thanks to [Lemma 6.5](#) when $y \in \{\mathbf{Ls}, \mathbf{Ex}\}$ (33, 34, 35, 36,
2788 37, 38). The remaining $y = \mathbf{Cn}$ case (39, 40, 41, 42) is the most complicated. First, we note
2789 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”
2790 is of the correct shape. The rewrite itself is valid because \leq is stable under changing the
2791
2792
2793

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 \equiv .

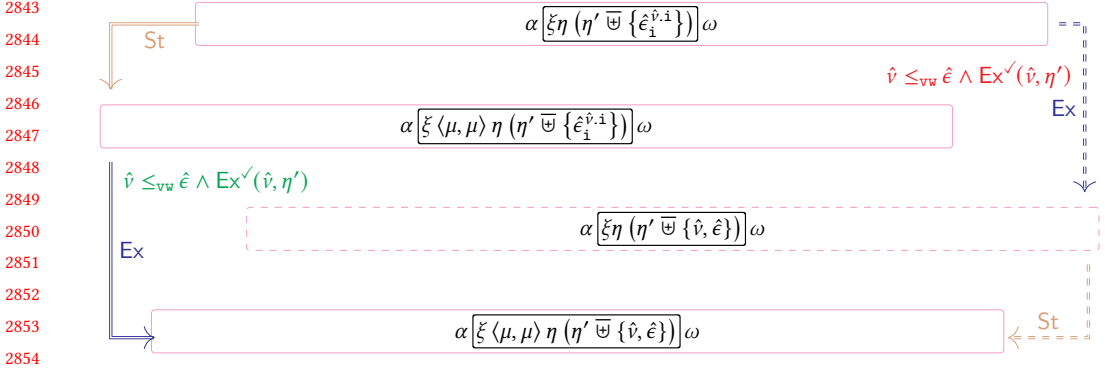
- For cases of $x \sqsubseteq St$ where $x \in a$, the required condition in the cases of $x \in \{Ti, Ab\}$ (43, 44, 45, 46) holds because there remains a message at each timestamp where there was a message originally, and the initial view remains the same. The remaining $y = Di$ case (47, 48) holds because pointing-to is stable under pulling along a message; and if the initial view pointed to the Di'ee then after pulling it, it will point to the Di'er pulled along the Di'ee.
- The cases of $x \sqsubseteq Mu$ where $x \in a$ (49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60) do not require special considerations regarding conditions.
- For cases of $x \sqsubseteq y$ where $x \in a$ and $y \in \{Fw, Rw\}$, the required condition in the cases of $x \in \{Ti, Ab\}$ (61, 62, 63, 64) holds because pointing downwards into a memory is stable under "loosening" a message within the memory ($v \leq \epsilon$). The case of $x \in Di$ (65, 66) hold thanks to Lemma 6.5, 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 Di'ee then after pulling it, it will point to the Di'er pulled along the Di'ee.



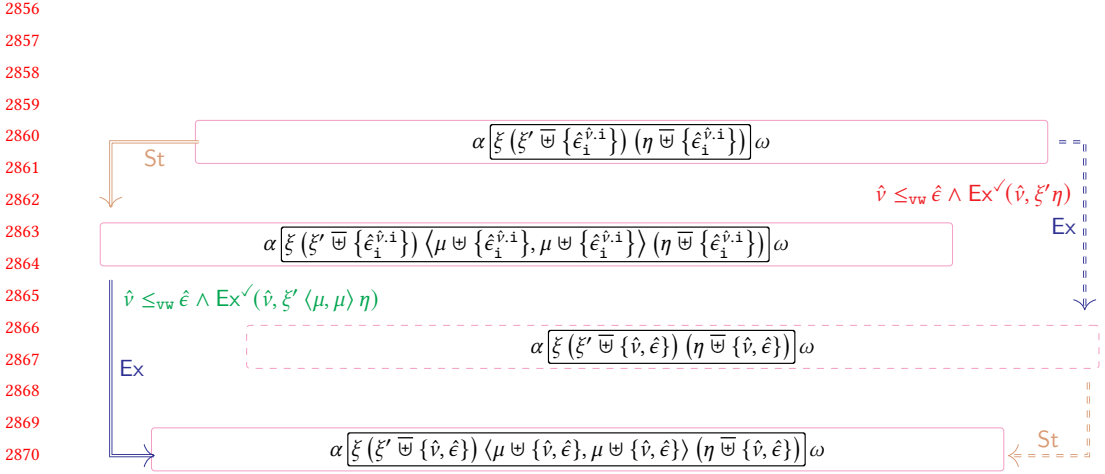
1. The $St \sqsubseteq Ls$ case when the Ls' ee does not appear across the St' ee



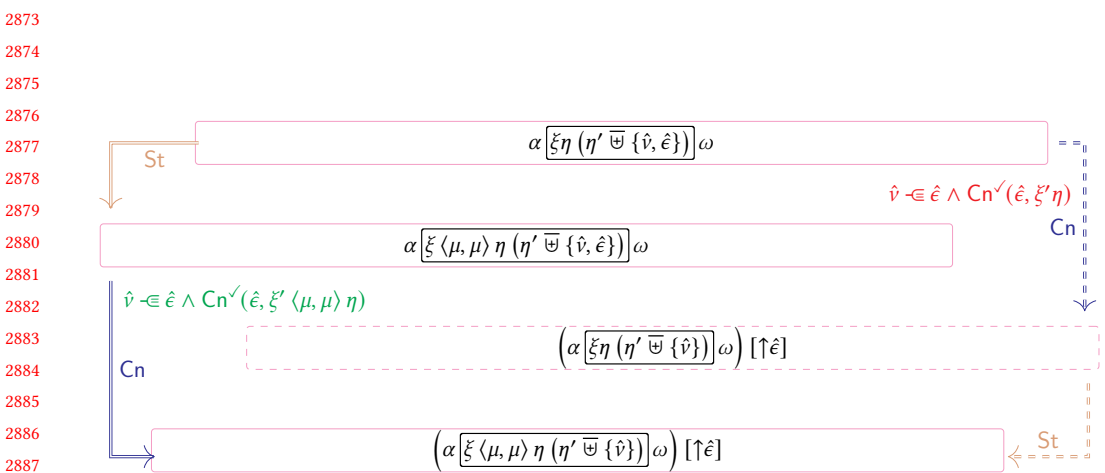
2. The $St \sqsubseteq Ls$ case when the Ls' ee appears across the St' ee



3. The $\text{St} \rightleftharpoons \text{Ex}$ case when the Ex'ee does not appear across the St'ee

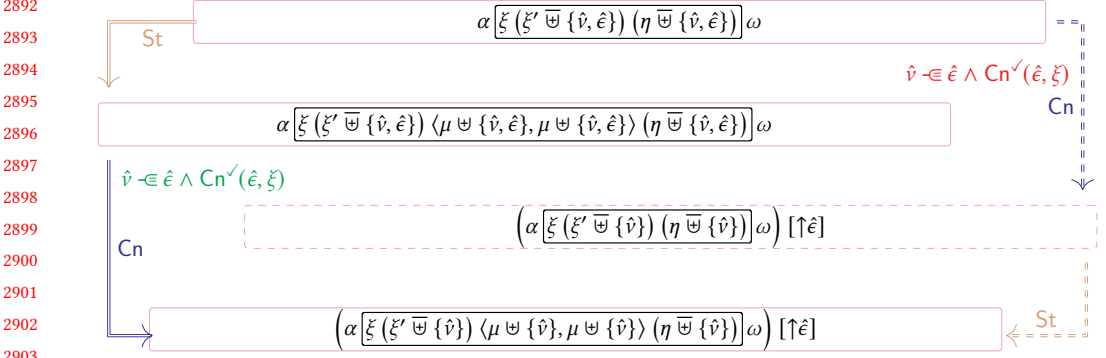


4. The $\text{St} \rightleftharpoons \text{Ex}$ case when the Ex'ee appears across the St'ee

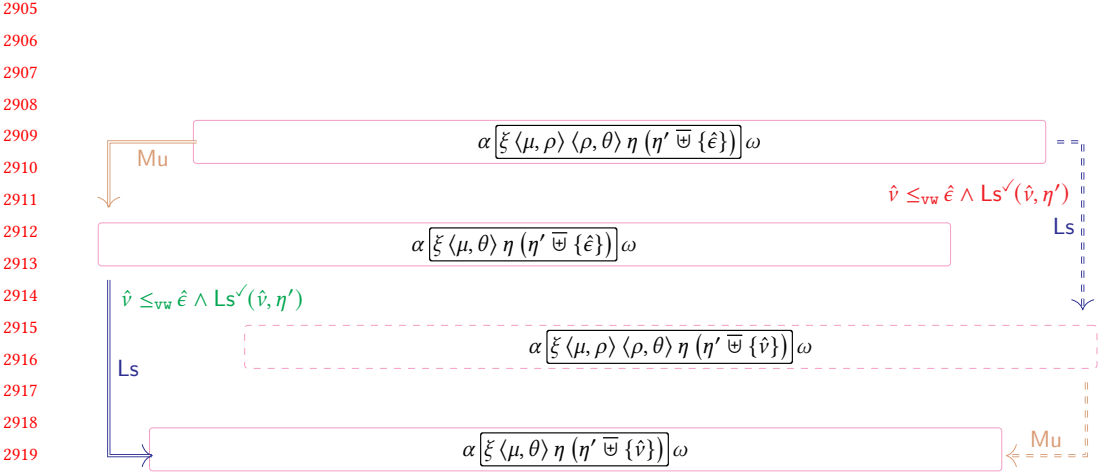


5. The $\text{St} \rightleftharpoons \text{Cn}$ case when the Cn'ee does not appear across the St'ee

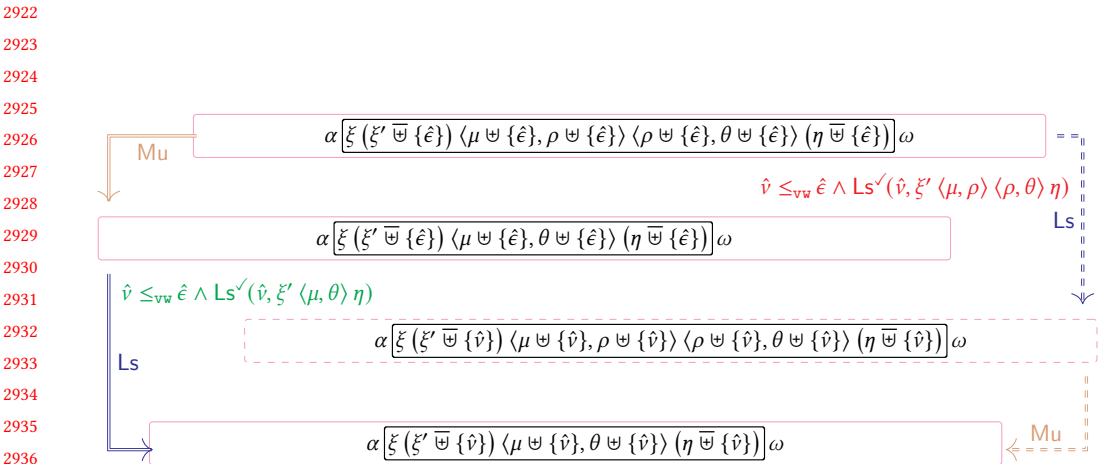




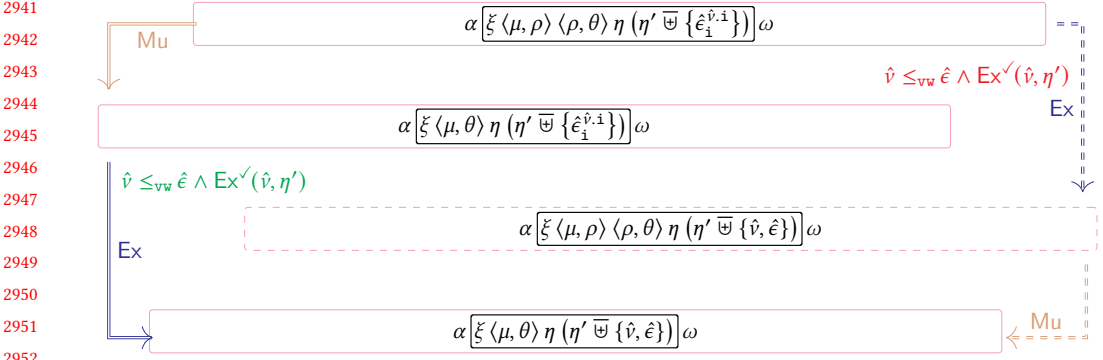
6. The $\text{St} \Leftrightarrow \text{Cn}$ case when the Cn' ee appears across the St' ee



