Compositional Semantics for Shared-Variable Concurrency

MIKHAIL SVYATLOVSKIY, Tel Aviv University, Israel
SHAI MERMELSTEIN, Tel Aviv University, Israel
ORI LAHAV, Tel Aviv University, Israel

We revisit the fundamental problem of defining a compositional semantics for a concurrent programming language under sequentially consistent memory with the aim of equating the denotations of pieces of code if and only if these pieces induce the same behavior under all program contexts. While the denotational semantics presented by Brookes [Information and Computation 127, 2 (1996)] has been considered a definitive solution, we observe that Brookes's full abstraction result crucially relies on the availability of an impractical whole-memory atomic read-modify-write instruction. In contrast, we consider a language with standard primitives, which apply to a single variable. For that language, we propose an alternative denotational semantics based on traces that track program write actions together with the writes expected from the environment, and equipped with several closure operators to achieve necessary abstraction. We establish the adequacy of the semantics, and demonstrate full abstraction for the case that the analyzed code segment is loop-free. Furthermore, we show that by including a whole-memory atomic read in the language, one obtains full abstraction for programs with loops. To gain confidence, our results are fully mechanized in Coq.

CCS Concepts: • **Theory of computation → Concurrency**; **Denotational semantics**; • **Software and its engineering → Semantics**; **Compilers**.

Additional Key Words and Phrases: Denotational Semantics; Concurrency; Shared-Memory; Compiler Optimizations

ACM Reference Format:
Mikhail Svyatlovskiy, Shai Mermelstein, and Ori Lahav. 2024. Compositional Semantics for Shared-Variable Concurrency. *Proc. ACM Program. Lang.* 8, PLDI, Article 169 (June 2024), 24 pages. https://doi.org/10.1145/3656399

1 INTRODUCTION

Denotational semantics aims to define the meaning of a piece of code independently of the context under which it is executed. Generally speaking, such semantics assigns a denotation $[\![C]\!]$ to every command C of a given programming language in a way that satisfies the following desiderata:

Compositionality: The denotation of a command should be determined from the denotations of the command's immediate constituents. For instance, assuming a sequential composition operator, ";", we require that $[\![C_1 \,;\, C_2]\!]$ is a function of $[\![C_1]\!]$ and $[\![C_2]\!]$.

Adequacy: Assuming a given operational semantics, the denotations should only consider equivalent commands that operationally behave the same when plugged in an arbitrary program context. When denotations are partially ordered, we also want the semantics to admit a *directional* version of adequacy that targets contextual refinement under the operational semantics instead of contextual equivalence. For instance, assuming that denotations are sets, as is the case in our

Authors' Contact Information: Mikhail Svyatlovskiy, Tel Aviv University, Israel, mikhail.svyatlovskiy@phystech.edu; Shai Mermelstein, Tel Aviv University, Israel, shai.mermelstein@gmail.com; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il.

ACM 2475-1421/2024/6-ART169
https://doi.org/10.1145/3656399

development, (directional) adequacy ensures that $[\![C_1]\!] \subseteq [\![C_2]\!]$ implies that for every program context $P[-]$, every behavior of $P[C_1]$ under the operational semantics is also a behavior of $P[C_2]$. This makes denotations beneficial in supporting modular reasoning about the operational semantics, which by itself is only able to capture complete closed programs. In particular, an adequate denotational semantics can be used for formally justifying local program transformations, as performed by optimizing compilers. Indeed, adopting contextual refinement as the correctness criteria of program transformations, adequacy allows one to derive the correctness of a local transformation $C_{\mathrm{src}} \rightsquigarrow C_{\mathrm{tgt}}$ from $[\![C_{\mathrm{tgt}}]\!] \subseteq [\![C_{\mathrm{src}}]\!]$.

Full abstraction: Ideally, it is desirable for a denotational semantics to equate *all* pairs of commands that are contextually equivalent under the given operational semantics. A directional version requires that $[\![C_1]\!] \not\subseteq [\![C_2]\!]$ implies that for some program context $P[-]$, some behavior of $P[C_1]$ under the operational semantics is not a behavior of $P[C_2]$. Conceptually, full abstraction, together with compositionality and adequacy, means that $[\![C]\!]$ is indeed a precise compositional counterpart of the given operational semantics. A fully abstract denotational semantics provides a *complete* reasoning principle for correctness of local program transformations. Full abstraction is sometimes considered as the holy grail of denotational semantics and it is typically very difficult to obtain [Cardone 2021].

In this paper we consider concurrent programs that employ shared variables for inter-thread synchronization, governed by a non-deterministic scheduler that cannot be controlled by the program. For this domain, developing compositional, adequate, and fully abstract semantics is highly challenging. Indeed, the standard approach for (non-deterministic) sequential programs, which models programs as transformations from an initial state to a set of final states, fails to provide compositional semantics for parallelism, since the state transformation induced by a parallel composition $C_1 \parallel C_2$ cannot be determined from those of C_1 and C_2. One needs more detailed structures to capture the behaviors of C_1 and C_2, but being too concrete risks full abstraction.

This problem was addressed by Brookes [1996] (see there also a discussion about earlier attempts). In Brookes's approach, the semantics $[\![C]\!]$ of a command C is given by a set of sequences of transitions from memory to memory, assuming arbitrary environment interference between transitions. For example, a sequence of the form $\langle s_1, s_1' \rangle, \langle s_2, s_2' \rangle$ consisting of two transitions represents the case that C did some steps to transform s_1 to s_1'; then the environment did some steps transforming s_1' to s_2; and then C continued its execution from s_2 and terminated in s_2'. Brookes showed how these sequences can be derived from a given command by first deriving a concrete set of sequences, and then closing it under two closure operators, called *mumble* and *stutter*. In particular, $[\![C_1 \parallel C_2]\!]$ is obtained by considering all interleavings of sequences of $[\![C_1]\!]$ with sequences of $[\![C_2]\!]$, and closing the resulting set under the two closure operators. Brookes demonstrated compositionality, adequacy, and full abstraction for this semantics.

However, the programming language assumed in [Brookes 1996] employs a command of the form (`await` B `then` C) that implements a "conditional critical region": it blocks the execution as long as B is unmet, and then in a single atomic step it verifies that B holds and fully runs C. Since B and C may involve multiple variables, this construct can implement arbitrary atomic (finite) memory-to-memory transformations (e.g., $[\mathtt{x}_1 \mapsto 0, \ldots, \mathtt{x}_{100} \mapsto 0]$ to $[\mathtt{x}_1 \mapsto 1, \ldots, \mathtt{x}_{100} \mapsto 100]$), which requires all other components to be suspended and is unrealistic in practical concurrency. Removing (or restricting) `await` does not harm compositionality or adequacy, but, Brookes's full-abstraction proof relies on `await` instructions for building a concurrent context $P[-]$ that precisely mimics the environment transitions in a given sequence. In fact, the starting point for the current work is our observation that there are commands C_1 and C_2 that behave the same when plugged in program

contexts without `await`, but can be differentiated by contexts that use `await` (see Examples 3.7, 4.6 and 5.1 below). Therefore, Brookes's semantics is too concrete for a language without `await`.

The main contribution of this work is a novel denotational semantics that addresses this problem. We propose two models:

- A "concrete semantics" in which denotations track the write operations performed by the command interleaved with environment writes. For example, $\mathtt{W}(\mathtt{x},1),\bar{\mathtt{W}}(\mathtt{x},2),\mathtt{W}(\mathtt{y},1)$ represents the case that C writes 1 to x, expects the environment to write 2 to x, and then writes 1 to y. The concrete semantics is compositional and adequate, but it is not fully abstract. Nevertheless, since, in contrast to [Brookes 1996], this semantics reflects the property of the operational semantics that each transition updates at most one variable and since, again in contrast to [Brookes 1996], we do not record read operations in our denotations, the concrete semantics suffices for validating a wide variety of contextual refinements (see Fig. 4 below).
- An "abstract semantics" obtained by closing concrete denotations under four rewrite rules, each of which mimics a certain operational simulation argument allowing one to hide and introduce component writes from the concrete trace. We show that the abstract semantics is also compositional and adequate, whereas full abstraction holds up to some level:
  - Full abstraction fully holds if we have a "snapshot" instruction that blocks the execution until some condition is met (a restriction of `await` to instances of the form `await` B `then skip`).
  - Without "snapshot", we establish a restricted version of full abstraction: if C_2 is loop-free and $[\![C_1]\!] \not\subseteq [\![C_2]\!]$, then there exists a context $P[-]$ such that some behavior of $P[C_1]$ is not a behavior of $P[C_2]$. Thus, the abstract semantics is always *sound* for validating local program transformations $C_{\mathrm{src}} \rightsquigarrow C_{\mathrm{tgt}}$, and it provides a *complete* reasoning principle when C_{src} is loop-free. When C_{src} has loops, we provide a (rather complicated) counterexample for full abstraction of our abstract semantics (see Example 4.28 below).

Instead of `await` instructions the language we assume employs standard read-modify-write (RMW) constructs that perform an atomic update of a *single* variable at a time. A natural question is whether, like `await`, RMWs allow concurrent contexts to distinguish between commands that are indistinguishable for contexts consisting solely of reads and writes. We answer this question affirmatively by demonstrating such cases (see Example 5.1 below). Moreover, we show that by strengthening one of the four rewrite rules used to define the abstract semantics, we obtain a denotational semantics that enables compositional reasoning about program transformations under the assumption that the context cannot perform RMW operations.

Finally, we note certain limitations of the current work (see also §6). All of them raise interesting questions for future work to which our approach may constitute a starting point.

- We assume that the underlying memory ensures *sequential consistency* (SC, for short) [Lamport 1979]—the strongest memory model with simple operational semantics based on interleaving concurrent manipulations of a standard variables-to-values mapping.
- Our notion of behavior under the operational semantics is based on *partial correctness*, that is: we only consider terminating executions as inducing program behaviors. Accordingly, contextual refinement ensures that the target program preserves safety properties of the source, but it is termination-insensitive, where a diverging program refines every program. Since a compositional characterization of partial correctness is already challenging, we left the question of termination to future work. This is in line with multiple previous works that consider only terminating executions [Liang et al. 2012, 2014; Turon and Wand 2011]. Nevertheless, Brookes [1996, §10] includes an extension to termination-sensitive refinement, using infinite sequences and assuming certain fairness conditions on the operational semantics.

expressions	$E ::=$	$v \mid a \mid E + E \mid E = E \mid \ldots$
let expressions	$L ::=$	$E \mid x \mid \texttt{XCHG}(x, E) \mid \texttt{FAA}(x, E)$
commands	$C ::=$	$\texttt{skip} \mid x := E \mid \texttt{let}\ a = L\ \texttt{in}\ C \mid$ $C\,;C \mid C \parallel C \mid C \oplus C \mid \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ x\ \texttt{do}\ C \mid$ $\texttt{assume}(E) \mid \texttt{snapshot}(s) \mid x := * \mid \texttt{while}\ *\ \texttt{do}\ C$
contexts	$P ::=$	$- \mid \texttt{let}\ a = L\ \texttt{in}\ P \mid P\,;C \mid C\,;P \mid P \parallel C \mid C \parallel P \mid C \oplus P \mid P \oplus C \mid$ $\texttt{if}\ E\ \texttt{then}\ P\ \texttt{else}\ C \mid \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ P \mid \texttt{while}\ x\ \texttt{do}\ P \mid \texttt{while}\ *\ \texttt{do}\ P$

Fig. 1. Syntax: Expressions, Let Expressions, Commands, and Contexts

- Our programming language is a first-order language. Fully abstract semantics for higher-order languages have proved elusive [Cardone 2021], but we hope that our model can be useful for a higher-order language with a full abstraction guarantee that applies to its first-order fragment.

Outline. The rest of this paper is structured as follows. In §2 we present the syntax and operational semantics of the language studied in this paper. In §3 we present the concrete denotational semantics, establish its compositionality and adequacy, and demonstrate various transformations it validates (Fig. 4). In §4 we present the abstract denotational semantics, establish its compositionality (§4.1), adequacy (§4.2), and (restricted as discussed above) full abstraction (§4.3 and §4.4), and demonstrate transformations validated by the abstract semantics but not by the concrete one (Fig. 5). In §5 we present the modification of the abstract semantics under the assumption that the context does not perform RMWs. Finally, in §6 we discuss related and future work.

Artifact. Our results are fully mechanized in Coq, and the proof scripts are available in https://doi.org/10.5281/zenodo.10925596.

2 SYNTAX, OPERATIONAL SEMANTICS, AND CONTEXTUAL REFINEMENT

In this section we present the syntax of the studied programming language, its operational semantics, and the notion of contextual refinement w.r.t. that semantics.

Syntax. We assume a set $\mathsf{Var} \triangleq \{\mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots\}$ of shared variables, ranged over by x, y; a set $\mathsf{LVar} \triangleq \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots\}$ of local variables, ranged over by a, b; and a set $\mathsf{Val} \triangleq \{0, 1, 2, \ldots\}$ of values, ranged over by v. We define a *state* s to be a function in $\mathsf{State} \triangleq \mathsf{Var} \to \mathsf{Val}$. In some examples below, we use $\mathsf{s}_0 \triangleq \lambda x.\, 0$ as the initial state.

Figure 1 presents the grammar for expressions, let expressions, commands, and contexts. Expressions are defined standardly and are composed of values (v) and local variables (a). Let expressions are used on the right-hand side of let bindings, and include standard expressions, shared variables, and RMW primitives. The latter are used to atomically execute a read from memory followed by a write to memory. We consider two kinds of RMWs, whose intuitive semantics is as follows:[1]

- Exchange (XCHG) loads from a shared variable and modifies it to a given argument.
- Fetch-And-Add (FAA) increments a shared variable by a given argument.

These instructions return the value they read, before the modification was performed.

Commands are mostly customary for a (first-order) imperative parallel language, with several choices that may deserve attention:

[1] Our Coq development also has Compare-And-Swap (CAS) instructions, which are elided here.

$$\frac{v = [\![E]\!]}{\langle E, s\rangle \xrightarrow{\varepsilon} \langle v, s\rangle} \qquad \frac{v = s(x)}{\langle x, s\rangle \xrightarrow{\varepsilon} \langle v, s\rangle} \qquad \frac{v = s(x) \quad v' = [\![E]\!] \quad s' = s[x \mapsto v']}{\langle \mathtt{XCHG}(x, E), s\rangle \xrightarrow{\mathtt{W}(x,v')} \langle v, s'\rangle} \qquad \frac{v = s(x) \quad v' = v + [\![E]\!] \quad s' = s[x \mapsto v']}{\langle \mathtt{FAA}(x, E), s\rangle \xrightarrow{\mathtt{W}(x,v')} \langle v, s'\rangle}$$

$$\frac{v = [\![E]\!] \quad s' = s[x \mapsto v]}{\langle x := E, s\rangle \xrightarrow{\mathtt{W}(x,v)} \langle \mathtt{skip}, s'\rangle} \qquad \frac{\langle L, s\rangle \xrightarrow{\gamma} \langle v, s'\rangle}{\langle \mathtt{let}\ a = L\ \mathtt{in}\ C, s\rangle \xrightarrow{\gamma} \langle C\{v/a\}, s'\rangle} \qquad \frac{\langle C_1, s\rangle \xrightarrow{\gamma} \langle C_1', s'\rangle}{\langle C_1 ; C_2, s\rangle \xrightarrow{\gamma} \langle C_1' ; C_2, s'\rangle} \qquad \frac{}{\langle \mathtt{skip} ; C, s\rangle \xrightarrow{\varepsilon} \langle C, s\rangle}$$

$$\frac{\langle C_1, s\rangle \xrightarrow{\gamma} \langle C_1', s'\rangle}{\langle C_1 \parallel C_2, s\rangle \xrightarrow{\gamma} \langle C_1' \parallel C_2, s'\rangle} \qquad \frac{\langle C_2, s\rangle \xrightarrow{\gamma} \langle C_2', s'\rangle}{\langle C_1 \parallel C_2, s\rangle \xrightarrow{\gamma} \langle C_1 \parallel C_2', s'\rangle} \qquad \frac{}{\langle C \parallel \mathtt{skip}, s\rangle \xrightarrow{\varepsilon} \langle C, s\rangle} \qquad \frac{}{\langle \mathtt{skip} \parallel C, s\rangle \xrightarrow{\varepsilon} \langle C, s\rangle}$$

$$\frac{[\![E]\!] \neq 0 \implies i = 1 \quad [\![E]\!] = 0 \implies i = 2}{\langle \mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s\rangle \xrightarrow{\varepsilon} \langle C_i, s\rangle} \qquad \frac{s(x) \neq 0}{\langle \mathtt{while}\ x\ \mathtt{do}\ C, s\rangle \xrightarrow{\varepsilon} \langle C ; \mathtt{while}\ x\ \mathtt{do}\ C, s\rangle} \qquad \frac{s(x) = 0}{\langle \mathtt{while}\ x\ \mathtt{do}\ C, s\rangle \xrightarrow{\varepsilon} \langle \mathtt{skip}, s\rangle}$$

$$\frac{[\![E]\!] \neq 0}{\langle \mathtt{assume}(E), s\rangle \xrightarrow{\varepsilon} \langle \mathtt{skip}, s\rangle} \qquad \frac{}{\langle \mathtt{snapshot}(s), s\rangle \xrightarrow{\varepsilon} \langle \mathtt{skip}, s\rangle} \qquad \frac{s' = s[x \mapsto v]}{\langle x := *, s\rangle \xrightarrow{\mathtt{W}(x,v)} \langle \mathtt{skip}, s'\rangle}$$

$$\frac{i \in \{1, 2\}}{\langle C_1 \oplus C_2, s\rangle \xrightarrow{\varepsilon} \langle C_i, s\rangle} \qquad \frac{}{\langle \mathtt{while} * \mathtt{do}\ C, s\rangle \xrightarrow{\varepsilon} \langle C ; \mathtt{while} * \mathtt{do}\ C, s\rangle} \qquad \frac{}{\langle \mathtt{while} * \mathtt{do}\ C, s\rangle \xrightarrow{\varepsilon} \langle \mathtt{skip}, s\rangle}$$

Fig. 2. Small-Step Semantics: $\langle L, s\rangle \xrightarrow{\gamma} \langle v, s'\rangle$ and $\langle C, s\rangle \xrightarrow{\gamma} \langle C', s'\rangle$

- Parallel composition, "$\parallel$", is a first class construct that can be employed arbitrarily deep inside other commands, rather than top-level parallel composition which is sometimes assumed when studying semantics of parallel languages.
- We include non-deterministic choices—between commands ($C_1 \oplus C_2$), stored values ($x := *$), and as a loop termination condition ($\mathtt{while} * \mathtt{do}\ C$).
- Less standardly, we use functional-style let bindings for assigning values to local variables. This allows us to restrict the scope of these variables inside a command, in a way that a parallel context cannot change or directly observe. Loops use global variables in the termination condition.
- A non-standard $\mathtt{snapshot}(s)$ command is used to block the execution until the memory is in state s (see operational semantics below).

