A Denotational Approach to Release/Acquire Concurrency

Yotam Dvir[1], Ohad Kammar[2], and Ori Lahav[1]

[1] Tel Aviv University yotamdvir@mail.tau.ac.il orilahav@tau.ac.il
[2] University of Edinburgh ohad.kammar@ed.ac.uk

Abstract. 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.

Keywords: Weak memory models · Release/Acquire · Shared state · Shared memory · Concurrency · Denotational semantics · Monads · Program refinement · Program equivalence · Compiler optimizations

1 Introduction

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

ordered traces of M closed under certain operations, called stutter and mumble. Traces consist of sequences of memory snapshots that M guarantees to provide while relying on its environment to make other memory snapshots. Brookes [12] used the insights behind this semantics to develop a semantic model for separation logic, and Turon and Wand [46] used them to design a separation logic for refinement. Additionally, Xu et al. [48] used traces as a foundation for the Rely/Guarantee approach for verification of concurrent programs, and Liang et al., Liang et al. [34, 35] used a trace-based program logic for refinement.

A *memory model* decides what outcomes are possible from the execution of a program. Brookes established the adequacy of the trace-based denotational semantics w.r.t. the operational semantics of the strongest model, known as *sequential consistency* (SC), where every memory access happens instantaneously and immediately affects all concurrent threads. However, SC is too strong to model real-world shared memory, whether it be of modern hardware, such as x86-TSO [40, 44] and ARM, or of programming languages such as C/C++ and Java [4, 37]. These runtimes follow *weak memory models* that allow performant implementations, but admit more behaviors than SC.

Do weak memory models admit adequate Brookes-style denotational semantics? This question has been answered affirmatively once, by Jagadeesan et al. [25], who closely followed Brookes to define denotational semantics for x86-TSO. Other weak memory models, in particular, models of *programming languages*, and *non-multi-copy-atomic* models, where writes can be observed by different threads in different orders, have so far been out of reach of Brookes's totally ordered traces, and were only captured by much more sophisticated models based on *partial orders* [15, 19, 24, 26, 28, 41].

In this paper we target the Release/Acquire memory model (RA, for short). This model, obtained by restricting the C/C++11 memory model to Release/Acquire atomics, is a well-studied fundamental memory model weaker than x86-TSO, which, roughly speaking, ensures "causal consistency" together with "per-location-SC" and "RMW (read-modify-write) atomicity" [29, 30]. These assurances make RA sufficiently strong for implementing common synchronization idioms. RA allows more performant implementations than SC, since, in particular, it allows the reordering of a write followed by a read from a different location, which is commonly performed by hardware, and it is non-multi-copy-atomic, thus allowing less centralized architectures like POWER [45].

Our first contribution is a Brookes-style denotational semantics for RA. As Brookes's traces are totally ordered, this result may seem counterintuitive. The standard semantics for RA is a declarative (a.k.a. axiomatic) memory model, in the form of acyclicity consistency constraints over partially ordered candidate execution graphs. Since these graphs are not totally ordered, one might expect that Brookes's traces are insufficient. Nevertheless, our first key observation is that an *operational* presentation of RA as an interleaving semantics of a weak memory system lends itself to Brookes-style semantics. For that matter, we develop a notion of traces compatible with Kang et al.'s "view-based" machine [27], an operational semantics that is equivalent to RA's declarative formulation. Our

main technical result is the (directed) adequacy of the proposed Brookes-style semantics w.r.t. that operational semantics of RA.

A main challenge when developing a denotational semantics lies in making it sufficiently abstract. While *full* abstraction is often out of reach, as a yardstick, we want our semantics to be able to justify various compiler transformations/optimizations that are known to be sound under RA [47]. Indeed, an immediate practical application of a denotational semantics is the ability to provide *local* formal justifications of program transformations, such as those performed by optimizing compilers. In this setting, to show that an optimization $N \twoheadrightarrow M$ is valid amounts to showing that replacing N by M anywhere in a larger program does not introduce new behaviors, which follows from $[\![M]\!] \leq [\![N]\!]$ given a directionally adequate denotation function $[\![-]\!]$.

To support various compiler transformations, we close our denotations under certain operations, including analogs to Brookes's stutter and mumble, but also several RA-specific operations, that allow us to relate programs which would naively correspond to rather different sets of traces. Given these closure operations, our semantics validates standard program transformations, including structural transformations, algebraic laws of parallel programming, and all known thread-local RA-valid compiler optimizations. Thus, the denotational semantics is instrumental in formally establishing validity of transformations under RA, which is a non-trivial task [19, 47].

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 [38] 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, 36], 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 lay the groundwork for the rest of the paper by introducing the programming language that we will use (§2.1), the main ideas that underpin Brookes's trace-based denotational semantics (§2.2), and the operational RA model (§2.3). In §3 we present the core aspects of our denotational semantics. First, we discuss our extension of RA's operations semantics with first-class

parallelism, which enables denotations to be defined for concurrent composition (§3.1). We then present RA traces (§3.2) and use them to define the denotations of key program constructs (§3.3). Next, we show how the restriction of traces within denotations (§3.4) and the addition of closure operations (§3.5) make our denotational semantics more abstract. The denotational semantics extends to the entire programming language standardly using Moggi's monad-based approach (§3.6). With the denotational semantics in place, we present our main results in §4. Finally, we conclude and discuss related work in §5. More details are available in the extended version of this paper [21].

2 Preliminaries

We first introduce the language and its operational semantics under the Sequential Consistency (SC) memory model (§2.1). We then outline Brookes's denotational semantics for SC (§2.2). Finally, we introduce Kang et al.'s operational presentation of Release/Acquire (RA) (§2.3).

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

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

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

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

We can use intermediate results in subsequent computations via let binding: $\mathbf{let}\, a = M\, \mathbf{in}\, N$ binds the result of M to a in N. Thus, we execute M first,

and substitute the resulting value V for a in N before executing $N[a \mapsto V]$. Similarly, we deconstruct pairs by matching: **match** M **with** $\langle a, b\rangle . N$ binds the components of the pair that M returns to a and b respectively in N. The first and second projections **fst** and **snd**, as well as the operation **swap** that swaps the pair constituents, are defined using **match** standardly.

Sequential consistency. In the strongest memory model of Sequential Consistency (SC), every value stored is immediately made available to every thread, and every dereference must load the latest stored value. Thus the underlying memory model uses maps from locations to values for the memory state that evolves during program execution. Given an initial state, the behavior of a program in SC depends only on the choice of interleaving of steps. Though any such map can serve as an initial state, litmus tests are traditionally designed with the memory that sets all values to 0 in mind. In (MP) the order of the two stores and the two loads ensures that executions under SC may return $\langle \langle\rangle , \langle 0, 0\rangle\rangle$, $\langle \langle\rangle , \langle 0, 1\rangle\rangle$, and $\langle \langle\rangle , \langle 1, 1\rangle\rangle$, *but not* $\langle \langle\rangle , \langle 1, 0\rangle\rangle$.

Observations. An *observable behavior* of an entire program is a value it may evaluate to from given initial memory values. While programs may internally interact and observe the memory, we do not consider it feasible to observe the memory directly.

2.2 Overview of Brookes's Trace-based Semantics

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

Adequacy in SC. A prominent approach to define compositional semantics for concurrent programs is due to Brookes [13], who defined a denotational semantics for SC by taking $[\![M]\!]$ to be a set of traces of M closed under certain rewrite rules as we detail below. Brookes established a (directional) adequacy theorem: if $[\![M]\!] \supseteq [\![N]\!]$ then the transformation $M \twoheadrightarrow N$ is valid under SC. The latter means that, when assuming SC-based operational semantics, M can be replaced by N within a program without introducing new observable behaviors for it.

Thus, adequacy formally grounds the intuition that the denotational semantics soundly captures behavior of program terms.

As a particular practical benefit, 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 $[\![\mathtt{x} := v \,;\, \mathtt{x} := w]\!] \supseteq [\![\mathtt{x} := w]\!]$ holds in Brookes's semantics. Thanks to adequacy, this justifies Write-Write Elimination (WW-Elim) $\mathtt{x} := v \,;\, \mathtt{x} := w \twoheadrightarrow \mathtt{x} := w$ 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 μ in order to guarantee the memory state ρ.

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

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

Rewrite rules in SC. Were denotations in Brookes's semantics defined to *only* include the traces explicitly mentioned above, it would not be abstract enough to justify (WW-Elim), which eliminates redundant writes. Indeed, we only saw traces with two transitions in $[\![\mathtt{x} := v \,;\, \mathtt{x} := w]\!]$, but in $[\![\mathtt{x} := w]\!]$ we saw traces with one. The semantics would still be adequate, but it would lack abstraction. This is where Brookes's second main idea comes into play, making the denotations more abstract by closing them under two operations that rewrite traces:

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

Mumble combines a couple of subsequent transitions of the form $\langle \mu, \rho \rangle \langle \rho, \theta \rangle$ into a single transition $\langle \mu, \theta \rangle$ anywhere in the trace. Intuitively, a program

term can always omit a guarantee to the environment, and rely on its own omitted guarantee instead of relying on the environment.