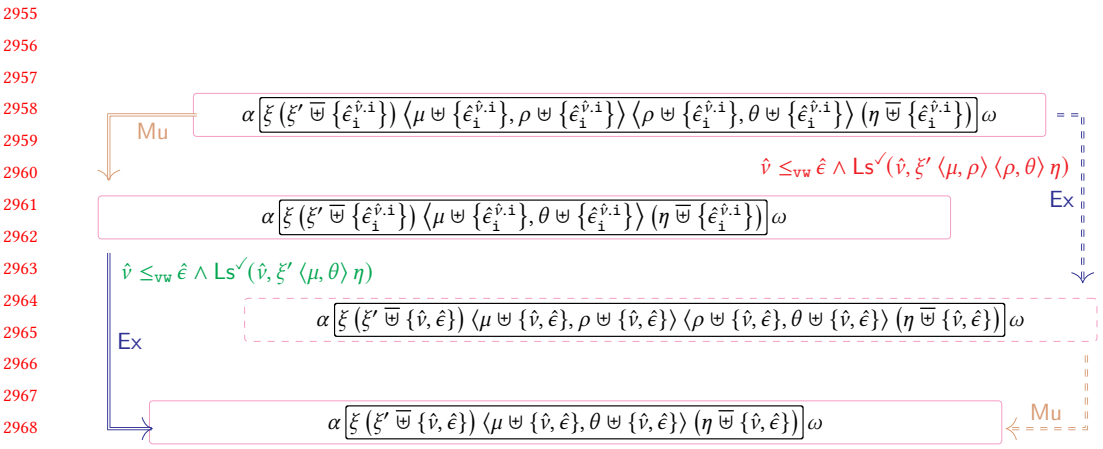
7. The $\text{Mu} \Leftrightarrow \text{Ls}$ case when the Ls' ee does not appear across the Mu' ee



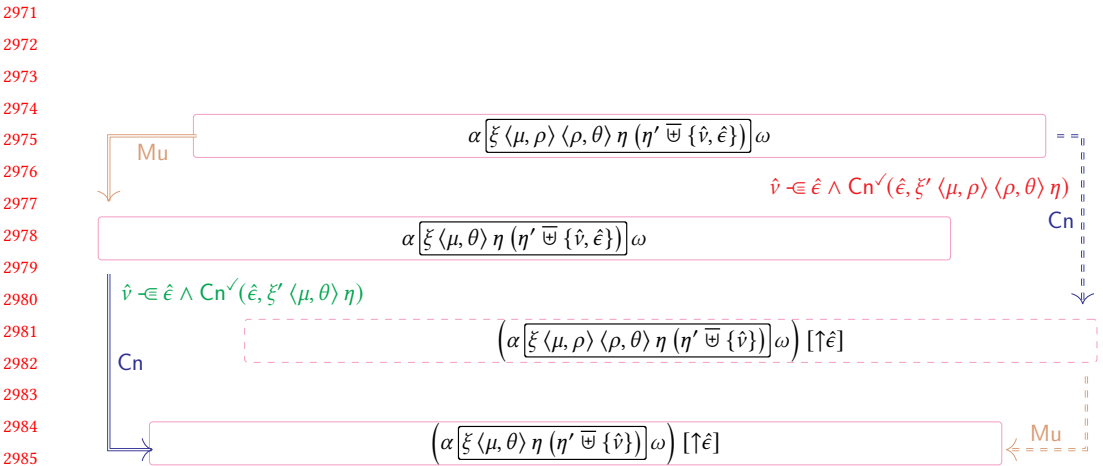
8. The $\text{Mu} \Leftrightarrow \text{Ls}$ case when the Ls' ee appears across the Mu' ee



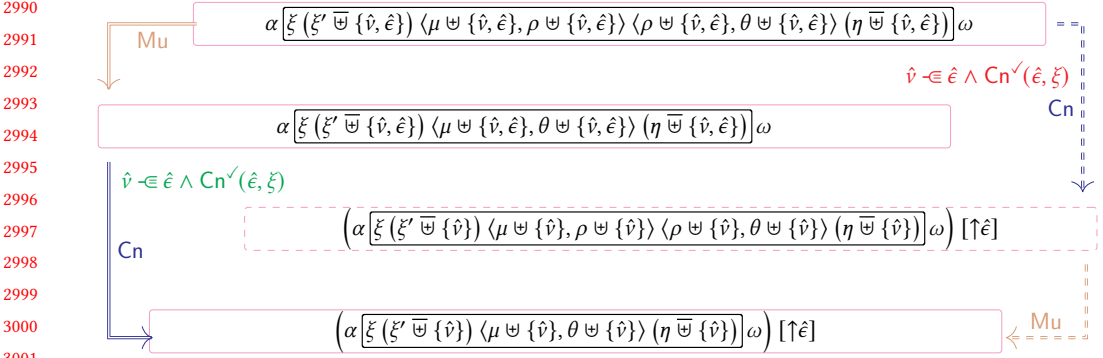
9. The $\text{Mu} \rightleftharpoons \text{Ex}$ case when the Ex 'ee does not appear across the Mu 'ee



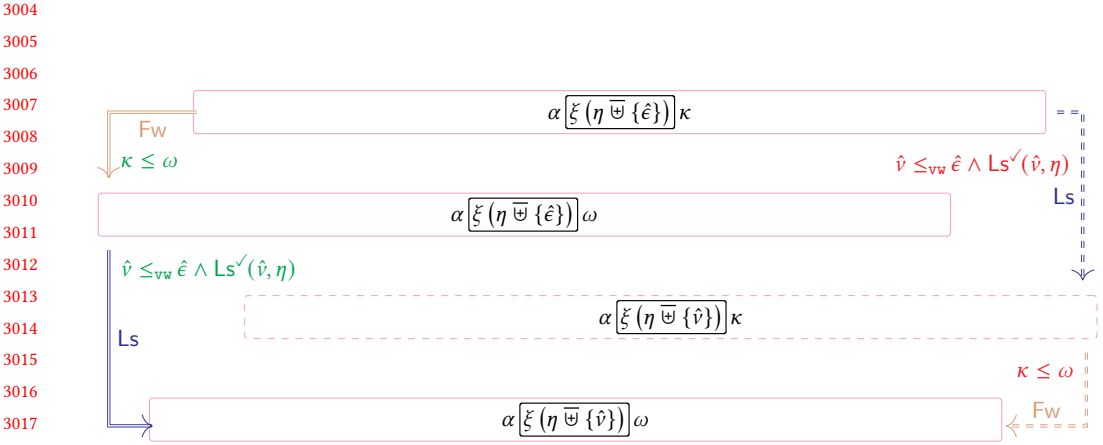
10. The $\text{Mu} \rightleftharpoons \text{Ex}$ case when the Ex 'ee appears across the Mu 'ee



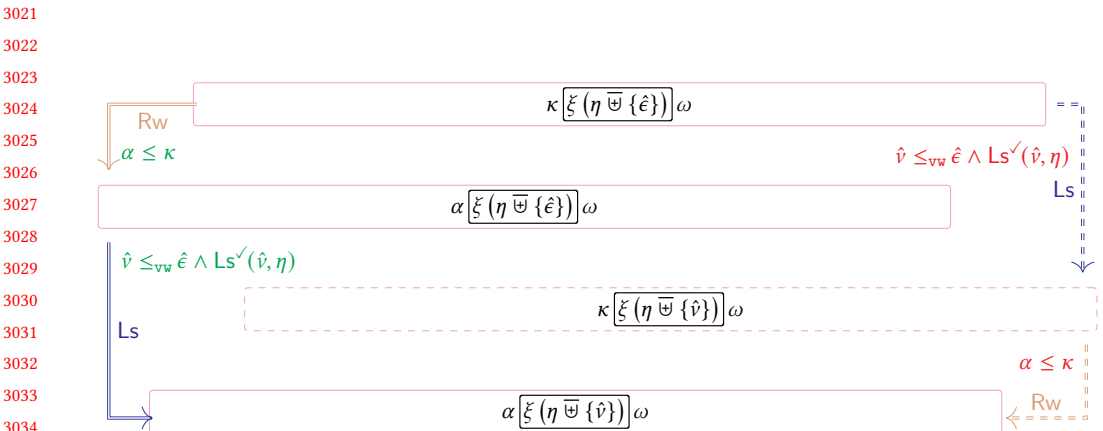
11. The $\text{Mu} \rightleftharpoons \text{Cn}$ case when the Cn 'ee does not appear across the Mu 'ee



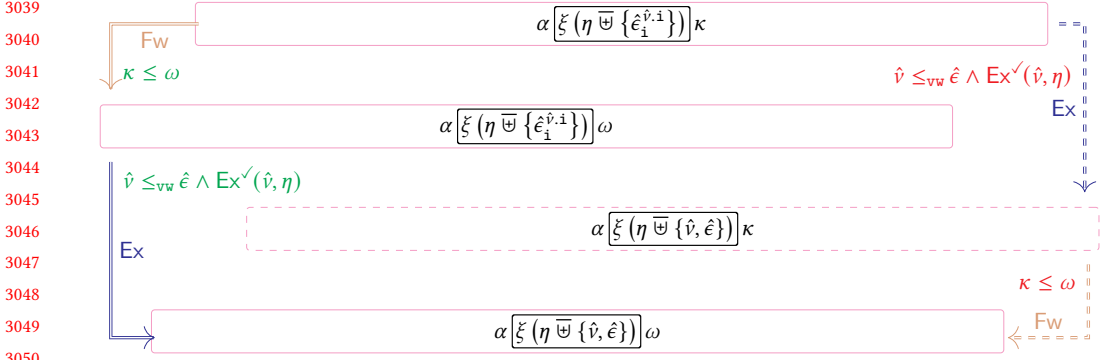
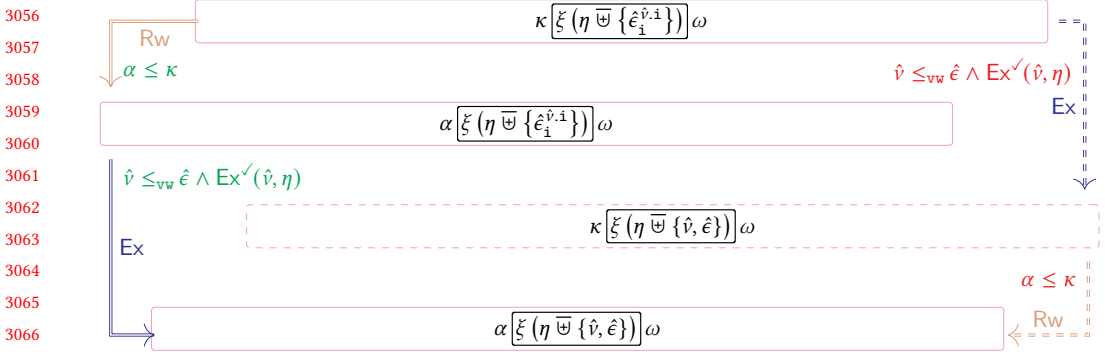
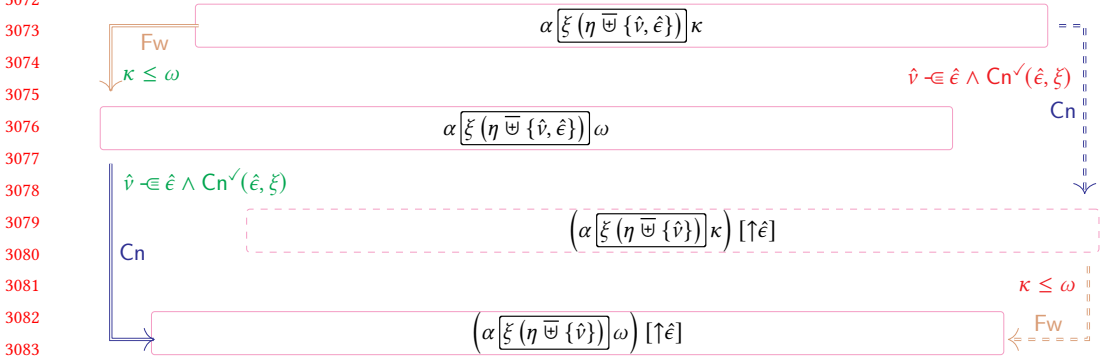
12. The $\text{Mu} \Leftrightarrow \text{Cn}$ case when the Cn 'ee appears across the Mu 'ee