In examples, we also use ($\mathtt{if}\ E\ \mathtt{then}\ C$) for ($\mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ \mathtt{skip}$), and employ syntactic sugar incorporating loads inside expressions, such as $x := y$ for $\mathtt{let}\ \mathtt{a} = y\ \mathtt{in}\ x := \mathtt{a}$ and $\mathtt{assume}(x = v)$ for $\mathtt{let}\ \mathtt{a} = x\ \mathtt{in}\ \mathtt{assume}(\mathtt{a} = v)$. We denote by $\mathsf{fv}(E)$ (respectively, $\mathsf{fv}(C)$) the set of local variables that occur free in an expression E (command C), and call an expression E (command C) *closed* if $\mathsf{fv}(E) = \emptyset$ ($\mathsf{fv}(C) = \emptyset$). We write $C\{v/a\}$ for the command obtained from C by substituting the free occurrences of a by v.

Finally, Fig. 1 specifies "contexts" which are defined standardly as commands with one "hole". We write $P[C]$ for the command obtained by "plugging in" the command C in P, that is: substituting the unique $-$ in P by C.

Operational Semantics. We assume that closed expressions are evaluated to values using a function $[\![\cdot]\!]$ in a standard way. The operational semantics of commands is given in Fig. 2 as a "small-step" transition relation between configurations, which are tuples of the form $\langle C, s\rangle$, where C is a

command and $s \in \mathsf{State}$. For a uniform definition of let bindings, it uses a "helper" relation which defines how let expressions are evaluated to values and affect the state.

The operational semantics is mostly standard. We use syntactic substitution to handle let bindings, so that steps execute only on closed commands. Parallelism is captured by arbitrary interleaving of component steps, with non-preemptive scheduling, in the sense that there are no explicit language constructs for controlling the scheduler. The shared memory follows the SC model, where each read reads the latest written value recorded in the state. To assist later definitions, the transitions are labeled with a write label of the form $\mathtt{W}(x, v)$ (with $x \in \mathsf{Var}$ and $v \in \mathsf{Val}$) or with an ε label when no write is performed. We often omit the label from the transition, writing $\langle C, s\rangle \to \langle C', s'\rangle$ to mean that $\langle C, s\rangle \xrightarrow{\gamma} \langle C', s'\rangle$ for some γ.

Note that no steps are associated with `skip` or with $\mathtt{assume}(E)$ when $[\![E]\!] = 0$. Intuitively, a state of the form $\langle \mathtt{skip}, s\rangle$ is a valid final state ("a value"). We write $\langle C, s\rangle \downarrow s'$ if $\langle C, s\rangle \to^* \langle \mathtt{skip}, s'\rangle$.[2]

Contextual Refinement. Contextual refinement under the operational semantics is identified with soundness of local program transformations, which is defined as follows:

Definition 2.1. A transformation from a command C_{src} to a command C_{tgt} is *sound*, denoted by $C_{\mathrm{src}} \rightsquigarrow C_{\mathrm{tgt}}$, if $\langle P[C_{\mathrm{tgt}}], s\rangle \downarrow s'$ implies $\langle P[C_{\mathrm{src}}], s\rangle \downarrow s'$ for every context P such that $P[C_{\mathrm{src}}]$ and $P[C_{\mathrm{tgt}}]$ are closed. We write $C_1 \leftrightsquigarrow C_2$ when both $C_1 \rightsquigarrow C_2$ and $C_2 \rightsquigarrow C_1$ hold.

Example 2.2. For $C_1 = \mathtt{x := x + 1}$ and $C_2 = \mathtt{let\ a = FAA(x, 1)\ in\ skip}$, we have $C_1 \rightsquigarrow C_2$ but $C_2 \not\rightsquigarrow C_1$. For the former, we can execute a load followed by a store in one atomic step to simulate the effect of FAA. (The denotational semantics below provides a formal account.) For the latter, with $P = - \parallel \mathtt{x := 1 ; x := 3}$, we have $\langle P[C_i], \mathtt{s}_0\rangle \downarrow \mathtt{s}_0[\mathtt{x} \mapsto 2]$ for $i = 1$ but not for $i = 2$.

The following transitivity and congruence properties are easy to establish:

Lemma 2.3. *If $C_1 \rightsquigarrow C_2$ and $C_2 \rightsquigarrow C_3$, then $C_1 \rightsquigarrow C_3$.*

Proof. Suppose that $P[C_1]$ and $P[C_3]$ are closed and $\langle P[C_3], s\rangle \downarrow s'$. Let $a_1, \ldots, a_n$ be an enumeration of $\mathsf{fv}(P[C_2])$ and $P' = \mathtt{let}\ a_1 = 0\ \mathtt{in}\ (\mathtt{let}\ a_2 = 0\ \mathtt{in}\ (\ldots \mathtt{let}\ a_n = 0\ \mathtt{in}\ P)\ \ldots))$. Then, $P'[C_1]$, $P'[C_2]$, and $P'[C_3]$ are all closed, and for $i \in \{1, 3\}$, we have $\langle P'[C_i], s\rangle \downarrow s'$ iff $\langle P[C_i], s\rangle \downarrow s'$. Then, $C_2 \rightsquigarrow C_3$ implies that $\langle P'[C_2], s\rangle \downarrow s'$, and, then, $C_1 \rightsquigarrow C_2$ implies that $\langle P[C_1], s\rangle \downarrow s'$, □

Lemma 2.4. *If $C_{\mathrm{src}} \rightsquigarrow C_{\mathrm{tgt}}$, then $P[C_{\mathrm{src}}] \rightsquigarrow P[C_{\mathrm{tgt}}]$ for every context P.*

Proof. Easily follows from the fact that for every two contexts P and P', there exists a context P'' such that $P''[C] = P'[P[C]]$ for every command C. □

Comparison with [Brookes 1996]. We conclude this section by discussing the relation of the above definitions to the corresponding ones in [Brookes 1996], and motivating the differences. We note that Brookes [1996] introduces two operational semantics for his language that differ in the level of granularity, and we compare ours to the one with finer levels of granularity [Brookes 1996, §9].

Putting `snapshot` aside, in the above operational semantics, every instruction involves at most one shared variable, which allows us to easily prove the following property:

Lemma 2.5. *If $\langle C, s\rangle \xrightarrow{\gamma} \langle C', s'\rangle$ and C is snapshot-free, then there exists some $x \in \mathsf{Var}$ such that for every s_1 with $s_1(x) = s(x)$, we have $\langle C, s_1\rangle \xrightarrow{\gamma} \langle C', s_1[x \mapsto s'(x)]\rangle$.*

[2] We denote by $S^?$ and S^* the reflexive and reflexive-transitive closures of a binary relation S (respectively) and write $S_1 ; S_2$ for relational composition, i.e., $S_1 ; S_2 \triangleq \{\langle x, z\rangle \mid \exists y.\, \langle x, y\rangle \in S_1 \wedge \langle y, z\rangle \in S_2\}$.

This is in contrast with [Brookes 1996], which has "await" instructions of the form `await` B `then` C, where B is a boolean condition, which may read from several shared variables, and C is a finite sequence of assignments that read and write to shared variables.

To formulate `await` in our terminology, we use *extended expressions*, ranged over by $\mathcal{E}$, that consist of values as well as *shared variables* (e.g., $\mathcal{E} = \mathtt{x} + \mathtt{y} + 9$). A standard function $[\![\mathcal{E}]\!]_s$ evaluates $\mathcal{E}$ at state s. Then, the operational semantics of `await` is formalized as follows:

$$\textsc{await}\ \frac{[\![\mathcal{E}]\!]_{s_0} \neq 0 \qquad \forall 1 \leq i \leq n.\, s_i = s_{i-1}[x_i \mapsto [\![\mathcal{E}_i]\!]_{s_{i-1}}]}{\langle \mathtt{await}\ \mathcal{E}\ \mathtt{then}\ (x_1 := \mathcal{E}_1\ ;\ \ldots\ ;\ x_n := \mathcal{E}_n), s_0\rangle \to \langle \mathtt{skip}, s_n\rangle}$$

In one atomic step, the system performs the loads from memory necessary to evaluate $\mathcal{E}$ and (conditionally) executes multiple assignments involving any number of additional loads in stores.

While being instrumental in the full abstraction proof, `await` is not standardly available in real-world shared-memory concurrent programming. Indeed, to implement `await`, one has to block all other concurrent processes from accessing any of the variables that are read/written in the `await` instruction. (Note that it does not suffice to only block concurrent `await`'s, we also need to block primitive loads and stores.) Instead, programming languages and multicore architectures provide atomic instructions that atomically manipulate a single address, including loads, stores, and RMWs. Locks, transactional libraries, concurrent objects, and other synchronization mechanisms are implemented on top of these basic instructions. Such implementations necessarily involve races—cases in which two different threads are concurrently accessing the same variable, and at least one of them is writing. Our focus is on concurrent implementations at this level of abstraction.

As we show in §4.3 and §4.4, we are only able to develop fully abstract denotational semantics when the source program is loop-free. With loops, our proposed denotational semantics is adequate but not fully abstract. To get full abstraction with loops, we use `snapshot`, which, like `await`, we consider to be unrealistic. The `snapshot` command uses only the "condition part" of the `await`, and can be thought of the restriction of `await` to the form `await` B `then skip`. (Since every program uses only finitely many variables, the state s used in `snapshot` can be always translated into an extended expression.) Thus, our results provide a full abstraction statement similar to [Brookes 1996] but without the full power of `await`. The (pretended) implementation of `snapshot` has to block all other concurrent processes from *writing* to shared variables, but unlike `await`, reads can proceed concurrently.

Finally, we note that for a single variable, we also have `assume`$(x = v)$ behaving like Brookes's `await` $x = v$ `then skip`, generating "no behavior" if the condition (on a single variable) is not met. We use assume commands in the full abstraction proof, but since we only consider terminating behaviors in this work, it is also possible to use busy-loops that wait until $x = v$.

3 CONCRETE DENOTATIONAL SEMANTICS

In this section we present the "concrete" denotational semantics and establish its compositionality and adequacy. The main ingredient for this semantics is our notion of a *trace*, which consists of an initial memory state, an initial *store*, which assigns values to local variables, and a *chronicle*, which is a sequence of *actions* performed by the command along with those expected by the concurrent context. Next, we formally define these objects and the required operations on them.

Notation 3.1 (Sequences). For a finite alphabet Σ, we denote by Σ^* the set of all (finite) sequences over Σ. We use ε to denote the empty sequence. We write $s_1 \cdot s_2$ for the concatenation of sequences, which is lifted to concatenation of sets of sequences in the obvious way. We identify symbols with sequences of length 1 or their singletons when needed (e.g., in expressions like $\sigma \cdot S$).

Stores. A *store* is a function $\theta \in \mathsf{Store} \triangleq \mathsf{LVar} \to \mathsf{Val}$. Stores are extended to expressions in the standard way. We also lift stores to let expressions by applying them inside (e.g., $\theta(\mathtt{FAA}(x, E)) = \mathtt{FAA}(x, \theta(E))$). In some examples below, we use $\theta_0 \triangleq \lambda a.\, 0$ as the initial store.

Actions. An *action* α is either a *component write* of the form $\mathtt{W}(x, v)$ with $x \in \mathsf{Var}$ and $v \in \mathsf{Val}$, or an *environment write* of the form $\bar{\mathtt{W}}(x, v)$ with $x \in \mathsf{Var}$ and $v \in \mathsf{Val}$. We write Act, CmpW, and EnvW for the set of all actions, component writes, and environment writes (respectively).

Chronicles. A *chronicle* c is a finite sequence of actions. We denote by Chro the set of all chronicles, by CmpChro the set of all chronicles consisting solely of component writes, and by EnvChro the set of all chronicles consisting solely of environment writes. A chronicle c induces a function from states to states, recursively defined by: $\varepsilon(s) \triangleq s$ and $(\mathtt{W}(x, v) \cdot c)(s) = (\bar{\mathtt{W}}(x, v) \cdot c)(s) \triangleq c(s[x \mapsto v])$.

Traces. A *trace* is a triple $t = \langle s, \theta, c \rangle \in \mathsf{Trace} \triangleq \mathsf{State} \times \mathsf{Store} \times \mathsf{Chro}$. We refer to the three components as the initial state (s), the initial store (θ), and the chronicle (c) of t, and to the state $c(s)$ as the *final state* of the trace t.

Sequential Composition of Traces. The sequential composition of $t = \langle s, \theta, c \rangle$ and $t' = \langle c(s), \theta, c' \rangle$, denoted by $t\,;\,t'$, is the trace $\langle s, \theta, c \cdot c' \rangle$. When the final state of t does not coincide with the initial state of t' or the two traces do not have the same initial store, then $t\,;\,t'$ is undefined.

Parallel Composition of Traces. Parallel composition is defined for actions, chronicles, and traces:

(1) The *dual* of an action α, denoted by $\bar{\alpha}$, is defined by $\bar{\alpha} \triangleq \bar{\mathtt{W}}(x, v)$ if $\alpha = \mathtt{W}(x, v)$ and $\bar{\alpha} \triangleq \mathtt{W}(x, v)$ if $\alpha = \bar{\mathtt{W}}(x, v)$. Two actions α and α' are *parallelly composable* if either $\alpha = \bar{\alpha'}$ or $\alpha = \alpha' \in \mathsf{EnvW}$. In that case, their *parallel composition*, denoted by $\alpha \parallel \alpha'$, is given by:

$$\alpha \parallel \alpha' \triangleq \begin{cases} \mathtt{W}(x, v) & \alpha = \bar{\alpha'} \in \{\mathtt{W}(x, v), \bar{\mathtt{W}}(x, v)\} \\ \alpha & \alpha = \alpha' \in \mathsf{EnvW} \end{cases}$$

(2) The *parallel composition* of two chronicles $c = \alpha_1 \cdots \alpha_n$ and $c' = \alpha'_1 \cdots \alpha'_n$, denoted by $c_1 \parallel c_2$, is defined by $c \parallel c' \triangleq (\alpha_1 \parallel \alpha'_1) \cdots (\alpha_n \parallel \alpha'_n)$. If some $\alpha_i \parallel \alpha'_i$ is undefined or the chronicles are not of the same length, then $c \parallel c'$ is undefined.

(3) The *parallel composition* of two traces $t = \langle s, \theta, c \rangle$ and $t' = \langle s, \theta, c' \rangle$, denoted by $t \parallel t'$, is the trace $\langle s, \theta, c \parallel c' \rangle$. When the two traces do not have the same initial state and store or $c \parallel c'$ is undefined, then $t \parallel t'$ is undefined.