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

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

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

Nevertheless, even without full abstraction, one can still provide evidence that an adequate semantics is abstract by ensuring that it supports known transformations. As an example, we show directly that $[\![\mathtt{x} := v\,;\,\mathtt{x} := w]\!] \supseteq [\![\mathtt{x} := w]\!]$ holds in Brookes's semantics. Since $[\![\mathtt{x} := v\,;\,\mathtt{x} := w]\!]$ is closed, it suffices to show that $[\![\mathtt{x} := v\,;\,\mathtt{x} := w]\!] \supseteq \left\{ \boxed{\langle \mu, \mu\,[\mathtt{x} := w]\rangle}\,\therefore\langle\rangle \;\middle|\; \text{memory } \mu \right\}$. For a memory μ, we have $\boxed{\langle \mu, \mu\,[\mathtt{x} := v]\rangle\,\langle \rho, \rho\,[\mathtt{x} := w]\rangle}\,\therefore\langle\rangle \in [\![\mathtt{x} := v\,;\,\mathtt{x} := w]\!]$ for every memory ρ, in particular when $\rho = \mu\,[\mathtt{x} := v]$. Since $\rho\,[\mathtt{x} := w] = \mu\,[\mathtt{x} := v]\,[\mathtt{x} := w] = \mu\,[\mathtt{x} := w]$, we have $\boxed{\langle \mu, \mu\,[\mathtt{x} := v]\rangle\,\langle \mu\,[\mathtt{x} := v], \mu\,[\mathtt{x} := w]\rangle}\,\therefore\langle\rangle \in [\![\mathtt{x} := v\,;\,\mathtt{x} := w]\!]$. After applying mumble, we have $\boxed{\langle \mu, \mu\,[\mathtt{x} := w]\rangle}\,\therefore\langle\rangle \in [\![\mathtt{x} := v\,;\,\mathtt{x} := w]\!]$.

2.3 Overview of Release/Acquire Operational Semantics

Memory accesses in RA are more subtle in than in SC. To address this we adopt Kang et al.'s "view-based" machine [27], an operational presentation of RA proven to be equivalent to the original declarative formulation of RA [e.g. 30]. In this model, rather than the memory holding only the latest value written to every variable, the memory accumulates a set of memory update messages for each location. Each thread maintains its own *view* that captures which messages the thread can observe, and is used to constrain the messages that the thread

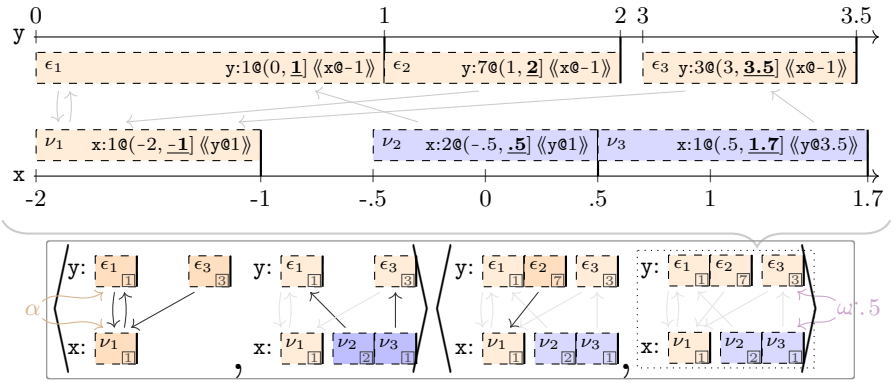

Fig. 1. Illustrations of a memory (top) and a trace (bottom), in the setting of two memory locations, `x` and `y`. **Top:** A memory holding six messages. The timelines are purposefully misaligned and not to scale to emphasize that timestamps for different locations are incomparable and that only the 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. ν_3 dovetails with ν_2, which does not dovetail with ν_1. **Bottom:** A trace with two transitions: $\alpha \boxed{\langle \mu_1, \rho_1 \rangle \langle \mu_2, \rho_2 \rangle} \omega \therefore 5$. The memory illustrated on top is ρ_2. Messages and edges that are not part of a previous memory are highlighted. The local messages are ν_2 and ν_3, and the rest are environment messages.

may read and write. The messages in the memory carry views as well, which are inherited from the thread that wrote the message, and passed to any thread that reads the message. Thus views indirectly maintain a causal relationship between messages in memory throughout the evolution of the system.

More concretely, causality is enforced by timestamping messages, thus placing them on their location's *timeline*. To capture the atomicity of RMWs, each message occupies a half-open segment $(q, t]$ on their location's timeline, where t is the message's timestamp. It *dovetails* with a message at the same location with timestamp q. An RMW "modifies" a message by dovetailing with it.

A view κ associates a timestamp $\kappa(\ell)$ to each location ℓ, obscuring the portion of ℓ's timeline before $\kappa(\ell)$. The view *points to* a message at ℓ with timestamp $\kappa(\ell)$. A view ω *dominates* a view α, written $\alpha \leq \omega$, if $\alpha(\ell) \leq \omega(\ell)$ for every ℓ.

Messages point to messages via the view they carry, and must point to themselves. So when specifying a message, the value its view takes at its location may be omitted. For example, assuming of two location, `x` and `y`, we denote by $\mathtt{x}{:}1@(.5, \underline{\mathbf{1.7}}]\ \langle\!\langle \mathtt{y}@3.5 \rangle\!\rangle$ the message at location `x` that carries the value 1, occupies the segment $(.5, 1.7]$ on `x`'s timeline, and carries the view κ such that $\kappa(\mathtt{x}) = 1.7$ and $\kappa(\mathtt{y}) = 3.5$. An example memory is depicted on the top of Figure 1.

When a thread writes to 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

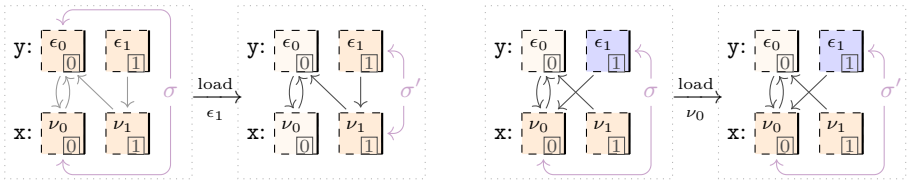

Fig. 2. Depictions of a step during an execution of a litmus test, with the view of the right thread changing from σ to σ'. The value each message carries is in its bottom-right corner. Views are illustrated implicitly in the graph structure that they impose. Obscured messages are faded. **Left:** As 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.

revealed messages, and when it reads, its view increases as needed to dominate the view of the loaded message. This may obscure messages at 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*:

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

By considering the possible interleavings, one can check that no execution in SC returns $\langle 0, 0\rangle$. However, in RA some do. Indeed, even if the left thread stores to x before the right thread loads from x, the right thread's view allows it to load 0, as 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 $\mathrm{FAA}\,(\mathtt{z}, 0)$ to a fresh location z. Thus, compare (SB) to the *store buffering with fences* litmus test:

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

Both of the $\mathrm{FAA}\,(\mathtt{z}, 0)$ 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 Denotational Semantics for Release/Acquire

We start this section by explaining how we support first-class concurrent composition ($\|$) in the operational semantics of Release/Acquire (§3.1). In the rest of the section we present the core of our denotational semantics. First, we present our notion of a trace, adapted to RA, along with four basic rewrite rules that our denotations are closed under (§3.2). Next, we define the denotations of the key program constructs (§3.3). We then present further aspects of the denotational semantics that make it more abstract: restrictions that traces in denotations must uphold (§3.4), and three more rewrite rules under which denotations are closed (§3.5). For completeness, we show how to give denotations to the whole language standardly, using Moggi's approach (§3.6).

3.1 First-class Concurrent Composition

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 $[\![M \parallel N]\!]$ 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-labelled 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.

