Concurrent Incorrectness Separation Logic

AZALEA RAAD, Imperial College London and Meta, UK
JOSH BERDINE, Meta, UK
DEREK DREYER, MPI-SWS, Germany
PETER W. O'HEARN, Meta and University College London, UK

Incorrectness separation logic (ISL) was recently introduced as a theory of under-approximate reasoning, with the goal of proving that compositional bug catchers find actual bugs. However, ISL only considers sequential programs. Here, we develop *concurrent incorrectness separation logic* (CISL), which extends ISL to account for bug catching in concurrent programs. Inspired by the work on Views, we design CISL as a parametric framework, which can be instantiated for a number of bug catching scenarios, including race detection, deadlock detection, and memory safety error detection. For each instance, the CISL meta-theory ensures the *soundness* of incorrectness reasoning for free, thereby guaranteeing that the bugs detected are true positives.

CCS Concepts: • **Theory of computation → Concurrency**; **Semantics and reasoning**; • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: Concurrency, program logics, separation logic, bug catching

ACM Reference Format:
Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent Incorrectness Separation Logic. *Proc. ACM Program. Lang.* 6, POPL, Article 34 (January 2022), 29 pages. https://doi.org/10.1145/3498695

1 INTRODUCTION

Recently there has been a successful trend in automated static analysis tools that use *under-approximate* techniques to detect bugs in *concurrent* programs. The idea behind under-approximation is to focus on a *subset* of program behaviours, to ensure one detects only *true positives* (real bugs) rather than *false positives* (spurious bug reports). For instance, RacerD [Blackshear et al. 2018] uses under-approximation to detect data races in Java programs, while Brotherston et al. [2021] utilise under-approximation for deadlock detection. RacerD is the state-of-the-art tool in race detection, significantly outperforming other race detectors in terms of bugs found and fixed: over 3k races found by RacerD have been fixed before reaching production (see [Blackshear et al. 2018]). More significantly, it supported the conversion of Facebook's Android app to a multi-threaded UI model, performing counter-factual reasoning to detect races that could exist *if* a class were placed in a multi-threaded context, *before* it was placed there; thousands of classes were converted this way, leading to performance gains in Facebook's Android app. The deadlock detector of Brotherston et al. [2021] has also been useful in practice, with over two-hundred deadlocks fixed by Facebook engineers. Fixing deadlocks is especially impactful as they can lead to "app not responding" behaviour, which can be more difficult for engineers to detect than plain crashes.

Authors' addresses: Azalea Raad, Imperial College London and Meta, UK, azalea.raad@imperial.ac.uk; Josh Berdine, Meta, UK, josh@berdine.net; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Peter W. O'Hearn, Meta and University College London, UK, p.ohearn@ucl.ac.uk.

2475-1421/2022/1-ART34
https://doi.org/10.1145/3498695

These static tools are compositional and run quickly on code changes, often in time proportional to the size of a code change rather than that of the global codebase. This distinguishes them from whole-program dynamic testing tools, making them especially well-suited to deployment at code review time within continuous integration systems (a deployment model emphasised by Google and Facebook in articles on static analysis at scale [Sadowski et al. 2018; Distefano et al. 2019]).

Unfortunately, useful though they may be, there is currently no well-understood or unifying theory underpinning such under-approximate analyses. As a result, each time such a tool is developed, the authors spend a significant amount of technical effort proving a *no-false-positives* (NFP) theorem, stating that the bugs found by the tool are indeed real bugs. For instance, in the case of RacerD this took a separate technical effort by Gorogiannis et al. [2019] to establish its NFP theorem. In the case of [Brotherston et al. 2021], the authors have developed a large corpus of lemmas building towards their NFP theorem, requiring significant technical investment and expertise. Furthermore, each such technical effort proves an NFP theorem for a bespoke tool or technique, and one cannot easily port those results to other under-approximate tools or techniques.

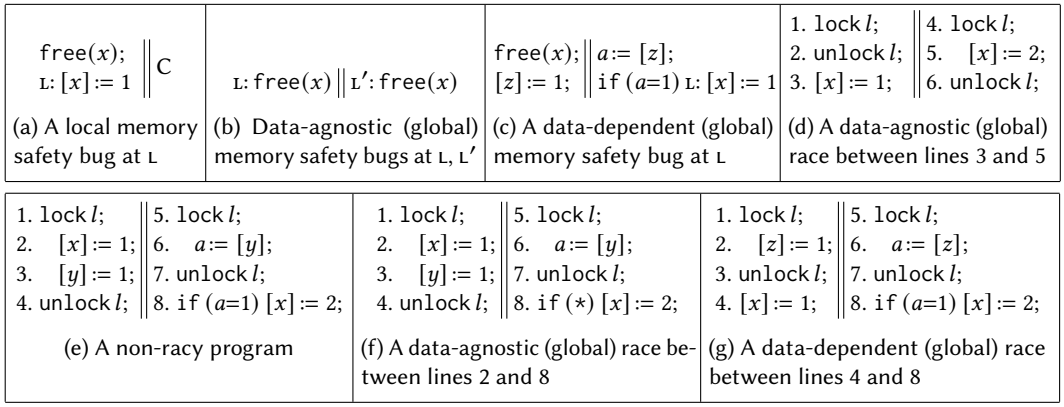
Ideally, one would have a unifying theory that underpins *concurrent under-approximate reasoning* (for proving *incorrectness*, i.e. the presence of bugs), in the same way that program logics such as [O'Hearn 2004; Owicki and Gries 1976] provide a foundation for concurrent over-approximate reasoning (for proving *correctness*, i.e. the absence of bugs). The key advantage of such a theory is that tools and techniques underpinned by it are accompanied by an NFP theorem *for free*.

Fortunately, the recent work of O'Hearn [2019] on *incorrectness logic* (IL) and its later extension to *incorrectness separation logic* (ISL) [Raad et al. 2020] have paved the way toward such a theory. IL lays the groundwork for a general theory of compositional under-approximate reasoning, and ISL has extended that to account for pointers and memory errors. However, these logics only apply to *sequential* programs and do not support reasoning about *concurrent* bug catching analyses.

In this paper, we close the gap by extending ISL to support concurrency. This task is far from straightforward, as witnessed in the correctness setting when extending separation logic (SL) [O'Hearn et al. 2001] with concurrency. Specifically, the advent of SL at the turn of the century led to a large body of work extending SL with concurrency, e.g. [O'Hearn 2004; Vafeiadis and Parkinson 2007; Dinsdale-Young et al. 2010; Nanevski et al. 2014; Jung et al. 2015; Raad et al. 2015], using different techniques to address different verification needs. This led to Parkinson's observation, in "The next 700 separation logics" [Parkinson 2010], that there has been "a disturbing trend for each new library or concurrency primitive to require a new separation logic". He rightfully argued "we shouldn't be inventing new separation logics, but should find the right logic to reason about interference". This insight eventually led to the concurrent Views framework [Dinsdale-Young et al. 2013], a parametric meta-theory of concurrent reasoning that distils the essence of separation logic, and can be instantiated to reason about different concurrent scenarios.

In the same spirit, in order to avoid inventing the "next 700 *incorrectness* separation logics", we develop *concurrent incorrectness separation logic* (CISL, pronounced "sizzle"), a unifying framework for concurrent under-approximate reasoning that can be instantiated to prove true-positives theorems for a range of concurrent bug catching scenarios. To our knowledge, CISL is the very *first* formal theory for concurrent under-approximate reasoning and bug catching. As with ISL, a key advantage of CISL is that it supports *compositional* reasoning, as needed to account for compositional analyses like those mentioned above. Moreover, CISL adds compositionality in a new dimension, namely *thread-locality*: we can reason about each concurrent thread in isolation.

Thanks to the soundness of CISL, each CISL instance is automatically accompanied by an NFP theorem. In §4–7, we instantiate CISL to detect concurrency bugs such as data races, deadlocks, and memory safety errors, along with a proof that the bugs found are real. Our CISL instantiations for races and deadlocks are inspired by the under-approximate analyses of RacerD and [Brotherston



(a) A local memory safety bug at L

(b) Data-agnostic (global) memory safety bugs at L, L′

(c) A data-dependent (global) memory safety bug at L

(d) A data-agnostic (global) race between lines 3 and 5

(e) A non-racy program

(f) A data-agnostic (global) race between lines 2 and 8

(g) A data-dependent (global) race between lines 4 and 8

Fig. 1. Several examples of memory safety bugs and races where all memory locations x, y, z initially hold 0

et al. 2021], respectively. Additionally, we strengthen our instantiations to catch races that RacerD could not, and to produce more informative assertions (i.e. with a witness trace) than those of [Brotherston et al. 2021].

Contributions and Outline. Our contributions (detailed in §2) are as follows. In §3 we present the general meta-theory of CISL. In §4 we instantiate CISL for race detection and compare it to RacerD. In §5 we instantiate CISL for deadlock detection and compare it to [Brotherston et al. 2021]. In §6 we further generalise CISL to account for thread interference. Using this generalisation, in §7 we instantiate CISL for subvariant reasoning (an under-approximate analogue of invariant reasoning), and use it to detect memory safety bugs. We discuss related work and conclude in §8.

2 OVERVIEW OF CISL

CISL at a Glance. As with its sequential counterpart ISL, CISL allows us to prove triples of the form $[p]\ \mathsf{C}\ [\epsilon : q]$, stating that *every* state in q is reachable by executing C starting in *some* state in p. The ϵ denotes an *exit condition* that may be either ok to denote normal (non-erroneous) execution, or $\epsilon \in \textsc{ErExit}$ to denote a buggy (erroneous) execution (ErExit is supplied to CISL as a parameter to distinguish different classes of bugs such as memory safety errors, races and so forth). A key part of CISL is its parallel composition rule, Par in Fig. 5 (p. 9), stating that if we prove $[p_i]\ \mathsf{C}_i\ [ok : q_i]$ for each $i \in \{1, 2\}$ in isolation, then we can also prove $[p_1 * p_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [ok : q_1 * q_2]$. Note that Par is identical to its over-approximate analogue in [O'Hearn 2004] and is similarly *compositional*: we can identify a normal execution of $\mathsf{C}_1 \,||\, \mathsf{C}_2$ by considering each thread in isolation. Observe that Par only allows us to prove normal (ok) triples; as we describe below, CISL provides different techniques for proving buggy triples, depending on the *bug category*.