From Commands to Traces. Figure 3 presents an inductive definition of the concrete semantics, which is a function $\lfloor \cdot \rfloor$ that maps commands to sets of traces. The `skip` command does not perform any component writes and tolerates any environment interference, thus it is associated with traces with arbitrary environment chronicles (rule SKIP). A store instruction generates a component write, and allows arbitrary environment interference before and after (rule STORE). The value to be stored is determined according to the initial store. (Unlike the operational semantics, this semantics assigns meaning to open programs as well.) Let bindings (rule LET) start with environment interference e and then possibly generate a component write $\mathtt{W}(x, v)$ following their operational semantics (reusing the first part of Fig. 2). Note that the memory visible to the let expression is the one obtained by applying e on the initial state. In turn, the continuation is given by C starting from a modified state and store. The resulting chronicle is the concatenation of $e, \gamma \in \{\mathtt{W}(x, v), \varepsilon\}$, and a chronicle c of the continuation. Here we use the transition labels from the operational semantics as component actions or the empty chronicles. Sequential composition of commands is handled by sequential composition of traces (rule SEQ). The denotation of parallel composition uses a (partial) operation for parallel composition of traces (rule PAR). Intuitively speaking, a component action

$$\textsc{skip}\ \frac{e \in \mathsf{EnvChro}}{\langle s,\theta,e\rangle \in \lfloor \mathtt{skip}\rfloor} \qquad \textsc{store}\ \frac{e_1,e_2 \in \mathsf{EnvChro} \quad \alpha = \mathtt{W}(x,\theta(E))}{\langle s,\theta,e_1\cdot\alpha\cdot e_2\rangle \in \lfloor x := E\rfloor} \qquad \textsc{let}\ \frac{e \in \mathsf{EnvChro} \quad \langle \theta(L), e(s)\rangle \xrightarrow{\gamma} \langle v, s'\rangle \quad \langle s', \theta[a\mapsto v], c\rangle \in \lfloor C\rfloor}{\langle s,\theta,e\cdot\gamma\cdot c\rangle \in \lfloor \mathtt{let}\ a = L\ \mathtt{in}\ C\rfloor}$$

$$\textsc{seq}\ \frac{t_1 \in \lfloor C_1\rfloor \quad t_2 \in \lfloor C_2\rfloor}{t_1 ; t_2 \in \lfloor C_1 ; C_2\rfloor} \qquad \textsc{par}\ \frac{t_1 \in \lfloor C_1\rfloor \quad t_2 \in \lfloor C_2\rfloor}{t_1 \parallel t_2 \in \lfloor C_1 \parallel C_2\rfloor} \qquad \textsc{choice}\ \frac{t \in \lfloor C_1\rfloor \cup \lfloor C_2\rfloor}{t \in \lfloor C_1 \oplus C_2\rfloor} \qquad \textsc{if}\ \frac{\theta(E)\neq 0 \implies \langle s,\theta,c\rangle \in \lfloor C_1\rfloor \quad \theta(E) = 0 \implies \langle s,\theta,c\rangle \in \lfloor C_2\rfloor}{\langle s,\theta,c\rangle \in \lfloor \mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\rfloor}$$

$$\textsc{while-true}\ \frac{\langle e(s),\theta,c\rangle \in \lfloor C\rfloor \quad e(s)(x)\neq 0 \quad t \in \lfloor \mathtt{while}\ x\ \mathtt{do}\ C\rfloor}{\langle s,\theta,e\cdot c\rangle ; t \in \lfloor \mathtt{while}\ x\ \mathtt{do}\ C\rfloor} \qquad \textsc{while-false}\ \frac{e_1,e_2 \in \mathsf{EnvChro} \quad e_1(s)(x) = 0}{\langle s,\theta,e_1\cdot e_2\rangle \in \lfloor \mathtt{while}\ x\ \mathtt{do}\ C\rfloor} \qquad \textsc{assume}\ \frac{e \in \mathsf{EnvChro} \quad \theta(E)\neq 0}{\langle s,\theta,e\rangle \in \lfloor \mathtt{assume}(E)\rfloor}$$

$$\textsc{snapshot}\ \frac{e_1,e_2 \in \mathsf{EnvChro} \quad e_1(s) = s'}{\langle s,\theta,e_1\cdot e_2\rangle \in \lfloor \mathtt{snapshot}(s')\rfloor} \qquad \textsc{havoc}\ \frac{e_1,e_2 \in \mathsf{EnvChro} \quad \alpha = \mathtt{W}(x,v)}{\langle s,\theta,e_1\cdot\alpha\cdot e_2\rangle \in \lfloor x := *\rfloor} \qquad \textsc{nd-while-true}\ \frac{t_1 \in \lfloor C\rfloor \quad t_2 \in \lfloor \mathtt{while}\ *\ \mathtt{do}\ C\rfloor}{t_1 ; t_2 \in \lfloor \mathtt{while}\ *\ \mathtt{do}\ C\rfloor} \qquad \textsc{nd-while-false}\ \frac{e \in \mathsf{EnvChro}}{\langle s,\theta,e\rangle \in \lfloor \mathtt{while}\ *\ \mathtt{do}\ C\rfloor}$$

Fig. 3. Concrete Trace Semantics: $t \in \lfloor C\rfloor$

on one side has to match the environment action expected from the other side, and together they form a component action for their external environment. In addition, if both sides expect the same environment action, then that action is also expected from the external environment of the parallel composition. The concrete semantics of other language constructs follow similar ideas aiming to match their operational semantics. As expected, for loops, the definition is recursive.

The concrete denotations admit some invariants, which are useful in our proofs. In particular, they are closed over environment actions before and after that command's effects:

Proposition 3.2. *If $\langle s,\theta,c\rangle \in \lfloor C\rfloor$, then $\langle s',\theta,e_1\cdot c\cdot e_2\rangle \in \lfloor C\rfloor$ for every environment chronicles e_1, e_2 and state s' such that $e_1(s') = s$.*

This invariant is explicitly enforced in some rules (e.g., skip, store), whereas other rules close over the prefix (e.g., let) or not close at all (e.g., seq) since they inherit the closure from their parts.

Example 3.3. For $C = \mathtt{let\ a = x\ in\ (x := a + 1)}$, $\lfloor C\rfloor$ consists of all traces of the form $\langle s,\theta,e_1\cdot e_2\cdot \mathtt{W(x}, v+1)\cdot e_3\rangle$ where $e_1,e_2,e_3 \in \mathsf{EnvChro}$ and $v = e_1(s)(\mathtt{x})$. In addition, $\lfloor \mathtt{x := 1}\rfloor$ consists of all traces of the form $\langle s,\theta,e_1\cdot \mathtt{W(x,1)}\cdot e_2\rangle$ where $e_1,e_2 \in \mathsf{EnvChro}$. For their parallel composition, $\lfloor C \parallel \mathtt{x := 1}\rfloor$ consists of all traces of the form $\langle s,\theta,e_1\cdot e_2\cdot \mathtt{W(x},v+1)\cdot e_3\cdot \mathtt{W(x,1)}\cdot e_4\rangle$ or $\langle s,\theta,e_1\cdot e_2\cdot \mathtt{W(x,1)}\cdot e_3\cdot \mathtt{W(x},v+1)\cdot e_4\rangle$ where $e_1,e_2,e_3,e_4 \in \mathsf{EnvChro}$ and $v = e_1(s)(\mathtt{x})$; and $\langle s,\theta,e_1\cdot \mathtt{W(x,1)}\cdot e_2\cdot e_3\cdot \mathtt{W(x},v+1)\cdot e_4\rangle$ where $e_1,e_2,e_3,e_4 \in \mathsf{EnvChro}$ and $v = e_2(s[\mathtt{x}\mapsto 1])(\mathtt{x})$.

Compositionality. From the definition of the semantics, it is easy to see that the concrete semantics is compositional. More formally, the following property is proved by standard induction on contexts (with an inner induction on the derivation of $t \in \lfloor \mathtt{while\ \_\ do}\ C\rfloor$ for loops):

Lemma 3.4. *If $\lfloor C_1\rfloor \subseteq \lfloor C_2\rfloor$, then $\lfloor P[C_1]\rfloor \subseteq \lfloor P[C_2]\rfloor$ for every context P.*

As a corollary, we obtain the compositionality of $\lfloor\cdot\rfloor$: for every command C whose immediate sub-commands are $C_1,\ldots,C_n$, we have that $\lfloor C\rfloor$ is a function of $\lfloor C_1\rfloor,\ldots,\lfloor C_n\rfloor$. To see this, consider for instance the case of $C = C_1 \parallel C_2$, and suppose that $\lfloor C_1\rfloor = \lfloor C_1'\rfloor$ and $\lfloor C_2\rfloor = \lfloor C_2'\rfloor$. By Lemma 3.4 applied to the context $P = - \parallel C_2$ and the commands C_1, C_1', we obtain $\lfloor C_1 \parallel C_2\rfloor = \lfloor C_1' \parallel C_2\rfloor$.

Then, again by Lemma 3.4 applied to the context $P = C_1' \parallel -$ and the commands C_2, C_2', we obtain $\lfloor C_1' \parallel C_2 \rfloor = \lfloor C_1' \parallel C_2' \rfloor$. Together, it follows that $\lfloor C_1 \parallel C_2 \rfloor = \lfloor C_1' \parallel C_2' \rfloor$.

Adequacy. The next lemma provides the key for the adequacy of the concrete semantics.

Lemma 3.5. *For a closed command C, we have $\langle C, s\rangle \downarrow s'$ iff $\langle s, \theta, c\rangle \in \lfloor C \rfloor$ for some store θ and component chronicle c such that $c(s) = s'$.*

Proof. For the proof we inductively define an auxiliary relation $\overset{c}{\Longrightarrow}$ between configurations labeled with a chronicle c, which represents an operational execution interrupted with the environment writes along c (akin to Brookes [1996]'s "state trace" behaviors):

$$\frac{}{\langle C, s\rangle \overset{\varepsilon}{\Longrightarrow} \langle C, s\rangle} \qquad \frac{\alpha = \bar{\mathsf{W}}(x, v) \qquad \langle C, s[x \mapsto v]\rangle \overset{c}{\Longrightarrow} \langle C', s'\rangle}{\langle C, s\rangle \overset{\alpha \cdot c}{\Longrightarrow} \langle C', s'\rangle} \qquad \frac{\langle C, s\rangle \xrightarrow{\gamma} \langle C', s'\rangle \qquad \langle C', s'\rangle \overset{c}{\Longrightarrow} \langle C'', s''\rangle}{\langle C, s\rangle \overset{\gamma \cdot c}{\Longrightarrow} \langle C'', s''\rangle}$$

Then, the claim of the lemma is a direct corollary of the following two claims. First, when c is a component chronicle, $\overset{c}{\Longrightarrow}$ trivially coincides with the operational semantics:
Claim 3.5.1: $\langle C, s\rangle \downarrow s'$ iff $\langle C, s\rangle \overset{c}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$ for some $c \in \mathsf{CmpChro}$ such that $c(s) = s'$.
Second, $\lfloor C \rfloor$ lies in tight correspondence with $\overset{c}{\Longrightarrow}$:
Claim 3.5.2: Let C be a command, and let $a_1, \ldots, a_n$ be an enumeration of $\mathrm{fv}(C)$. Then, $\langle s, \theta, c\rangle \in \lfloor C \rfloor$ iff $\langle C\{\theta(a_1)/a_1\} \ldots \{\theta(a_n)/a_n\}, s\rangle \overset{c}{\Longrightarrow} \langle \mathtt{skip}, c(s)\rangle$. The proof of each direction in this claim proceeds by induction on C, where the interesting cases follow from the fact that $\overset{c}{\Longrightarrow}$ is compatible with sequential and parallel compositions. More concretely, for the left-to-right direction, we prove that: (1) if $\langle C_1, s\rangle \overset{c_1}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$ and $\langle C_2, s'\rangle \overset{c_2}{\Longrightarrow} \langle \mathtt{skip}, s''\rangle$, then $\langle C_1 ; C_2, s\rangle \overset{c_1 \cdot c_2}{\Longrightarrow} \langle \mathtt{skip}, s''\rangle$; and (2) if $\langle C_1, s\rangle \overset{c_1}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$ and $\langle C_2, s\rangle \overset{c_2}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$, then $\langle C_1 \parallel C_2, s\rangle \overset{c_1 \| c_2}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$. For the converse, we prove: (3) if $\langle C_1 ; C_2, s\rangle \overset{c}{\Longrightarrow} \langle \mathtt{skip}, s''\rangle$, then $\langle C_1, s\rangle \overset{c_1}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$ and $\langle C_2, s'\rangle \overset{c_2}{\Longrightarrow} \langle \mathtt{skip}, s''\rangle$ for some chronicles c_1, c_2 such that $c = c_1 \cdot c_2$ and state s'; and (4) if $\langle C_1 \parallel C_2, s\rangle \overset{c}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$, then $\langle C_1, s\rangle \overset{c_1}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$ and $\langle C_2, s\rangle \overset{c_2}{\Longrightarrow} \langle \mathtt{skip}, s'\rangle$ for some chronicles c_1, c_2 such that $c = c_1 \parallel c_2$. □

Adequacy of the concrete semantics is now a corollary:

Theorem 3.6. *If $\lfloor C_{\mathrm{tgt}} \rfloor \subseteq \lfloor C_{\mathrm{src}} \rfloor$, then $C_{\mathrm{src}} \rightsquigarrow C_{\mathrm{tgt}}$.*

Proof. Suppose that $\lfloor C_{\mathrm{tgt}} \rfloor \subseteq \lfloor C_{\mathrm{src}} \rfloor$. Let P be a context such that $P[C_{\mathrm{src}}]$ and $P[C_{\mathrm{tgt}}]$ are closed, and suppose that $\langle P[C_{\mathrm{tgt}}], s\rangle \downarrow s'$. Since $\langle P[C_{\mathrm{tgt}}], s\rangle \downarrow s'$, by Lemma 3.5, we have $\langle s, \theta, c\rangle \in \lfloor P[C_{\mathrm{tgt}}] \rfloor$ for some store θ and component chronicle $c \in \mathsf{CmpChro}$ such that $c(s) = s'$. Since $\lfloor C_{\mathrm{tgt}} \rfloor \subseteq \lfloor C_{\mathrm{src}} \rfloor$, by Lemma 3.4, it follows that $\langle s, \theta, c\rangle \in \lfloor P[C_{\mathrm{src}}] \rfloor$. By Lemma 3.5, it follows that $\langle P[C_{\mathrm{src}}], s\rangle \downarrow s'$. □

Figure 4 presents examples of program transformations that are validated by the concrete semantics. Among RMWs, we only list transformations involving FAA, but similar transformations can be shown for XCHG and CAS (and some are included in our Coq development).

Many of the transformations in Fig. 4 are *structural transformations* revealing the algebraic properties of the language operators. In particular, *generalized sequencing* reduces parallel composition to sequential composition. Indeed, by introducing and eliminating `skip` instructions, using generalized sequencing we obtain that $C_1 \parallel C_2 \rightsquigarrow C_1 ; C_2$ for every C_1 and C_2. This transformation is typically considered counterproductive for performance (although it saves the time it takes to spawn a thread), but it shows the expected monotonicity property of the operational semantics, which does not hold under some weak memory models [Lahav and Vafeiadis 2016].

Other transformations involve memory accesses. As a concrete example of the style of reasoning in these proofs, consider the case of unused load elimination/introduction. The traces in $\lfloor \mathtt{let}\ a =$

Algebraic laws of sequential composition

$(C_1 ; C_2) ; C_3 \leftrightsquigarrow C_1 ; (C_2 ; C_3)$ $\quad$ $\texttt{skip} ; C \leftrightsquigarrow C$ $\quad$ $C ; \texttt{skip} \leftrightsquigarrow C$

Reordering of local operations

$\texttt{let}\ a = E\ \texttt{in let}\ a' = E'\ \texttt{in}\ C \leftrightsquigarrow \texttt{let}\ a' = E'\ \texttt{in let}\ a = E\ \texttt{in}\ C$ provided that $a \neq a'$, $a \notin \mathrm{fv}(E')$, and $a' \notin \mathrm{fv}(E)$

$\texttt{let}\ a = E\ \texttt{in}\ x := E ; C \leftrightsquigarrow \texttt{let}\ a = E\ \texttt{in}\ x := a ; C$ provided that $a \notin \mathrm{fv}(E)$

$\texttt{if}\ E\ \texttt{then}\ (\texttt{let}\ a = E'\ \texttt{in}\ C_1)\ \texttt{else}\ (\texttt{let}\ a = E'\ \texttt{in}\ C_2) \leftrightsquigarrow \texttt{let}\ a = E'\ \texttt{in}\ (\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2)$ provided that $a \notin \mathrm{fv}(E)$

Unused assignment elimination

$\texttt{let}\ a = E\ \texttt{in}\ C \leftrightsquigarrow C$ provided that $a \notin \mathrm{fv}(C)$

Loop unrolling

$\texttt{while}\ x\ \texttt{do}\ C \leftrightsquigarrow \texttt{let}\ a = x\ \texttt{in}\ (\texttt{if}\ a\ \texttt{then}\ (C ; \texttt{while}\ x\ \texttt{do}\ C))$ provided that $a \notin \mathrm{fv}(C)$

Algebraic laws of parallel composition

$\texttt{skip} \parallel C \leftrightsquigarrow C$ $\quad$ $C_1 \parallel C_2 \leftrightsquigarrow C_2 \parallel C_1$ $\quad$ $(C_1 \parallel C_2) \parallel C_3 \leftrightsquigarrow C_1 \parallel (C_2 \parallel C_3)$

Generalized Sequencing (a.k.a. thread inlining/sequentialization)

$(C_1 ; C_1') \parallel (C_2 ; C_2') \rightsquigarrow (C_1 \parallel C_2) ; (C_1' \parallel C_2')$

Algebraic laws of non-deterministic choice and distributivity over non-deterministic choice

$C_1 \oplus C_2 \leftrightsquigarrow C_2 \oplus C_1$ $\quad$ $(C_1 \oplus C_2) \oplus C_3 \leftrightsquigarrow C_1 \oplus (C_2 \oplus C_3)$ $\quad$ $C \oplus C \leftrightsquigarrow C$ $\quad$ $C_1 \oplus C_2 \rightsquigarrow C_1$ $\quad$ $C_1 \oplus C_2 \rightsquigarrow C_2$