An *initial state* consists of a memory with a single message at each location, and a view which points to these messages' timestamps. The example below shows how threads inherit their parent's view upon activation and combine their views as they synchronize:

Example. In the following, $\rightsquigarrow$ is the execution step relation, $\rightsquigarrow^*$ is its reflexive-transitive closure, μ_0 is an initial memory, $\dot{\kappa}$ is the κ-labelled view-leaf, $T \mathbin{\widehat{}} R$ is the view-tree that consists of a node connected to the view-trees T and R, and ω is the least view that dominates both ω_1 and ω_2:

$$\begin{aligned}\langle \mu_0, \dot{\alpha} \rangle, M \,;\, (N_1 \parallel N_2) &\rightsquigarrow^* \langle \mu_1, \dot{\alpha}' \rangle, N_1 \parallel N_2 \rightsquigarrow \left\langle \mu_1, \dot{\alpha}' \mathbin{\widehat{}} \dot{\alpha}' \right\rangle, N_1 \parallel N_2 \\ &\rightsquigarrow^* \left\langle \rho, \dot{\omega}_1 \mathbin{\widehat{}} \dot{\omega}_2 \right\rangle, V_1 \parallel V_2 \rightsquigarrow \langle \rho, \dot{\omega} \rangle, \langle V_1, V_2 \rangle\end{aligned}$$

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. Finally, once both threads return a value, they synchronize.

Handling parallel composition as a first-class construct allows us to decompose Write-Read Reordering (WR-Reord) $(\mathtt{x} := v)\,;\mathtt{y?} \twoheadrightarrow \mathbf{fst}\,\langle \mathtt{y?}, (\mathtt{x} := v)\rangle$, a crucial reordering of memory accesses valid under RA but not under SC, into a

combination of Write-Read Deorder (WR-Deord) $\langle (\mathtt{x} := v)\,, \mathtt{y?}\rangle \twoheadrightarrow (\mathtt{x} := v) \parallel \mathtt{y?}$ together with structural transformations and laws of parallel programming:

$$
\begin{array}{l}
(\mathtt{x} := v)\,;\mathtt{y?} \overset{\text{Structural}}{\twoheadrightarrow} \mathbf{snd}\ \langle (\mathtt{x} := v)\,, \mathtt{y?}\rangle \overset{(\text{WR-Deord})}{\twoheadrightarrow} \mathbf{snd}\ ((\mathtt{x} := v) \parallel \mathtt{y?}) \\
\overset{\text{Par. Prog. Law: Symmetry}}{\twoheadrightarrow} \mathbf{snd}\ (\mathbf{swap}\ (\mathtt{y?} \parallel (\mathtt{x} := v))\,) \overset{\text{Structural}}{\twoheadrightarrow} \mathbf{fst}\ (\mathtt{y?} \parallel (\mathtt{x} := v)) \overset{\text{Par. Prog. Law: Sequencing}}{\twoheadrightarrow} \mathbf{fst}\ \langle \mathtt{y?}, (\mathtt{x} := v)\rangle
\end{array}
$$

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).

3.2 Traces for Release/Acquire

Adapting Brookes's SC-traces, our RA-traces also 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 ground these intuition formally in the rewind and forward rewrite rules below.

We write the trace as $\alpha\,[\xi]\,\omega \therefore r$. See an illustration on the bottom of Figure 1.

Stutter & Mumble. We define the stutter (St) and mumble (Mu) rewrite rules:

$$
\alpha\,[\xi\eta]\,\omega \therefore r \xrightarrow{\mathsf{St}} \alpha\,[\xi\langle\mu,\mu\rangle\eta]\,\omega \therefore r \qquad \alpha\,[\xi\langle\mu,\rho\rangle\langle\rho,\theta\rangle\eta]\,\omega \therefore r \xrightarrow{\mathsf{Mu}} \alpha\,[\xi\langle\mu,\theta\rangle\eta]\,\omega \therefore r
$$

As in Brookes's semantics, their role is to make the semantics more abstract by divorcing the length of the sequence from the individual steps taken in the operational semantics, while maintaining the transitions' Rely/Guarantee character.

Rewind & Forward. The rewind (Rw) rewrite rules establish the fact that the term only relies on certain messages being *revealed*, not on messages being obscured. The rewind rule modifies the initial view, making it point to earlier messages on the timelines. Thus, relied upon messages will remain available after the rewrite. Similarly, the forward (Fw) rewrite rule establish the fact that the term only guarantees that certain messages are revealed. The forward rule modifies the final view, making it point to later messages on the timelines. Thus, any message guaranteed to be available was already guaranteed beforehand. The rules are schematically depicted in Figure 3.

Fig. 3. Schematic depictions of the rewind and forward rewrite rule, focusing on a single location, where the initial/final view points to ν before and points to ϵ after. The messages ν and ϵ may coincide, dovetail, or be separated. **Left:** The initial view α is "rewound" to α'. **Right:** The final view ω is "forwarded" to ω'.

3.3 Introducing Denotations for RA

We present denotations of key constructs of the programming language. By referring to the notion of a *closed set* below, we mean a set that is closed under certain rewrite rules, such as stutter, mumble, rewind, and forward from §3.2.

Pure. A pure (i.e. effect-free) computation guarantees a returned value, and otherwise can only guarantee what it relies on. For example, define $[\![2+3]\!]$ as least closed set with all traces of the form $\kappa\,\boxed{\langle\mu,\mu\rangle}\,\kappa\mathrel{.\,.}5$.

Sequence. In denoting sequential composition we must make sure that the first component does not obscure any message that the second component relies on. Thus, define $[\![\langle M,N\rangle]\!]$ as least closed set with all traces of the form $\alpha\,\boxed{\xi\eta}\,\omega\mathrel{.\,.}\langle r,s\rangle$, where there exists a view κ such that $\alpha\,\boxed{\xi}\,\kappa\mathrel{.\,.}r\in[\![M]\!]$ and $\kappa\,\boxed{\eta}\,\omega\mathrel{.\,.}s\in[\![N]\!]$. The *existence* of the revealed messages is implicit: ξ's closing memory must be contained in the memory that follows it, which is η's opening memory. The definition of $[\![M\,;N]\!]$ is the same, except that the first component of the returned pair is discarded. That is, with traces of the form $\alpha\,\boxed{\xi\eta}\,\omega\mathrel{.\,.}s$.

Parallel. Threads composed in parallel rely on the same preceding sequential environment and guarantee to the same succeeding sequential environment. Thus, define $[\![M_1\parallel M_2]\!]$ as the least closed set with all traces of the form $\alpha\,\boxed{\xi}\,\omega\mathrel{.\,.}\langle r_1,r_2\rangle$, where there exist sequences ξ_1 and ξ_2 such that and ξ is obtained by interleaving their transitions, and $\alpha\,\boxed{\xi_i}\,\omega\mathrel{.\,.}r_i\in[\![M_i]\!]$ (for $i\in\{1,2\}$).

Dereference. We define $[\![\ell\mathbf{?}]\!]$ to be the least closed set with all traces of the form $\alpha\,\boxed{\langle\mu,\mu\rangle}\,\omega\mathrel{.\,.}v$, where $\ell{:}v@(q,\alpha(\ell)]\!\langle\!\langle\kappa\rangle\!\rangle\in\mu$ for some timestamp q and view κ, and both $\alpha\le\omega$ and $\kappa\le\omega$.

Assignment. Define $[\![\ell:=v]\!]$ as the least closed set with all traces of the form $\alpha\,\boxed{\langle\mu,\rho\rangle}\,\omega\mathrel{.\,.}\langle\rangle$ where ρ is obtained by adding the message $\ell{:}v@(q,\omega(\ell)]\!\langle\!\langle\omega\rangle\!\rangle$ to μ for some timestamp q, and $\alpha\le\omega$.

Read-modify-write. The definition of $[\![\mathrm{FAA}\,(\ell,w)]\!]$ combines the two above, along with a dovetailing requirement. Specifically, it is the least closed set with all traces of the form $\alpha\,\boxed{\langle\mu,\rho\rangle}\,\omega\mathrel{.\,.}v$, where $\ell{:}v@(q,\alpha(\ell)]\!\langle\!\langle\kappa\rangle\!\rangle\in\mu$ for some timestamp q and view κ, both $\alpha\le\omega$ and $\kappa\le\omega$, and ρ is obtained by adding the message $\ell{:}\,(v{+}w)\,@(\alpha(\ell),\omega(\ell)]\!\langle\!\langle\omega\rangle\!\rangle$ to μ. The semantics of other RMWs is defined similarly.

Example. We show that $[\![\ell := v\,;\, v]\!] \subseteq [\![\ell := v\,;\, \ell?]\!]$. When sequencing two traces, the final view of the first must match the initial view of the second, so traces in $[\![\ell := v\,;\, v]\!]$ have the form $\alpha\boxed{\langle \mu, \rho\rangle\ \langle \theta, \theta\rangle}\omega \therefore v$, where ρ is obtained by adding the message $\ell{:}v@(q, \omega(\ell)]\!\langle\omega\rangle\!\rangle$ to μ for some timestamp q, and $\alpha \leq \omega$. Since ω points to this added message, and since $\rho \subseteq \theta$ as memories along a trace's sequence, $\omega\boxed{\langle \theta, \theta\rangle}\omega \therefore v \in [\![\ell?]\!]$. By sequencing, $\alpha\boxed{\langle \mu, \rho\rangle\ \langle \theta, \theta\rangle}\omega \therefore v \in [\![\ell := v\,;\, \ell?]\!]$.