The Three Faces of Concurrent Bugs (Errors). When using CISL to detect different classes of bugs (e.g. memory safety errors and races), we have identified three bug categories: 1) *local* (interleaving-agnostic) bugs; 2) *global* (interleaving-dependent), *data-agnostic* bugs; and 3) *global, data-dependent* bugs. We describe these three categories through several examples in Fig. 1, where we write τ_1 and τ_2 for the left and right threads in each example, respectively.

Local bugs are due to one thread, say τ, in that they arise even when τ executes in isolation (i.e. sequentially) and are thus *interleaving-agnostic*. An example of this is shown in Fig. 1a: regardless of the behaviour of C in τ_2 (the right thread), executing τ_1 (the left thread) leads to a use-after-free (memory safety) bug at L as x is accessed after it is deallocated. As we describe later in §3, local bugs can be detected using the ParEr rule of CISL in Fig. 5 (on p. 9): due to the short-circuit semantics of

errors (whereby execution is terminated upon encountering an error), a bug is reached by executing a concurrent program, $[p]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [\epsilon : q]$, if it is reached by executing one thread, $[p]\ \mathsf{C}_1\ [\epsilon : q]$.

In contrast to local bugs, global bugs are due to how concurrent threads interact with one another and arise only under certain interleavings, i.e. they are *interleaving-dependent*. For instance, Figures 1b and 1c both depict examples of global bugs. In particular, in an interleaving of Fig. 1b where τ_1 is executed after (resp. before) τ_2, we reach a use-after-free bug at L (resp. L′). Analogously, in an interleaving of Fig. 1c where τ_2 is executed after τ_1, the condition of the if statement is satisfied and thus we reach a use-after-free bug at L. Note that there is a *data dependency* between τ_1 and τ_2 in Fig. 1c in that τ_1 may affect the *control flow* of τ_2: the value read in $a := [z]$, and subsequently the condition of `if` and whether L: $[x] := 1$ is executed, depends on whether τ_2 executes $a := [z]$ before or after τ_1 executes $[z] := 1$. As such, the bug at L is *data-dependent*. By contrast, the threads in Fig. 1b cannot affect the control flow of one another and thus the bugs at L and L′ are *data-agnostic*. Intuitively, data dependence arises through *deterministic* (non-random) conditions in if/loop statements, prescribing a certain execution path. Specifically, were we to replace the (deterministic) condition a=1 in Fig. 1c with the *non-deterministic* expression $\star$ (which evaluates to an arbitrary value), then the memory-safety error at L would be rendered data-agnostic.

Detecting Global Bugs. Due to their global nature, global bugs (be they data-agnostic/dependent) cannot be detected using ParEr. Instead, as they manifest only under certain interleavings, they can be detected using the ParSeq, ParL and ParR rules of CISL (in Fig. 5), which enable us to consider certain interleavings. For instance, let $\mathsf{C}_1 = \mathsf{C}_3; \mathsf{C}_4$ and $\mathsf{C} = \mathsf{C}_1 \,||\, \mathsf{C}_2$, and let ϵ denote an error that manifests only in an interleaving of C in which C_2 is executed between C_3 and C_4 as $[p]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [\epsilon : q]$. We can then detect ϵ as follows. First, we use ParL to execute C_3 normally, (the $[p]\ \mathsf{C}_3\ [ok : r]$ premise); and then show that the continuation $\mathsf{C}_4 \,||\, \mathsf{C}_2$ yields error ϵ (the $[r]\ \mathsf{C}_4 \,||\, \mathsf{C}_2\ [\epsilon : q]$ premise). To do this, we then use ParSeq to execute C_2 before C_4, i.e. we show $[r]\ \mathsf{C}_2; \mathsf{C}_4\ [\epsilon : q]$.

Unfortunately, however, ParSeq, ParL and ParR are *not compositional* as they consider multiple threads at every proof step, rather than examining each thread in isolation, and require the user to determine an interleaving *a priori*. However, as we discuss below, in most cases we *can* detect global bugs compositionally in CISL by encoding buggy executions as normal ones and then using the compositional Par rule. We elaborate on this below through several examples of data races.

Data Races. Unlike memory safety bugs that may be local or global, *data races* (hereafter simply races) belong solely to the global category. Specifically, two accesses (reads and writes) of a given program C race with one another if 1) they are *conflicting*, i.e. they are by distinct threads, on the same location, and at least one of them is a write; and 2) they appear next to each other in a given interleaving (a.k.a. history) of C. As such, races are attributed to two threads (and are thus global) and may manifest only under certain interleavings. To see this, consider the example in Fig. 1d, and let us write $H = [1, \cdots, n]$ for an interleaving where the instructions numbered $1 \cdots n$ are executed in order. Note that the program in Fig. 1d induces several (partial) interleavings, including $H = [1, 2, 4, 3, 5]$ and $H' = [4, 5, 6, 1, 2, 3]$. The two conflicting accesses on x (lines 3 and 5) *race* in H as they are adjacent in H. By contrast, they *do not race* in H': thanks to the mutual exclusion induced by the lock on l, they cannot appear as adjacent accesses when τ_2 acquires l first.

Data-Dependent Bugs in Practice. As shown by Blackshear et al. [2018] and Brotherston et al. [2021], data-dependent bugs (due to deterministic conditions in if/while statements) make the task of detecting true bugs (true positives) much more difficult. To see this, consider the example in Fig. 1e and note that it does not allow any data races: both threads access y while holding the lock l, and τ_2 accesses x only if $a = 1$, i.e. τ_2 accesses x only when its critical section is executed after that in τ_1, forcing a *happens-before* order from $[x] := 1$ to $[x] := 2$. On the other hand, it is always

possible to find a race in Fig. 1f, when the value non-deterministically picked by $*$ is non-zero, as witnessed by the history $[5, 6, 7, 1, 2, 8]$. Note that the program in Fig. 1f only differs from the one in Fig. 1e in that the deterministic condition (a=1) is replaced with the non-deterministic $*$.

Although we can generalise the CISL *theory* to detect data-dependent bugs, ruling out data-dependent bugs is non-trivial in *practice* (e.g. due to the happens-before order induced under certain interleavings). This difficulty led Blackshear et al. [2018] and Brotherston et al. [2021] to focus only on data-agnostic bugs by assuming that all program conditionals are *non-deterministic* ($*$) as in Fig. 1f. As such, in the remainder of the paper, we follow the same practical approach and focus on providing a theoretical foundation for *data-agnostic* bug catching.

CISL for Detecting Data-Agnostic Bugs. Returning to Fig. 1d, note that the race between lines 3 and 5 in $H = [1, 2, 4, 3, 5]$ is data-agnostic. As mentioned above, we can detect data-agnostic bugs *compositionally* by treating buggy (here racy) executions as normal (non-erroneous) executions and using the Par rule to analyse each thread in isolation and combine their results. More concretely, to enable compositional reasoning, we do not treat races as errors. Rather, we compute a local (sequential) history of each thread in isolation, combine them together using Par, and ultimately examine them to detect data races and construct a global (concurrent) history *witnessing* the race.

To see this, consider the CISL proof sketch of the race in Fig. 1d in Fig. 2, where the sequential histories of τ_1 and τ_2 are initially both $[]$, as denoted by $\tau_1 \mapsto [] * \tau_2 \mapsto []$. Using Par, for $i \in \{1, 2\}$ we reason about τ_i in isolation starting from $\tau_i \mapsto []$. Subsequently, we obtain $\tau_1 \mapsto [1, 2, 3]$ and $\tau_2 \mapsto [4, 5, 6]$ separately, and combine them using Par into $\tau_1 \mapsto [1, 2, 3] * \tau_2 \mapsto [4, 5, 6]$. Finally, we use the CISL rule of consequence, Cons, to additionally obtain $\mathsf{race}(3, 5, [1, 2, 4, 3, 5])$, describing a race between lines 3 and 5, witnessed by *global history* $[1, 2, 4, 3, 5]$. A global history of $\tau_1 \mapsto H_1$ and $\tau_2 \mapsto H_2$ is obtained by computing all permutations of $H_1 +\!\!+ H_2$ (where $+\!\!+$ denotes concatenation) and then filtering out those that are not *well-formed*. Intuitively, well-formed histories respect the mutual exclusion semantics of locks. For instance, $[1, 4, 2, 3, 5, 6]$ in Fig. 1d is not well-formed: it allows τ_2 to acquire l (4) while it is held by τ_1 (at 1). We formalise well-formed histories in §4.

$[\tau_1 \mapsto [] * \tau_2 \mapsto []]$

$[\tau_1 \mapsto []]$	$[\tau_2 \mapsto []]$
1. lock l;	4. lock l;
$[ok\colon \tau_1 \mapsto [1]]$	$[ok\colon \tau_2 \mapsto [4]]$
2. unlock l;	5. $[x] := 2$;
$[ok\colon \tau_1 \mapsto [1, 2]]$	$[ok\colon \tau_2 \mapsto [4, 5]]$
3. $[x] := 1$;	6. unlock l;
$[ok\colon \tau_1 \mapsto [1, 2, 3]]$	$[ok\colon \tau_2 \mapsto [4, 5, 6]]$

$[ok\colon \tau_1 \mapsto [1, 2, 3] * \tau_2 \mapsto [4, 5, 6]]$ // Par

$\left[ok\colon \begin{array}{l}\tau_1 \mapsto [1, 2, 3] * \tau_2 \mapsto [4, 5, 6] \\ \wedge\ \mathsf{race}(3, 5, [1, 2, 4, 3, 5])\end{array}\right]$ // Cons

Fig. 2. A CISL proof of the race in Fig. 1d

Lastly, our presentation of histories thus far was abbreviated by merely recording line numbers. However, to identify races, rather than recording line numbers (which requires examining the code), we record execution *events*. For instance, the sequential history of τ_1 in Fig. 1f is $\tau_1 \mapsto H_1$ with $H_1 = [\mathsf{L}(\tau_1, l), \mathsf{W}(\tau_1, 2, x), \mathsf{W}(\tau_1, 3, y), \mathsf{U}(\tau_1, l)]$ (abbreviated as $[1, 2, 3, 4]$), where $\mathsf{L}(\tau_1, l)$ and $\mathsf{U}(\tau_1, l)$ respectively denote (the events for) locking and unlocking l on lines 1 and 4, and $\mathsf{W}(\tau_1, 2, x)$ and $\mathsf{W}(\tau_1, 3, y)$ respectively denote writing to x and y on lines 2 and 3. Similarly, a sequential history of τ_2 is $\tau_2 \mapsto H_2$ with $H_2 = [\mathsf{L}(\tau_2, l), \mathsf{R}(\tau_2, 6, y), \mathsf{U}(\tau_2, l), \mathsf{W}(\tau_2, 8, x)]$. We combine H_1 and H_2 into $H = [\mathsf{L}(\tau_2, l), \mathsf{R}(\tau_2, 6, y), \mathsf{U}(\tau_2, l), \mathsf{L}(\tau_1, l), \mathsf{W}(\tau_1, 2, x), \mathsf{W}(\tau_2, 8, x)]$, witnessing the race via the adjacent conflicting accesses $\mathsf{W}(\tau_1, 2, x)$ and $\mathsf{W}(\tau_2, 8, x)$. Note that we do not record the labels (line numbers) of lock/unlock events: labels are used to locate races which can only occur between memory accesses. Moreover, as we focus on data-agnostic races, we need not record the values read/written. That is, the values read/written do not affect the control flow and have no bearing on races. In the remainder of the paper, we use both abbreviated histories (for exposition brevity) and full histories.