$C ; (C_1 \oplus C_2) \leftrightsquigarrow (C ; C_1) \oplus (C ; C_2)$ $\quad$ $(C_1 \oplus C_2) ; C \leftrightsquigarrow (C_1 ; C) \oplus (C_2 ; C)$ $\quad$ $C \parallel (C_1 \oplus C_2) \leftrightsquigarrow (C \parallel C_1) \oplus (C \parallel C_2)$

Load-after-load elimination

$\texttt{let}\ a = x\ \texttt{in}\ (\texttt{let}\ b = x\ \texttt{in}\ C) \rightsquigarrow \texttt{let}\ a = x\ \texttt{in}\ (\texttt{let}\ b = a\ \texttt{in}\ C)$

Load-after-store elimination

$x := E ; \texttt{let}\ a = x\ \texttt{in}\ C \rightsquigarrow x := E ; \texttt{let}\ a = E\ \texttt{in}\ C$

Unused load elimination/introduction

$\texttt{let}\ a = x\ \texttt{in}\ C \leftrightsquigarrow C$ provided that $a \notin \mathrm{fv}(C)$

Load-after-FAA elimination

$\texttt{let}\ a = \texttt{FAA}(x, E)\ \texttt{in let}\ a' = x\ \texttt{in}\ C \rightsquigarrow \texttt{let}\ a = \texttt{FAA}(x, E)\ \texttt{in let}\ a' = a + E\ \texttt{in}\ C$ provided that $a \notin \mathrm{fv}(E)$

Load-before-FAA elimination

$\texttt{let}\ a = x\ \texttt{in let}\ a' = \texttt{FAA}(x, E)\ \texttt{in}\ C \rightsquigarrow \texttt{let}\ a = \texttt{FAA}(x, E)\ \texttt{in let}\ a' = a\ \texttt{in}\ C$ provided that $a \notin \mathrm{fv}(E)$

Load-store to FAA

$\texttt{let}\ a = x\ \texttt{in}\ (x := a + E ; C) \rightsquigarrow \texttt{let}\ a = \texttt{FAA}(x, E)\ \texttt{in}\ C$ provided that $a \notin \mathrm{fv}(E)$

Assume introduction/elimination

$\texttt{skip} \rightsquigarrow \texttt{assume}(E)$ $\quad$ $C \rightsquigarrow \texttt{assume}(0)$ $\quad$ $\texttt{assume(x = 0)} \leftrightsquigarrow \texttt{while x do skip}$

Specializing non-deterministic values

$x := * \rightsquigarrow x := v$ $\quad$ $\texttt{while}\ *\ \texttt{do}\ C \rightsquigarrow \texttt{while}\ x\ \texttt{do}\ C$

Fig. 4. Examples of program transformations validated by the concrete semantics

$x\ \texttt{in}\ C\rfloor$ are by definition of the form $\langle s, \theta, e \cdot c\rangle$ with $e \in \mathsf{EnvChro}$ and $\langle e(s), \theta[a \mapsto e(s)(x)], c\rangle \in \lfloor C\rfloor$. When $a \notin \mathrm{fv}(C)$, the latter holds iff $\langle e(s), \theta, c\rangle \in \lfloor C\rfloor$. Then, by picking $e = \varepsilon$, we obtain $\lfloor C\rfloor \subseteq \lfloor \texttt{let}\ a = x\ \texttt{in}\ C\rfloor$ (load elimination). The converse (load introduction) follows from the observation that $\langle e(s), \theta, c\rangle \in \lfloor C\rfloor$ implies that $\langle s, \theta, e \cdot c\rangle \in \lfloor C\rfloor$ for every command C. Our Coq development provides general lemmas that are repeatedly used in these arguments.

Example 3.7. The concrete semantics captures some refinements that are invalid in [Brookes 1996]. Indeed, every command in the language we study changes at most one shared variable. This is reflected in traces since every action in them mentions one variable. For instance, using the

concrete semantics we can show that $C_1 \,;\, C_2 \,;\, C_3 \rightsquigarrow C_1 \,;\, C_3$ for:[3]

$$C_1 = \mathtt{x := 1 \,;\, y := 1}$$
$$C_2 = \mathtt{let\ a = x\ in\ (let\ b = y\ in\ (assume((a = 2 \wedge b \neq 2) \vee (a \neq 2 \wedge b = 2))))}$$
$$C_3 = \mathtt{let\ a = x\ in\ (let\ b = y\ in\ (assume((a = 2 \wedge b = 2))))}$$

In Brookes's setting, this refinement fails to hold. For example, the condition in C_2 will never be satisfied in the context $- \parallel \mathtt{await\ true\ then\ (x := 2 \,;\, y := 2)}$.

4 ABSTRACT SEMANTICS

The semantics above fully tracks the sequence of writes performed by a command. There are, however, contextual refinements in which writes are eliminated or introduced. The "abstract semantics" presented in this section supports such refinements. The main idea is to close the concrete sets of traces under certain rewrite rules that hide or introduce actions that can be safely assumed to be unobservable by the concurrent environment. Then, it may be the case that some traces in $\lfloor C_{\mathsf{tgt}} \rfloor$ are not in $\lfloor C_{\mathsf{src}} \rfloor$, but they are in the closure of $\lfloor C_{\mathsf{src}} \rfloor$ under these rewrites.

The main technical challenge lies in identifying these rewrite rules and proving the required properties for this semantics. In §4.1, we establish the compositionality property, which, unlike the case of the concrete semantics, is not a direct corollary of the definition, and requires a new argument. Then, in §4.2, we show how adequacy of the abstract semantics follows from its compositionality and Lemma 3.5 about the concrete semantics. In §4.3, we show that the set of rules is "complete" by establishing full abstraction. In §4.4, we consider full abstraction in the absence of `snapshot`.

Notation 4.1 (Rewrite Rules and Closures). A *rewrite rule* x is a binary relation on syntactic objects. We use the notation $a \xrightarrow{x} b$ to mean that $\langle a, b \rangle \in x$. For a set X of rewrite rules, we write $a \xrightarrow{X} b$ if $a \xrightarrow{x} b$ for some $x \in X$. A set A is *closed* under X if $b \in A$ whenever $a \xrightarrow{X} b$ for some $a \in A$. Assuming some universal set $\mathcal{A}$, the *closure* of A under X, denoted by A^X, is defined as the smallest subset of $\mathcal{A}$ that contains A and is closed under X.

The following general propositions are useful in the sequel.

Proposition 4.2. *For every $A \subseteq \mathcal{A}$ and set X of rewrite rules, $A^X = \{b \in \mathcal{A} \mid \exists a \in A.\, a \xrightarrow{X}{}^{*}\ b\}$*

Proposition 4.3. *For every $A, B \subseteq \mathcal{A}$ and set X of rewrite rules, $A \subseteq B^X$ implies $A^X \subseteq B^X$.*

We define the abstract semantics using four rewrite rules. The rules aim to match operational arguments for cases where it is possible to eliminate a redundant idempotent write, eliminate several writes that cancel each other, or introduce an invisible write. The examples following the definition provide the intuition behind each rewrite rule.

Definition 4.4. The *abstract denotation* of a command C, denoted by $[\![C]\!]$, is defined by $[\![C]\!] \triangleq \lfloor C \rfloor^{\mathcal{R}}$, where $\mathcal{R}$ consists of the following rewrite rules on traces:

COALESCE: $\langle s, \theta, c_1 \cdot m_1 \cdot \mathtt{W}(x, v) \cdot m_2 \cdot c_2 \rangle \xrightarrow{\text{COALESCE}} \langle s, \theta, c_1 \cdot \mathtt{W}(x, v) \cdot c_2 \rangle$ provided that $m_1, m_2 \in \mathsf{CmpChro}$ and $(m_1 \cdot \mathtt{W}(x, v) \cdot m_2)(c_1(s)) = c_1(s)[x \mapsto v]$.

$\overline{\text{COALESCE}}$: $\langle s, \theta, c_1 \cdot m_1 \cdot \bar{\mathtt{W}}(x, v) \cdot m_2 \cdot c_2 \rangle \xrightarrow{\overline{\text{COALESCE}}} \langle s, \theta, c_1 \cdot \bar{\mathtt{W}}(x, v) \cdot c_2 \rangle$ provided that $m_1, m_2 \in \mathsf{CmpChro}$, $(m_1 \cdot \bar{\mathtt{W}}(x, v) \cdot m_2)(c_1(s)) = c_1(s)[x \mapsto v]$, and $m_1(c_1(s))(x) = c_1(s)(x)$.

DEL-RED: $\langle s, \theta, c_1 \cdot \mathtt{W}(x, v) \cdot c_2 \rangle \xrightarrow{\text{DEL-RED}} \langle s, \theta, c_1 \cdot c_2 \rangle$ provided that $c_1(s)(x) = v$.

ADD-RED: $\langle s, \theta, c_1 \cdot c_2 \rangle \xrightarrow{\text{ADD-RED}} \langle s, \theta, c_1 \cdot \mathtt{W}(x, v) \cdot c_2 \rangle$ provided that $c_1(s)(x) = v$.

[3] To assist the reader, we highlight the commands eliminated by a transformation.

Below we freely use Prop. 4.3 and show $\lfloor C_{\text{tgt}} \rfloor \subseteq [\![C_{\text{src}}]\!]$ instead of $[\![C_{\text{tgt}}]\!] \subseteq [\![C_{\text{src}}]\!]$.

Example 4.5 (Rule COALESCE*).* Rule COALESCE permits to combine consecutive component writes into one "atomic block". The condition $(m_1 \cdot \mathtt{W}(x,v) \cdot m_2)(c_1(s)) = c_1(s)[x \mapsto v]$ ensures that the effect of the formed block is the same as the effect of a single write. As an example, let $C_{\text{src}} = \mathtt{let\ a = y\ in\ (y := 1\,;x := 1\,;y := a)}$. Intuitively speaking, we can always execute it atomically, without letting the environment to interfere in between the first load and the final store. In that case, it behaves like $C_{\text{tgt}} = \mathtt{x := 1}$. Such reasoning is impossible in the concrete semantics, but the rule COALESCE of the abstract semantics is allowing us exactly that, thus justifying $C_{\text{src}} \rightsquigarrow C_{\text{tgt}}$. Indeed, to show that $\lfloor C_{\text{tgt}} \rfloor \subseteq [\![C_{\text{src}}]\!]$ observe that every trace t in $\lfloor C_{\text{tgt}} \rfloor$ has the form $\langle s, \theta, e_1 \cdot \mathtt{W(x,1)} \cdot e_2 \rangle$. We can start from a corresponding trace in $[\![C_{\text{src}}]\!]$ of the form $\langle s, \theta, e_1 \cdot \mathtt{W(y,1)} \cdot \mathtt{W(x,1)} \cdot \mathtt{W}(\mathtt{y}, e_1(s)(\mathtt{y})) \cdot e_2 \rangle$ and rewrite by COALESCE with $c_1 = e_1$, $m_1 = \mathtt{W(y,1)}$, $m_2 = \mathtt{W}(\mathtt{y}, e_1(s)(\mathtt{y}))$, and $c_2 = e_2$ to obtain t.

Example 4.6 (Rule $\overline{\text{COALESCE}}$*).* Rule $\overline{\text{COALESCE}}$ allows one to "attach" component actions to an environment action, provided that the composed block has the same effect as the single environment action. To see this in action, let $C_{\text{src}} = \mathtt{let\ a = y\ in\ y := 3\,;if\ x \neq 2\ then\ (if\ x = 2\ then\ y := a)}$. The two if conditions are satisfied only if the concurrent environment changes x from non-zero value to zero. In this case, we can encompass that environment store of x with the load from y and the store of 3 to y just before, and the store of the previous value of y just after, and in this case C_{src} behaves like $C_{\text{tgt}} = \mathtt{if\ x \neq 2\ then\ (if\ x = 2\ then\ skip\ else\ y := 3)\ else\ y := 3}$. The rule $\overline{\text{COALESCE}}$ of the abstract semantics is needed for that, thus justifying $C_{\text{src}} \rightsquigarrow C_{\text{tgt}}$. Indeed, to show that $\lfloor C_{\text{tgt}} \rfloor \subseteq [\![C_{\text{src}}]\!]$ observe that the traces in $\lfloor C_{\text{tgt}} \rfloor$ are either of the form $\langle s, \theta, e_1 \cdot \bar{\mathtt{W}}(\mathtt{x},2) \cdot e_2 \rangle$ with $e_1(s)(\mathtt{x}) \neq 2$ or of the form $\langle s, \theta, e_1 \cdot \mathtt{W(y,3)} \cdot e_2 \rangle$. Traces of the latter form are directly in $\lfloor C_{\text{src}} \rfloor$. For a trace t of the first form, we start from a corresponding trace in $\lfloor C_{\text{src}} \rfloor$ of the form $\langle s, \theta, e_1 \cdot \mathtt{W(y,3)} \cdot \bar{\mathtt{W}}(\mathtt{x},2) \cdot \mathtt{W}(\mathtt{y}, e_1(s)(\mathtt{y})) \cdot e_2 \rangle$ and rewrite by $\overline{\text{COALESCE}}$ with $c_1 = e_1$, $m_1 = \mathtt{W(y,3)}$, $m_2 = \mathtt{W}(\mathtt{y}, e_1(s)(\mathtt{y}))$, and $c_2 = e_2$ to obtain t. As in COALESCE, the condition $(m_1 \cdot \bar{\mathtt{W}}(x,v) \cdot m_2)(c_1(s)) = c_1(s)[x \mapsto v]$ of $\overline{\text{COALESCE}}$ ensures that the formed atomic block affects the memory exactly as the single environment store. We note that in Brookes's setting, the transformation $C_{\text{src}} \rightsquigarrow C_{\text{tgt}}$ fails to hold. For the context $P = - \parallel \mathtt{await\ (x \neq 2 \wedge y \neq 3)\ then\ x := 2}$, starting from a state with $\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 2$, only $P[C_{\text{tgt}}]$ terminates in a state with $\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 2$.

Example 4.7 (Second side-condition of $\overline{\text{COALESCE}}$*).* Attaching component actions to an environment write $\bar{\mathtt{W}}(x,v)$ may fail if these actions modify x and the environment write is due to an RMW on x. This is the reason for the condition $m_1(c_1(s))(x) = c_1(s)(x)$ in $\overline{\text{COALESCE}}$. For example, using x instead of y in the commands in Example 4.6, without this condition, we would obtain:

$$\mathtt{let\ a = x\ in\ x := 3\,;if\ x \neq 2\ then\ (if\ x = 2\ then\ x := a)} \rightsquigarrow \mathtt{if\ x \neq 2\ then\ (if\ x = 2\ then\ skip\ else\ x := 3)\ else\ x := 3}$$

However, starting from $\mathtt{x} \mapsto 0$, in parallel to $\mathtt{FAA(x,2)}$, only the target can terminate with $\mathtt{x} \mapsto 2$.

Example 4.8 (Rule DEL-RED*).* Executing $C_{\text{src}} = \mathtt{let\ a = x\ in\ x := a}$ atomically is invisible for the concurrent environment, behaving like skip. In the concrete semantics, we cannot prove $C_{\text{src}} \rightsquigarrow \mathtt{skip}$ since all chronicles of $\lfloor C_{\text{src}} \rfloor$ have one component write, whereas those of $\lfloor \mathtt{skip} \rfloor$ have none. The rule DEL-RED is needed here. Indeed, to show that $\lfloor \mathtt{skip} \rfloor \subseteq [\![C_{\text{src}}]\!]$, we start with an arbitrary trace t in $\lfloor \mathtt{skip} \rfloor$, which must have the form $\langle s, \theta, e \rangle$. Then, a corresponding trace in $\lfloor C_{\text{src}} \rfloor$ of the form $\langle s, \theta, e \cdot \mathtt{W}(\mathtt{x}, e(s)(\mathtt{x})) \rangle$ can be rewritten to t by DEL-RED with $c_1 = e$ and $c_2 = \varepsilon$.

Example 4.9 (Rule ADD-RED*).* In the operational semantics fetch-and-add by 0 is equivalent to a read. In particular, for $C_{\text{src}} = \mathtt{let\ a = x\ in\ y := a}$ and $C_{\text{tgt}} = \mathtt{let\ a = FAA(x,0)\ in\ y := a}$, we have

$C_{\text{src}} \rightsquigarrow C_{\text{tgt}}$. This cannot be shown by the concrete semantics since chronicles of C_{src} have only one component write, while those of C_{tgt} have two such writes. To show that $\lfloor C_{\text{tgt}} \rfloor \subseteq [\![C_{\text{src}}]\!]$, we start with an arbitrary trace t in $\lfloor C_{\text{tgt}} \rfloor$, which must have the form $\langle s, \theta, e_1 \cdot \mathtt{W}(\mathsf{x}, v) \cdot e_2 \cdot \mathtt{W}(\mathsf{y}, v) \cdot e_3 \rangle$ with $v = e_1(s)(\mathsf{x})$. Then, a corresponding trace in $\lfloor C_{\text{src}} \rfloor$ of the form $\langle s, \theta, e_1 \cdot e_2 \cdot \mathtt{W}(\mathsf{y}, v) \cdot e_3 \rangle$ can be rewritten to t by ADD-RED with $c_1 = e_1$ and $c_2 = e_2 \cdot \mathtt{W}(\mathsf{y}, v) \cdot e_3$.