3.4 Correspondence to the Operational Semantics

Traces in denotations, if unconstrained, may represent behaviors that include operationally unreachable states. Forbidding such redundant traces eliminates a source of differentiation between denotations, thus increasing their abstraction.

Reachable states. Consider the transformation $\mathtt{x?\,;\,y?} \twoheadrightarrow \mathtt{y?}$, a consequence of the RA-valid Irrelevant Read Elimination (R-Elim) $\mathtt{x?\,;}\,\langle\rangle \twoheadrightarrow \langle\rangle$ and structural equivalences. Consider the state S that consists of the memory at the top of Figure 1 and the view that points to ν_3 and ϵ_2. The only step $\mathtt{x?\,;\,y?}$ can take from the state S is to load ν_3, inheriting the view that ν_3 carries, which changes the thread's view to point to ϵ_3. Only ϵ_3 is available in the following step, which means the term returns 3. In contrast, starting from S, the term $\mathtt{y?}$ can load from ϵ_2 to return 7. This analysis does not invalidate the transformation because the state S is unreachable by an execution starting from an initial state, and should therefore be ignored when determining observable behaviors.

Internalizing invariants. Just as we ignore unreachable states in the operational semantics, we discard "unreachable" traces to refine our denotational semantics. We consider a state to be valid if it adheres to the following invariants.

Scattering: segments in memory never overlap.
Pointing: views always point to messages.
Dominating: views always dominate the views of the messages to which they point. This invalidates the state S above, because the view of the thread does not dominate the view of ν_3 even though it points to it.
Descending: a path from a message along the view-induced graph structure cannot end in another message with a greater timestamp at the same location. Demonstrated both positively and negatively in Figure 4.
Acyclicity: a cycle along the view-induced graph structure consists solely of messages which have the smallest timestamp on their timeline.

Memory snapshots in traces are required to obey each of the invariants above. The initial and final view must point to and dominate the opening and closing memory respectively. This means that there must be a message to load that allows the initial and final view to be equal, and we obtain $[\![\mathtt{x?\,;}\,\langle\rangle]\!] \supseteq [\![\langle\rangle]\!]$.

We also uphold requirements that correspond to the relation between the states across a possibly-interrupted series of steps in the operational semantics:

Accumulating: the memory after contains the memory before. We require that every memory snapshot contains the one before it.

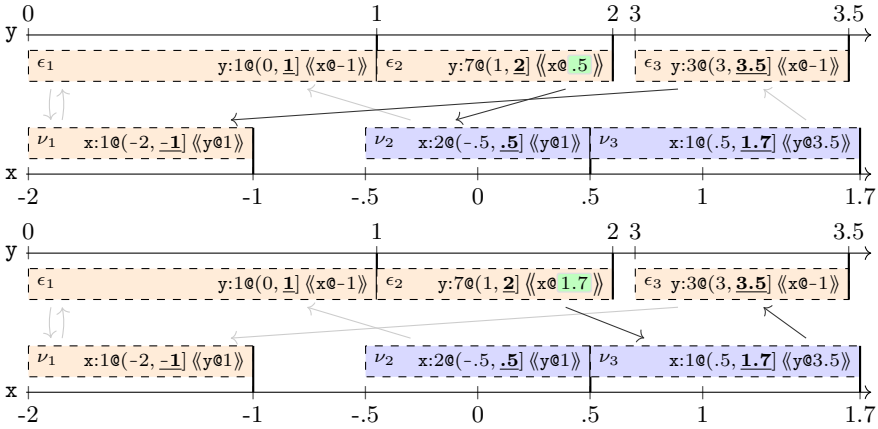

Fig. 4. Two variations on the memory illustrated in Figure 1. **Top:** This can function as a memory snapshot in a trace. It demonstrates that the views of messages along a timeline do not have to be ordered: ϵ_2 appears earlier than ϵ_3 on $\mathtt{y}$'s timeline but points to a later message on $\mathtt{x}$'s timeline. **Bottom:** This cannot function as a memory snapshot in a trace, because it contains an ascending path. Intuitively, no thread could have written ϵ_2 because the view that ϵ_2 carries indicates that the thread would have already "known" about ν_3 and therefore, following the causality chain, about ϵ_3 as well. Thus, the thread would have been forbidden from picking ϵ_2's timestamp.

Delimiting: if the view-trees before and after are leaves, then the view after dominates the view before, and the view of any written message dominates the view before and is dominated by the view after. We impose the analogous requirement on the initial and final views, and on the local messages.

The trace in Figure 1 adheres to the invariants and relationships we have listed.

Concrete operational correspondence. We call the rewrite rules that were defined in §3.2 *concrete* because they maintain a certain concrete interpretation of traces. To see this, consider the operational semantics for RA augmented with an additional kind of step, which any term can take. The only change along this step is that a view in the view-tree inherits the view from a message that is available to it. This addition does not change the observable behaviors of whole programs, and maintains the above invariants.

Each trace in the denotations of §3.3, if closed only under the concrete rewrite rules, corresponds to an interrupted execution in the augmented operational semantics. The correspondence is similar to that from Brookes's semantics in terms of the sequence of transitions and return value. The initial and final views determine the views at the beginning and the end of the interrupted execution.

The introduction of the rewrite rules in §3.5 will mean that traces do not have such a clear operational interpretation. The key to our proof of adequacy

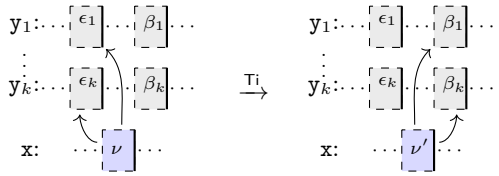

Fig. 5. Schematic depiction of the tighten rewrite rule, that focuses on a particular memory snapshot within the trace, in the setting of $k{+}1$ locations. The message ν is "tightened" to ν', such that for each i it points to β_i instead of ϵ_i. This includes the case that β_i and ϵ_i are the same message in some locations.

is to partially recover this operational correspondence in terms of the overall observable behaviors (§4).

3.5 Abstract Rewrite Rules

Transitions in RA traces consist of sets of messages, which record much more information about the operational execution than the mappings from locations to values we had in SC. This makes the trace-based semantics too concrete. We resolve the memory-concreteness issue by introducing three *abstract* rewrite rules that obfuscate information about local messages. This makes the denotations more abstract by blurring the distinctions that denotations can make.

Tighten. Recall the transformation (WR-Deord) that we wish to support. Let $\tau_1 \in [\![\mathtt{x} := v]\!]$ and $\tau_2 \in [\![\mathtt{y?}]\!]$, such that they compose sequentially to form a trace from $[\![\langle (\mathtt{x} := v)\,, \mathtt{y?} \rangle]\!]$. Then τ_1's final view κ must equal τ_2's initial view. The view κ dominates the view σ of the local message ν_1 stored by τ_1, and κ cannot obscure the message ν_2 from which τ_2 loaded its value. Thus, σ cannot obscure ν_2. In contrast, consider τ_1 and τ_2 that compose in parallel to form a trace from $[\![(\mathtt{x} := v) \parallel \mathtt{y?}]\!]$. Here, the view of the local message may very well obscure the loaded message. Indeed, the final view of τ_1 may dominate the initial view of τ_2.

To resolve this, observe that the purpose of recording views in messages is to encumber its loaders. Under this perspective, the view of a local message guarantees to the environment that loading the local message will keep certain messages revealed. Therefore, making the view larger only weakens the guarantee. Thus, we introduce the tighten (Ti) rewrite rule that makes the view of a local message larger. The rule is depicted in Figure 5, and Figure 6 provides a concrete example. Using tighten, we can show that $[\![\langle (\mathtt{x} := v)\,, \mathtt{y?} \rangle]\!] \supseteq [\![(\mathtt{x} := v) \parallel \mathtt{y?}]\!]$.