Summary: Compositional Reasoning with CISL. CISL proof rules (Fig. 5) support *compositional* reasoning via: (1) Par for normal executions; and (2) ParEr for detecting local bugs. Moreover,

(3) one can detect data-agnostic bugs by encoding buggy executions as normal ones and using Par to detect them. In the case of (2), ParEr reduces *concurrent* bug detection to *sequential*, which can be automated as in [Raad et al. 2020]. In the case of (3), we instantiate CISL to detect races (CISL_{RD}), deadlocks (CISL_{DD}), and memory safety errors (CISL_{SV}); CISL_{RD} and CISL_{DD} are inspired by the under-approximate analyses of the [Blackshear et al. 2018] and [Brotherston et al. 2021] tools.

3 THE CISL FRAMEWORK

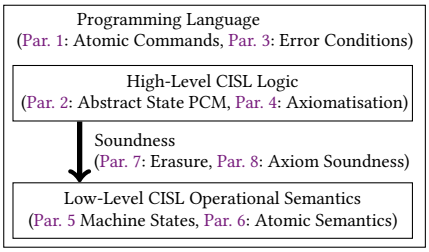


Fig. 3. An overview of the CISL framework

We present the CISL meta-theory. The CISL framework is parametric and may be instantiated for concurrent, under-approximate reasoning for a multitude of applications, including e.g. CISL_{DC} (CISL with disjoint concurrency) for detecting memory safety bugs, CISL_{RD} (CISL with race detection) for detecting races on shared memory, and CISL_{DD} (CISL with deadlock detection) for detecting deadlock scenarios. To instantiate CISL, one must supply CISL with the specified parameters; the soundness of the instantiated CISL reasoning then follows immediately from the soundness of the framework (see Thm. 3.8). For clarity, we delineate the CISL parameters enclosed in solid boxes. To provide a clearer account of CISL, we often follow CISL parameters with their CISL_{DC} instantiation for detecting memory safety bugs.

Programming Language. We build CISL on a simple programming language comprising standard composite commands, and parametrised by a set of *atomic* commands. This allows us to instantiate CISL for a number of use-cases without changing the underlying meta-theory. Our programming language is given by the C grammar in Def. 3.1, and includes atomic commands (a) supplied as a parameter (Par. 1), as well as the standard constructs of `skip`, sequential composition ($C_1; C_2$), non-deterministic choice ($C_1 + C_2$), loops ($C^\star$) and parallel composition ($C_1 \,||\, C_2$).

Parameter 1 (Atomic commands). Assume a set of *atomic* commands, Atom, ranged over by a.

Definition 3.1 (CISL language). The *CISL programming language* is defined as follows:

$$\text{Comm} \ni C ::= a \mid \texttt{skip} \mid C_1; C_2 \mid C_1 + C_2 \mid C^\star \mid C_1 \,||\, C_2$$

Example 3.2 (CISL_{DC} atomics). The *CISL_{DC} atomic commands*, Atom_{DC}, are defined as follows:

$$\text{Atom}_{\text{DC}} \ni a ::= \text{L}\colon \texttt{error} \mid x := v \mid \texttt{assume}(B) \mid x := \texttt{alloc()} \mid \text{L}\colon \texttt{free}(x) \mid \text{L}\colon x := [y] \mid \text{L}\colon [x] := y$$

The CISL_{DC} atomic commands include explicit error statements (`error`), assume statements ($\texttt{assume}(B)$) to model deterministic conditionals/loops, assignment ($x := v$) and heap-manipulating commands for allocation ($x := \texttt{alloc()}$), disposal ($\texttt{free}(x)$), lookup (reading from the heap, $x := [y]$) and mutation (writing to the heap $[x] := y$). We assume a set BAst of *Boolean assertions*; we use B as a metavariable for a Boolean assertion, $v, v' \cdots$ for values, and x, y, z for variables.

To better track memory safety errors and connect them to culprit instructions, we annotate instructions that may cause such errors with a label $\text{L} \in \text{Label}$. As we demonstrate shortly, when an error is encountered we report the label of the offending instruction (e.g. L). As such, we only consider *well-formed* programs: those with unique labels across their constituent instructions. For brevity, we drop the instruction labels when they are immaterial to the discussion.

3.1 CISL Logic and Proof Rules

State PCM. Program logics, and reasoning frameworks in general, do not typically reason directly about the low-level machine states. Rather, they provide a high-level (abstract) representation of the state, often equipped with additional instrumentation that supports certain reasoning principles. For instance, in order to reason about concurrent accesses to the memory, one can model the state as a shared heap of memory locations, with each location instrumented with a *fractional permission* $\pi \in (0, 1]$, where $\pi = 1$ on location x denotes *full ownership* on x granting permission for writing to x, while $0 < \pi < 1$ denotes *partial ownership* on x sufficient for reading from x.

In separation logic and its family of descendants, the high-level states are typically modelled by a *partial commutative monoid* (PCM) of the form $(\textsc{State}, \circ, \textsc{State}^0)$, where $\textsc{State}$ denotes the set of states; $\circ : \textsc{State} \times \textsc{State} \rightharpoonup \textsc{State}$ denotes the partial state composition operator that is commutative and associative; and $\textsc{State}^0 \subseteq \textsc{State}$ denotes the set of unit states. The reasoning is then carried out via $p, q \in \textsc{View} \triangleq \mathcal{P}(\textsc{State})$, describing *views* (sets of states).

Parameter 2 (State PCM). Assume a *partial commutative monoid (PCM) for states*, $(\textsc{State}, \circ, \textsc{State}^0)$, where $\textsc{State}^0 \subseteq \textsc{State}$ and:

- the composition function, $\circ : \textsc{State} \times \textsc{State} \rightharpoonup \textsc{State}$, is commutative and associative;
- for all $s \in \textsc{State}$, there exists $s_0 \in \textsc{State}^0$ such that $s \circ s_0 = s$; and
- for all $s, s' \in \textsc{State}$ and $s_0 \in \textsc{State}^0$, if $s \circ s_0 = s'$ then $s = s'$.

Example 3.3 (CISL_{DC} States). We model a CISL_{DC} state $s \in \textsc{State}_{\text{DC}}$ as a partial map associating each program variable or location with a value and a *fractional permission* $\pi \in (0, 1]$: $\textsc{State}_{\text{DC}} \triangleq (\textsc{Var} \xrightarrow{\text{fin}} \textsc{Val} \times (0, 1]) \cup (\textsc{Loc} \xrightarrow{\text{fin}} (\textsc{Val} \uplus \{\bot\}) \times (0, 1])$. The designated value $\bot \notin \textsc{Val}$ is used to track those locations that have been *deallocated*. That is, given $l \in \textsc{Loc}$, if $s(l)=(v, -)$ and $v \in \textsc{Val}$ then l is allocated in s and holds value v; and if $v=\bot$ then l has been deallocated.

CISL_{DC} composition is standard: $(s_1 \circ_{\text{DC}} s_2)(x)$ is 1) $s_i(x)$ if $x \in dom(s_i) \setminus dom(s_{3-i})$ for $i \in \{1, 2\}$; 2) $(v, \pi_1{+}\pi_2)$ if $s_1(x) = (v, \pi_1)$, $s_2(x) = (v, \pi_2)$ and $\pi_1{+}\pi_2 \leq 1$. If there exists $x \in dom(s_1) \cap dom(s_2)$ s.t. the conditions of 2) are not satisfied, the entire composition $s_1 \circ_{\text{DC}} s_2$ is undefined. CISL_{DC} unit set is $\textsc{State}^0_{\text{DC}} \triangleq \{\emptyset\}$, where $\emptyset$ is an empty function. CISL_{DC} state PCM is $(\textsc{State}_{\text{DC}}, \circ_{\text{DC}}, \textsc{State}^0_{\text{DC}})$.

Definition 3.4 (Views). The set of *views* is $\textsc{View} \triangleq \mathcal{P}(\textsc{State})$.

Notation. We use p, q, r as meta-variables for views (i.e. $p, q, r \in \textsc{View}$). We write $p * q$ for $\{s \circ s' \mid s \in p \wedge s' \in q\}$; $p \wedge q$ for $p \cap q$; $p \vee q$ for $p \cup q$; false for $\emptyset$; and true for $\textsc{View}$. In the context of CISL_{DC}, we write emp for $\{\emptyset\}$ and $l \overset{\pi}{\mapsto} v$ (resp. $x \overset{\pi}{\mapsto} v$) for $\{[l \mapsto (v, \pi)]\}$ (resp. $\{[x \mapsto (v, \pi)]\}$). We write $l \mapsto v$ (resp. $x \mapsto v$) for $l \overset{1}{\mapsto} v$ (resp. $x \overset{1}{\mapsto} v$), $l \overset{\pi}{\not\mapsto}$ for $l \overset{\pi}{\mapsto} \bot$, and $l \not\mapsto$ for $l \overset{1}{\not\mapsto}$.