Example 4.10. Rule ADD-RED is also necessary for a language without RMWs. For:

$$C_{\text{src}} = \mathtt{let\ a = y\ in\ (y := 1\ ;\ (if\ x \neq 0\ then\ x := 0)\ ;\ y := a)} \qquad C_{\text{tgt}} = \mathtt{assume(x = 0)\ ;\ x := 0}$$

we have $C_{\text{src}} \rightsquigarrow C_{\text{tgt}}$, but $\lfloor C_{\text{tgt}} \rfloor \subseteq [\![C_{\text{src}}]\!]$ cannot be established without ADD-RED.

Remark 4.11. In the presence of COALESCE and ADD-RED, the rule DEL-RED can be strengthened:

DEL-RED′: $\langle s, \theta, c_1 \cdot m \cdot c_2 \rangle \xrightarrow{\text{DEL-RED}'} \langle s, \theta, c_1 \cdot c_2 \rangle$ provided that $m \in \mathsf{CmpChro}$ and $m(c_1(s)) = c_1(s)$.

Indeed, we can rewrite as follows using an arbitrary $x \in \mathsf{Var}$, and then apply DEL-RED:

$$\langle s, \theta, c_1 \cdot m \cdot c_2 \rangle \xrightarrow{\text{ADD-RED}} \langle s, \theta, c_1 \cdot m \cdot \mathtt{W}(x, c_1(s)(x)) \cdot c_2 \rangle \xrightarrow{\text{COALESCE}} \langle s, \theta, c_1 \cdot \mathtt{W}(x, c_1(s)(x)) \cdot c_2 \rangle.$$

4.1 Compositionality

We establish the compositionality of $[\![C]\!]$. First, to handle sequential composition, we observe that the rules of $\mathcal{R}$ can be applied inside sequential composition of traces:

PROPOSITION 4.12. *The following hold for every* $\mathsf{r} \in \mathcal{R}$*:*

- *If* $t_1 \xrightarrow{\mathsf{r}} t_1'$ *and* $t_1' \,;\, t_2$ *is defined, then* $t_1 \,;\, t_2 \xrightarrow{\mathsf{r}} t_1' \,;\, t_2$.
- *If* $t_2 \xrightarrow{\mathsf{r}} t_2'$ *and* $t_1 \,;\, t_2'$ *is defined, then* $t_1 \,;\, t_2 \xrightarrow{\mathsf{r}} t_1 \,;\, t_2'$.

From this property, we obtain the following proposition, which solves the case of sequential composition in the compositionality proof.

PROPOSITION 4.13. *If* $\lfloor C_1 \rfloor \subseteq [\![C_1']\!]$*, then* $\lfloor C_1 \,;\, C_2 \rfloor \subseteq [\![C_1' \,;\, C_2]\!]$*. Similarly, if* $\lfloor C_2 \rfloor \subseteq [\![C_2']\!]$*, then* $\lfloor C_1 \,;\, C_2 \rfloor \subseteq [\![C_1 \,;\, C_2']\!]$.

PROOF. We prove the first claim and the second proof is symmetric. Suppose that $\lfloor C_1 \rfloor \subseteq [\![C_1']\!]$. Let $t \in \lfloor C_1 \,;\, C_2 \rfloor$. By definition, we have $t = t_1 \,;\, t_2$ for some $t_1 \in \lfloor C_1 \rfloor$ and $t_2 \in \lfloor C_2 \rfloor$. Our assumption entails that $t_1 \in [\![C_1']\!]$. Let $t_1' \in \lfloor C_1' \rfloor$ such that $t_1' \xrightarrow{\mathcal{R}}{}^{*} t_1$. By Prop. 4.12, $t_1' \,;\, t_2 \xrightarrow{\mathcal{R}}{}^{*} t$. In particular, $t_1' \,;\, t_2$ is defined, and thus by definition we have $t_1' \,;\, t_2 \in \lfloor C_1' \,;\, C_2 \rfloor$. It follows that $t \in [\![C_1' \,;\, C_2]\!]$. □

Handling parallel composition is more difficult. Indeed, a claim like Prop. 4.12 does not hold for parallel composition instead of sequential composition: since the rewrite rules change the chronicle in the trace, it may be that $t_1 \xrightarrow{\mathcal{R}} t_1'$ and $t_1' \parallel t_2$ is defined, but $t_1 \parallel t_2$ is undefined. We address this problem by showing that in such cases there must be another trace t_2' that satisfies $t_1 \parallel t_2' \xrightarrow{\mathcal{R}}{}^{*} t_1' \parallel t_2$ and belongs to any concrete denotation that t_2 belongs to. For the formal argument, we introduce a set $\mathcal{D} \triangleq \{\xrightarrow{\overline{\text{DISPERSE}}}, \xrightarrow{\text{DISPERSE}}, \xrightarrow{\text{ADD-}\overline{\text{RED}}}, \xrightarrow{\text{DEL-}\overline{\text{RED}}}\}$ of "dual" rewrite rules:

$\overline{\text{DISPERSE}}$: $\langle s, \theta, c_1 \cdot \bar{\mathtt{W}}(x, v) \cdot c_2 \rangle \xrightarrow{\overline{\text{DISPERSE}}} \langle s, \theta, c_1 \cdot e_1 \cdot \bar{\mathtt{W}}(x, v) \cdot e_2 \cdot c_2 \rangle$ provided that
$e_1, e_2 \in \mathsf{EnvChro}$ and $(e_1 \cdot \bar{\mathtt{W}}(x, v) \cdot e_2)(c_1(s)) = c_1(s)[x \mapsto v]$.

DISPERSE: $\langle s, \theta, c_1 \cdot \mathtt{W}(x, v) \cdot c_2 \rangle \xrightarrow{\text{DISPERSE}} \langle s, \theta, c_1 \cdot e_1 \cdot \mathtt{W}(x, v) \cdot e_2 \cdot c_2 \rangle$ provided that
$e_1, e_2 \in \mathsf{EnvChro}$, $(e_1 \cdot \mathtt{W}(x, v) \cdot e_2)(c_1(s)) = c_1(s)[x \mapsto v]$, and $e_1(c_1(s))(x) = c_1(s)(x)$.

ADD-$\overline{\text{RED}}$: $\langle s, \theta, c_1 \cdot c_2 \rangle \xrightarrow{\text{ADD-}\overline{\text{RED}}} \langle s, \theta, c_1 \cdot \bar{\mathtt{W}}(x, v) \cdot c_2 \rangle$ provided that $c_1(s)(x) = v$.

DEL-$\overline{\text{RED}}$: $\langle s, \theta, c_1 \cdot \bar{\mathtt{W}}(x, v) \cdot c_2 \rangle \xrightarrow{\text{DEL-}\overline{\text{RED}}} \langle s, \theta, c_1 \cdot c_2 \rangle$ provided that $c_1(s)(x) = v$.

Proposition 4.14. *For every command C, $\lfloor C \rfloor$ is closed under $\mathcal{D}$.*

Proof. By induction on $t \in \lfloor C \rfloor$ using the following claims for the inductive step:
Claim 4.14.1: For every $\mathsf{d} \in \mathcal{D}$, if $t_1 \,;\, t_2 \xrightarrow{\mathsf{d}} t'$, then either $t' = t_1' \,;\, t_2$ for some t_1' such that $t_1 \xrightarrow{\mathsf{d}} t_1'$ or $t' = t_1 \,;\, t_2'$ for some t_2' such that $t_2 \xrightarrow{\mathsf{d}} t_2'$.
Claim 4.14.2: If $t_1 \parallel t_2 \xrightarrow{\mathsf{d}} t'$ for $\mathsf{d} \in \{\overline{\textsc{disperse}}, \textsc{add-red}, \textsc{del-red}\}$, then $t' = t_1' \parallel t_2'$ for some t_1' and t_2' such that $t_1 \xrightarrow{\mathsf{d}} t_1'$ and $t_2 \xrightarrow{\mathsf{d}} t_2'$.
Claim 4.14.3: If $t_1 \parallel t_2 \xrightarrow{\textsc{disperse}} t'$, then $t' = t_1' \parallel t_2'$ for some t_1' and t_2' satisfying one of the following:

$$t_1 \xrightarrow{\textsc{disperse}} t_1' \text{ and } t_2 \xrightarrow{\overline{\textsc{disperse}}} t_2' \quad \text{or} \quad t_1 \xrightarrow{\overline{\textsc{disperse}}} t_1' \text{ and } t_2 \xrightarrow{\textsc{disperse}} t_2'.$$

□

Proposition 4.15. *Suppose that $t_1' \parallel t_2'$ is defined.*

- *If $\mathsf{r} \in \mathcal{R}$ and $t_1 \xrightarrow{\mathsf{r}} t_1'$, then there exists t_2 such that $t_2' \xrightarrow{\mathcal{D}} t_2$ and $t_1 \parallel t_2 \xrightarrow{\mathsf{r}} t_1' \parallel t_2'$.*
- *If $\mathsf{r} \in \mathcal{R}$ and $t_2 \xrightarrow{\mathsf{r}} t_2'$, then there exists t_1 such that $t_1' \xrightarrow{\mathcal{D}} t_1$ and $t_1 \parallel t_2 \xrightarrow{\mathsf{r}} t_1' \parallel t_2'$.*

With Propositions 4.14 and 4.15, we obtain the variant of Prop. 4.13 to handle parallel composition:

Proposition 4.16. *If $\lfloor C_1 \rfloor \subseteq [\![C_1']\!]$, then $\lfloor C_1 \parallel C_2 \rfloor \subseteq [\![C_1' \parallel C_2]\!]$. Similarly, if $\lfloor C_2 \rfloor \subseteq [\![C_2']\!]$, then $\lfloor C_1 \parallel C_2 \rfloor \subseteq [\![C_1 \parallel C_2']\!]$.*

Proof. We prove the first claim and the second proof is symmetric. Suppose that $\lfloor C_1 \rfloor \subseteq [\![C_1']\!]$. Let $t \in \lfloor C_1 \parallel C_2 \rfloor$. By definition, $t = t_1 \parallel t_2$ for some $t_1 \in \lfloor C_1 \rfloor$ and $t_2 \in \lfloor C_2 \rfloor$. Our assumption entails that $t_1 \in [\![C_1']\!]$. To show that $t \in [\![C_1' \parallel C_2]\!]$, it suffices to show that for every t_1' such that $t_1' \xrightarrow{\mathcal{R}}{}^* t_1$, there exists $t_2' \in \lfloor C_2 \rfloor$ such that $t_1' \parallel t_2' \xrightarrow{\mathcal{R}}{}^* t$. By Prop. 4.14, it suffices to show that for every t_1' such that $t_1' \xrightarrow{\mathcal{R}}{}^* t_1$, there exists t_2' such that $t_2 \xrightarrow{\mathcal{D}}{}^* t_2'$ and $t_1' \parallel t_2' \xrightarrow{\mathcal{R}}{}^* t$. We prove this claim by induction on the number of rewrite steps in $t_1' \xrightarrow{\mathcal{R}}{}^* t_1$. In the base case we have $t_1' = t_1$ and we can take $t_2' = t_2$ and $t_1' \parallel t_2' = t$. For the induction step, suppose that for t_1' there exists t_2' such that $t_2 \xrightarrow{\mathcal{D}}{}^* t_2'$ and $t_1' \parallel t_2' \xrightarrow{\mathcal{R}}{}^* t$, and let t_1'' such that $t_1'' \xrightarrow{\mathcal{R}} t_1'$. By Prop. 4.15, there exists t_2'' such that $t_2' \xrightarrow{\mathcal{D}} t_2''$ and $t_1'' \parallel t_2'' \xrightarrow{\mathcal{R}} t_1' \parallel t_2'$. Thus, we have $t_2 \xrightarrow{\mathcal{D}}{}^* t_2''$ and $t_1'' \parallel t_2'' \xrightarrow{\mathcal{R}}{}^* t$. □

Using Propositions 4.13 and 4.16 for handling sequential and parallel composition, and similar lemmas for other constructs, we can easily establish the following lemma by induction on P:

Lemma 4.17. *If $[\![C_1]\!] \subseteq [\![C_2]\!]$, then $[\![P[C_1]]\!] \subseteq [\![P[C_2]]\!]$ for every context P.*

As discussed above for the concrete semantics (see discussion after Lemma 3.4), the compositionality of $[\![\cdot]\!]$ follows from Lemma 4.17. This also entails that there exists a (mathematical) function that maps *the denotations* of the immediate sub-commands of C to the denotation of C. To see this, consider again the case of $C = C_1 \parallel C_2$. Given $[\![C_1]\!]$ and $[\![C_2]\!]$, we can arbitrarily "pick" some commands C_1' and C_2' with $[\![C_1']\!] = [\![C_1]\!]$ and $[\![C_2']\!] = [\![C_2]\!]$, and "return" $\lfloor C_1' \parallel C_2' \rfloor^{\mathcal{R}}$. Since $[\![C_1']\!] = [\![C_1]\!]$ and $[\![C_2']\!] = [\![C_2]\!]$, the compositionality of $[\![\cdot]\!]$ ensures that $[\![C]\!] = [\![C_1 \parallel C_2]\!] = [\![C_1' \parallel C_2']\!] = \lfloor C_1' \parallel C_2' \rfloor^{\mathcal{R}}$.

Remark 4.18. Candidates for a direct compositional definition of $[\![C_1 \,;\, C_2]\!]$ and $[\![C_1 \parallel C_2]\!]$ are to take the $\mathcal{R}$-closure of the set obtained by taking all possible sequential/parallel compositions of traces from $[\![C_1]\!]$ and $[\![C_2]\!]$. This works for sequential composition, as we have $[\![C_1 \,;\, C_2]\!] = \{t_1 \,;\, t_2 \mid t_1 \in [\![C_1]\!], t_2 \in [\![C_2]\!]\}^{\mathcal{R}}$. However, for parallel composition, we only have $[\![C_1 \parallel C_2]\!] \subseteq \{t_1 \parallel t_2 \mid t_1 \in [\![C_1]\!], t_2 \in [\![C_2]\!]\}^{\mathcal{R}}$. To see that the converse does not hold, let:

$$C_1 = \mathsf{x} := 1 \,;\, \mathtt{assume}(\mathsf{y} = 0) \,;\, \mathtt{assume}(\mathsf{z} \neq 1) \,;\, \mathtt{assume}(\mathsf{z} = 1) \,;\, \mathsf{x} := 0$$

$$C_2 = \mathsf{y} := 1 \,;\, \mathtt{assume}(\mathsf{x} = 0) \,;\, \mathtt{assume}(\mathsf{z} \neq 1) \,;\, \mathtt{assume}(\mathsf{z} = 1) \,;\, \mathsf{y} := 0$$

Using $\overline{\textsc{coalesce}}$ on $\langle \mathsf{s}_0, \theta_0, \mathtt{W}(\mathsf{x}, 1) \cdot \bar{\mathtt{W}}(\mathsf{z}, 1) \cdot \mathtt{W}(\mathsf{x}, 0) \rangle \in \lfloor C_1 \rfloor$ and $\langle \mathsf{s}_0, \theta_0, \mathtt{W}(\mathsf{y}, 1) \cdot \bar{\mathtt{W}}(\mathsf{z}, 1) \cdot \mathtt{W}(\mathsf{y}, 0) \rangle \in \lfloor C_2 \rfloor$, we obtain that $t = \langle \mathsf{s}_0, \theta_0, \bar{\mathtt{W}}(\mathsf{z}, 1) \rangle \in \{t_1 \parallel t_2 \mid t_1 \in [\![C_1]\!], t_2 \in [\![C_2]\!]\}^{\mathcal{R}}$. But, $t \notin [\![C_1 \parallel C_2]\!]$. Indeed, no trace in $\lfloor C_1 \parallel C_2 \rfloor$ has a single environment write to z, and all rules of $\mathcal{R}$ preserve the environment actions.

Store-before-store elimination
$x := E\,;x := E' \rightsquigarrow x := E'$

Store-after-load eliminations
$\texttt{let}\ a = x\ \texttt{in}\ (x := a\,;C) \rightsquigarrow \texttt{let}\ a = x\ \texttt{in}\ C$
$\texttt{let}\ a = x\ \texttt{in}\ (\texttt{if}\ E\ \texttt{then}\ (x := a\,;C_1)\ \texttt{else}\ C_2) \rightsquigarrow \texttt{let}\ a = x\ \texttt{in}\ (\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2)$
$\texttt{let}\ a = x\ \texttt{in}\ (y := E\,;x := a) \rightsquigarrow y := E$ provided that $x \neq y$ and $a \notin \mathsf{fv}(E)$
$\texttt{let}\ a = x\ \texttt{in}\ (\texttt{let}\ b = y\ \texttt{in}\ x := a) \rightsquigarrow \texttt{skip}$ provided that $a \neq b$

Unused FAA-before-store elimination
$\texttt{let}\ a = \mathsf{FAA}(x, E)\ \texttt{in}\ x := v \rightsquigarrow x := v$

FAA-after-FAA elimination
$\texttt{let}\ a = \mathsf{FAA}(x, E)\ \texttt{in}\ \texttt{let}\ b = \mathsf{FAA}(x, E')\ \texttt{in}\ C \rightsquigarrow \texttt{let}\ a = \mathsf{FAA}(x, E + E')\ \texttt{in}\ \texttt{let}\ b = a + E\ \texttt{in}\ C$ provided that $a \notin \mathsf{fv}(E) \cup \mathsf{fv}(E')$

FAA-after-store elimination
$x := v\,;(\texttt{let}\ a = \mathsf{FAA}(x, E)\ \texttt{in}\ C) \rightsquigarrow \texttt{let}\ a = v\ \texttt{in}\ (x := v + [\![E]\!]\,;C)$ provided that $a \notin \mathsf{fv}(E)$

Redundant FAA elimination/introduction
$\texttt{let}\ a = x\ \texttt{in}\ C \leftrightsquigarrow \texttt{let}\ a = \mathsf{FAA}(x, 0)\ \texttt{in}\ C$

Fig. 5. Examples of program transformations validated by the abstract semantics

4.2 Adequacy

We show that adequacy of the abstract semantics is a corollary of its compositionality and of Lemma 3.5. For that, we first observe that the rewrite rules only manipulate chronicles leaving the initial state, initial store, and (derived) final state intact, and that only component traces are mapped to component traces by the rewrite rules in $\mathcal{R}$.