Absorb. Recall the transformation (WW-Elim) that we wish to support. To show this we aim to replicate, as far as we can, the reasoning we have used to show $[\![\mathtt{x} := v\,;\mathtt{x} := w]\!] \supseteq [\![\mathtt{x} := w]\!]$ in Brookes's semantics. Recall that, to use mumble, we made the memories match across the two transitions of $[\![\mathtt{x} := v\,;\mathtt{x} := w]\!]$. Doing so here, we end up with two local messages, whereas traces from $[\![\mathtt{x} := w]\!]$ only have a single local message. Roughly speaking, the equality concerning SC

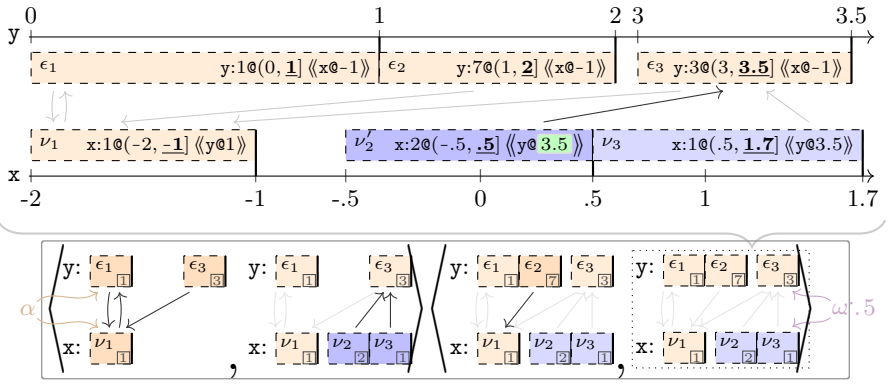

Fig. 6. A possible result from rewriting the trace from Figure 1 using tighten. Since ν_2 is local in the trace from Figure 1, tighten can advance its view to point to ϵ_3 instead of ϵ_1. The same replacement is applied throughout the trace's sequence, not just the closing memory.

memories $\mu\,[\mathtt{x} := v]\,[\mathtt{x} := w] = \mu\,[\mathtt{x} := w]$ does not transfer to RA where memory, by accumulating messages, is more concrete. We resolve this by adding the absorb (Ab) rewrite rule, which replaces two dovetailed local messages with one that carries the second message's value. The rule is depicted in Figure 7, and Figure 8 provides a specific example.

Dilute. There is another known family of transformations that are valid under RA memory, yet we cannot justify with the rules we presented. These introduce non-modifying atomic updates, such as Read to FAA (R-FAA) $\ell\mathtt{?} \twoheadrightarrow \mathrm{FAA}\,(\ell, 0)$.

Running within some context, $\mathrm{FAA}\,(\ell, 0)$ reads a message ν, to which it dovetails another message ϵ with the same value. It's possible that some β dovetails with ϵ later in the execution. In the same context, we can simulate this behavior with $\ell\mathtt{?}$ instead, by having the context provide ν' instead of ν, with the difference that it takes up the same segment that ν and ϵ have taken up combined. If there is a β as mentioned, it can now dovetail with ν' to the same effect. In this scenario, ν is an environment message, but we must also account for the case that it is local to allow for composition, such as in $\ell\mathtt{:=}v\mathtt{;}\ell\mathtt{?} \twoheadrightarrow \ell\mathtt{:=}v\mathtt{;}\mathrm{FAA}\,(\ell, 0)$.

We internalize the idea behind this argument as the dilute (Di) rewrite rule, in which a message is replaced by two message that together occupy the same segment, the second being a local message that cannot appear before the first in the trace and must carry the same value. With dilute, $[\![\ell\mathtt{?}]\!] \supseteq [\![\mathrm{FAA}\,(\ell, 0)]\!]$. The rule is depicted in Figure 7, and Figure 9 provides a specific example.

3.6 Monadic Presentation

One of the contributions of this work is to bridge research of weak-memory models with Moggi's monad-based approach [38] to denotational semantics. In this approach, one start by defining a monad, which has three components. The

Fig. 7. Schematic depictions of the absorb (left) and dilute (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 that they all have the same view. The *elliptical* clouds represent views—including the initial and final view, as well as other messages—that point to each of the dovetailing messages. **Left:** The message ν is "absorbed" into the message ϵ to become ϵ'. No view may point to ν. **Right:** The message ν' "dilutes" into ν and ϵ. While ϵ must be a local message, ν and ν' can appear anywhere the trace's sequence, as long as they appear in the same places in the sequence, and that ϵ does not appear before. The views that point to ν' before diluting can point either to ν or to ϵ after diluting.

first associates for every set X, which we think of as representing returned values, to a set $\underline{\mathcal{T}}X$ representing computations that return values from X. In our case, $\underline{\mathcal{T}}X$ consists of countable sets of traces closed under rewrite rules.

Denotations are then defined according to their *typing judgments*. For example, $a, b : \mathsf{Loc} \vdash \langle a, b\mathbf{?}\rangle : (\mathsf{Loc} \times \mathsf{Val})$ means that in the context that the free variables a and b are locations, the term $\langle a, b\mathbf{?}\rangle$ is a location-value pair. Given a function γ that maps a and b to locations, $[\![\langle a, b\mathbf{?}\rangle]\!]\,\gamma \in \underline{\mathcal{T}}\,(\mathsf{Loc} \times \mathsf{Val})$. For $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$, we generalize containment $[\![N]\!] \supseteq [\![M]\!]$ pointwise: if γ maps variables in Γ appropriately by their type, then $[\![N]\!]\,\gamma \supseteq [\![M]\!]\,\gamma$. This degenerates when Γ is empty, i.e. when M and N are *closed terms*.

The second monad component is a function $\mathrm{return}^{\mathcal{T}}_X : X \to \underline{\mathcal{T}}X$ maps values to pure computations that return that value. The third component sequences computations, such that the latter depends on the value returned by the former: $(\gg\!=^{\mathcal{T}}_{X,Y}) : (\underline{\mathcal{T}}X) \times (X \to \underline{\mathcal{T}}Y) \to \underline{\mathcal{T}}Y$. Omitting the indices, the monad components must satisfy certain axioms that formalize the stated intuition: $\mathrm{return}\,r \gg\!= f = f(r)$, $P \gg\!= \mathrm{return} = P$ and $(P \gg\!= f) \gg\!= g = P \gg\!= \lambda r.\,(f(r) \gg\!= g)$.

In our case, we define $\mathrm{return}\,r$ as the least closed set with all traces of the form $\kappa\,\boxed{\langle \mu, \mu\rangle}\,\kappa \therefore r$; and $P \gg\!= f$ as the least closed set with all traces of the form $\alpha\,\boxed{\xi\eta}\,\omega \therefore s$, where $\alpha\,\boxed{\xi}\,\kappa \therefore r \in P$ and $\kappa\,\boxed{\eta}\,\omega \therefore s \in f(r)$ for some κ.

Denotations. This approach comes read-made with denotations for standard language constructs. For example, $[\![\langle M, N\rangle]\!]\,\gamma \coloneqq [\![M]\!]\,\gamma \gg\!= \lambda r.\,([\![N]\!]\,\gamma \gg\!= \lambda s.\,\langle r, s\rangle)$. Similarly, $[\![\mathbf{match}\,M\,\mathbf{with}\,\langle a, b\rangle.\,N]\!]\,\gamma \coloneqq [\![M]\!]\,\gamma \gg\!= \lambda\langle r, s\rangle.\,[\![N]\!]\,\gamma\,[a \mapsto r]\,[b \mapsto s]$, where $\gamma\,[a \mapsto r]$ is obtained from γ by mapping a to r. Pure computations use the return function, e.g. $[\![v]\!] = \mathrm{return}\,v$.

Program effects can be modularly introduced in this approach, such as memory access, where $[\![\ell \coloneqq v]\!] \in \underline{\mathcal{T}}\,\{\langle\rangle\}$ and $[\![\ell\mathbf{?}]\!], [\![\mathrm{FAA}\,(\ell, v)]\!] \in \underline{\mathcal{T}}\mathsf{Val}$; and par-

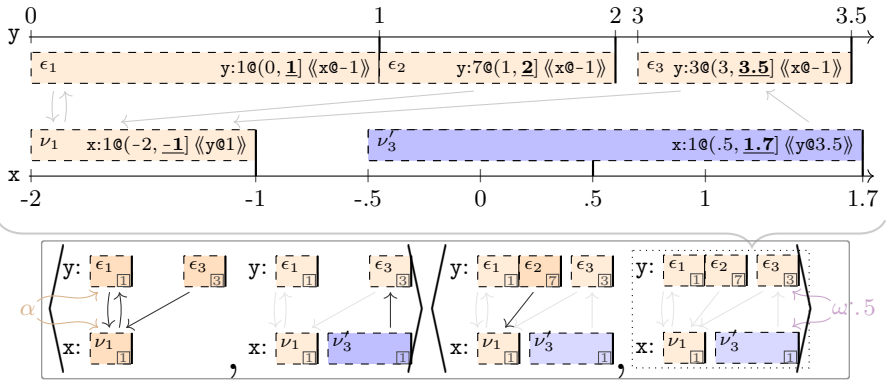

Fig. 8. A possible result from rewriting of the trace from Figure 6 using absorb. The dovetailed messages ν_2 and ν_3 are local in the trace from Figure 1, added within the same transition, so by rewriting by absorb they can be replaced by ν'_3 obtained by stretching ν_3's segment to cover ν_2's segment.

allel composition, a function $(|||^{\mathcal{T}}_{X,Y}) : \underline{\mathcal{T}}X \times \underline{\mathcal{T}}Y \to \underline{\mathcal{T}}(X \times Y)$ with which $[\![M \parallel N]\!]\,\gamma \coloneqq [\![M]\!]\,\gamma \,|||\, [\![N]\!]\,\gamma$. The definition remains the same: we obtain traces in $P \,|||\, Q$ by interleaving transitions and pairing returned values of traces with matching views, one from P and one from Q.