Exit Conditions. The CISL theory uses *under-approximate triples* [O'Hearn 2019] of the form $[p]\ \text{C}\ [\epsilon : q]$, interpreted as: q describes a *subset* of the states that can be reached from p by executing C, where ϵ denotes an *exit condition* indicating either normal or erroneous termination.

We define the set of exit conditions as $\textsc{Exit} \triangleq \{ok\} \uplus \textsc{ErExit}$, where ok denotes normal termination, and $\epsilon \in \textsc{ErExit}$ denotes an erroneous termination. Erroneous conditions are reasoning-specific and thus supplied as a CISL parameter. For instance, an error condition in CISL_{DC} denotes either a memory safety bug or an explicit error (after executing `error`). Concretely, we define CISL_{DC} error conditions as $\textsc{ErExit}_{\text{DC}} \triangleq \{mse(\text{L}), er(\text{L}) \mid \text{L} \in \textsc{Label}\}$, where $mse(\text{L})$ denotes a memory safety bug encountered at the L-labelled instruction and $er(\text{L})$ denotes an explicit error at L.

Parameter 3 (Error conditions). Assume a set of *error conditions*, $\textsc{ErExit}$, where $ok \notin \textsc{ErExit}$.

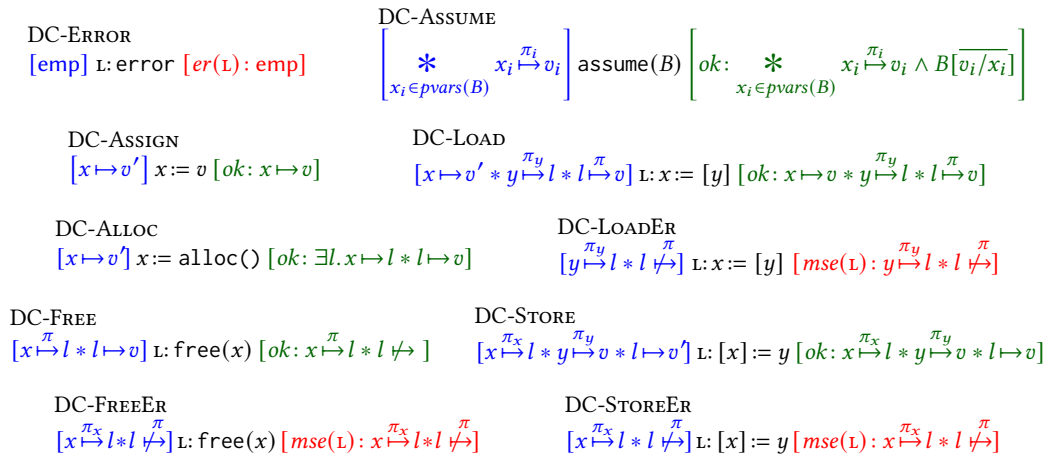


Fig. 4. The CISL_{DC} axioms

Atomic Axioms. We shortly define the under-approximate proof system of CISL. As atomic commands are supplied as a parameter, the CISL proof system is accordingly parametrised by their set of under-approximate *axioms* (Par. 4). An atomic axiom is a tuple of the form $(p, \mathrm{a}, \epsilon, q)$, with $p, q \in \textsc{View}$, $\mathrm{a} \in \textsc{Atom}$ and $\epsilon \in \textsc{Exit}$, and is lifted to the CISL proof rule $[p]$ a $[\epsilon : q]$ (see Atom).

Parameter 4 (Axioms). Assume a set of *axioms* $\textsc{Axiom} \subseteq \textsc{View} \times \textsc{Atom} \times \textsc{Exit} \times \textsc{View}$.

Example 3.5 (CISL_{DC} axioms). The CISL_{DC} axioms, $\textsc{Axiom}_{\text{DC}}$, are given in Fig. 4 and are analogous to those of Raad et al. [2020]. For better readability, we present the axioms as inference rules with CISL triples of the form $[p]$ a $[\epsilon : q]$ in their conclusion, rather than tuples of the form $(p, \mathrm{a}, \epsilon, q)$.

CISL Proof Rules. We present the under-approximate CISL proof system in Fig. 5. As in [Raad et al. 2020; O'Hearn 2019], our triples are of the form: $\vdash [p]\ \mathrm{C}\ [\epsilon : q]$, denoting that every state in the postcondition q is reachable from some state in the precondition p under ϵ. That is, for each s_q in q, there exists s_p in p such that executing C on s_p terminates with ϵ and yields s_q.

The Skip, SeqEr, Seq, Loop1, Loop2, Choice, Cons and Disj rules are as in [O'Hearn 2019; Raad et al. 2020]). Similarly, the Frame rule is analogous to that of [Raad et al. 2020], except that the $*$ operator here denotes the general notion of composition defined by lifting the composition operator of the underlying PCM to views (sets of states), rather than that of heap composition in [Raad et al. 2020].

The Atom rule simply lifts atomic axioms as CISL proof rules. The Par rule is an under-approximate analogue of the disjoint concurrency rule in concurrent separation logic (CSL) [O'Hearn 2004], stating that if the states in q_1 (resp. q_2) are reachable from those in p_1 (resp. p_2) when executing C_1 (resp. C_2), then the combined states in $q_1 * q_2$ are reachable from $p_1 * p_2$ when executing $\mathrm{C}_1 \,||\, \mathrm{C}_2$. Note that, unlike in CSL where the composition operator imposes *disjointness*, the composition operator in CISL does not and merely mandates composability as defined by the PCM (Par. 2).

The ParEr rule is the concurrent analogue of SeqEr, describing the *short-circuiting* semantics of concurrent executions: given $i \in \{1, 2\}$, if running the smaller program C_i results in an error, then running the larger $\mathrm{C}_1 \,||\, \mathrm{C}_2$ also results in an error. Intuitively, this is because the behaviours of $\mathrm{C}_1; \mathrm{C}_2$ and $\mathrm{C}_2; \mathrm{C}_1$ are both included in those of $\mathrm{C}_1 \,||\, \mathrm{C}_2$, in that they describe two possible *interleavings* when executing $\mathrm{C}_1 \,||\, \mathrm{C}_2$. As such, as in SeqEr, if running C_1 (resp. C_2) yield an error, then the overall execution of the $\mathrm{C}_1; \mathrm{C}_2$ (resp. $\mathrm{C}_2; \mathrm{C}_1$) interleaving of $\mathrm{C}_1 \,||\, \mathrm{C}_2$ also yields an error.

$$\textsc{Skip}\ \vdash [p]\ \texttt{skip}\ [ok\colon p] \qquad \textsc{SeqEr}\ \frac{\vdash [p]\ \mathrm{C}_1\ [\epsilon\colon q] \quad \epsilon \in \textsc{ErExit}}{\vdash [p]\ \mathrm{C}_1;\mathrm{C}_2\ [\epsilon\colon q]} \qquad \textsc{Seq}\ \frac{\vdash [p]\ \mathrm{C}_1\ [ok\colon r] \quad \vdash [r]\ \mathrm{C}_2\ [\epsilon\colon q]}{\vdash [p]\ \mathrm{C}_1;\mathrm{C}_2\ [\epsilon\colon q]} \qquad \textsc{Loop1}\ \vdash [p]\ \mathrm{C}^\star\ [ok\colon p]$$

$$\textsc{Loop2}\ \frac{\vdash [p]\ \mathrm{C}^\star;\mathrm{C}\ [\epsilon\colon q]}{\vdash [p]\ \mathrm{C}^\star\ [\epsilon\colon q]} \qquad \textsc{Atom}\ \frac{(p,\mathrm{a},\epsilon,q) \in \textsc{Axiom}}{\vdash [p]\ \mathrm{a}\ [\epsilon\colon q]} \qquad \textsc{Choice}\ \frac{\vdash [p]\,\mathrm{C}_i\ [\epsilon\colon q] \qquad \text{for some } i\in\{1,2\}}{\vdash [p]\ \mathrm{C}_1+\mathrm{C}_2\ [\epsilon\colon q]} \qquad \textsc{Frame}\ \frac{\vdash [p]\ \mathrm{C}\ [\epsilon\colon q]}{\vdash [p*r]\ \mathrm{C}\ [\epsilon\colon q*r]}$$

$$\textsc{Cons}\ \frac{p'\subseteq p \quad \vdash [p']\ \mathrm{C}\ [\epsilon\colon q'] \quad q\subseteq q'}{\vdash [p]\ \mathrm{C}\ [\epsilon\colon q]} \qquad \textsc{Disj}\ \frac{\vdash [p_1]\ \mathrm{C}\ [\epsilon\colon q_1] \quad \vdash [p_2]\ \mathrm{C}\ [\epsilon\colon q_2]}{\vdash [p_1\vee p_2]\ \mathrm{C}\ [\epsilon\colon q_1\vee q_2]} \qquad \textsc{Par}\ \frac{\vdash [p_i]\ \mathrm{C}_i\ [ok\colon q_i] \qquad \text{for all } i\in\{1,2\}}{\vdash [p_1*p_2]\ \mathrm{C}_1\,||\,\mathrm{C}_2\ [ok\colon q_1*q_2]}$$

$$\textsc{ParEr}\ \frac{\epsilon \in \textsc{ErExit} \quad \vdash [p]\ \mathrm{C}_i\ [\epsilon\colon q] \qquad \text{for some } i\in\{1,2\}}{\vdash [p]\ \mathrm{C}_1\,||\,\mathrm{C}_2\ [\epsilon\colon q]} \qquad \textsc{ParSeq}\ \frac{\vdash [p]\ \mathrm{C}_1;\mathrm{C}_2\ [\epsilon\colon q] \quad \text{or} \quad \vdash [p]\ \mathrm{C}_2;\mathrm{C}_1\ [\epsilon\colon q]}{\vdash [p]\ \mathrm{C}_1\,||\,\mathrm{C}_2\ [\epsilon\colon q]}$$

$$\textsc{ParL}\ \frac{\mathrm{C}_1{=}\mathrm{C}_3;\mathrm{C}_4 \quad \vdash [p]\ \mathrm{C}_3\ [ok\colon r] \quad \vdash [r]\ \mathrm{C}_4\,||\,\mathrm{C}_2\ [\epsilon\colon q]}{\vdash [p]\ \mathrm{C}_1\,||\,\mathrm{C}_2\ [\epsilon\colon q]} \qquad \textsc{ParR}\ \frac{\mathrm{C}_2{=}\mathrm{C}_3;\mathrm{C}_4 \quad \vdash [p]\ \mathrm{C}_3\ [ok\colon r] \quad \vdash [r]\ \mathrm{C}_1\,||\,\mathrm{C}_4\ [\epsilon\colon q]}{\vdash [p]\ \mathrm{C}_1\,||\,\mathrm{C}_2\ [\epsilon\colon q]}$$