Proposition 4.19. *If* $\langle s, \theta, c\rangle \xrightarrow{\mathcal{R}} \langle s', \theta', c'\rangle$, *then* $s = s'$, $\theta = \theta'$, *and* $c(s) = c'(s')$.

Proposition 4.20. *If* $\langle s, \theta, c\rangle \xrightarrow{\mathcal{R}} \langle s', \theta', c'\rangle$ *and* $c' \in \mathsf{CmpChro}$, *then* $c \in \mathsf{CmpChro}$.

Theorem 4.21. *If* $[\![C_{\mathrm{tgt}}]\!] \subseteq [\![C_{\mathrm{src}}]\!]$, *then* $C_{\mathrm{src}} \rightsquigarrow C_{\mathrm{tgt}}$.

Proof. Suppose that $[\![C_{\mathrm{tgt}}]\!] \subseteq [\![C_{\mathrm{src}}]\!]$. Let P be a context such that $P[C_{\mathrm{src}}]$ and $P[C_{\mathrm{tgt}}]$ are closed, and suppose that $\langle P[C_{\mathrm{tgt}}], s\rangle \downarrow s'$. Since $\langle P[C_{\mathrm{tgt}}], s\rangle \downarrow s'$, by Lemma 3.5, we have $\langle s, \theta, c\rangle \in \lfloor P[C_{\mathrm{tgt}}]\rfloor$ for some store θ and component chronicle $c \in \mathsf{CmpChro}$ such that $c(s) = s'$. Since $\lfloor P[C_{\mathrm{tgt}}]\rfloor \subseteq [\![P[C_{\mathrm{tgt}}]]\!]$, we have $\langle s, \theta, c\rangle \in [\![P[C_{\mathrm{tgt}}]]\!]$. Since $[\![C_{\mathrm{tgt}}]\!] \subseteq [\![C_{\mathrm{src}}]\!]$, by Lemma 4.17, it follows that $\langle s, \theta, c\rangle \in [\![P[C_{\mathrm{src}}]]\!]$. Using Prop. 4.2, $t_0 \xrightarrow{\mathcal{R}}{}^{*} \langle s, \theta, c\rangle$ for some $t_0 \in \lfloor P[C_{\mathrm{src}}]\rfloor$. Then, by Propositions 4.19 and 4.20, $t_0 = \langle s, \theta, c'\rangle$ for some component chronicle $c' \in \mathsf{CmpChro}$ such that $c'(s) = c(s) = s'$. By Lemma 3.5, it follows that $\langle P[C_{\mathrm{src}}], s\rangle \downarrow s'$. □

By Prop. 4.3, we have $[\![C_{\mathrm{tgt}}]\!] \subseteq [\![C_{\mathrm{src}}]\!]$ iff $\lfloor C_{\mathrm{tgt}}\rfloor \subseteq [\![C_{\mathrm{src}}]\!]$. It follows that every program transformation that is validated by the concrete semantics is also validated by the abstract one:

Proposition 4.22. *If* $\lfloor C_{\mathrm{tgt}}\rfloor \subseteq \lfloor C_{\mathrm{src}}\rfloor$, *then* $[\![C_{\mathrm{tgt}}]\!] \subseteq [\![C_{\mathrm{src}}]\!]$.

Figure 5 presents examples of refinements that are validated by the abstract semantics but not by the concrete semantics. Again, among RMWs, we only list transformations involving FAA.

4.3 Full Abstraction

We establish full abstraction for the abstract semantics. The proof uses the notation $\bar{c}$ for the *dual* of a chronicle c, defined by $\bar{c} \triangleq \bar{\alpha_1} \cdots \bar{\alpha_n}$ for $c = \alpha_1 \cdots \alpha_n$.

Theorem 4.23. *If* $[\![C_{\mathrm{tgt}}]\!] \not\subseteq [\![C_{\mathrm{src}}]\!]$, *then* $C_{\mathrm{src}} \not\rightsquigarrow C_{\mathrm{tgt}}$.

Proof. Suppose that $[\![C_{\mathrm{tgt}}]\!] \not\subseteq [\![C_{\mathrm{src}}]\!]$. By Prop. 4.3, we have $\lfloor C_{\mathrm{tgt}} \rfloor \not\subseteq [\![C_{\mathrm{src}}]\!]$. Let $t_{\mathrm{tgt}} = \langle s_0, \theta, c_{\mathrm{tgt}} \rangle \in \lfloor C_{\mathrm{tgt}} \rfloor \setminus [\![C_{\mathrm{src}}]\!]$. Using t_{tgt}, we construct a context that demonstrates that $C_{\mathrm{src}} \not\rightsquigarrow C_{\mathrm{tgt}}$. Suppose first that t_{tgt} is non-empty, and let $\alpha_1, \ldots, \alpha_n \in \mathrm{Act}$ such that $c_{\mathrm{tgt}} = \alpha_1 \cdots \alpha_n$. For every $1 \leq i \leq n$, let:

$$s_i \triangleq (\alpha_1 \cdots \alpha_i)(s_0) \qquad C_i \triangleq \begin{cases} \texttt{let a = XCHG}(x, v) \texttt{ in assume(a = } s_{i-1}(x)\texttt{)} & \alpha_i = \bar{\mathtt{W}}(x, v) \\ \texttt{skip} & \text{otherwise} \end{cases}$$

Let $a_1, \ldots, a_k$ be an enumeration of $\mathsf{fv}(C_{\mathrm{src}}) \cup \mathsf{fv}(C_{\mathrm{tgt}})$, and define:

$$C_{\mathrm{ctx}} \triangleq C_1 \,;\, \texttt{snapshot}(s_1) \,;\, C_2 \,;\, \texttt{snapshot}(s_2) \,;\, \ldots \,;\, C_{n-1} \,;\, \texttt{snapshot}(s_{n-1}) \,;\, C_n$$
$$P \triangleq \texttt{let } a_1 = \theta(a_1) \texttt{ in } (\texttt{let } a_2 = \theta(a_2) \texttt{ in } (\ldots (\texttt{let } a_k = \theta(a_k) \texttt{ in } (C_{\mathrm{ctx}} \parallel -)) \ldots))$$

Intuitively speaking, the snapshots used in the context ensure that every execution of C_{ctx} visits the states $s_1, \ldots, s_{n-1}$ in this order.

Clearly, $P[C_{\mathrm{src}}]$ and $P[C_{\mathrm{tgt}}]$ are closed. We claim that (i) $\langle P[C_{\mathrm{tgt}}], s_0 \rangle \downarrow s_n$, but (ii) $\langle P[C_{\mathrm{src}}], s_0 \rangle \not\downarrow s_n$. For (i), observe that $\langle s_0, \theta, c_{\mathrm{tgt}}^{-} \parallel c_{\mathrm{tgt}} \rangle \in \lfloor P[C_{\mathrm{tgt}}] \rfloor$ for any store θ. Then, since $c_{\mathrm{tgt}}^{-} \parallel c_{\mathrm{tgt}}$ is a component chronicle and $c_{\mathrm{tgt}}(s_0) = s_n$, $\langle P[C_{\mathrm{tgt}}], s_0 \rangle \downarrow s_n$ follows by Lemma 3.5.

To prove (ii), it suffices to prove the following claim:

Claim 4.23.1: $\langle s_0, \theta, \bar{c} \rangle \xrightarrow{\mathcal{R}}{}^{*} \langle s_0, \theta, c_{\mathrm{tgt}} \rangle$ for every $c \in \mathsf{Chro}$ such that $c(s_0) = s_n$ and $\langle s_0, \theta, c \rangle \in \lfloor C_{\mathrm{ctx}} \rfloor$.

Indeed, from this claim we obtain that $\langle s_0, \theta, \bar{c} \rangle \notin \lfloor C_{\mathrm{src}} \rfloor$ for every chronicle c such that $c(s_0) = s_n$ and $\langle s_0, \theta, c \rangle \in \lfloor C_{\mathrm{ctx}} \rfloor$, which implies that $\langle s_0, \theta, c \rangle \notin \lfloor P[C_{\mathrm{src}}] \rfloor$ for every component chronicle c satisfying $c(s_0) = s_n$. Then, (ii) follows by Lemma 3.5.

Next, we prove Claim 4.23.1. Let c be a chronicle such that $c(s_0) = s_n$ and $\langle s_0, \theta, c \rangle \in \lfloor C_{\mathrm{ctx}} \rfloor$. Due to the use of snapshots in C_{ctx}, since $\langle s_0, \theta, c \rangle \in \lfloor C_{\mathrm{ctx}} \rfloor$, we have that $c = c_1 \cdots c_n$ for some chronicles $c_1, \ldots, c_n$ such that $c_i(s_{i-1}) = s_i$ and $\langle s_{i-1}, \theta, c_i \rangle \in \lfloor C_i \rfloor$ for every $1 \leq i \leq n$. We show that for every $1 \leq i \leq n$, we have $\langle s_{i-1}, \theta, \bar{c_i} \rangle \xrightarrow{\mathcal{R}}{}^{*} \langle s_{i-1}, \theta, \alpha_i \rangle$. By repeatedly applying this rewrite, using Prop. 4.12, the desired $\langle s_0, \theta, \bar{c} \rangle \xrightarrow{\mathcal{R}}{}^{*} \langle s_0, \theta, c_{\mathrm{tgt}} \rangle$ follows.

Let $1 \leq i \leq n$, and consider the possible cases:

- $\alpha_i = \bar{\mathtt{W}}(x, v)$ is an environment write: In this case, $\langle s_{i-1}, \theta, c_i \rangle \in \lfloor C_i \rfloor$ implies (by rules let,assume) that there exists environment chronicles e', e such that $c_i = e' \cdot \mathtt{W}(x, v) \cdot e$ and $e'(s_{i-1})(x) = s_{i-1}(x)$. Then, since we also have $c_i(s_{i-1}) = s_i$, using $\overline{\textsc{coalesce}}$, we can rewrite $\langle s_{i-1}, \theta, \bar{c_i} \rangle$ into $\langle s_{i-1}, \theta, \alpha_i \rangle$.
- $\alpha_i = \mathtt{W}(x, v)$ is an component write: In this case, c_i is an environment chronicle, and either $c_i = e' \cdot \bar{\mathtt{W}}(x, v) \cdot e$ for some environment chronicles e', e or $\bar{\alpha_i}$ is not inside c_i. In the first case, since $c_i(s_{i-1}) = s_i$, using coalesce, we can rewrite $\langle s_{i-1}, \theta, \bar{c_i} \rangle$ into $\langle s_{i-1}, \theta, \alpha_i \rangle$. In the second case, $c_i(s_{i-1}) = s_i = \alpha_i(s_{i-1})$ implies that $s_{i-1} = s_i$. Using add-red, we can rewrite $\langle s_{i-1}, \theta, \bar{c_i} \rangle$ into $\langle s_{i-1}, \theta, \alpha_i \cdot \bar{c_i} \rangle$. Then, using the del-red$'$ rewrite rule (a combination of coalesce and add-red and del-red, see Remark 4.11), we rewrite $\langle s_{i-1}, \theta, \alpha_i \cdot \bar{c_i} \rangle$ into $\langle s_{i-1}, \theta, \alpha_i \rangle$.

Finally, consider the case that $c_{\mathrm{tgt}} = \varepsilon$. In this case we define $C_{\mathrm{ctx}} \triangleq \texttt{skip}$ and also define P as above (using C_{ctx}). We have $\langle s_0, \theta, \varepsilon \rangle \in \lfloor P[C_{\mathrm{tgt}}] \rfloor$ for any store θ, and by Lemma 3.5 we obtain that $\langle P[C_{\mathrm{tgt}}], s_0 \rangle \downarrow s_0$. In turn, as above, $\langle P[C_{\mathrm{src}}], s_0 \rangle \not\downarrow s_0$ follows from the fact that $\langle s_0, \theta, \bar{c} \rangle \xrightarrow{\mathcal{R}}{}^{*} \langle s_0, \theta, \varepsilon \rangle$ for every chronicle c such that $c(s_0) = s_0$ and $\langle s_0, \theta, c \rangle \in \lfloor C_{\mathrm{ctx}} \rfloor$. To prove this fact, let c such that $c(s_0) = s_0$ and $\langle s_0, \theta, c \rangle \in \lfloor C_{\mathrm{ctx}} \rfloor$. Then, $\langle s_0, \theta, c \rangle \in \lfloor C_{\mathrm{ctx}} \rfloor$ implies that c is an environment chronicle. Using del-red$'$, we rewrite $\langle s_0, \theta, \bar{c} \rangle$ into $\langle s_0, \theta, \varepsilon \rangle$. □

Given Thm. 4.23, we can use the abstract semantics to easily *invalidate* certain transformations. Next, we present two such examples.

Example 4.24. Store-before-store elimination is invalid with an intervening load. For instance, for $C_{\mathrm{src}} = \texttt{x := 1 ; let a = y in (x := 2 ; z := a)}$ and $C_{\mathrm{tgt}} = \texttt{let a = y in (x := 2 ; z := a)}$, we have $C_{\mathrm{src}} \not\rightsquigarrow C_{\mathrm{tgt}}$. Since $\langle \mathsf{s}_0, \theta_0, \bar{\mathtt{W}}(\mathsf{y}, 1) \cdot \mathtt{W}(\mathsf{x}, 2) \cdot \mathtt{W}(\mathsf{z}, 0)\rangle \in [\![C_{\mathrm{tgt}}]\!] \setminus [\![C_{\mathrm{src}}]\!]$, this follows from Thm. 4.23.

Example 4.25. A repeated store cannot be eliminated when there is an intervening store. For instance, for $C_{\mathrm{src}} = \texttt{x := 1 ; y := 1 ; x := 1}$ and $C_{\mathrm{tgt}} = \texttt{x := 1 ; y := 1}$, we have $C_{\mathrm{src}} \not\rightsquigarrow C_{\mathrm{tgt}}$. Since $\langle \mathsf{s}_0, \theta_0, \mathtt{W}(\mathsf{x}, 1) \cdot \bar{\mathtt{W}}(\mathsf{x}, 2) \cdot \mathtt{W}(\mathsf{y}, 1)\rangle \in t \in [\![C_{\mathrm{tgt}}]\!] \setminus [\![C_{\mathrm{src}}]\!]$, this follows from Thm. 4.23.

From the full abstraction proof, we observe that although multiple rewrites of a trace may be necessary, these rewrites do not overlap. We only apply them to disjoint parts of the chronicle. Formally, we let $\mathcal{R}^{\mathrm{loc}}$ be the set consisting of "local" variants of the rules:

$\textsc{coalesce}^{\mathrm{loc}}$: $\langle s, \theta, m_1 \cdot \mathtt{W}(x, v) \cdot m_2\rangle \xrightarrow{\textsc{coalesce}^{\mathrm{loc}}} \langle s, \theta, \mathtt{W}(x, v)\rangle$ provided that $m_1, m_2 \in \mathsf{CmpChro}$ and $(m_1 \cdot \mathtt{W}(x, v) \cdot m_2)(s) = s[x \mapsto v]$.

$\overline{\textsc{coalesce}}^{\mathrm{loc}}$: $\langle s, \theta, m_1 \cdot \bar{\mathtt{W}}(x, v) \cdot m_2\rangle \xrightarrow{\overline{\textsc{coalesce}}^{\mathrm{loc}}} \langle s, \theta, \bar{\mathtt{W}}(x, v)\rangle$ provided that $m_1, m_2 \in \mathsf{CmpChro}$, $(m_1 \cdot \bar{\mathtt{W}}(x, v) \cdot m_2)(s) = s[x \mapsto v]$, and $m_1(s)(x) = s(x)$.

$\textsc{del-red}^{\mathrm{loc}}$: $\langle s, \theta, m\rangle \xrightarrow{\textsc{del-red}^{\mathrm{loc}}} \langle s, \theta, \varepsilon\rangle$ provided that $m \in \mathsf{CmpChro}$ and $m(s) = s$.

$\textsc{add-red}^{\mathrm{loc}}$: $\langle s, \theta, \varepsilon\rangle \xrightarrow{\textsc{add-red}^{\mathrm{loc}}} \langle s, \theta, \mathtt{W}(x, v)\rangle$ provided that $s(x) = v$.