Adhering to left-to-right evaluation both operationally and denotationally, $M \mathbin{:=} N$ is equivalent to $\mathbf{match}\,\langle M, N\rangle\,\mathbf{with}\,\langle a, b\rangle.\, a \mathbin{:=} b$. In traces of assignment, the added local message is free to dovetail with a previous message, unlike in RMW traces where it must. Therefore, we have $[\![\ell \mathbin{:=} (\ell\mathbf{?} + v)]\!] \supseteq [\![\mathrm{FAA}\,(\ell, v)]\!]$.

Structural reasoning. Among the general results and proof techniques this approach supplies are *structural equivalences*. These are denotational equations that hold due to the properties of the core calculus, and are preserved by modular expansions with program effects. For instance, if K is effect-free, then $[\![\mathbf{if}\,K\,\mathbf{then}\,M\,\mathbf{;}\,N\,\mathbf{else}\,M\,\mathbf{;}\,N'\,]\!] = [\![M\,\mathbf{;}\,\mathbf{if}\,K\,\mathbf{then}\,N\,\mathbf{else}\,N'\,]\!]$. Equivalences such as this one may otherwise require challenging ad-hoc proofs [e.g. 24, 26].

More generally, structural reasoning composes to derive further equivalences. For example, from $[\![\langle\rangle]\!] = [\![\ell\mathbf{?}\,\mathbf{;}\,\langle\rangle]\!]$ and structural equivalences, namely "left neutrality" $[\![K]\!] = [\![\langle\rangle\,\mathbf{;}\,K]\!]$ and "associativity" $[\![(M\,\mathbf{;}\,N)\,\mathbf{;}\,K]\!] = [\![M\,\mathbf{;}\,(N\,\mathbf{;}\,K)]\!]$:

$$[\![K]\!] = [\![\langle\rangle\,\mathbf{;}\,K]\!] = [\![(\ell\mathbf{?}\,\mathbf{;}\,\langle\rangle)\,\mathbf{;}\,K]\!] = [\![\ell\mathbf{?}\,\mathbf{;}\,(\langle\rangle\,\mathbf{;}\,K)]\!] = [\![\ell\mathbf{?}\,\mathbf{;}\,K]\!] \qquad (\star)$$

Structural reasoning generalizes to program transformations. For example, $(\gg\!=)$ is monotonic, so we can also derive:

$$[\![\langle\rangle]\!] = [\![\ell\mathbf{?}\,\mathbf{;}\,\langle\rangle]\!] = [\![\ell\mathbf{?}]\!] \gg\!= \lambda v.[\![\langle\rangle]\!] \supseteq [\![\mathrm{FAA}\,(\ell, 0)]\!] \gg\!= \lambda v.[\![\langle\rangle]\!] = [\![\mathrm{FAA}\,(\ell, 0)\,\mathbf{;}\,\langle\rangle]\!]$$

Since $(|||)$ is also monotonic, we can use this to show that $[\![(\mathrm{SB})]\!] \supseteq [\![(\mathrm{SB{+}F})]\!]$.

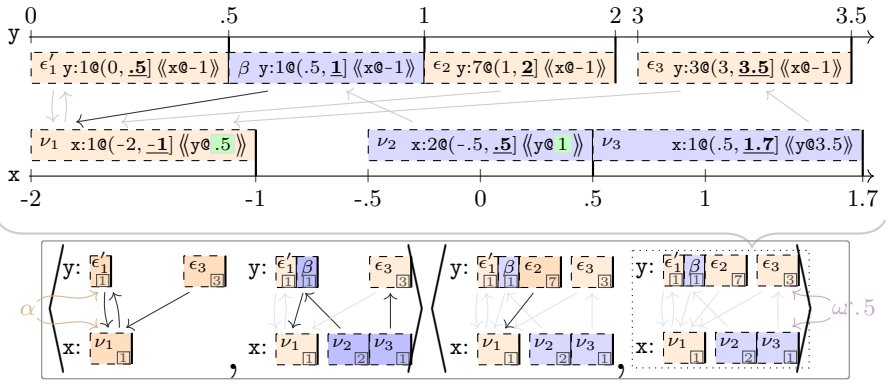

Fig. 9. A possible result from rewriting of the trace from Figure 1 using dilute. The message ϵ_1 from Figure 1 was replaced with ϵ_1', with the same value 1. The local message β—which takes up the rest of the missing space left behind by ϵ_1—always appears with ϵ_1', dovetailing with it and carrying the same value. The message ϵ_2, that used to dovetail with ϵ_1, now dovetails with β.

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 returned 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.

We classify a term as 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.

4 Main Results

We present the main results that we have proven about our denotational semantics. Moggi's semantic toolkit features ubiquitously in their proofs.

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. We have used this in (⊠). 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:

Lemma 1. *If* $[\![M]\!] \subseteq [\![N]\!]$ *then* $[\![\Xi\,[M]]\!] \subseteq [\![\Xi\,[N]]\!]$ *for any program context* $\Xi\,[-]$.

Compositionality is a consequence of its monadic design using monotonic operators, and is not substantially different from previous work [e.g. 20].

Observability correspondence. The abstract rewrite rules break the direct correspondence between traces and interrupted executions. 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.

Still, some connection must remain to obtain a proof of 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:

Lemma 2. *For every execution of a program* M *in the operational semantics of RA, there exists* $\alpha \boxed{\langle \mu, \rho \rangle} \omega \therefore r \in [\![M]\!]$ *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:

Lemma 3. *For every program* M *and* $\alpha \boxed{\langle \mu, \rho \rangle} \omega \therefore r \in [\![M]\!]$ *there is an observable behavior of* M *with initial state* $\langle \alpha, \mu \rangle$ *and return value matching* r.

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

Overcoming the concreteness-abstraction hurdle. The most technically challenging step in proving Lemma 3 is to prove the application of abstract rewrite rules can be deferred to the end. We define the *basic denotation* of a term M by $\underline{[\![M]\!]}$, which is the denotation were it defined using only the concrete rewrite rules. Denoting its closure under the abstract rewrite rules by $\underline{[\![M]\!]}^{\dagger}$, we claim:

Lemma 4. *If* M *is a program, then* $\underline{[\![M]\!]}^{\dagger} = [\![M]\!]$.

Thus, to obtain all of the traces that result from the regular denotational construction, where all of the rewrite rules are applied throughout the entire denotational construction, it is enough to close only under the concrete rewrite rules as the denotation of a program is built-up from its subterms, applying the abstract rewrite rules only at the top level.

The intuition that guides the inductive proof of Lemma 4 is that the abstract rewrite rules can be percolated out. To get the main idea across while keeping the discussion self-contained, we focus on the $\underline{[\![M_1 \parallel M_2]\!]}^{\dagger} \supseteq [\![M_1 \parallel M_2]\!]$ case.

Let $\pi \in [\![M_1 \parallel M_2]\!]$. By definition, π is obtained by first composing some $\tau_1 \in [\![M_1]\!]$ in parallel with some $\tau_2 \in [\![M_2]\!]$, i.e. interleaving transitions and pairing return values, and then rewriting the resulting trace τ with concrete

and abstract rules. By the inductive hypothesis, $\underline{[\![M_i]\!]}^\dagger \supseteq [\![M_i]\!]$. So $\tau_i \in \underline{[\![M_i]\!]}^\dagger$, meaning that τ_i is the result of rewriting some $\tau_i' \in \underline{[\![M_i]\!]}$ with abstract rules.

To warm up, we first address the case where $\tau_1' \xrightarrow{\mathsf{Ab}} \tau_1$ and $\tau_2' = \tau_2$. We would hope, naively, that we can compose τ_1' with τ_2' to obtain some $\tau' \in \underline{[\![M_1 \parallel M_2]\!]}$ such that $\tau' \xrightarrow{\mathsf{Ab}} \tau$, and thus τ' rewrites to π. However, they do not compose because τ_1' has two local message, and τ_2' has only the one environment message that matches the result of "absorbing" the two messages. Rather, τ_1' can compose with a trace $\bar{\tau}_2$ which is equal to τ_2' except for having the required two environment messages instead of the combined one.