Fig. 5. The CISL proof rules

Finally, ParL, ParR and ParSeq are concurrent analogues of Seq, stating that the states in q are reachable from those in p by executing $\mathrm{C}_1\,||\,\mathrm{C}_2$, if they are reachable under a particular *interleaving* of $\mathrm{C}_1\,||\,\mathrm{C}_2$. For instance, when $\mathrm{C}_1 = \mathrm{C}_3;\mathrm{C}_4$, ParL captures the $\mathrm{C}_3;(\mathrm{C}_4\,||\,\mathrm{C}_2)$ interleaving of $\mathrm{C}_1\,||\,\mathrm{C}_2$: if executing C_3 from p terminates normally and yields r ($\vdash\ [p]\ \mathrm{C}_3\ [ok\colon r]$) and executing the continuation $\mathrm{C}_4\,||\,\mathrm{C}_2$ from r yields q under ϵ, then executing $\mathrm{C}_1\,||\,\mathrm{C}_2$ from p results in q under ϵ.

Observe that Par can be *derived* from ParSeq, Seq and Frame as follows:

$$\dfrac{\dfrac{\dfrac{\vdash [p_1]\ \mathrm{C}_1\ [ok\colon q_1]}{\vdash [p_1*p_2]\ \mathrm{C}_1\ [ok\colon q_1*p_2]}\ \textsc{Frame} \qquad \dfrac{\vdash [p_2]\ \mathrm{C}_2\ [ok\colon q_2]}{\vdash [q_1*p_2]\ \mathrm{C}_2\ [ok\colon q_1*q_2]}\ \textsc{Frame}}{\vdash [p_1*p_2]\ \mathrm{C}_1;\mathrm{C}_2\ [ok\colon q_1*q_2]}\ \textsc{Seq}}{\vdash [p_1*p_2]\ \mathrm{C}_1\,||\,\mathrm{C}_2\ [ok\colon q_1*q_2]}\ \textsc{ParSeq}$$

Note that the Par rule enables *compositional* reasoning in that it is *interleaving-agnostic*, allowing us to reason about the behaviour of each thread in isolation, i.e. locally, combining the results at the end. Intuitively, this is because of the resources accessed by distinct threads are compatible with one another, then their combined result is reachable regardless of their interleaving. That is, the combined result is reachable under *all* possible interleavings of concurrent threads. Similarly, ParEr enables compositional bug-catching by allowing us to reason about the erroneous behaviour of one thread in isolation. This is thanks to the short-circuiting semantics of CISL for concurrent executions: if one thread terminates erroneously, then the overall program also terminates erroneously. By contrast, ParL, ParR and ParSeq are not compositional and correspond to a particular interleaving.

As discussed in §2, ParEr is used to detect *local* bugs, i.e. those bugs that are *interleaving-agnostic* and can manifest by executing a single thread, regardless of the behaviour of concurrent threads. For instance, when concurrent threads access disjoint heap resources, then memory safety bugs such as use-after-free are instances of local bugs: if a thread τ accesses a memory location owned by τ after it has already been freed (deallocated) by τ, then a use-after-free error can always be encountered regardless of the behaviour of other threads running concurrently with τ.

However, as discussed in §2, certain bugs are *global* (i.e. they depend on the behaviour of two or more threads) and are *interleaving-dependent* in that they only manifest under certain interleavings. For instance, data races and deadlocks are examples of bugs that may only occur under certain interleavings of two or more threads. As such, ParEr cannot be used to detect such global bugs. Although ParL and ParR can indeed be used to detect global bugs such as data races, they are not ideal due to their non-compositionality. However, as discussed in §2, we can detect (global) data-agnostic errors *compositionally* by encoding *bugs as non-errors*, whereby we treat buggy executions as normal (non-erroneous) ones and use Par to analyse them compositionally. As such, we show that Par and ParEr are indeed sufficient to detect a significant number of bugs compositionally.

3.2 CISL Model and Semantics

We next present the CISL operational semantics, parametrised by *machine states* (Par. 5 below). Note that while the states in Par. 2 provide a high-level (often instrumented) state representation, the machine states denote a low-level representation of the machine (without instrumentation). For instance, CISL_{DC} machine states forgo the fractional instrumentation of their high-level counterpart.

Parameter 5 (Machine states). Assume a set of *machine states*, MState, ranged over by m.

Example 3.6 (CISL_{DC} machine states). A CISL_{DC} machine state $m \in \text{MState}_{\text{DC}}$ is a partial function from variables and locations to $\text{Val} \uplus \{\bot\}$: $\text{MState}_{\text{DC}} : (\text{Var} \xrightarrow{\text{fin}} \text{Val}) \cup (\text{Loc} \xrightarrow{\text{fin}} \text{Val} \uplus \{\bot\})$.

Atomic Semantics. As the set of atomic commands is supplied as a parameter to the CISL programming language (Par. 1), the CISL operational semantics is further parametrised by the *semantics of atomic commands* (Par. 6 below), defined as (machine) state transformers. For instance, the CISL_{DC} atomic semantics is analogous to its counterpart in [Raad et al. 2020].

Parameter 6 (Atomic semantics). Assume an *atomic semantics function* $[\![.]\!]_{\text{A}} : \text{Atom} \to \text{Exit} \to \mathcal{P}(\text{MState} \times \text{MState})$.

CISL operational semantics. We define the CISL operational semantics by separating its *control flow transitions* from its state-transforming transitions. The former describe the sequential execution steps in each thread, e.g. how a loop is unrolled; while the latter describe how the underlying machine states determine the overall execution of a (concurrent) program.

The CISL control flow transitions at the top of Fig. 6 are standard and are of the form $\text{C} \xrightarrow{l} \text{C}'$, where $l \in \text{Lab} \triangleq \text{Atom} \uplus \{\text{id}\}$ denotes the *transition label*. A transition label l may be either id for silent transitions (no-ops), or $\text{a} \in \text{Atom}$ for executing the atomic command a.

We define the *state-transforming function* $[\![.]\!] : \text{Lab} \to \text{Exit} \to \mathcal{P}(\text{MState} \times \text{MState})$ as an extension of $[\![.]\!]_{\text{A}}$ as follows. Given a transition label l, we write $[\![l]\!]\epsilon$ for 1) $[\![l]\!]_{\text{A}}\epsilon$ when $l \in \text{Atom}$; 2) $\{(m, m) \mid m \in \text{MState}\}$ when l=id and ϵ=*ok*; and 3) $\emptyset$ when l=id and $\epsilon \in \text{ErExit}$. That is, atomic transitions transform the state according to their semantics as prescribed by $[\![.]\!]_{\text{A}}$ (Par. 6), while no-op transitions (denoted by id) always execute normally and leave the state unchanged.

The CISL state-transforming transitions are given at the bottom of Fig. 6 and are of the form $\text{C}, m \overset{n}{\Longrightarrow} \epsilon, m'$ with $\text{C} \in \text{Comm}$, $m, m' \in \text{MState}$, $\epsilon \in \text{Exit}$ and $n \in \mathbb{N}$, stating that starting from state m, program C terminates after n steps in state m' under exit condition ϵ . The first transition states that `skip` trivially terminates (after zero steps) successfully (with exit condition *ok*) and leaves the underlying state unchanged. The second transition states that starting from m a program C terminates erroneously (with $\epsilon \in \text{ErExit}$) after one step in m' if it takes an erroneous step. Finally, the last (inductive) transition states that if C takes one normal (non-erroneous) step transforming

$$\frac{}{\mathtt{a} \xrightarrow{\mathtt{a}} \mathtt{skip}} \qquad \frac{\mathtt{C}_1 \xrightarrow{l} \mathtt{C}_1'}{\mathtt{C}_1;\mathtt{C}_2 \xrightarrow{l} \mathtt{C}_1';\mathtt{C}_2} \qquad \frac{}{\mathtt{skip};\mathtt{C} \xrightarrow{\mathsf{id}} \mathtt{C}} \qquad \frac{i \in \{1,2\}}{\mathtt{C}_1 + \mathtt{C}_2 \xrightarrow{\mathsf{id}} \mathtt{C}_i} \qquad \frac{}{\mathtt{C}^\star \xrightarrow{\mathsf{id}} \mathtt{skip}} \qquad \frac{}{\mathtt{C}^\star \xrightarrow{\mathsf{id}} \mathtt{C};\mathtt{C}^\star}$$

$$\frac{\mathtt{C}_1 \xrightarrow{l} \mathtt{C}_1'}{\mathtt{C}_1 \,||\, \mathtt{C}_2 \xrightarrow{l} \mathtt{C}_1' \,||\, \mathtt{C}_2} \qquad \frac{\mathtt{C}_2 \xrightarrow{l} \mathtt{C}_2'}{\mathtt{C}_1 \,||\, \mathtt{C}_2 \xrightarrow{l} \mathtt{C}_1 \,||\, \mathtt{C}_2'} \qquad \frac{}{\mathtt{skip} \,||\, \mathtt{C} \xrightarrow{\mathsf{id}} \mathtt{C}} \qquad \frac{}{\mathtt{C} \,||\, \mathtt{skip} \xrightarrow{\mathsf{id}} \mathtt{C}}$$

$$\frac{}{\mathtt{skip}, m \overset{0}{\Rightarrow} \mathit{ok}, m} \qquad \frac{\epsilon \in \textsc{ErExit} \quad \mathtt{C} \xrightarrow{l} \mathtt{C}' \quad (m, m') \in [\![l]\!]\epsilon}{\mathtt{C}, m \overset{1}{\Rightarrow} \epsilon, m'} \qquad \frac{\mathtt{C} \xrightarrow{l} \mathtt{C}' \qquad (m, m'') \in [\![l]\!]\mathit{ok} \qquad \mathtt{C}', m'' \overset{n}{\Rightarrow} \epsilon, m'}{\mathtt{C}, m \overset{n+1}{\Longrightarrow} \epsilon, m'}$$

Fig. 6. The CISL control flow transitions (above); the CISL operational semantics (below)

m to m'', and the resulting program $\mathtt{C}''$ subsequently terminates after n steps with ϵ transforming m'' to m', then the overall program terminates after n+1 steps with ϵ transforming m to m'.