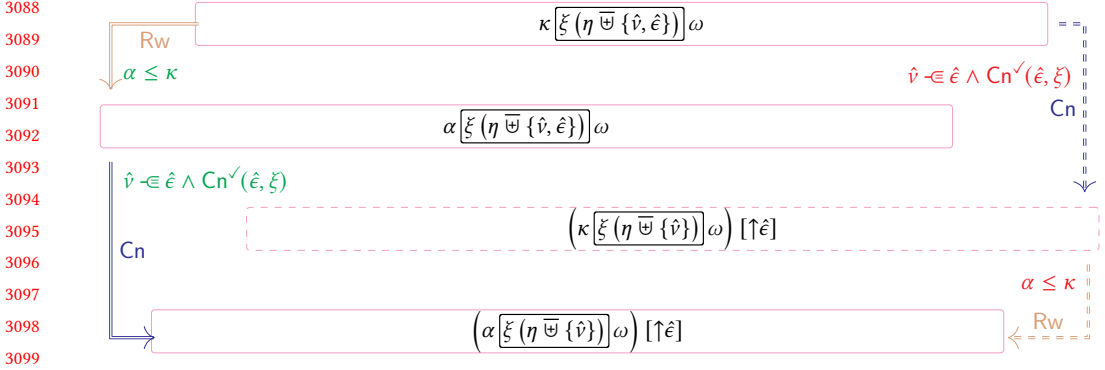
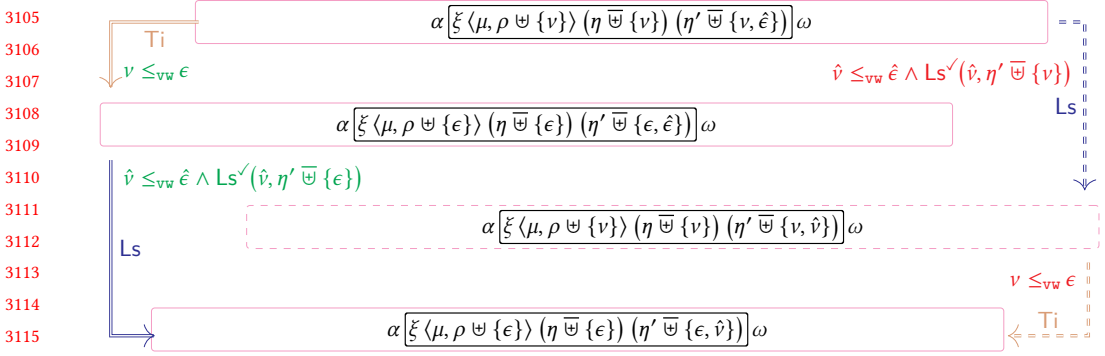
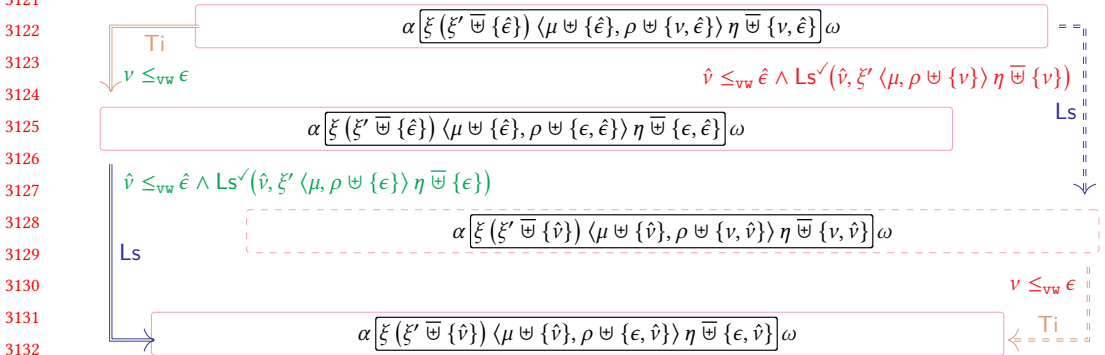


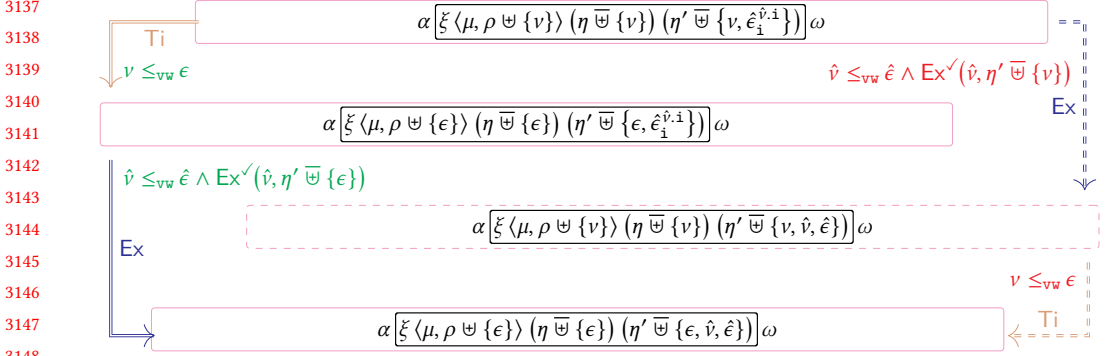
13. The $\text{Fw} \Leftrightarrow \text{Ls}$ case



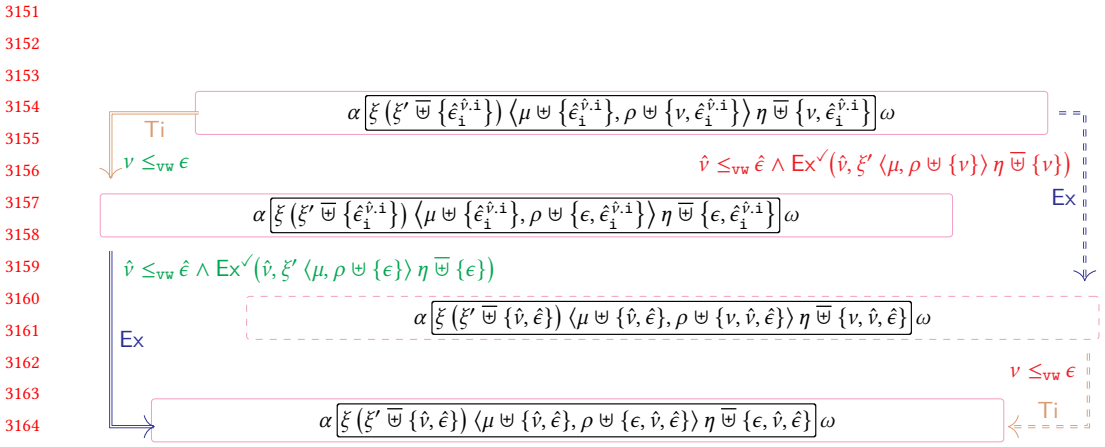
14. The $\text{Rw} \Leftrightarrow \text{Ls}$ case


 15. The Fw \Leftrightarrow Ex case

 16. The Rw \Leftrightarrow Ex case

 17. The Fw \Leftrightarrow Cn case

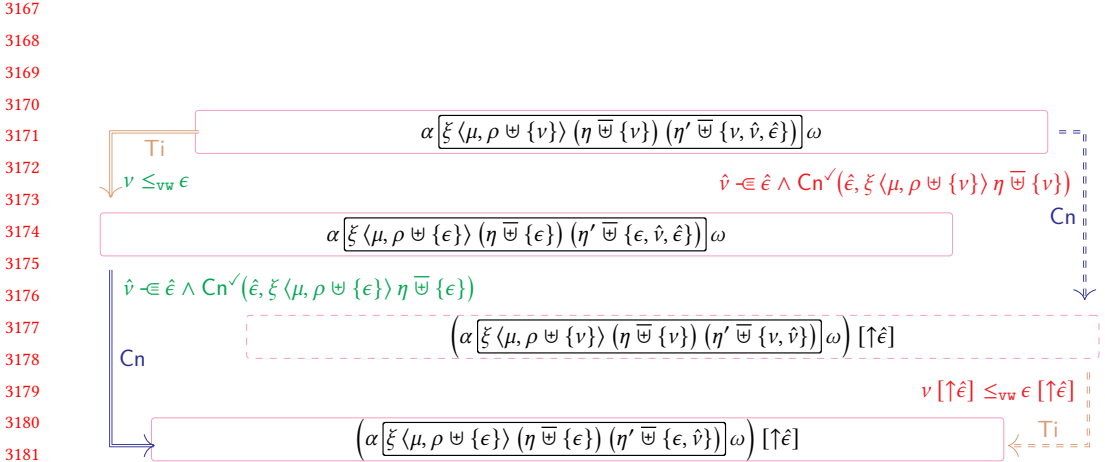

 18. The $\text{Rw} \Leftrightarrow \text{Cn}$ case

 19. The $\text{Ti} \Leftrightarrow \text{Ls}$ case when the $\text{Ls}'\text{ee}$ appears after the $\text{Ti}'\text{ee}$

 20. The $\text{Ti} \Leftrightarrow \text{Ls}$ case when the $\text{Ls}'\text{ee}$ appears before the $\text{Ti}'\text{ee}$



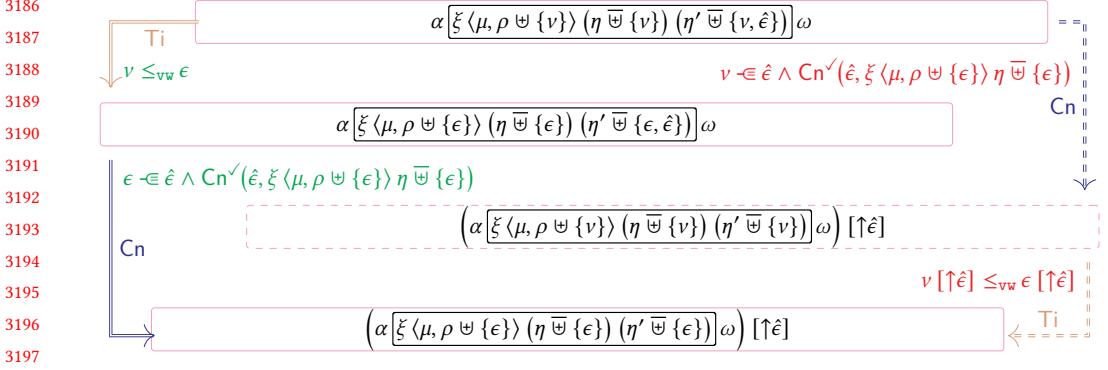
21. The $Ti \rightleftharpoons Ex$ case when the Ex'ee appears after the Ti'ee