We formalize this by introducing a dual auxiliary rewrite rule $\bar{\mathsf{x}}$ for each abstract rule x. For example, the dual of absorb is expel, which splits up an environment message dually to how absorb combines local messages. The auxiliary rewrite rules keep us within the basic denotations:

Lemma 5. *If $\tau \in \underline{[\![M]\!]}$ and $\tau \xrightarrow{\mathsf{z}} \pi$ for some auxiliary rule z, then $\pi \in \underline{[\![M]\!]}$.*

Then we apply $\tau_2' \xrightarrow{\bar{\mathsf{x}}} \bar{\tau}_2 \in \underline{[\![M_i]\!]}$, and obtain the required τ' by composing τ_1' in parallel with $\bar{\tau}_2$. This process of applying the dual rewrite in order to percolate an abstract rewrite out holds for sequential composition too. We summarize:

Lemma 6. *If $\pi' \xrightarrow{\mathsf{x}} \pi$ for some abstract x, and π composes in parallel with ϱ to obtain τ, then there exist $\varrho' \xrightarrow{\bar{\mathsf{x}}} \varrho$ and $\tau' \xrightarrow{\mathsf{x}} \tau$, such that π' composes in parallel with ϱ' to obtain τ'. Similarly for sequential composition.*

In the case where there are more abstract rewrite rules needed to obtain τ_1 from τ_1', we can repeat the process. Yet two problems remain.

The first problem is that π is obtained from $\tau' \in \underline{[\![M_1 \parallel M_2]\!]}$ by both concrete and abstract rewrites, starting with the abstract rewrites that we have "peeled off" τ_1. To show that $\pi \in \underline{[\![M_1 \parallel M_2]\!]}^\dagger$, we need the concrete rewrites to come before the abstract rewrites.

The second problem appears once we remove our simplifying assumption that $\tau_2' = \tau_2$. In the general case, we obtain $\bar{\tau}_2$ from τ_2' using abstract rewrites followed by auxiliary rewrites. If we could replace the sequence of rewrites with one in which the abstract rewrites follow the auxiliary rewrites, then τ_2' could be rewritten with auxiliary rules to some $\bar{\tau}_2' \in \underline{[\![M_2]\!]}$ by using Lemma 5, which in turn could be rewritten with abstract rewrites to $\bar{\tau}_2 \in \underline{[\![M_2]\!]}^\dagger$. This would allow the proof to continue by repeating the process to the other side.

Both problems are solved by commuting the abstract rewrites outwards:

Lemma 7. *For any rewrite sequence starting with τ and ending with π, there exists one in which all of the abstract rewrites appear last.*

Thus, we can do as we planned and repeat the process to the other side, "peeling off" the abstract rewrites from $\bar{\tau}_2$ to obtain $\bar{\tau}_2' \in \underline{[\![M_2]\!]}$, rewriting τ_1' with the dual auxiliary rules in lockstep, resulting in some $\bar{\tau}_1' \in \underline{[\![M_1]\!]}$ by Lemma 5. By Lemma 6, these compose in parallel to some $\bar{\tau} \in \underline{[\![M_1 \parallel M_2]\!]}$ that rewrites with

concrete and abstract rules to τ, and thus to π. By Lemma 7, we can rewrite $\bar{\tau}$ with concrete rules to some $\bar{\tau}' \in [\![M_1 \parallel M_2]\!]$ first, and with abstract rules afterwards, obtaining $\pi \in [\![M_1 \parallel M_2]\!]^{\dagger}$.

Having established Lemma 4, the rest is relatively straightforward. First, traces in basic denotations correspond to interrupted executions, and in particular, an analog of Lemma 3 holds for basic denotations:

Lemma 8. *For every program M and $\alpha \langle \mu, \rho \rangle \omega . \cdot . r \in [\![M]\!]$ there is an observable behavior of M with initial state $\langle \alpha, \mu \rangle$ and return value matching r.*

Next, it is clear from their definition that the abstract rules do not change the number of transitions. Thus, thanks to Lemma 4, the single-transition traces in $[\![M]\!]$ are the result of rewriting single-transition traces in $[\![M]\!]$ by abstract rules, which correspond to observable behaviors of M by Lemma 8.

Lemma 3 follows from the fact that the abstract rules preserve the correspondence between traces and observable behavior of programs. For example, due to absorb there is a trace which only adds one message in the denotation of a program that adds two messages; yet the initial view, the opening memory, and the returned value are maintained. The tighten rule similarly preserves these. In both cases, the execution exhibiting the behavior can remain unchanged. The dilute rule may replace an initial message's timestamp with a smaller one, in which case the execution exhibiting the behavior needs to use the new timestamp accordingly, but otherwise remains the same.

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

Theorem 9. *If $[\![M]\!] \subseteq [\![N]\!]$, then for all program contexts $\Xi[-]$, every observable behavior of $\Xi[M]$ is an observable behavior of $\Xi[N]$.*

In particular, $[\![M]\!] \subseteq [\![N]\!]$ implies that $N \twoheadrightarrow M$ is valid under RA, because the effect of applying it is unobservable.

Adequacy follows immediately from the above results. Indeed, using soundness, an observable behavior of $\Xi[M]$ corresponds to a single-transition $\tau \in [\![\Xi[M]]\!]$; by the assumption and compositionality $\tau \in [\![\Xi[N]]\!]$; and using the other direction, τ corresponds to an observable behavior of $\Xi[N]$.

Higher-order subtleties. When applying the above results in the presence of higher order, one must pay attention to the *program* assumption. Indeed, suppose $[\![M]\!] \supseteq [\![M']\!]$. Compositionality does not entail that $[\![\lambda a.\, M]\!] \supseteq [\![\lambda a.\, M']\!]$. Indeed, a function $\lambda a.\, M$ is a value, i.e. it does not execute, and in particular it does not perform any effects, regardless of M. Accordingly, $[\![\lambda a.\, M]\!]$ consists of closures of traces of the form $\kappa \langle \mu, \mu \rangle \kappa . \cdot . f$, where f is a function that returns sets of traces obtained from $[\![M]\!]$. The fact that $[\![M]\!] \supseteq [\![M']\!]$ is not helpful, because traces in $[\![\lambda a.\, M']\!]$ have different returned values f' from traces in $[\![\lambda a.\, M]\!]$.

Directional compositionality is still useful in the presence of abstractions. For example, if M is a program that returns a location, then from $[\![a := v \,;\, a := w]\!] \supseteq [\![a := w]\!]$ it follows that $[\![(\lambda a.\, a := v \,;\, a := w)\, M]\!] \supseteq [\![(\lambda a.\, a := w)\, M]\!]$.

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

Fig. 10. A selective list of supported non-structural transformations. Along with Symmetry, the denotational semantics supports all symmetric-monoidal laws with the binary operator ($\parallel$) and the unit $\langle\rangle$. Similar transformations, replacing FAA with other RMWs, are supported too. The abstract rewrite rules used to validate a transformation is mentioned, if there is one.

To deal with the need to prove properties "pointwise" that abstractions bring about, such as containment of denotations in the proof of directional compositionality, we use logical relations. Moggi's toolkit provides a standard way to define these, thereby lifting properties to their higher-order counterparts.

Transformations exhibiting abstraction. To the best of our knowledge, all transformations $N \twoheadrightarrow M$ proven to be valid under RA in the existing literature are supported by our denotational semantics, i.e. $[\![N]\!] \supseteq [\![M]\!]$. Structural transformations are supported by virtue of using Moggi's standard semantics. Our semantics also validates "algebraic laws of parallel programming", such as sequencing $M \parallel N \twoheadrightarrow \langle M, N\rangle$ and its generalization that Hoare and van Staden [22] recognized, $(M_1\,;M_2) \parallel (N_1\,;N_2) \twoheadrightarrow (M_1 \parallel N_1)\,;(M_2 \parallel N_2)$, which in the functional setting can take the more expressive form in which the values returned are passed on to the following computation. See Figure 10 for a partial list.

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

5 Related Work and Concluding Remarks

Our work follows the approach of Brookes [13] and its extension to higher-order functions using monads by Benton et al. [6]. Brookes developed a denotational semantics for shared memory concurrency under standard sequentially consistency [33], and established full abstraction w.r.t. a language that has a global atomic **await** instruction that locks the entire memory. The concepts behind this approach had been used in multiple related developments, e.g. [12, 34, 35, 46]. We hope that our work that targets RA will pave the way for similar continuations.