3.3 CISL Soundness

Erasure. In order to relate the CISL proof system to its operational semantics, it is necessary to define a relationship between the abstract states and the concrete machine states. To this end, as the abstract and machine states are both supplied as parameters to CISL, we further parametrise CISL by an *erasure* function, relating each abstract state to a set of machine states. For instance, in the case of CISL_{DC}, the erasure function simply removes the instrumentation given by permissions.

Parameter 7 (Erasure). Assume an *erasure function* $\lfloor . \rfloor : \textsc{State} \to \mathcal{P}(\textsc{MState})$.

We lift the erasure function to views (sets of states) and define $\lfloor p \rfloor \triangleq \bigcup_{s \in p} \lfloor s \rfloor$ for $p \in \textsc{View}$.

Example 3.7 (CISL_{DC} erasure). The *CISL_{DC} erasure function*, $\lfloor . \rfloor_{\text{DC}}$, is defined as follows: where:

$$\lfloor s \rfloor_{\text{DC}} = \{m\} \overset{\text{def}}{\iff} dom(m){=}dom(s) \land \forall x, v.\ m(x){=}v \implies s(x){=}(v, -)$$

Semantic Incorrectness Triples. We next present the formal interpretation of CISL triples. Recall that intuitively a CISL triple $[p]\ \mathtt{C}\ [\epsilon : q]$ states that every state in q is reachable from some state in p under ϵ. Put formally: $\models [p]\ \mathtt{C}\ [\epsilon : q] \overset{\text{def}}{\iff} \exists n \in \mathbb{N}.\ \text{reach}_n(p, \mathtt{C}, \epsilon, q)$, denoting that each state in q is reachable from some state in p in at most n steps, with:

$$\text{reach}_n(p, \mathtt{C}, \epsilon, q) \overset{\text{def}}{\iff} \forall m_q \in \lfloor q \rfloor.\ \exists m_p \in \lfloor p \rfloor, k \leq n.\ \mathtt{C}, m_p \overset{k}{\Rightarrow} \epsilon, m_q$$

Atomic Soundness. In order to show that the CISL proof system is sound, we must show that its (syntactic) triples in Fig. 5 induce valid semantics triples: if a triple $\vdash [p]\ \mathtt{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 5, then $\models [p]\ \mathtt{C}\ [\epsilon : q]$ holds. Note that we must also show this for the atomic axioms in Par. 4 as they are lifted to proof rules via Atom. Since atomic axioms are a CISL parameter, we thus require (Par. 8 below) that they 1) induce valid semantic triples; and 2) preserve all $*$-compatible views. The former ensures that if $(p, \mathtt{a}, \epsilon, q) \in \textsc{Axiom}$, then $\models [p]\ \mathtt{a}\ [\epsilon : q]$ holds; i.e. for all $m_q \in \lfloor q \rfloor$, there exists $m_p \in \lfloor p \rfloor$ such that $(m_p, m_q) \in [\![\mathtt{a}]\!]_{\text{A}}\epsilon$. The latter ensures that atomic commands of one thread preserve the states of concurrent threads in the environment and is necessary for establishing the soundness of Frame. Putting the two conditions together, we require:

Parameter 8 (Atomic soundness). Assume that for all $(p, \mathsf{a}, \epsilon, q) \in \textsc{Axiom}$ the following holds:

$$\forall s \in \textsc{State},\, m_q \in \lfloor q * \{s\} \rfloor.\ \exists m_p \in \lfloor p * \{s\} \rfloor.\ (m_p, m_q) \in [\![\mathsf{a}]\!]_{\mathsf{A}} \epsilon$$

We show the soundness of the CISL_{DC} atomic axioms in the technical appendix [Raad et al. 2022]. Finally, in the following theorem we show that the CISL proof system is *sound*, with its full proof given in the technical appendix [Raad et al. 2022].

Theorem 3.8 (Soundness). *For all $p, \mathsf{C}, \epsilon, q$, if $\vdash [p]\ \mathsf{C}\ [\epsilon : q]$ is derivable using the rules in Fig. 5, then $\models [p]\ \mathsf{C}\ [\epsilon : q]$ holds.*

3.4 Generalising the Rule of Consequence (View Shifts)

View Shifts. As shown in the literature [Dinsdale-Young et al. 2013; Jung et al. 2015], it is common to instrument abstract states with additional *ghost* states (e.g. auxiliary variables [Owicki and Gries 1976]) that do not have a counterpart in the underlying machine states. As such, when updating the ghost state, one can alter the abstract state without changing the underlying machine state. To capture such updates, following the literature we define a *view shift* relation. A view shift from abstract states p to those in q, written $p \preceq q$, describes updating each abstract state $s_p \in p$ to some abstract state $s_q \in q$ without altering the underlying machine state and while preserving all $*$-compatible states (i.e. frames). This is captured in Def. 3.9 below. We can then *generalise* the rule of consequence Cons in Fig. 5 to use view shifts in its premise rather than implications, as shown in GCons below. In the technical appendix [Raad et al. 2022] we show that GCons is sound.

$$\textsc{GCons}\ \frac{p' \preceq p \quad \vdash [p']\ \mathsf{C}\ [\epsilon : q'] \quad q \preceq q'}{\vdash [p]\ \mathsf{C}\ [\epsilon : q]}$$

Definition 3.9 (View shift). The *view shift relation*, $\preceq \subseteq \textsc{View} \times \textsc{View}$, is defined as follows:

$$p \preceq q \overset{\text{def}}{\iff} \forall s \in \textsc{State}.\ \lfloor p * \{s\} \rfloor \subseteq \lfloor q * \{s\} \rfloor$$

4 CISL_{RD}: CISL FOR RACE DETECTION

We present CISL_{RD} for detecting data-agnostic races such as that in Fig. 1f. We follow the approach of the RacerD tool [Blackshear et al. 2018] and assume all program conditions are non-deterministic. We compare CISL_{RD} with RacerD and show how it detects races that RacerD cannot.

Lazy versus Eager Race Reporting. Recall from §2 that we use CISL_{RD} to detect races by encoding erroneous (here racy) executions as normal ones. This has an additional advantage in that it allows us to uncover *all* potential races at once. More concretely, when combining the sequential histories of threads, we can either adopt a *lazy* approach where we only report the first global history that witnesses a race, or adopt an *eager* approach where we consider and find a witness for each possible race. For instance, let C denote extending the program in Fig. 1d with a third thread τ_3 executing $7.\ [x] := 3$, thus yielding the sequential history $[7]$. The lazy approach then produces a trace witnessing a single race, e.g. $[1, 2, 3, 7]$, whereas the eager approach produces a witness trace for each of the three data races, namely between lines 3 and 5 (witness $[1, 2, 4, 3, 5]$), lines 3 and 7 (witness $[1, 2, 3, 7]$), and lines 5 and 7 (witness $[4, 5, 7]$).

Simplifying Assumption: Races. As CISL (and by extension CISL_{RD}) is an under-approximate analysis, it is sound to aim for a subset of races. As such, for simplicity, rather than aiming to detect all races, we initially focus on a specific class of races. Specifically, we target races between conflicting accesses e and e' where the set of locks held by at least one access is empty. This way, given two sequential histories of the form $H = H_0 +\!\!+ e +\!\!+ -$ and $H' = H'_0 +\!\!+ e' +\!\!+ -$, when the set of locks

$$H \in \textsc{Hist} \triangleq \left\{E \in \textsc{Seq}\langle\textsc{Event}\rangle \;\middle|\; \mathsf{wf}(E)\right\} \qquad e \in \textsc{Event} \triangleq \left\{\begin{array}{l|l}\mathsf{R}(\mathrm{L},\tau,x), \mathsf{W}(\mathrm{L},\tau,x), & \mathrm{L} \in \textsc{Label} \wedge \tau \in \textsc{Tid} \\ \mathsf{L}(\tau,x), \mathsf{U}(\tau,x) & \wedge\ x \in \textsc{Loc}\end{array}\right\}$$

$$s \in \textsc{State}_{\mathrm{RD}} \triangleq \left\{f \in \textsc{Tid} \xrightarrow{\text{fin}} \textsc{Hist} \;\middle|\; \forall \tau, H, e.\ f(\tau){=}H \wedge e \in H \Rightarrow \mathsf{tid}(e){=}\tau\right\} \qquad m \in \textsc{MState}_{\mathrm{RD}} \triangleq \textsc{Hist}$$

$$\begin{array}{rl}\mathsf{wf}(H) \overset{\text{def}}{\iff} & \forall x, \tau, H'.\ H = H' +\!\!+ \mathsf{U}(\tau,x) +\!\!+ - \implies x \in \mathsf{alocks}(H',\tau) \\ & \wedge\ \forall x, \tau, H'.\ H = H' +\!\!+ \mathsf{L}(\tau,x) +\!\!+ - \implies \forall \tau'.\ x \notin \mathsf{alocks}(H',\tau')\end{array}$$

$$\mathsf{alocks}([],\tau) \triangleq \emptyset \qquad \mathsf{alocks}(e +\!\!+ H,\tau) \triangleq \begin{cases}\{x\} \cup \mathsf{alocks}(H,\tau) & \text{if } e = \mathsf{L}(\tau,x) \wedge \mathsf{U}(\tau,x) \notin H \\ \mathsf{alocks}(H,\tau) & \text{otherwise}\end{cases}$$

Fig. 7. The CISL_{RD} model domain

held at e is empty, then we can always construct the well-formed *witness* history $H_0 +\!\!+ H_0' +\!\!+ e +\!\!+ e'$. For instance, when e and e' respectively denote the accesses on lines 3 and 5 of Fig. 1d, then the set of locks held by τ_1 at e is empty. As such, given the sequential histories $H = [1,2,3]$ and $H' = [4,5,6]$, we can construct the history $[1,2,4,3,5]$ witnessing the race, i.e. $H_0 = [1,2]$ and $H_0' = [4]$. As we describe shortly in §4.1, this is because when the set of locks held at e is empty (i.e. H_0 contains no active locks), then the history $H_0 +\!\!+ H_0' +\!\!+ e +\!\!+ e'$ is always well-formed.