The relation $\rightrightarrows$ between traces is inductively defined as follows:

$$\frac{}{\langle s, \theta, \varepsilon\rangle \rightrightarrows \langle s, \theta, \varepsilon\rangle} \qquad \frac{t_1 \xrightarrow{\mathcal{R}^{\mathrm{loc}}}{}^{?} t_1' \qquad t_2 \rightrightarrows t_2'}{t_1 \,;\, t_2 \rightrightarrows t_1' \,;\, t_2'}$$

Then, the full abstraction proof shows that $C_{\mathrm{src}} \not\rightsquigarrow C_{\mathrm{tgt}}$ whenever $\lfloor C_{\mathrm{tgt}} \rfloor \not\subseteq \{t' \mid \exists t \in \lfloor C_{\mathrm{src}} \rfloor .\, t \rightrightarrows t'\}$. In fact, by analyzing the rewrite rules we prove the following:

Lemma 4.26. *For every set T of traces, we have $T^{\mathcal{R}} = \{t' \mid \exists t \in T .\, t \rightrightarrows t'\}$.*

4.4 Full Abstraction Without Snapshots

The full abstraction proof above relies on the availability of the `snapshot` command, which gives the parallel context the ability to simultaneously observe the values of all variables. Next, we show that snapshots can be avoided in that proof provided that C_{src} is loop-free. Roughly speaking, we show in this case it is possible to achieve the effect of a snapshot executing in parallel to C_{src} by repeatedly reading shared variables a number of times that can be determined from C_{src}. This means that when C_{src} is loop-free snapshots do no not increase the distinguishing power of the parallel context. In turn, we present a delicate example of a command C_{src} with loops, where a certain refinement holds for snapshot-free contexts but fails to hold for contexts with snapshot. In particular, this implies that in the language without snapshot, full abstraction of the abstract semantics does not hold for code fragments with loops.

Formally, we say that a transformation from a command C_{src} to a command C_{tgt} is *sound for no-snapshot context*, denoted by $C_{\mathrm{src}} \rightsquigarrow_{\not\mathtt{snapshot}} C_{\mathrm{tgt}}$, if $\langle P[C_{\mathrm{tgt}}], s\rangle \downarrow s'$ implies $\langle P[C_{\mathrm{src}}], s\rangle \downarrow s'$ for every snapshot-free context P such that $P[C_{\mathrm{src}}]$ and $P[C_{\mathrm{tgt}}]$ are closed.

Theorem 4.27. *If C_{src} is loop-free and $[\![C_{\mathrm{tgt}}]\!] \not\subseteq [\![C_{\mathrm{src}}]\!]$, then $C_{\mathrm{src}} \not\rightsquigarrow_{\not\mathtt{snapshot}} C_{\mathrm{tgt}}$.*

Proof Sketch. In the proof Thm. 4.23, snapshot is needed in order to ensure that a certain state is reached when C_{src} is executed concurrently. When C_{src} is loop-free, we can achieve this result by repeatedly reading the shared variables used in C_{src}, and checking their values one-by-one. More precisely, given a state s, let $C_s \triangleq \mathtt{assume}(x_1 = s(x_1)) \,;\, \ldots \,;\, \mathtt{assume}(x_n = s(x_n))$ where $x_1, \ldots, x_n$ is an

enumeration of all shared variables occurring in C_{src}. When C_{src} is loop-free, there exists a bound $N \in \mathbb{N}$ on the number of writes performed by C_{src} (i.e., the number of component actions in $\lfloor C_{\text{src}} \rfloor$). We use a sequential composition $C_s \,;\, \ldots \,;\, C_s$ consisting of $N+1$ copies of C_s instead of $\texttt{snapshot}(s)$. If after every execution of C_s we reach a state different than s, then for the next execution of C_s to terminate, we need at least one write by the concurrent context. Since the C_{src} is performing at most N writes, executing C_s $N+1$ times in a row ensures that at some point we visit s. □

The above implication fails if C_{src} has loops. The simplest example we found is presented next.

Example 4.28. For the commands $C_{\text{src}} = \texttt{while} * \texttt{do}\ (\texttt{y} := 0 \,;\, \texttt{x} := * \,;\, \texttt{x} := 0 \,;\, \texttt{y} := *)$ and $C_{\text{tgt}} = \texttt{y} := 0 \,;\, \texttt{x} := 1 \,;\, \texttt{y} := 1 \,;\, \texttt{x} := 0$, we have $C_{\text{src}} \not\rightsquigarrow C_{\text{tgt}}$ but $C_{\text{src}} \rightsquigarrow_{\not\text{snapshot}} C_{\text{tgt}}$. The former follows from Thm. 4.23 since we have $\langle \mathsf{s}_0, \theta_0, \mathtt{W}(\mathsf{y},0) \cdot \mathtt{W}(\mathsf{x},1) \cdot \mathtt{W}(\mathsf{y},1) \cdot \mathtt{W}(\mathsf{x},0) \rangle \in \lfloor C_{\text{tgt}} \rfloor \setminus [\![C_{\text{src}}]\!]$.

To see that $C_{\text{src}} \rightsquigarrow_{\not\text{snapshot}} C_{\text{tgt}}$, we have to resort to cumbersome operational reasoning, and provide a simulation relation that relates operational executions of $P[C_{\text{tgt}}]$ to those of $P[C_{\text{src}}]$. Roughly speaking, the main idea is to execute $\texttt{y} := 0$ and $\texttt{x} := *$ (with 1 for $*$) in the source when the target executes $\texttt{y} := 0$ and $\texttt{x} := 1$, respectively. Then, when the target executes $\texttt{y} := 1$, the source executes $\texttt{x} := 0 \,;\, \texttt{y} := *$ (with 1 for $*$). This creates a mismatch between the target's state that has $\texttt{x} = 1$ and the source's state that has $\texttt{x} = 0$. Nevertheless, whenever the concurrent context relies on the value of x, the source can do another half-iteration and execute $\texttt{y} := 0 \,;\, \texttt{x} := *$ to fix the value of x as it is in the target's state, moving the mismatch between the target and the source to y. This way, we are able to use the source's non-deterministic loop, to provide the concurrent context with whatever value it needs for x and y, one at a time. Finally, when the target executes $\texttt{x} := 0$ the source executes $\texttt{x} := 0 \,;\, \texttt{y} := *$ (with the final value of y in the target for $*$).

Making this intuition formal is rather challenging (which provides us with more confidence that the denotational semantics is beneficial for formal refinement proofs). In our Coq development, we do that by generalizing the notion of a command context, demonstrating how generalized contexts interact with the operational semantics, and using generalized contexts for defining the simulation.

Example 4.28 uses non-deterministic looping, $\texttt{while} * \texttt{do}\ C$, but, by using the following proposition, it is possible to devise a similar example without non-deterministic looping:

Proposition 4.29. *The following transformations are sound:*

- $x := * \,;\, \texttt{while}\ x\ \texttt{do}\ (x := * \,;\, C \,;\, x := *) \,;\, x := * \rightsquigarrow \texttt{while} * \texttt{do}\ C$.
- *For* $C = \texttt{while}\ y\ \texttt{do}\ (x := 0 \oplus \texttt{let}\ a = \mathtt{FAA}(x,1)\ \texttt{in skip})$, *we have* $\texttt{let}\ a = y\ \texttt{in}\ (y := 1 \,;\, (C \parallel y := 0) \,;\, y := a) \rightsquigarrow x := *$ *provided that* $x \neq y$.

Using Prop. 4.29, we can adapt Example 4.28 to use a command C'_{src} that does not use $\texttt{while} * \texttt{do}\ C$ and $x := *$ instead of C_{src} and have $C'_{\text{src}} \rightsquigarrow_{\not\text{snapshot}} C_{\text{tgt}}$. To see that $C'_{\text{src}} \not\rightsquigarrow C_{\text{tgt}}$, note that a concurrent snapshot observing $\texttt{x} = \texttt{y} = 1$ is possible for C_{tgt} but not for C'_{src}. Thus, snapshots strictly increase the distinguishing power of contexts also in a language without non-deterministic loops.

5 SEMANTICS FOR RMW-FREE CONTEXTS

In this section we show that RMWs strictly increase the power of contexts to distinguish between code fragments, and show how to modify the abstract semantics for the case of RMW-free contexts.

Formally, we say that a transformation from a command C_{src} to a command C_{tgt} is *sound for no-RMW context*, denoted by $C_{\text{src}} \rightsquigarrow_{\not\text{rmw}} C_{\text{tgt}}$, if for every RMW-free context P such that $P[C_{\text{src}}]$ and $P[C_{\text{tgt}}]$ are closed, we have that $\langle P[C_{\text{tgt}}], s \rangle \downarrow s'$ implies $\langle P[C_{\text{src}}], s \rangle \downarrow s'$.

The next example demonstrates a case where $C_{\text{src}} \rightsquigarrow_{\not\text{rmw}} C_{\text{tgt}}$ but $C_{\text{src}} \not\rightsquigarrow C_{\text{tgt}}$. (Example 4.7 provides another case in point.)

Example 5.1. Let: $C_{\text{src}} = \mathtt{x} := 1 \,;\, \mathtt{x} := 5 \,;\, \mathtt{if}\ \mathtt{x} = 2\ \mathtt{then}\ \mathtt{x} := 3\ \mathtt{else}\ \mathtt{x} := 4$
$C_{\text{tgt}} = \mathtt{x} := 1 \,;\, \mathtt{if}\ \mathtt{x} = 2\ \mathtt{then}\ \mathtt{x} := 3\ \mathtt{else}\ \mathtt{x} := 4$
An intuitive argument for $C_{\text{src}} \rightsquigarrow C_{\text{tgt}}$ could claim that when the "then" branch is taken, there must be a moment when the parallel context stores 2 in x and we can execute $\mathtt{x} := 5$ "just before" that moment; and when the "else" branch is taken we can execute $\mathtt{x} := 5;\mathtt{if}\ \mathtt{x} = 2\ \mathtt{then}\ \mathtt{x} := 3\ \mathtt{else}\ \mathtt{x} := 4$ as one atomic block at the time the target executes $\mathtt{x} := 4$. This argument, however, ignores the option that the context may not be able to store 2 in x if the value of x was modified to 5, which is possible when the context stores 2 in x using an RMW. Indeed, for $P = - \parallel \mathtt{let\ b = FAA(x, 1)\ in\ skip}$ we have $\langle P[C], \mathsf{s}_0\rangle \downarrow \mathsf{s}_0[\mathsf{x} \mapsto 3]$ for $C = C_{\text{tgt}}$ but not for $C = C_{\text{src}}$. Alternatively, using Thm. 4.23, $C_{\text{src}} \not\rightsquigarrow C_{\text{tgt}}$ follows from the fact that for $t = \langle \mathsf{s}_0, \theta_0, \mathtt{W}(\mathsf{x}, 1) \cdot \bar{\mathtt{W}}(\mathsf{x}, 2) \cdot \mathtt{W}(\mathsf{x}, 3)\rangle$, we have $t \in [\![C_{\text{tgt}}]\!] \setminus [\![C_{\text{src}}]\!]$. Using the semantics below, we will formally show that $C_{\text{src}} \rightsquigarrow_{\not\text{rmw}} C_{\text{tgt}}$.

From the discussion above, we observe that when the context is RMW-free, we would like to allow to attach component actions to environment actions even when the component actions write to the same variable that the environment modifies. This would formally justify the intuitive argument about the "then" branch, allowing us to rewrite $\langle \mathsf{s}_0, \theta_0, \mathtt{W}(\mathsf{x}, 1) \cdot \mathtt{W}(\mathsf{x}, 5) \cdot \bar{\mathtt{W}}(\mathsf{x}, 2) \cdot \mathtt{W}(\mathsf{x}, 3)\rangle$ into the target trace $\langle \mathsf{s}_0, \theta_0, \mathtt{W}(\mathsf{x}, 1) \cdot \bar{\mathtt{W}}(\mathsf{x}, 2) \cdot \mathtt{W}(\mathsf{x}, 3)\rangle$. We do so by omitting the second side condition of $\overline{\text{COALESCE}}$, using its following strengthening (in the sense that it allows more rewrites):

$\overline{\text{COALESCE}}^{\not\text{rmw}}$: $\langle s, \theta, c_1 \cdot m_1 \cdot \bar{\mathtt{W}}(x, v) \cdot m_2 \cdot c_2\rangle \xrightarrow{\overline{\text{COALESCE}}^{\not\text{rmw}}} \langle s, \theta, c_1 \cdot \bar{\mathtt{W}}(x, v) \cdot c_2\rangle$ provided that $m_1, m_2 \in \text{CmpChro}$ and $(m_1 \cdot \bar{\mathtt{W}}(x, v) \cdot m_2)(c_1(s)) = c_1(s)[x \mapsto v]$.

We let $\mathcal{R}_{\not\text{rmw}} \triangleq \{\xrightarrow{\text{COALESCE}}, \xrightarrow{\overline{\text{COALESCE}}^{\not\text{rmw}}}, \xrightarrow{\text{DEL-RED}}, \xrightarrow{\text{ADD-RED}}\}$ and $[\![C]\!]_{\not\text{rmw}} \triangleq \lfloor C \rfloor^{\mathcal{R}_{\not\text{rmw}}}$. Next, we prove the following compositionality property, analogous to 4.17:

Lemma 5.2. *If* $[\![C_1]\!]_{\not\text{rmw}} \subseteq [\![C_2]\!]_{\not\text{rmw}}$, *then* $[\![P[C_1]]\!] \subseteq [\![P[C_2]]\!]$ *for every RMW-free context* P.

Proof. The proof proceeds by induction on P. For sequential composition, we use the following analogue of Prop. 4.13:
Claim 5.2.1: If $\lfloor C_1 \rfloor \subseteq [\![C_1']\!]_{\not\text{rmw}}$, then $\lfloor C_1 \,;\, C_2 \rfloor \subseteq [\![C_1' \,;\, C_2]\!]_{\not\text{rmw}}$. Similarly, if $\lfloor C_2 \rfloor \subseteq [\![C_2']\!]_{\not\text{rmw}}$, then $\lfloor C_1 \,;\, C_2 \rfloor \subseteq [\![C_1 \,;\, C_2']\!]_{\not\text{rmw}}$.
For parallel composition, we define the following rule that is dual to $\overline{\text{COALESCE}}^{\not\text{rmw}}$:

DISPERSE$^{\not\text{rmw}}$: $\langle s, \theta, c_1 \cdot \mathtt{W}(x, v) \cdot c_2\rangle \xrightarrow{\text{DISPERSE}^{\not\text{rmw}}} \langle s, \theta, c_1 \cdot e_1 \cdot \mathtt{W}(x, v) \cdot e_2 \cdot c_2\rangle$ provided that $e_1, e_2 \in \text{EnvChro}$ and $(e_1 \cdot \mathtt{W}(x, v) \cdot e_2)(c_1(s)) = c_1(s)[x \mapsto v]$.

We define $\mathcal{D}_{\not\text{rmw}} \triangleq \{\xrightarrow{\overline{\text{DISPERSE}}}, \xrightarrow{\text{DISPERSE}^{\not\text{rmw}}}, \xrightarrow{\text{ADD-}\overline{\text{RED}}}, \xrightarrow{\text{DEL-}\overline{\text{RED}}}\}$, and show the following analogue of Prop. 4.14:
Claim 5.2.2: For every RMW-free command C, $\lfloor C \rfloor$ is closed under $\mathcal{D}^{\not\text{rmw}}$.

Then, we can prove the following variant of Prop. 4.16, which establishes the required property for parallel composition:
Claim 5.2.3: If $\lfloor C_1 \rfloor \subseteq [\![C_1']\!]_{\not\text{rmw}}$ and C_2 is RMW-free, then $\lfloor C_1 \parallel C_2 \rfloor \subseteq [\![C_1' \parallel C_2]\!]_{\not\text{rmw}}$. Similarly, if $\lfloor C_2 \rfloor \subseteq [\![C_2']\!]_{\not\text{rmw}}$ and C_1 is RMW-free, then $\lfloor C_1 \parallel C_2 \rfloor \subseteq [\![C_1 \parallel C_2']\!]_{\not\text{rmw}}$.
Other language constructs are handled similarly. □

Given Lemma 5.2, adequacy of $[\![\cdot]\!]_{\not\text{rmw}}$ is shown similarly to the proof of Thm. 4.21.

Theorem 5.3. *If* $[\![C_{\text{tgt}}]\!]_{\not\text{rmw}} \subseteq [\![C_{\text{src}}]\!]_{\not\text{rmw}}$, *then* $C_{\text{src}} \rightsquigarrow_{\not\text{rmw}} C_{\text{tgt}}$.