Jagadeesan et al. [25] adapted Brookes's semantics to the x86-TSO memory model [40]. They showed that for x86-TSO it suffices to include the final store buffer at the end of the trace and add two additional simple closure rules that emulate non-deterministic propagation of writes from store buffers to memory, and identify observably equivalent store buffers. The x86-TSO model, however, is much closer to sequential consistency than RA, which we study in this paper. In particular, unlike RA, x86-TSO is "multi-copy-atomic" (writes by one thread are made globally visible to *all* other threads at the same time) and successful RMW operations are immediately globally visible. Additionally, the parallel composition construct in Jagadeesan et al. [25] is rather strong: threads are forked and joined only when the store buffers are empty. Being non-multi-copy-atomic, RA requires a more delicate notion of traces and closure rules, but it has more natural meta-theoretic properties, which one would expect from a programming language concurrency model: sequencing, a.k.a. thread-inlining, is unsound under x86-TSO [see 25, 31] but sound under RA (see Figure 10).

Burckhardt et al. [14] developed a denotational semantics for hardware weak memory models (including x86-TSO) following an alternative approach. They represent sequential code blocks by sequences of operations that the code performs, and close them under certain rewrite rules (reorderings and eliminations) that characterize the memory model. This approach does not validates important optimizations, such as Read-Read Elimination. Moreover, unlike x86-TSO, RA cannot be characterized by rewrite operations on SC traces [31].

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

Others present non-compositional techniques and tools to check refinement under weak memory models between whole-thread sequential programs that apply for any concurrent context. Poetzl and Kroening [43] 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. [39] 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, 32]. 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, 28, 41] 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 [28] use partial orders, Castellan, Paviotti et al. [15, 41] 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 [42]. These allow to modularly combine shared memory concurrency with other computational effects. Birkedal et al. [11] develop semantics for a type-and-effect system for SC memory which they use to enhance compiler optimizations based on assumptions on the context that come from the type system. We hope to the current work can serve as a basis to extend such accounts to weaker models.

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

References

1. Abadi, M., Plotkin, G.: A model of cooperative threads. Log. Methods Comput. Sci. **6**(4) (2010), https://doi.org/10.2168/LMCS-6(4:2)2010
2. Aguirre, A., Katsumata, S.y., Kura, S.: Weakest preconditions in fibrations. Mathematical Structures in Computer Science **32**(4) (2022), https://doi.org/10.1017/S0960129522000330
3. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: ESOP, LNCS, vol. 9032, Springer (2015), https://doi.org/10.1007/978-3-662-46669-8_12
4. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL, ACM (2011), https://doi.org/10.1145/1926385.1926394
5. Benton, N., Hofmann, M., Nigam, V.: Abstract effects and proof-relevant logical relations. In: POPL, ACM (2014)
6. Benton, N., Hofmann, M., Nigam, V.: Effect-dependent transformations for concurrent programs. In: PPDP, ACM (2016), https://doi.org/10.1145/2967973.2968602
7. Benton, N., Hughes, J., Moggi, E.: Monads and effects. In: APPSEM (2000)
8. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations with dynamic allocation. In: PPDP, ACM (2007)
9. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations: higher-order store. In: PPDP, ACM (2009)
10. Benton, N., Leperchey, B.: Relational reasoning in a nominal semantics for storage. In: TLCA, Springer (2005)
11. Birkedal, L., Sieczkowski, F., Thamsborg, J.: A concurrent logical relation. In: CSL, LIPIcs, vol. 16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012), https://doi.org/10.4230/LIPIcs.CSL.2012.107
12. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1-3) (2007), https://doi.org/10.1016/j.tcs.2006.12.034
13. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Inf. Comput. **127**(2) (1996), https://doi.org/10.1006/inco.1996.0056
14. Burckhardt, S., Musuvathi, M., Singh, V.: Verifying local transformations on relaxed memory models. In: CC, LNCS, vol. 6011, Springer (2010), https://doi.org/10.1007/978-3-642-11970-5_7
15. Castellan, S.: Weak memory models using event structures. In: JFLA, Saint-Malo, France (Jan 2016), URL https://hal.inria.fr/hal-01333582
16. Chakraborty, S., Vafeiadis, V.: Validating optimizations of concurrent C/C++ programs. In: CGO, ACM (2016), https://doi.org/10.1145/2854038.2854051
17. Cho, M., Lee, S., Hur, C., Lahav, O.: Modular data-race-freedom guarantees in the promising semantics. In: PLDI, ACM (2021), https://doi.org/10.1145/3453483.3454082

18. Cho, M., Lee, S., Lee, D., Hur, C., Lahav, O.: Sequential reasoning for optimizing compilers under weak memory concurrency. In: PLDI, ACM (2022), https://doi.org/10.1145/3519939.3523718
19. Dodds, M., Batty, M., Gotsman, A.: Compositional verification of compiler optimisations on relaxed memory. In: ESOP, LNCS, vol. 10801, Springer (2018), https://doi.org/10.1007/978-3-319-89884-1_36
20. Dvir, Y., Kammar, O., Lahav, O.: An algebraic theory for shared-state concurrency. In: APLAS, LNCS, vol. 13658, Springer (2022), https://doi.org/10.1007/978-3-031-21037-2_1
21. Dvir, Y., Kammar, O., Lahav, O.: A denotational approach to release/acquire concurrency (2024), URL http://tiny.cc/esop24ra
22. Hoare, T., van Staden, S.: The laws of programming unify process calculi. Sci. Comput. Program. **85** (2014), https://doi.org/10.1016/j.scico.2013.08.012
23. Hofmann, M.: Correctness of effect-based program transformations. In: Formal Logical Methods for System Security and Correctness, IOS Press (2008)
24. Jagadeesan, R., Jeffrey, A., Riely, J.: Pomsets with preconditions: a simple model of relaxed memory. Proc. ACM Program. Lang. **4**(OOPSLA) (2020), https://doi.org/10.1145/3428262
25. Jagadeesan, R., Petri, G., Riely, J.: Brookes is relaxed, almost! In: FOSSACS, LNCS, vol. 7213, Springer (2012), https://doi.org/10.1007/978-3-642-28729-9_12
26. Jeffrey, A., Riely, J., Batty, M., Cooksey, S., Kaysin, I., Podkopaev, A.: The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. Proc. ACM Program. Lang. **6**(POPL) (2022), https://doi.org/10.1145/3498716
27. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL, ACM (2017), https://doi.org/10.1145/3009837.3009850
28. Kavanagh, R., Brookes, S.: A denotational semantics for SPARC TSO. In: MFPS, ENTCS, vol. 341, Elsevier (2018), https://doi.org/10.1016/j.entcs.2018.03.025
29. Lahav, O.: Verification under causally consistent shared memory. ACM SIGLOG News **6**(2) (Apr 2019), https://doi.org/10.1145/3326938.3326942
30. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL, ACM (2016), https://doi.org/10.1145/2837614.2837643
31. Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: FM, LNCS, vol. 9995 (2016), https://doi.org/10.1007/978-3-319-48989-6_29
32. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI, ACM (2017), https://doi.org/10.1145/3062341.3062352
33. Lamport, L.: 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

34. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: POPL, ACM (2012), https://doi.org/10.1145/2103656.2103711
35. Liang, H., Feng, X., Fu, M.: Rely-guarantee-based simulation for compositional verification of concurrent program transformations. ACM Trans. Program. Lang. Syst. **36**(1) (2014), https://doi.org/10.1145/2576235
36. Maillard, K., Hriţcu, C., Rivas, E., Van Muylder, A.: The next 700 relational program logics. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019), https://doi.org/10.1145/3371072
37. Manson, J., Pugh, W.W., Adve, S.V.: The Java memory model. In: POPL, ACM (2005), https://doi.org/10.1145/1040305.1040336
38. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1) (1991), https://doi.org/10.1016/0890-5401(91)90052-4
39. Morisset, R., Pawan, P., Nardelli, F.Z.: Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In: PLDI, ACM (2013), https://doi.org/10.1145/2491956.2491967
40. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: TPHOLs, LNCS, vol. 5674, Springer (2009), https://doi.org/10.1007/978-3-642-03359-9_27
41. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular relaxed dependencies in weak memory concurrency. In: ESOP, LNCS, vol. 12075, Springer (2020), https://doi.org/10.1007/978-3-030-44914-8_22
42. Plotkin, G., Power, J.: Notions of computation determine monads. In: FOSSACS, Springer Berlin Heidelberg (2002)
43. Poetzl, D., Kroening, D.: Formalizing and checking thread refinement for data-race-free execution models. In: TACAS, LNCS, vol. 9636, Springer (2016), https://doi.org/10.1007/978-3-662-49674-9_30
44. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8. Proc. ACM Program. Lang. **2**(POPL) (2018), https://doi.org/10.1145/3158107
45. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI, ACM (2011), https://doi.org/10.1145/1993498.1993520
46. Turon, A.J., Wand, M.: A separation logic for refining concurrent objects. In: POPL, ACM (2011), https://doi.org/10.1145/1926385.1926415
47. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Nardelli, F.Z.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: POPL, ACM (2015), https://doi.org/10.1145/2676726.2676995
48. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2) (1997), https://doi.org/10.1007/BF01211617