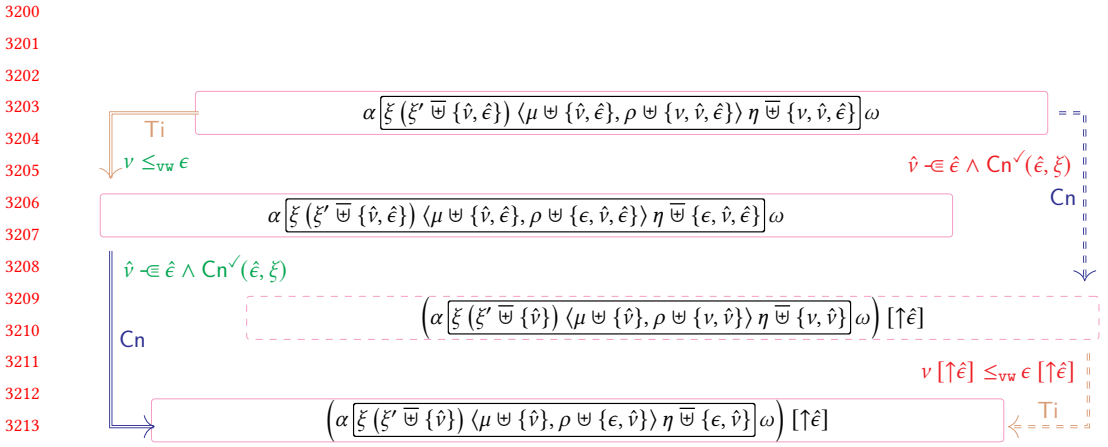
22. The $Ti \rightleftharpoons Ex$ case when the Ex'ee appears before the Ti'ee



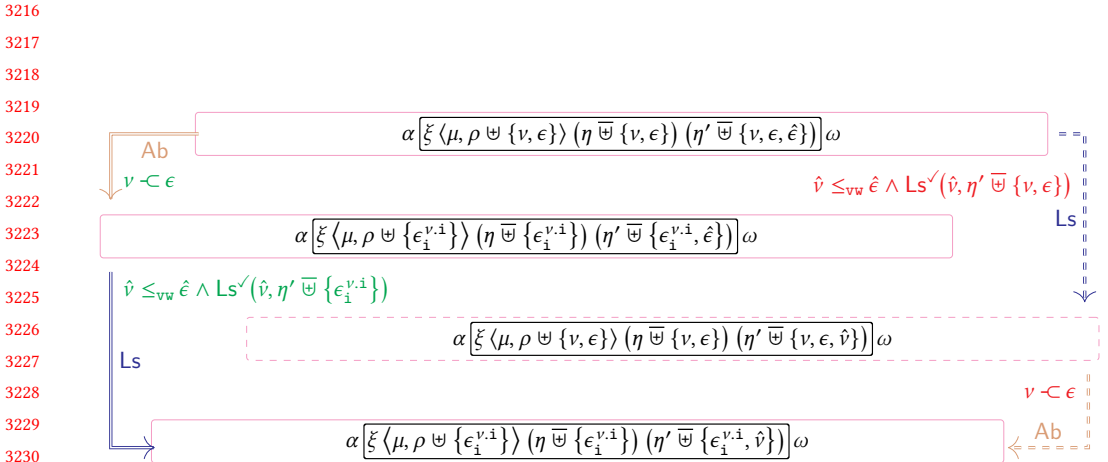
23. The $Ti \rightleftharpoons Cn$ case when the Cn'ee appears after the Ti'ee, and the Ti'ee is not the Cn'er



24. The $\text{Ti} \Leftrightarrow \text{Cn}$ case when the Cn' 'ee appears after the Ti' 'ee, and the Ti' 'ee is the Cn' 'er

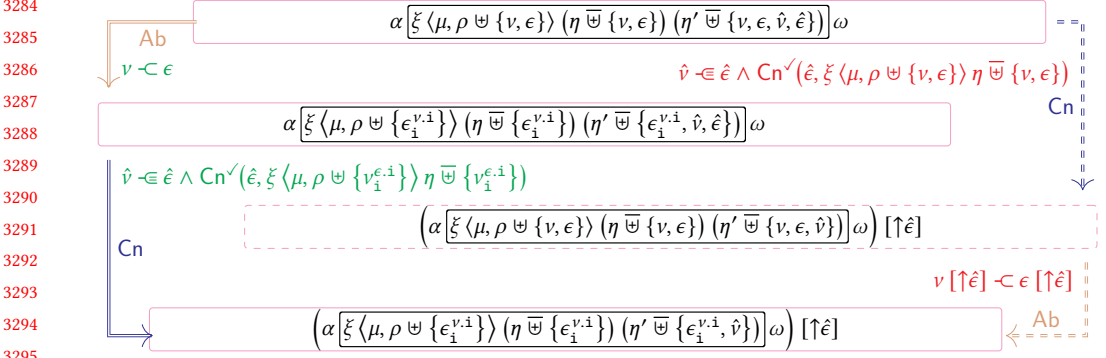


25. The $\text{Ti} \Leftrightarrow \text{Cn}$ case when the Cn' 'ee appears before the Ti' 'ee

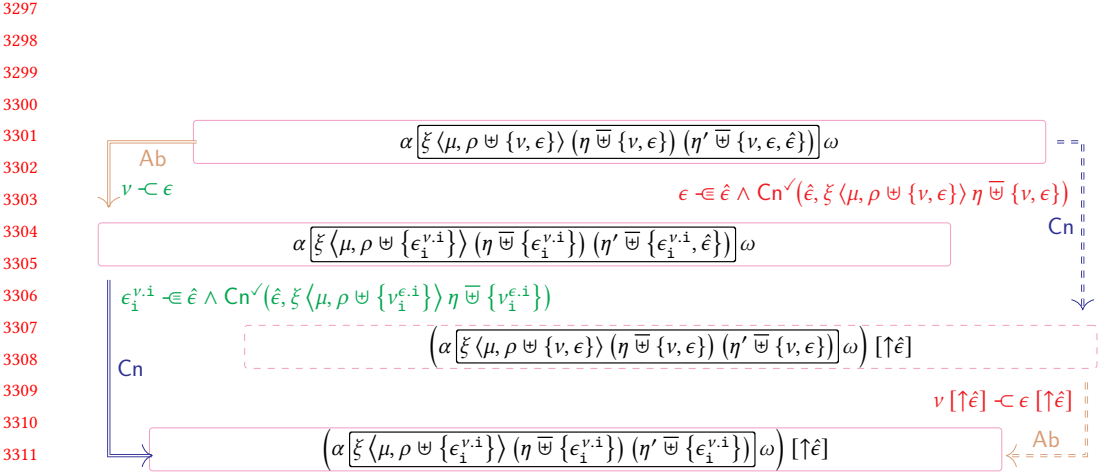


26. The $\text{Ab} \Leftrightarrow \text{Ls}$ case when the Ls' 'ee appears after the Ab' 'ee

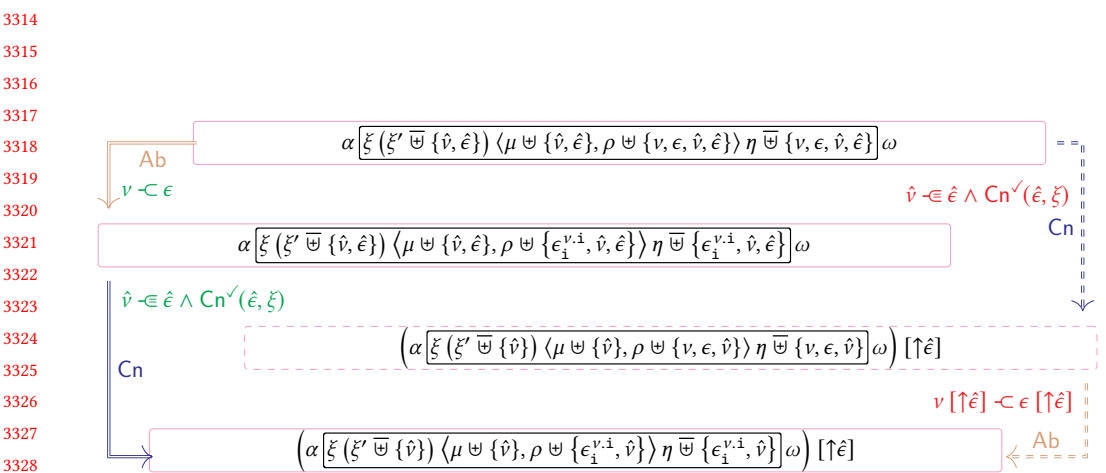
3231
3232
3233
3234



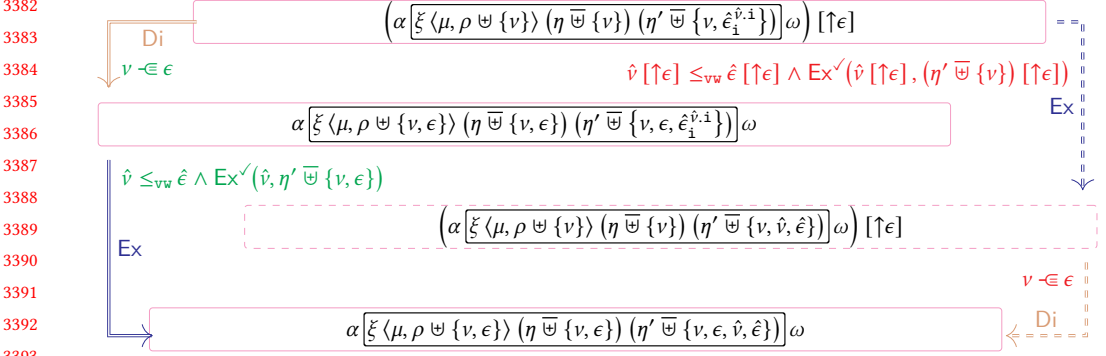
30. The $\text{Ab} \rightleftharpoons \text{Cn}$ case when the Cn 'ee appears after the Ab 'er, and the Ab 'er is not the Cn 'er



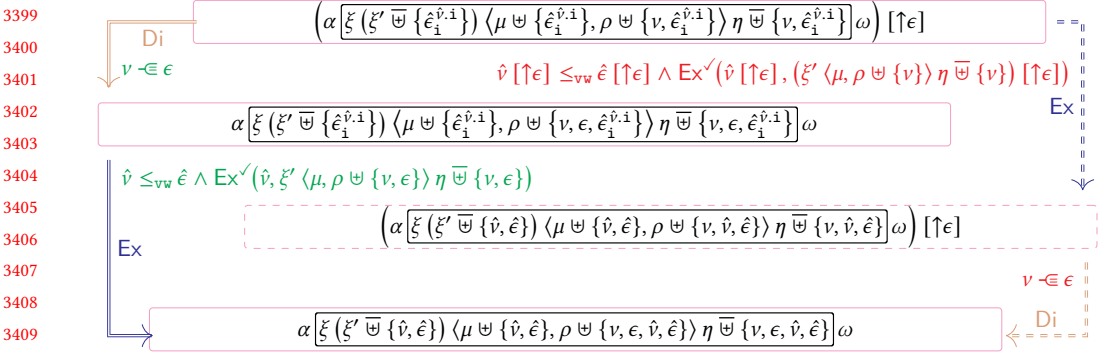
31. The $\text{Ab} \rightleftharpoons \text{Cn}$ case when the Cn 'ee appears after the Ab 'er, and the Ab 'er is the Cn 'er