Caveat and Relation to RacerD. The subset of racy executions described above coincide precisely with those detected by RacerD [Blackshear et al. 2018]. Nevertheless, both our simple approach described above and RacerD fail to detect races where e.g. two threads with conflicting accesses hold a disjoint set of locks, as shown in Race1. Specifically, as the two threads acquire two distinct locks (l and l'), Race1 induces the racy interleaving $H = [1,4,2,5]$ in which the conflicting accesses on x (on lines 2 and 5) are adjacent. However, neither RacerD nor our simple CISL_{RD} instantiation detect this race. Nevertheless, in §4.2 we show how we can generalise and strengthen CISL_{RD} to detect such races.

1. $\mathtt{lock}_\tau\ l$; ‖ 4. $\mathtt{lock}_{\tau'}\ l'$;
2. L: $[x] :=_\tau 1$; ‖ 5. L': $[x] :=_{\tau'} 2$;
3. $\mathtt{unlock}_\tau\ l$; ‖ 6. $\mathtt{unlock}_{\tau'}\ l'$;

(Race1)

4.1 CISL_{RD} Formalism

CISL_{RD} Atomic Commands (Par. 1). We assume a set of shared memory locations ranged over by x, y, l, and a set of local variables (registers) ranged over by a, b, c. We instantiate CISL_{RD} with a simple set of atomic commands comprising reads and writes (accesses) on shared memory locations, and locking constructs used as a synchronisation mechanism to avoid races:

$$\textsc{Atom}_{\text{RD}} \ni \mathsf{a} ::= \mathrm{L}\colon a :=_\tau [x] \mid \mathrm{L}\colon [x] :=_\tau a \mid \mathtt{lock}_\tau\ l \mid \mathtt{unlock}_\tau\ l$$

To identify races, we annotate instructions with a thread id τ. Moreover, to locate races we further annotate memory accesses with a label L. As such, we assume that a CISL_{RD} program C is *well-formed* in that (1) the labels across C are unique; (2) instructions of the same thread are associated with the same thread id; and (3) concurrent instructions in C have distinct thread ids.

Note that $\textsc{Atom}_{\text{RD}}$ does not include the `assume(.)` construct, and thus since the CISL programming language (Def. 3.1) only supports non-deterministic choice ($\mathsf{C}_1 + \mathsf{C}_2$) and loops ($\mathsf{C}^\star$), we rule out deterministic conditionals and loops from CISL_{RD}.[1]

CISL_{RD} State PCM (Par. 2). The CISL_{RD} states, $\textsc{State}_{\text{RD}}$, are as defined in Fig. 7, where a state s is a map from thread identifiers to *well-formed histories*. A history records a sequence of *events* executed by the thread thus far, where an event may be either 1) $\mathsf{R}(\mathrm{L},\tau,x)$ denoting a read access on x by τ at L; 2) $\mathsf{W}(\mathrm{L},\tau,x)$ denoting a write access on x by τ at L; 3) $\mathsf{L}(\tau,l)$ denoting a lock acquisition

[1] Recall that `if (b) then` C_1 `else` C_2 can be encoded as `(assume(b);` C_1`) + (assume(¬b);` C_2`)`, and `while (b)` C can be encoded as `(assume(b);` C`)`*`;assume(¬b)`.

on l by τ; or 4) $\mathsf{U}(\tau, l)$ denoting a lock release on l by τ. The functions tid and lab respectively return the thread and label of an event, where applicable; e.g. $\mathsf{tid}(e)=\tau$ and $\mathsf{lab}(e)=\mathtt{L}$ when $e=\mathsf{R}(\mathtt{L},\tau,x)$.

A history is well-formed, written $\mathsf{wf}(H)$, iff it respects the lock semantics. Specifically, if H contains a lock release on x by τ, then it must contain an earlier lock acquisition on x by τ (first conjunct); i.e. a lock can only be released by the thread that acquired it last. Moreover, if H contains a lock acquisition on x by τ, then no thread must have previously acquired x without releasing it (second conjunct). Note that given a history H and a thread τ, the $\mathsf{alocks}(H,\tau)$ denotes the *active locks* of τ in H, namely those locks that have been acquired by τ in H but not released.

The CISL_{RD} PCM is $(\textsc{State}_{\text{RD}}, \uplus, \emptyset)$, where $\uplus$ denotes disjoint function union and $\emptyset$ denotes a map with an empty domain. We write $\tau \mapsto H$ for the set $p=\{[\tau \mapsto H]\}$. We define the *conflict relation*, $\bowtie$, as follows, where R^{s} denotes the symmetric closure of R:

$$\bowtie \triangleq \left((\textsc{Event} \times \textsc{Event}) \cap \left\{(\mathsf{R}(\mathtt{L},\tau,x), \mathsf{W}(\mathtt{L}',\tau',x)), (\mathsf{W}(\mathtt{L},\tau,x), \mathsf{W}(\mathtt{L}',\tau',x)) \;\middle|\; \tau \neq \tau'\right\}\right)^{\mathsf{s}}$$

CISL_{RD} Error Conditions (Par. 3). As discussed above, we do not treat data races as errors and thus define the set of erroneous exit conditions for CISL_{RD} as the empty set: $\textsc{ErExit}_{\text{RD}} \triangleq \emptyset$

The Race Assertion. Recall from §4 that we aim to detect races between conflicting accesses e_1 and e_2 where the set of locks held by at least one access is empty. This is captured by $\mathsf{race}(\mathtt{L}_1, \mathtt{L}_2, H)$ below, denoting a data race between $\mathtt{L}_1$ and $\mathtt{L}_2$ with the *witness history* (trace) H:

$$\begin{aligned}\mathsf{race}(\mathtt{L}_1,\mathtt{L}_2,H) \overset{\text{def}}{\iff} \exists \tau_1,\tau_2,e_1,e_2,H_1,H_2.\ & \tau_1 \mapsto H_1 \between e_1 \between - * \tau_2 \mapsto H_2 \between e_2 \between - * e_1 \bowtie e_2 \\ & * \mathsf{lab}(e_1){=}\mathtt{L}_1 * \mathsf{lab}(e_2){=}\mathtt{L}_2 * \mathsf{alocks}(H_1,\tau_1) = \emptyset * H{=}H_1 \between H_2 \between [e_1,e_2]\end{aligned}$$

Specifically, if the history of τ_1 is of the form $H_1 \between e_1 \between -$ with e_1 at $\mathtt{L}_1$, the history of τ_2 is $H_2 \between e_2 \between -$ with e_2 at $\mathtt{L}_2$, the accesses e_1 and e_2 are conflicting, and (without loss of generality) the set of locks held by τ_1 prior to e_1 is empty, then there is a race between e_1 and e_2 which can be witnessed by $H_1 \between H_2 \between [e_1,e_2]$. For instance, let e_i denote the event associated with the instruction on line i in Fig. 1d; i.e. e_3 and e_5 denote the conflicting accesses on lines 3 and 5, respectively. Then $\mathsf{race}(3,5,[e_1,e_2,e_4,e_3,e_5])$ holds, capturing the race between e_3 and e_5.

Note that requiring that τ_1 hold an empty lock set upon the e_1 access simplifies the task of constructing a witness history. Specifically, as the set of locks held by τ_1 at the end of H_1 is empty, the combined history $H_1 \between H_2 \between [e_1,e_2]$ is always well-formed (see Fig. 7). We shortly demonstrate how race can be used for identifying races in an execution.

CISL_{RD} Axioms (Par. 4). The CISL_{RD} axioms, $\textsc{Axiom}_{\text{RD}}$, are defined below, where assignments require no resource (they do not extend the history as their behaviour is local), while each non-local instruction of τ simply extends its history thus far with an associated event. For instance, when the current state is $\tau \mapsto H$, i.e. the current history of τ is given by H, then executing $\mathtt{lock}_\tau\ l$ updates the state to $\tau \mapsto H \between \mathsf{L}(\tau,l)$. Note that when τ already holds the lock on l (i.e. $l \in \mathsf{alocks}(H,\tau)$), then the resulting history $H \between \mathsf{L}(\tau,l)$ is not well-formed, and thus $\tau \mapsto H \between \mathsf{L}(\tau,l)$ describes an empty set, rendering the triple vacuously true. For better readability, we present the axioms as inference rules with CISL triples of the form $[p]\ \mathtt{a}\ [\epsilon : q]$, rather than tuples of the form $(p, \mathtt{a}, \epsilon, q)$.