With Thm. 5.3, we can revisit Example 5.1 and derive $C_{\text{src}} \rightsquigarrow_{\not\text{rmw}} C_{\text{tgt}}$ from $\lfloor C_{\text{tgt}} \rfloor \subseteq [\![C_{\text{src}}]\!]_{\not\text{rmw}}$. Indeed, traces in $\lfloor C_{\text{tgt}} \rfloor$ are either of the form $\langle s, \theta, e_1 \cdot \mathtt{W}(\mathsf{x}, 1) \cdot e_2 \cdot \bar{\mathtt{W}}(\mathsf{x}, 2) \cdot e_3 \cdot \mathtt{W}(\mathsf{x}, 3) \cdot e_4\rangle$ or of

the form $\langle s, \theta, e_1 \cdot \mathtt{W}(\mathtt{x},1) \cdot e_2 \cdot \mathtt{W}(\mathtt{x},4) \cdot e_3 \rangle$. For t of the first form, we can start from a corresponding trace in $\lfloor C_{\mathrm{src}} \rfloor$ of the form $\langle s, \theta, e_1 \cdot \mathtt{W}(\mathtt{x},1) \cdot e_2 \cdot \mathtt{W}(\mathtt{x},5) \cdot \bar{\mathtt{W}}(\mathtt{x},2) \cdot e_3 \cdot \mathtt{W}(\mathtt{x},3) \cdot e_4 \rangle$ and rewrite by $\overline{\textsc{coalesce}}^{\not\mathrm{rmw}}$ with $c_1 = e_1 \cdot \mathtt{W}(\mathtt{x},1) \cdot e_2$, $m_1 = \mathtt{W}(\mathtt{x},5)$, $m_2 = \varepsilon$, and $c_2 = e_3 \cdot \mathtt{W}(\mathtt{x},3) \cdot e_4$ to obtain t. For t of the second form, we can start from a corresponding trace in $\lfloor C_{\mathrm{src}} \rfloor$ of the form $\langle s, \theta, e_1 \cdot \mathtt{W}(\mathtt{x},1) \cdot e_2 \cdot \mathtt{W}(\mathtt{x},5) \cdot \mathtt{W}(\mathtt{x},4) \cdot e_3 \rangle$ and rewrite by $\textsc{coalesce}$ with $c_1 = e_1 \cdot \mathtt{W}(\mathtt{x},1) \cdot e_2$, $m_1 = \mathtt{W}(\mathtt{x},5)$, $m_2 = \varepsilon$, and $c_2 = e_3$ to obtain t.

Finally, we establish a full abstraction property for the ~~rmw~~ semantics:

Theorem 5.4. *If* $[\![C_{\mathrm{tgt}}]\!]_{\not\mathrm{rmw}} \not\subseteq [\![C_{\mathrm{src}}]\!]_{\not\mathrm{rmw}}$, *then* $C_{\mathrm{src}} \not\rightsquigarrow_{\not\mathrm{rmw}} C_{\mathrm{tgt}}$.

Proof. The proof is similar to the proof of Thm. 4.23. Instead of RMW followed by assume, to construct the appropriate context we let $C_i \triangleq x := v$ for the case that $\alpha_i = \bar{\mathtt{W}}(x,v)$. Then we have to show that $\langle s_{i-1}, \theta, \bar{c}_i \rangle \xrightarrow{\mathcal{R}}{}^{*} \langle s_{i-1}, \theta, \alpha_i \rangle$ for this case. By the rule $\textsc{store}$, $\langle s_{i-1}, \theta, c_i \rangle \in \lfloor C_i \rfloor$ implies that there exist environment chronicles e', e such that $c_i = e' \cdot \mathtt{W}(x,v) \cdot e$. (Unlike the corresponding case in the proof of Thm. 4.23, we do not necessarily have $e'(s_{i-1})(x) = s_{i-1}(x)$.) Then, since we also have $c_i(s_{i-1}) = s_i$ (due to the use of snapshots), using $\overline{\textsc{coalesce}}^{\not\mathrm{rmw}}$, we can rewrite $\langle s_{i-1}, \theta, \bar{c}_i \rangle$ into $\langle s_{i-1}, \theta, \alpha_i \rangle$. □

A version that uses repeated reads instead of snapshots is proved by combining the proofs of Thm. 4.27 and Thm. 5.4. We write $C_{\mathrm{src}} \rightsquigarrow_{\not\mathrm{rmw},\not\mathrm{snapshot}} C_{\mathrm{tgt}}$, if $\langle P[C_{\mathrm{tgt}}], s \rangle \downarrow s'$ implies $\langle P[C_{\mathrm{src}}], s \rangle \downarrow s'$ for every rmw-free and snapshot-free context P such that $P[C_{\mathrm{src}}]$ and $P[C_{\mathrm{tgt}}]$ are closed.

Theorem 5.5. *If* C_{src} *is loop-free and* $[\![C_{\mathrm{tgt}}]\!]_{\not\mathrm{rmw}} \not\subseteq [\![C_{\mathrm{src}}]\!]_{\not\mathrm{rmw}}$, *then* $C_{\mathrm{src}} \not\rightsquigarrow_{\not\mathrm{rmw},\not\mathrm{snapshot}} C_{\mathrm{tgt}}$.

6 RELATED AND FUTURE WORK

We have already discussed the seminal work of Brookes [1996], from which we took a lot of inspiration. Our traces consist of write actions, rather than transitions (pairs of states) as in Brookes's traces, and are closer in spirit to models of Milner's CCS [Milner 1980] and Hoare's CSP [Hoare 1985]. This choice has several advantages. First, it directly reflects the property of the operational semantics that each transition updates at most one variable. Second, since reads are not recorded in traces, our concrete semantics, i.e., before imposing any closures, already validates a variety of refinements, including all those that do not involve writes. In contrast, in Brookes's traces reads are tracked as stuttering transitions, and closures are needed also for refinements of reads (and of `skip`). Third, explicit environment writes in traces allows us to have a rule like $\overline{\textsc{coalesce}}$ that mimics operational simulation that attaches component actions to one environment write.

Brookes's traces, which are very similar to the traces used for giving meaning to rely/guarantee judgements [Jones 1983; Xu et al. 1997], have provided a useful intuition and formal basis for multiple later frameworks, e.g., [Dingel 1999, 2002; Liang et al. 2012, 2014; Turon and Wand 2011], which propose relational program logics for reasoning about refinements. For example, [Dingel 1999, 2002] used Brookes's semantics for deriving a refinement calculus allowing one to develop full concurrent programs by repeatedly refining a specification.

Some works address the challenge of validating contextual refinements that are conditioned by some assumptions on the concurrent context. Our results on snapshot/RMW-free contexts go in this direction, but there is, of course, a variety of more fine-grained assumptions that will allow deriving useful refinements. For example, we would like to be able to reason about common concurrency primitives, such as locks and transactions. These can be implemented from standard shared memory constructs, but when studying full abstraction for them, one should only consider disciplined contexts, that, e.g., properly interleave lock and unlock commands. Some works, which provide sound techniques but do not consider full abstraction, addressed similar challenges. For example, [Liang et al. 2012, 2014] developed a framework for establishing contextual refinement that

handles assumptions such as data-race-freedom and data encapsulation in concurrent objects, and demonstrate that their technique is sufficiently expressive for verifying a complex garbage collector. More recently, Frumin et al. [2021]; Song et al. [2023] studied refinements conditioned by separation logic premises, and Khyzha and Lahav [2022]; Singh and Lahav [2023] studied refinements that assume that clients adhere to a given library call policy. We hope that our denotational semantics will form a basis for continuations along these lines.

Another line of work, see e.g., [Benton et al. 2016], attempts to capture shared-memory concurrency in general, and Brookes's semantics in particular, using monadic constructions following [Moggi 1991], or even as an algebraic theory [Abadi and Plotkin 2010; Dvir et al. 2022]. A prominent advantage of these approaches is their ability to capture higher-order programs, while we are limited to first-order programs. Additionally, this approach detaches structural refinements from effectful ones and paves the way to type-and-effect systems, enabling reasoning about refinements using assumptions from a type analysis (see e.g., [Birkedal et al. 2012; Kammar 2014]).

Our work handles shared variables admitting sequentially consistent semantics (SC). Jagadeesan et al. [2012] modified Brookes's semantics to apply for x86-TSO memory (see [Owens et al. 2009]), and achieved full abstraction using `await` instructions. Dvir et al. [2024] developed Brookes's semantics for the Release/Acquire memory model (see [Lahav et al. 2016]), but did not study full abstraction. A large body of work, e.g., [Jagadeesan et al. 2020; Jeffrey and Riely 2019; Jeffrey et al. 2022; Kavanagh and Brookes 2018, 2019; Paviotti et al. 2020], has been devoted to the study of compositional semantics for weakly consistent memory that is not necessarily accompanying an existing operational semantics like in our case. A prominent idea there is the use of partially ordered multisets ("pomsets") [Pratt 1986] or event structures [Winskel 1987] that generalize linearly ordered traces, like those we work with. This aligns with axiomatic approaches (see, e.g., [Alglave et al. 2014]), which, as is, like operational semantics, are restricted to apply on closed full programs.

In the realm of weak memory models, reasoning about correctness of local compiler optimizations is rather challenging and error-prone. Many works have addressed this issue in different levels of formality, e.g., [Burckhardt et al. 2010; Chakraborty and Vafeiadis 2016; Cho et al. 2022; Dodds et al. 2018; Morisset et al. 2013; Poetzl and Kroening 2016]. Interestingly, it is not always the case that a weaker memory model allows more optimizations than a stronger one (see, e.g., [Gopalakrishnan et al. 2023]). For instance, weak memory model usually do not support "store-after-load elimination" and "redundant FAA elimination" that are valid under SC (see Fig. 5). Attempting to allow local proofs of optimizations, some of these works develop compositional semantics, but these are restricted to top-level parallel composition. An noteworthy exception is the work of Dodds et al. [2018] who developed a denotational semantics for the Release/Acquire weak memory model. Their semantics is based on an axiomatic formulation, which they generalize to allow "block-local execution graphs" that iterate over all possible context execution graphs, and thus achieving full abstraction. Their blocks are, however, restricted to be sequential, which enables local validation of program transformations without actually showing that $[\![C_1 \parallel C_2]\!]$ is a function of $[\![C_1]\!]$ and $[\![C_2]\!]$.

Our notion of contextual refinement is based on partial correctness, and is insensitive to termination. In concurrent programs termination is interesting assuming scheduler fairness [Francez 1986], and, termination is generalized into a family of progress conditions [Liang and Feng 2020]. By using infinite traces, Brookes's semantics generalizes to fair infinite runs [Brookes 1996, §10], and is shown to be fully abstract w.r.t. operational "state-trace behaviors" consisting of sequences of states visited during the computation. We leave the task of incorporating this dimension into our semantics for future work, possibly by taking coinductive versions of our concrete semantics. For the abstract semantics, we expect that the local rewriting rules (see Lemma 4.26) will be handy.

ACKNOWLEDGMENTS

We thank Yotam Dvir and Ohad Kammar for fruitful discussions about this work, and the anonymous reviewers for their valuable feedback. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811) and the Israel Science Foundation (grant number 814/22).

DATA-AVAILABILITY STATEMENT

The artifact is available at [Svyatlovskiy et al. 2024].

REFERENCES

Martín Abadi and Gordon D. Plotkin. 2010. A Model of Cooperative Threads. *Log. Methods Comput. Sci.* 6, 4 (2010). https://doi.org/10.2168/LMCS-6(4:2)2010

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Nick Benton, Martin Hofmann, and Vivek Nigam. 2016. Effect-Dependent Transformations for Concurrent Programs. In *PPDP*. ACM, New York, NY, USA, 188–201. https://doi.org/10.1145/2967973.2968602

Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *CSL (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 16)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 107–121. https://doi.org/10.4230/LIPIcs.CSL.2012.107

Stephen Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Information and Computation* 127, 2 (1996), 145–163. https://doi.org/10.1006/inco.1996.0056

Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. 2010. Verifying Local Transformations on Relaxed Memory Models. In *CC*. Springer, Berlin, Heidelberg, 104–123. https://doi.org/10.1007/978-3-642-11970-5_7

Felice Cardone. 2021. Games, Full Abstraction and Full Completeness. In *The Stanford Encyclopedia of Philosophy* (Spring 2021 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2021/entries/games-abstraction/

Soham Chakraborty and Viktor Vafeiadis. 2016. Validating Optimizations of Concurrent C/C++ Programs. In *CGO*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2854038.2854051

Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential Reasoning for Optimizing Compilers under Weak Memory Concurrency. In *PLDI*. ACM, New York, NY, USA, 213–228. https://doi.org/10.1145/3519939.3523718

Jürgen Dingel. 1999. A Trace-Based Refinement Calculus for Shared-Variable Parallel Programs. In *AMAST*. Springer, Berlin, Heidelberg, 231–247. https://doi.org/10.1007/3-540-49253-4_18

Jürgen Dingel. 2002. A Refinement Calculus for Shared-Variable Parallel and Distributed Programming. *Form. Asp. Comput.* 14, 2 (dec 2002), 123–197. https://doi.org/10.1007/s001650200032

Mike Dodds, Mark Batty, and Alexey Gotsman. 2018. Compositional Verification of Compiler Optimisations on Relaxed Memory. In *ESOP*. Springer, Cham, 1027–1055. https://doi.org/10.1007/978-3-319-89884-1_36

Yotam Dvir, Ohad Kammar, and Ori Lahav. 2022. An Algebraic Theory for Shared-State Concurrency. In *APLAS*. Springer, Cham, 3–24. https://doi.org/10.1007/978-3-031-21037-2_1

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

Nissim Francez. 1986. *Fairness*. Springer. https://doi.org/10.1007/978-1-4612-4886-6

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). https://doi.org/10.46298/LMCS-17(3:9)2021

Akshay Gopalakrishnan, Clark Verbrugge, and Mark Batty. 2023. Memory Consistency Models for Program Transformations: An Intellectual Abstract. In *ISMM*. ACM, New York, NY, USA, 30–42. https://doi.org/10.1145/3591195.3595274

Charles Antony Richard Hoare. 1985. *Communicating sequential processes*. Prentice-Hall.

Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with Preconditions: A Simple Model of Relaxed Memory. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 194 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428262

Radha Jagadeesan, Gustavo Petri, and James Riely. 2012. Brookes Is Relaxed, Almost!. In *FoSSaCS*. Springer, Berlin, Heidelberg, 180–194. https://doi.org/10.1007/978-3-642-28729-9_12

Alan Jeffrey and James Riely. 2019. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. *Logical Methods in Computer Science* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:33)2019

Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency. *Proc. ACM Program. Lang.* 6, POPL, Article 54

(jan 2022), 30 pages. https://doi.org/10.1145/3498716
Cliff B. Jones. 1983. Tentative Steps toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (oct 1983), 596–619. https://doi.org/10.1145/69575.69577
Ohad Kammar. 2014. *Algebraic theory of type-and-effect systems.* Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/8910
Ryan Kavanagh and Stephen Brookes. 2018. A denotational account of C11-style memory. *CoRR* abs/1804.04214 (2018). arXiv:1804.04214 http://arxiv.org/abs/1804.04214
Ryan Kavanagh and Stephen Brookes. 2019. A Denotational Semantics for SPARC TSO. *Logical Methods in Computer Science* Volume 15, Issue 2 (May 2019). https://doi.org/10.23638/LMCS-15(2:10)2019
Artem Khyzha and Ori Lahav. 2022. Abstraction for Crash-Resilient Objects. In *ESOP*. Springer International Publishing, Cham, 262–289. https://doi.org/10.1007/978-3-030-99336-8_10
Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *POPL*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643
Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM*. Springer, 479–495. https://doi.org/10.1007/978-3-319-48989-6_29
Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439
Hongjin Liang and Xinyu Feng. 2020. Progress of Concurrent Objects. *Foundations and Trends in Programming Languages* 5, 4 (2020), 282–414. https://doi.org/10.1561/2500000041
Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A Rely-Guarantee-Based Simulation for Verifying Concurrent Program Transformations. In *POPL*. ACM, New York, NY, USA, 455–468. https://doi.org/10.1145/2103656.2103711
Hongjin Liang, Xinyu Feng, and Ming Fu. 2014. Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 3 (Mar. 2014), 55 pages. https://doi.org/10.1145/2576235
Robin Milner. 1980. *A calculus of communicating systems.* Springer.
Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4
Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *PLDI*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2491956.2491967
Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs*. Springer, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27
Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *ESOP*. Springer, Cham, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22
Daniel Poetzl and Daniel Kroening. 2016. Formalizing and Checking Thread Refinement for Data-Race-Free Execution Models. In *ESOP*. Springer, Berlin, Heidelberg, 515–530. https://doi.org/10.1007/978-3-662-49674-9_30
Vaughan Pratt. 1986. Modeling Concurrency with Partial Orders. *Int. J. Parallel Program.* 15, 1 (feb 1986), 33–71. https://doi.org/10.1007/BF01379149
Abhishek Kr Singh and Ori Lahav. 2023. An Operational Approach to Library Abstraction under Relaxed Memory Concurrency. *Proc. ACM Program. Lang.* 7, POPL, Article 53 (jan 2023), 31 pages. https://doi.org/10.1145/3571246
Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (jan 2023), 31 pages. https://doi.org/10.1145/3571232
Mikhail Svyatlovskiy, Shai Mermelstein, and Ori Lahav. 2024. *Coq Mechanization for "Compositional Semantics for Shared-Variable Concurrency" (PLDI 2024).* https://doi.org/10.5281/zenodo.10925596
Aaron Joseph Turon and Mitchell Wand. 2011. A Separation Logic for Refining Concurrent Objects. In *POPL*. ACM, New York, NY, USA, 247–258. https://doi.org/10.1145/1926385.1926415
Glynn Winskel. 1987. Event structures. In *ACPN*. Springer, Berlin, Heidelberg, 325–392. https://doi.org/10.1007/3-540-17906-2_31
Qiwen Xu, Willem P. de Roever, and Jifeng He. 1997. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects Comput.* 9, 2 (1997), 149–174. https://doi.org/10.1007/BF01211617

Received 2023-11-16; accepted 2024-03-31