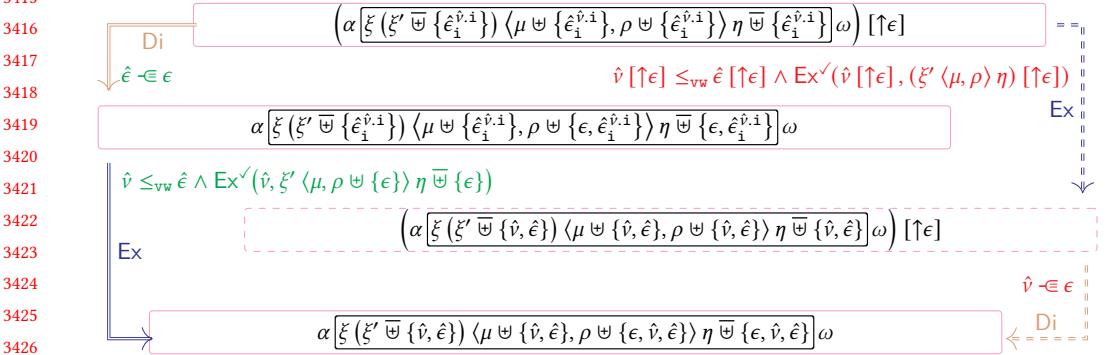
32. The $\text{Ab} \rightleftharpoons \text{Cn}$ case when the Cn 'ee appears before the Ab 'er



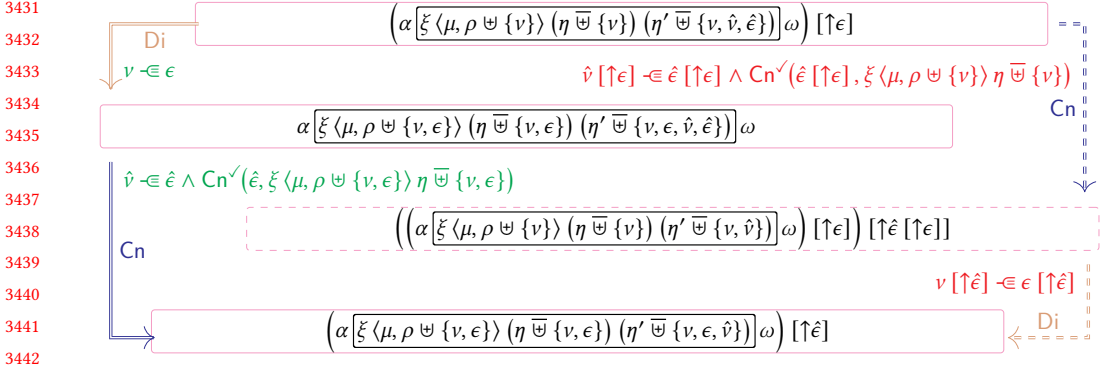
36. The $\text{Di} \Leftrightarrow \text{Ex}$ case when the Ex 'er appears after the Di 'ee



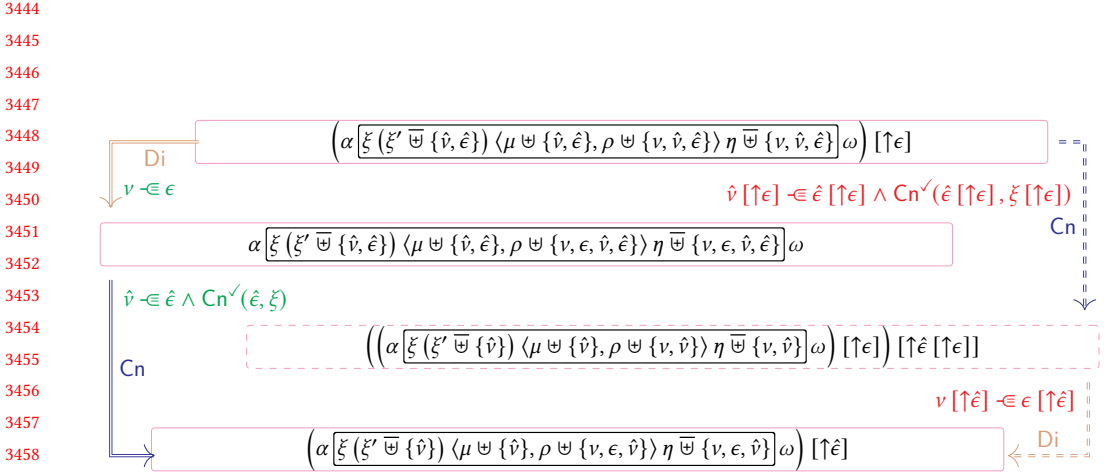
37. The $\text{Di} \Leftrightarrow \text{Ex}$ case when the Ex 'er appears before the Di 'ee, and the Di 'er is not the Ex 'er



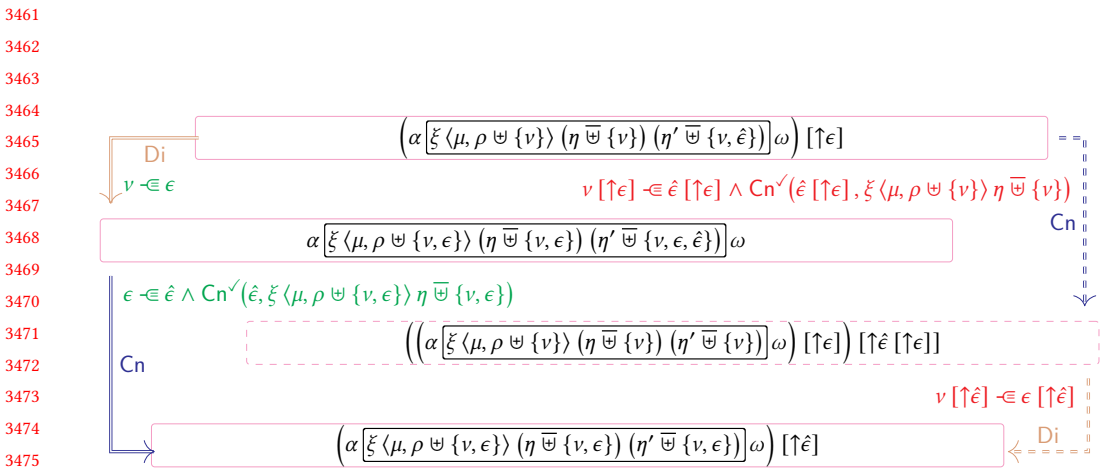
38. The $\text{Di} \Leftrightarrow \text{Ex}$ case when the Ex 'er appears before the Di 'ee, and the Di 'er is the Ex 'er



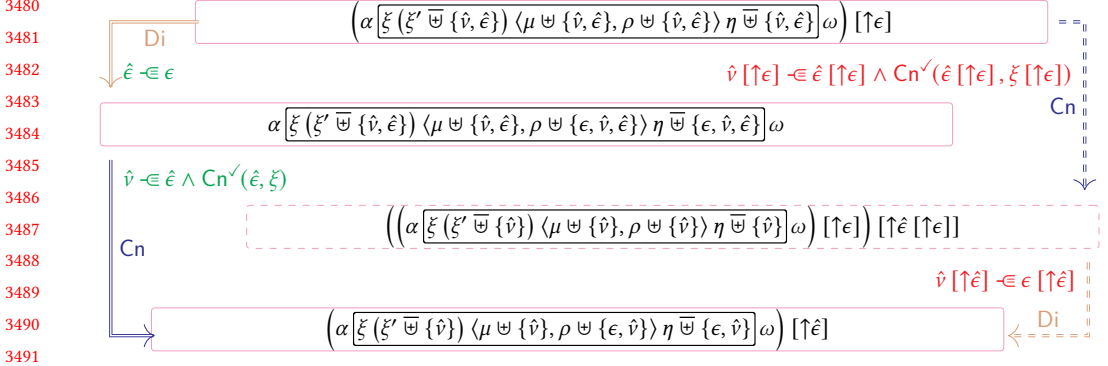
39. The $Di \Leftrightarrow Cn$ case when the Cn 'ee appears after the Di 'ee, and the Di 'ee is not the Cn 'er



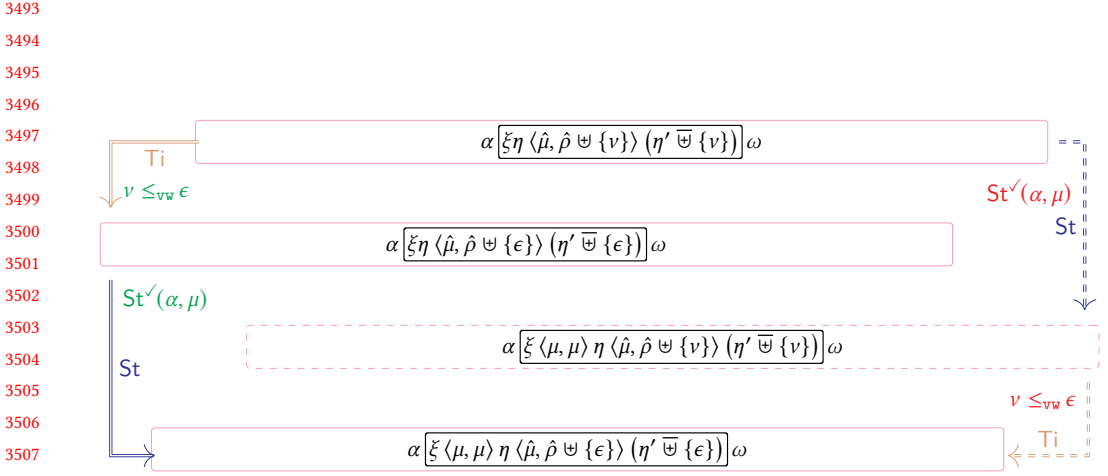
40. The $Di \Leftrightarrow Cn$ case when the Cn 'ee appears before the Di 'ee, and the Di 'er is not the Cn 'ee



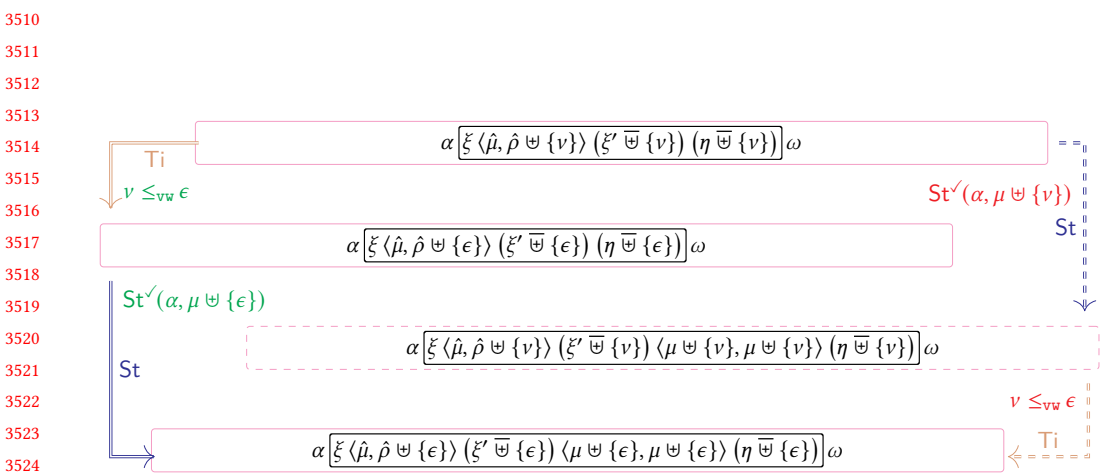
41. The $Di \Leftrightarrow Cn$ case when the Cn 'ee appears after the Di 'ee, and the Di 'ee is the Cn 'er



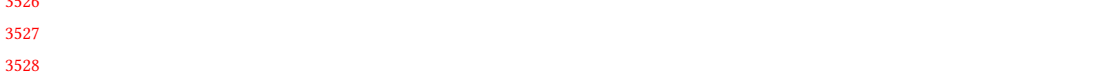
42. The $\text{Di} \rightleftharpoons \text{Cn}$ case when the Cn 'ee appears before the Di 'ee, and the Di 'er is the Cn 'ee

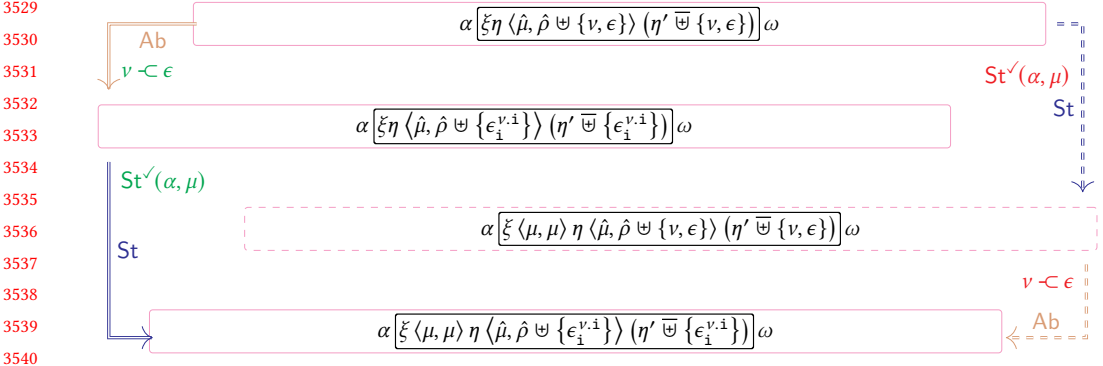


43. The $\text{Ti} \rightleftharpoons \text{St}$ case when the Ti 'ee does not appear across the St 'ee

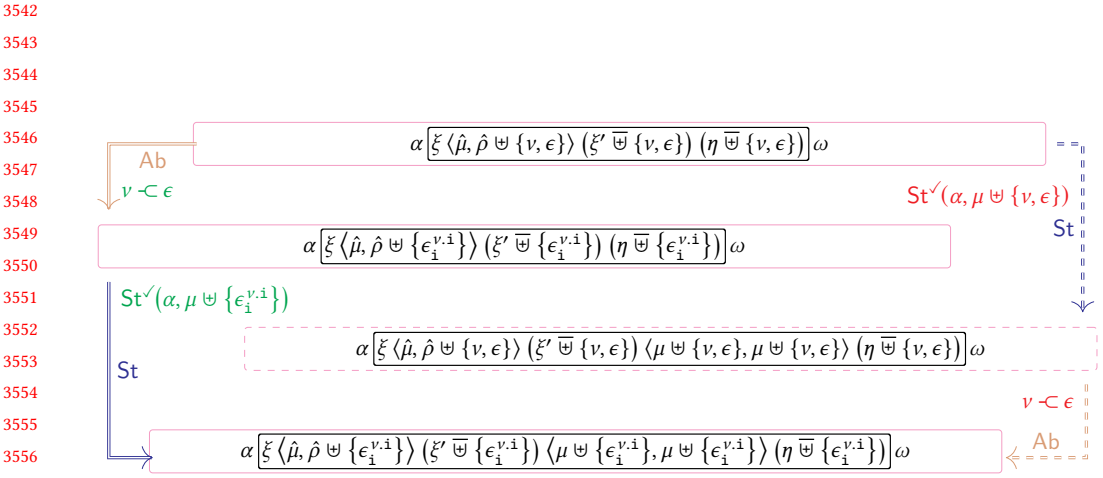


44. The $\text{Ti} \rightleftharpoons \text{St}$ case when the Ti 'ee appears across the St 'ee

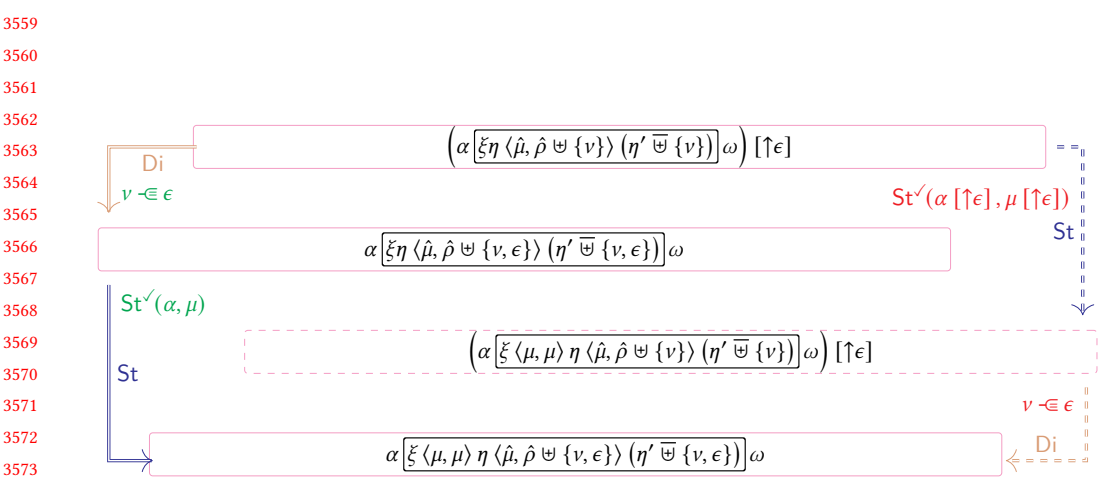




45. The Ab \Leftrightarrow St case when the Ab'ee does not appear across the St'ee

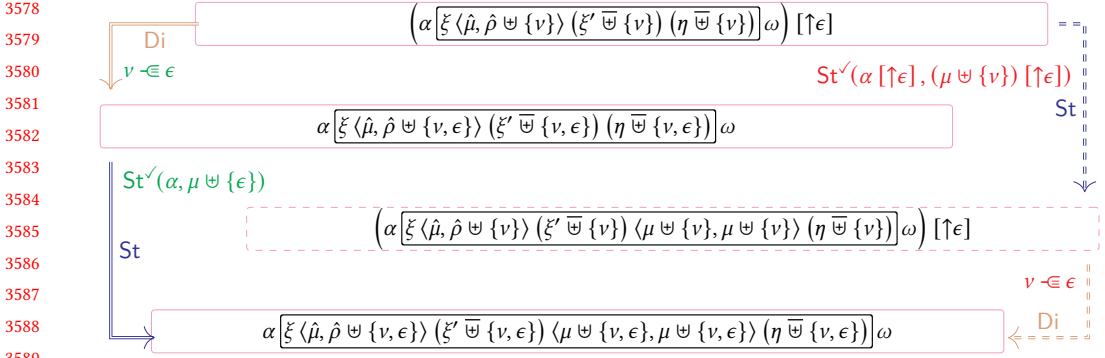


46. The Ab \Leftrightarrow St case when the Ab'ee appears across the St'ee

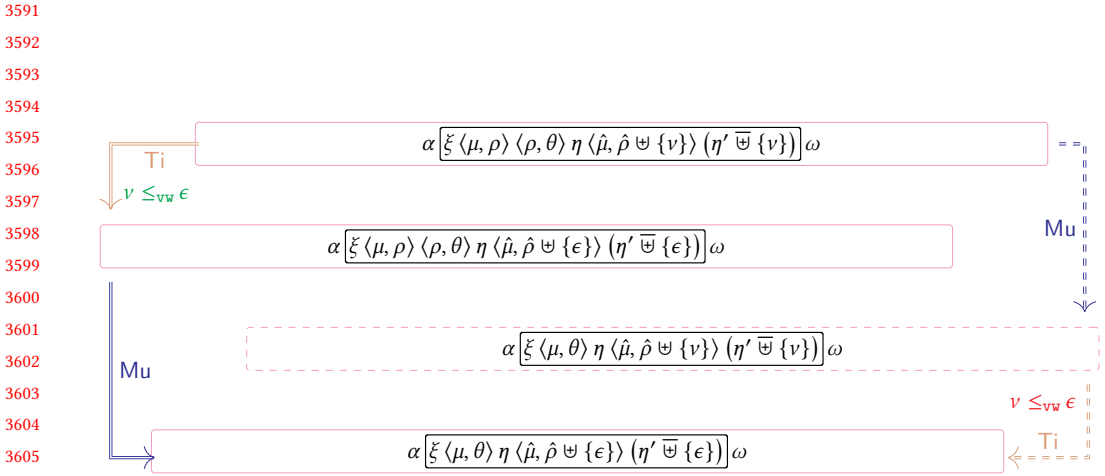


47. The Di \Leftrightarrow St case when the Di'ee does not appear across the St'ee

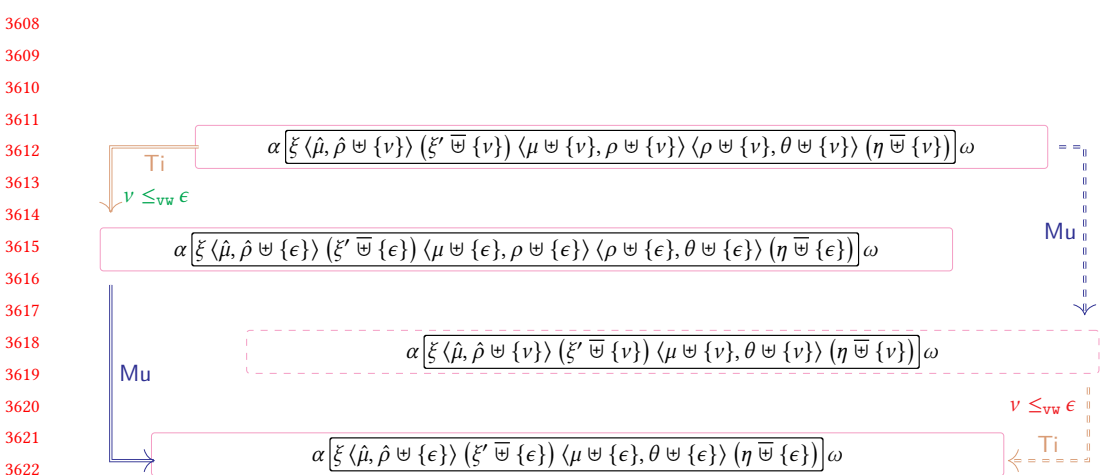




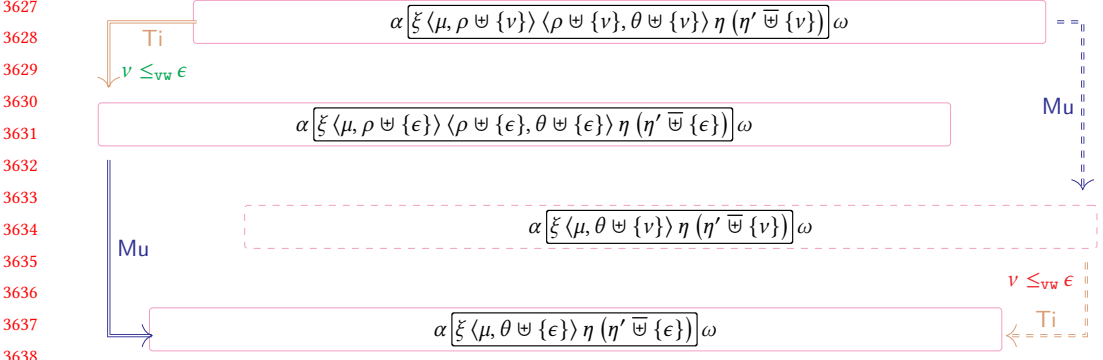
48. The $Di \Leftrightarrow St$ case when the Di'ee appears across the St'ee



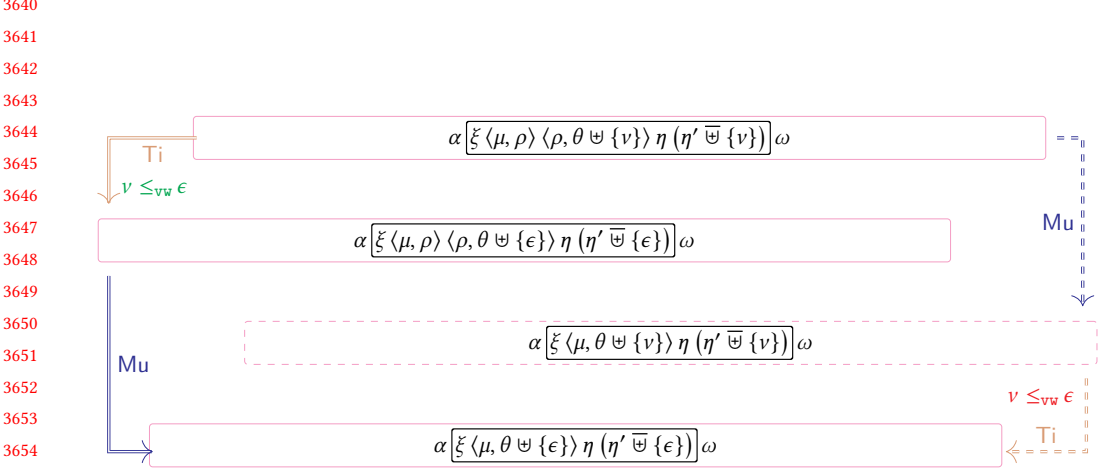
49. The $Ti \Leftrightarrow Mu$ case when the Ti'ee does not appear across the Mu'ee



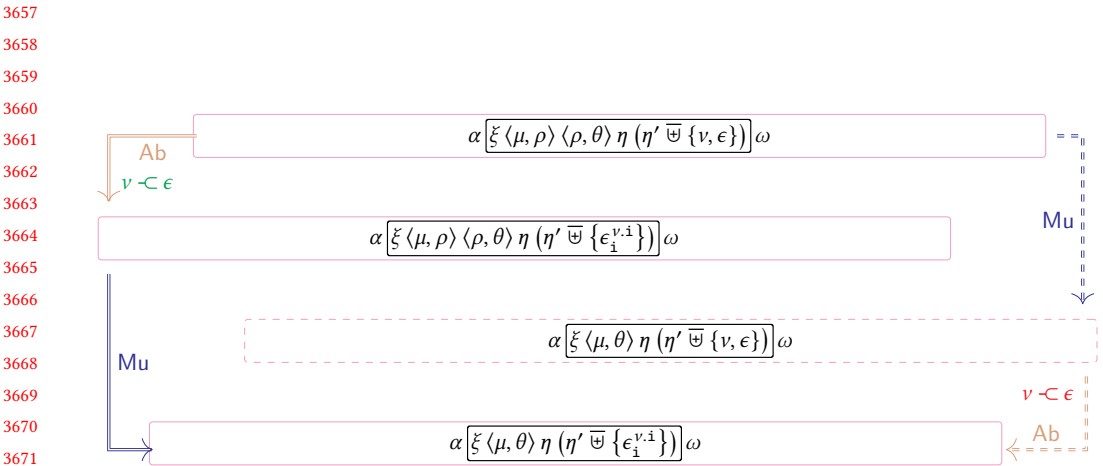
50. The $Ti \Leftrightarrow Mu$ case when the Ti'ee appears across the Mu'ee



51. The $Ti \rightleftharpoons Mu$ case when the Ti 'ee makes its appearance in the first Mu 'er

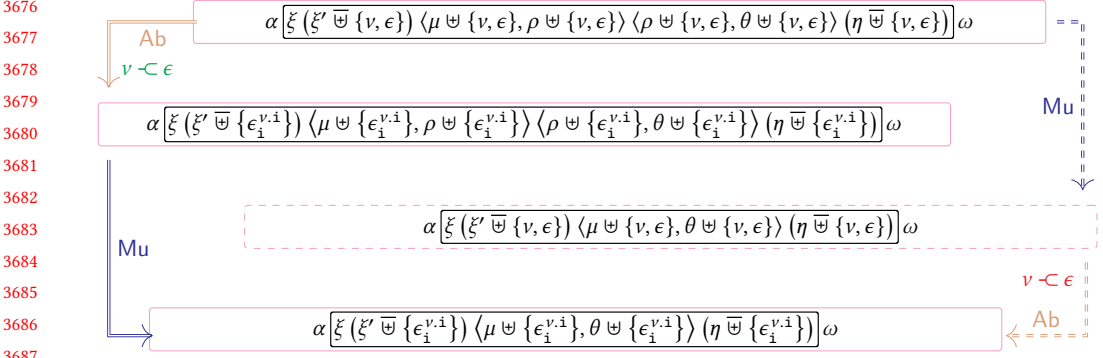


52. The $Ti \rightleftharpoons Mu$ case when the Ti 'ee makes its appearance in the second Mu 'er

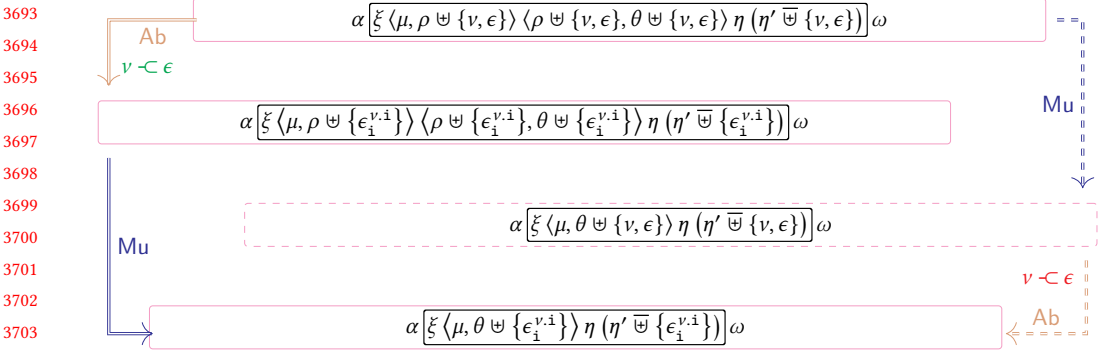


53. The $Ab \rightleftharpoons Mu$ case when the Ab 'ee does not appear across the Mu 'ee

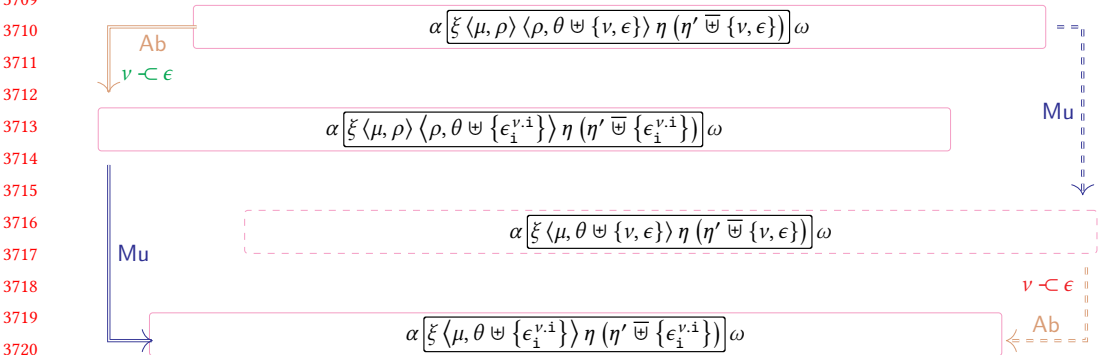
3672
3673
3674
3675



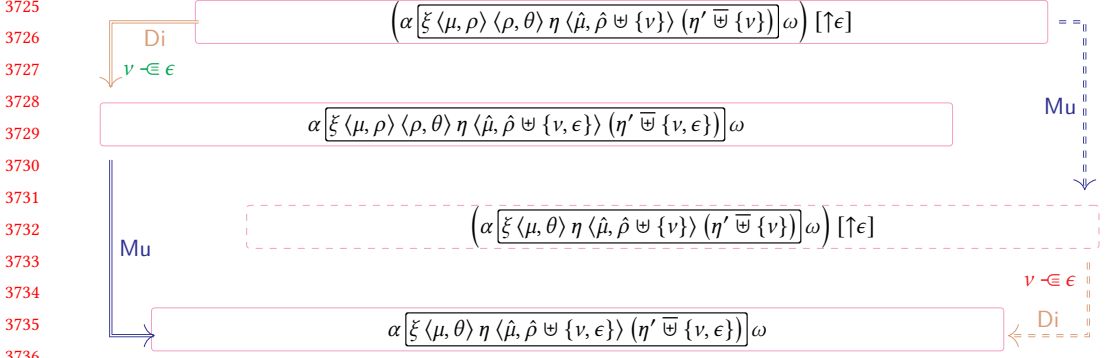
54. The Ab \Leftrightarrow Mu case when the Ab'ee appears across the Mu'ee



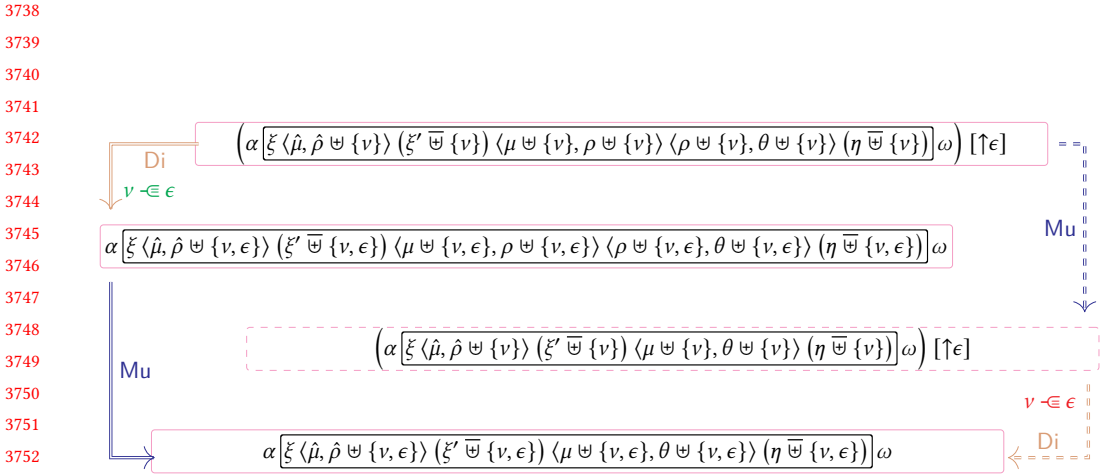
55. The Ab \Leftrightarrow Mu case when the Ab'ee makes its appearance in the first Mu'er



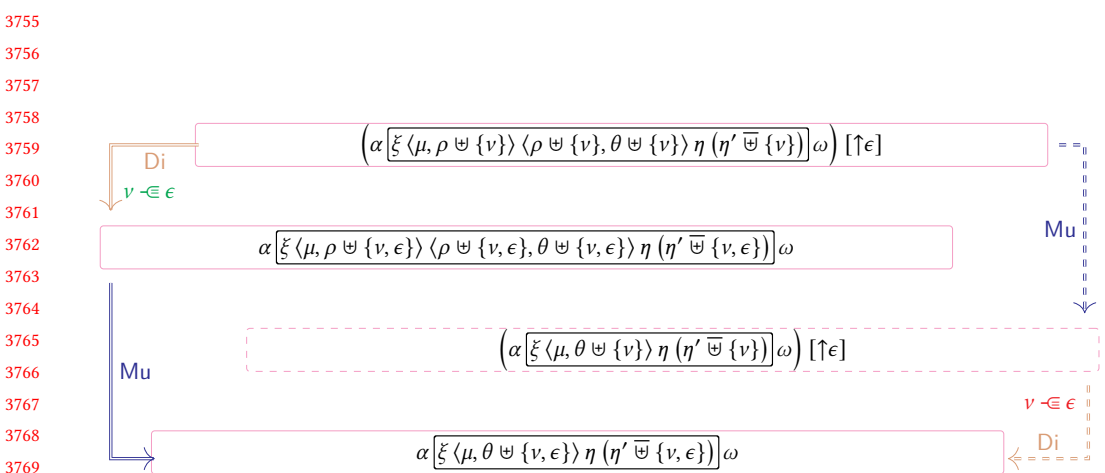
56. The Ab \Leftrightarrow Mu case when the Ab'ee makes its appearance in the second Mu'er



57. The $\text{Di} \Leftrightarrow \text{Mu}$ case when the Di 'ee does not appear across the Mu 'ee



58. The $\text{Di} \Leftrightarrow \text{Mu}$ case when the Di 'ee appears across the Mu 'ee



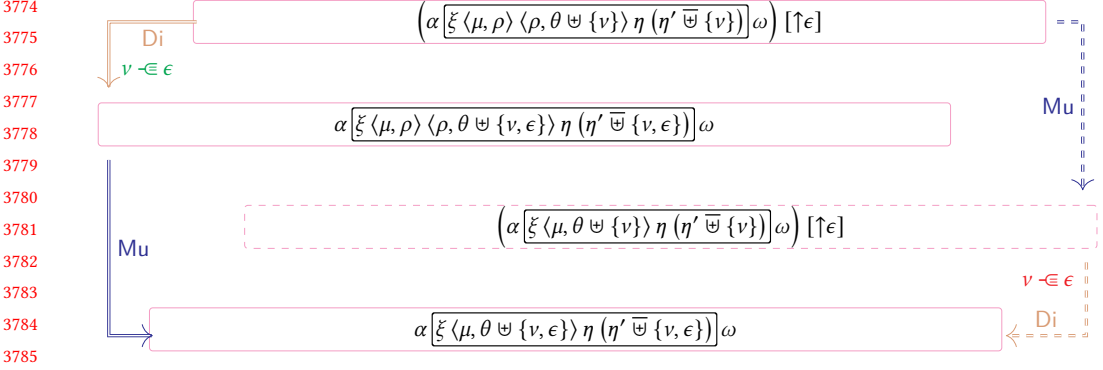
59. The $\text{Di} \Leftrightarrow \text{Mu}$ case when the Di 'ee makes its appearance in the first Mu 'ee

3770

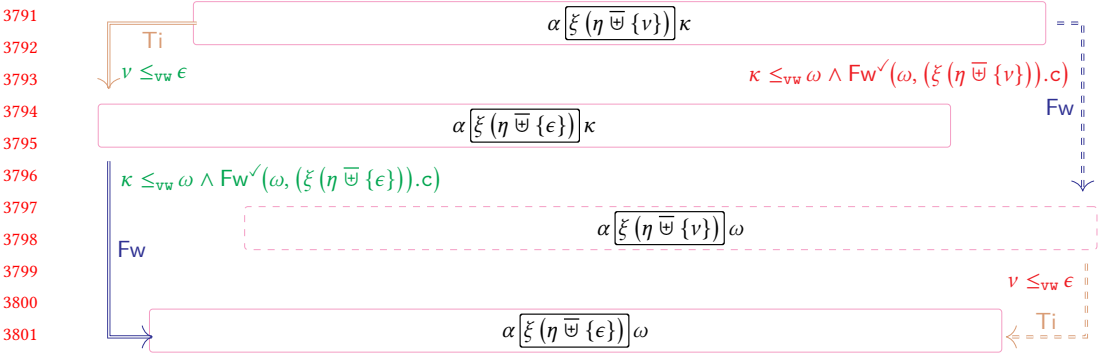
3771

3772

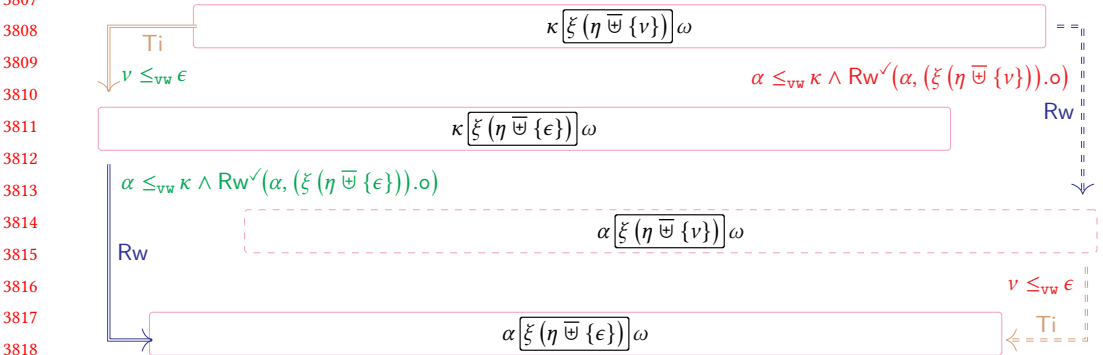
3773



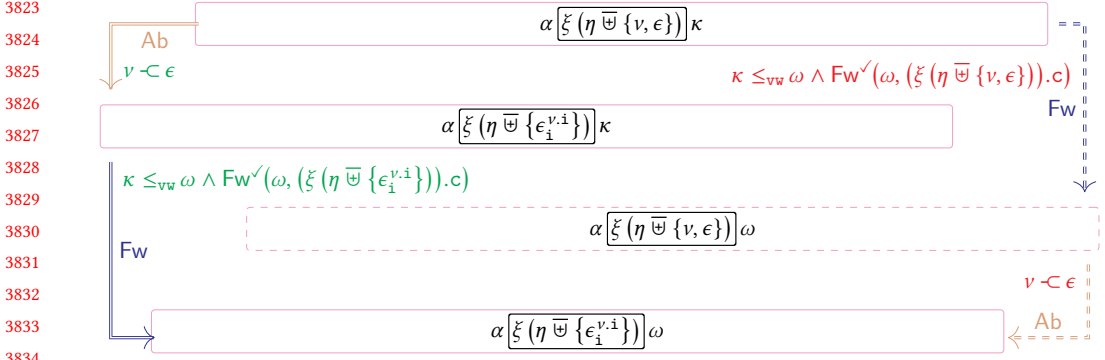
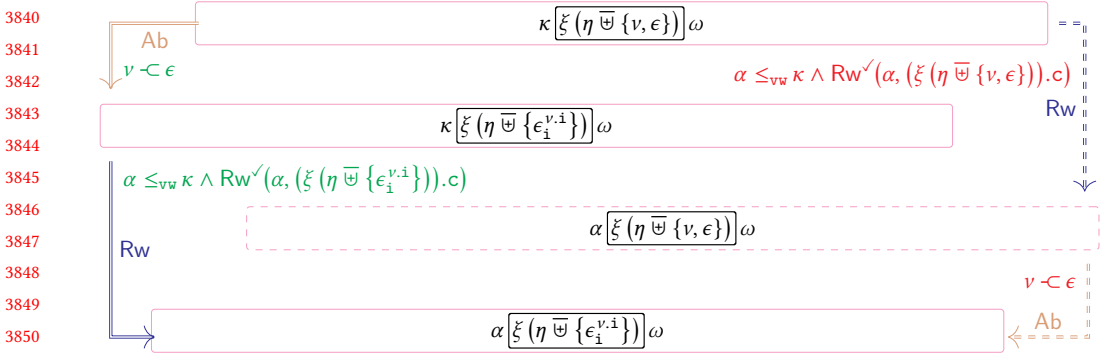
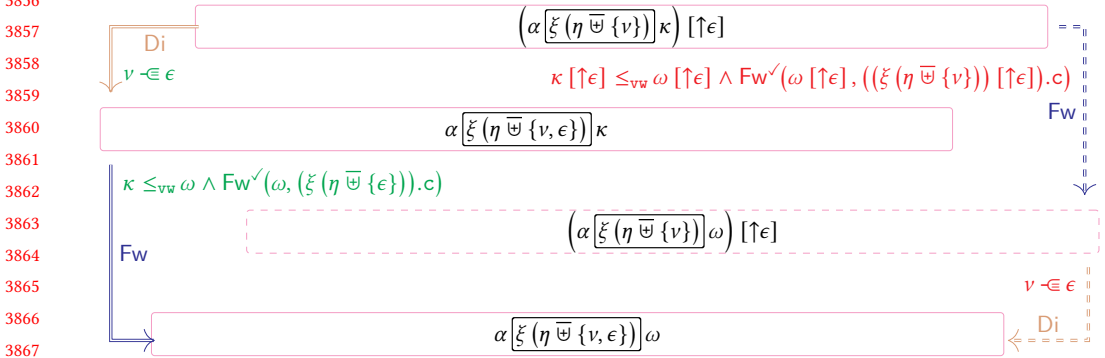
60. The Di \Leftrightarrow Mu case when the Di'ee makes its appearance in the second Mu'er



61. The Ti \Leftrightarrow Fw case

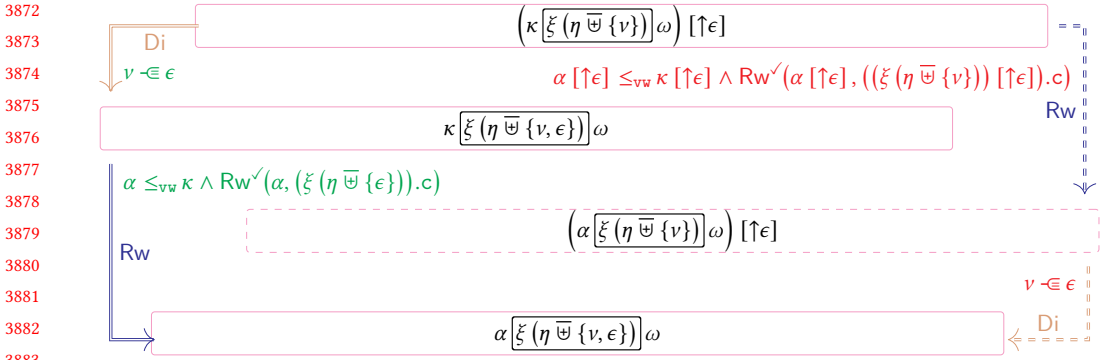


62. The Ti \Leftrightarrow Rw case


 63. The Ab \Leftrightarrow Fw case

 64. The Ab \Leftrightarrow Rw case

 65. The Di \Leftrightarrow Fw case

3870

3871



66. The $\text{Di} \Leftrightarrow \text{Rw}$ case

3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920