RD-Read
$$[\tau \mapsto H]\ \mathtt{L}{:}\ a :=_\tau [x]\ [ok{:}\ \tau \mapsto H \between \mathsf{R}(\mathtt{L},\tau,x)]$$

RD-Write
$$[\tau \mapsto H]\ \mathtt{L}{:}\ [x] :=_\tau a\ [ok{:}\ \tau \mapsto H \between \mathsf{W}(\mathtt{L},\tau,x)]$$

RD-Assign
$$[\mathsf{emp}]\ x := \mathsf{e}\ [ok{:}\ \mathsf{emp}]$$

RD-Lock
$$[\tau \mapsto H]\ \mathtt{lock}_\tau\ l\ [ok{:}\ \tau \mapsto H \between \mathsf{L}(\tau,l)]$$

RD-Unlock
$$[\tau \mapsto H]\ \mathtt{unlock}_\tau\ l\ [ok{:}\ \tau \mapsto H \between \mathsf{U}(\tau,l)]$$

$$[\tau_1 \mapsto [] * \tau_2 \mapsto []]$$

$[\tau_1 \mapsto []]$	$[\tau_2 \mapsto []]$
$\texttt{lock}_{\tau_1}\ l$; // RD-Lock	$\texttt{lock}_{\tau_2}\ l$; // RD-Lock
$[ok\colon \tau_1 \mapsto [\mathsf{L}(\tau_1, l)]]$	$[ok\colon \tau_2 \mapsto [\mathsf{L}(\tau_2, l)]]$
$\mathrm{L}_x\colon [x] :=_{\tau_1} 1$; // RD-Write	$\mathrm{L}'_y\colon a :=_{\tau_2} [y]$; // RD-Read
$[ok\colon \tau_1 \mapsto [\mathsf{L}(\tau_1, l), \mathsf{W}(\mathrm{L}_x, \tau_1, x)]]$	$[ok\colon \tau_2 \mapsto [\mathsf{L}(\tau_2, l), \mathsf{R}(\mathrm{L}'_y, \tau_2, y)]]$
$\mathrm{L}_y\colon [y] :=_{\tau_1} 1$; // RD-Write	$\texttt{unlock}_{\tau_1}\ l$; // RD-Unlock
$[ok\colon \tau_1 \mapsto [\mathsf{L}(\tau_1, l), \mathsf{W}(\mathrm{L}_x, \tau_1, x), \mathsf{W}(\mathrm{L}_y, \tau_1, y)]]$	$[ok\colon \tau_2 \mapsto [\mathsf{L}(\tau_2, l), \mathsf{R}(\mathrm{L}'_y, \tau_2, y), \mathsf{U}(\tau_2, l)]]$
$\texttt{unlock}_{\tau_1}\ l$; // RD-Unlock	$(\mathrm{L}'_x\colon [x] :=_{\tau_2} 2 + \texttt{skip})$ // Choice, RD-Write
$[ok\colon \tau_1 \mapsto [\mathsf{L}(\tau_1, l), \mathsf{W}(\mathrm{L}_x, \tau_1, x), \mathsf{W}(\mathrm{L}_y, \tau_1, y), \mathsf{U}(\tau_1, l)]]$	$[ok\colon \tau_2 \mapsto [\mathsf{L}(\tau_2, l), \mathsf{R}(\mathrm{L}'_y, \tau_2, y), \mathsf{U}(\tau_2, l), \mathsf{W}(\mathrm{L}'_x, \tau_2, x)]]$

// Par

$$[ok\colon \tau_1 \mapsto [\mathsf{L}(\tau_1, l), \mathsf{W}(\mathrm{L}_x, \tau_1, x), \mathsf{W}(\mathrm{L}_y, \tau_1, y), \mathsf{U}(\tau_1, l)] * \tau_2 \mapsto [\mathsf{L}(\tau_2, l), \mathsf{R}(\mathrm{L}'_y, \tau_2, y), \mathsf{U}(\tau_2, l), \mathsf{W}(\mathrm{L}'_x, \tau_2, x)]]$$

// Cons

$$\left[ok\colon \begin{array}{l} \tau_1 \mapsto [\mathsf{L}(\tau_1, l), \mathsf{W}(\mathrm{L}_x, \tau_1, x), \mathsf{W}(\mathrm{L}_y, \tau_1, y), \mathsf{U}(\tau_1, l)] * \tau_2 \mapsto [\mathsf{L}(\tau_2, l), \mathsf{R}(\mathrm{L}'_y, \tau_2, y), \mathsf{U}(\tau_2, l), \mathsf{W}(\mathrm{L}'_x, \tau_2, x)] \\ \wedge\ \mathsf{race}\Big(\mathrm{L}_x, \mathrm{L}'_x, [\mathsf{L}(\tau_2, l), \mathsf{R}(\mathrm{L}'_y, \tau_2, y), \mathsf{U}(\tau_2, l), \mathsf{L}(\tau_1, l), \mathsf{W}(\mathrm{L}_x, \tau_1, x), \mathsf{W}(\mathrm{L}'_x, \tau_2, x)]\Big) \end{array}\right]$$

Fig. 8. A proof sketch of the race in Fig. 1f (see Example 4.2)

Example 4.1. Let $e \triangleq \mathsf{W}(\mathrm{L}, \tau, x)$ and $e' \triangleq \mathsf{W}(\mathrm{L}', \tau', x)$. We then have:

$$\dfrac{\dfrac{}{[\tau \mapsto []]\ \mathrm{L}\colon [x] :=_{\tau} 1\ [ok\colon \tau \mapsto [e]]}\ \text{RD-Write} \quad \dfrac{}{[\tau' \mapsto []]\ \mathrm{L}'\colon [x] :=_{\tau'} 2\ [ok\colon \tau' \mapsto [e']]}\ \text{RD-Write}}{[\tau \mapsto [] * \tau' \mapsto []]\ \mathrm{L}\colon [x] :=_{\tau} 1 \,||\, \mathrm{L}'\colon [x] :=_{\tau'} 2\ [ok\colon \tau \mapsto [e] * \tau' \mapsto [e']]}\ \text{Par}$$
$$[\tau \mapsto [] * \tau' \mapsto []]\ \mathrm{L}\colon [x] :=_{\tau} 1 \,||\, \mathrm{L}'\colon [x] :=_{\tau'} 2\ [ok\colon (\tau \mapsto [e] * \tau' \mapsto [e']) \wedge \mathsf{race}(\mathrm{L}, \mathrm{L}', [e, e'])]\ \text{Cons}$$

At the last step of the derivation above, we apply the Cons rule of CISL to obtain $\mathsf{race}(\mathrm{L}, \mathrm{L}', [e, e'])$. Specifically, note that $(\tau \mapsto [e] * \tau' \mapsto [e']) \wedge \mathsf{race}(\mathrm{L}, \mathrm{L}', [e, e']) \Leftrightarrow \tau \mapsto [e] * \tau' \mapsto [e']$, and we are only required to show that $(\tau \mapsto [e] * \tau' \mapsto [e']) \wedge \mathsf{race}(\mathrm{L}, \mathrm{L}', [e, e']) \Rightarrow \tau \mapsto [e] * \tau' \mapsto [e']$.

Duplicate Races. Observe that using the same reasoning steps above, we can also derive $\mathsf{race}(\mathrm{L}', \mathrm{L}, [e', e])$, identifying the same race between L and L′ with a different witness history. However, duplicate race reporting can be avoided by simply inspecting the labels of racing instructions. For instance, once we report the race between L and L′ via $\mathsf{race}(\mathrm{L}, \mathrm{L}', [e, e'])$, we can suppress all other witnesses of the form $\mathsf{race}(\mathrm{L}, \mathrm{L}', -)$ or $\mathsf{race}(\mathrm{L}', \mathrm{L}, -)$.

Example 4.2 (Fig. 1f). We present a proof outline of the race in Fig. 1f in Fig. 8, where we have annotated instructions with their thread identifiers and labels (where applicable). Note that we have encoded $\texttt{if}\ (*)\ \mathrm{L}'_x\colon [x] :=_{\tau_2} 1$ in the right thread using the non-deterministic choice (+) of CISL as $\mathrm{L}'_x\colon [x] :=_{\tau_2} 2 + \texttt{skip}$. For brevity, rather than giving the full derivation, we follow the classical Hoare logic proof outline, annotating each line of the code with its pre- and post-condition. We further commentate each proof step and write e.g. // RD-Lock to denote an application of RD-Lock.

$\mathbf{CISL_{RD}}$ Machine States (Par. 5) and Erasure (Par. 7). While a $\mathrm{CISL_{RD}}$ state $s \in \textsc{State}_{\mathrm{RD}}$ records the local (sequential) history associated with each thread, a *$CISL_{RD}$ machine state*, $m \in \textsc{MState}_{\mathrm{RD}}$ in Fig. 7, records the global execution history. A global history is obtained (through the $\mathrm{CISL_{RD}}$ erasure function) by combining all local thread histories into a well-formed history. That is, the *$CISL_{RD}$ erasure function* is as defined below, where $H|_\tau$ denotes restricting H to the events of τ: $\lfloor s \rfloor_{\mathrm{RD}} \triangleq \left\{ H \,\middle|\, \forall \tau.\ s(\tau) = H' \Rightarrow H|_\tau = H' \right\}$.

$\mathbf{CISL_{RD}}$ Atomic Semantics (Par. 6). The $\mathrm{CISL_{RD}}$ atomic semantics is defined below, where each instruction of τ extends the current history (machine state) H_g by inserting an associated event

e in H_g such that e is the last event of τ in the extended history, and the extended history is well-formed (i.e. is in Hist). That is, whenever H_g can be split as $H_\tau +\!\!+ H$ such that H does not contain any events associated with τ, then executing an instruction associated with e may update H_g to $H_\tau +\!\!+ e +\!\!+ H$. Note that such splitting of H_g may not be unique; e.g. when $H_g = [e_1 \cdots e_n]$ and none of $e_1 \cdots e_n$ are associated with τ, then $\mathsf{R}(\iota, \tau, x)$ may be inserted anywhere within H_g. Intuitively, this captures the different ways executing the instructions of one thread may be interleaved with those of others.

$$
\begin{array}{rl}
[\![\iota\colon a :=_\tau [x]]\!]_{\mathsf{A}}\, ok \triangleq \left\{(H_\tau +\!\!+ H, H_\tau +\!\!+ e +\!\!+ H) \in \textsc{Hist}^2 \;\middle|\; e{=}\mathsf{R}(\iota, \tau, x) \wedge \forall e' \in H.\ e'.\mathsf{tid} \neq \tau\right\} \\
[\![\iota\colon [x] :=_\tau a]\!]_{\mathsf{A}}\, ok \triangleq \left\{(H_\tau +\!\!+ H, H_\tau +\!\!+ e +\!\!+ H) \in \textsc{Hist}^2 \;\middle|\; e{=}\mathsf{W}(\iota, \tau, x) \wedge \forall e' \in H.\ e'.\mathsf{tid} \neq \tau\right\} \\
[\![\mathtt{lock}_\tau\ l]\!]_{\mathsf{A}}\, ok \triangleq \left\{(H_\tau +\!\!+ H, H_\tau +\!\!+ e +\!\!+ H) \in \textsc{Hist}^2 \;\middle|\; e{=}\mathsf{L}(\tau, l) \wedge \forall e' \in H.\ e'.\mathsf{tid} \neq \tau\right\} \\
[\![\mathtt{unlock}_\tau\ l]\!]_{\mathsf{A}}\, ok \triangleq \left\{(H_\tau +\!\!+ H, H_\tau +\!\!+ e +\!\!+ H) \in \textsc{Hist}^2 \;\middle|\; e{=}\mathsf{U}(\tau, l) \wedge \forall e' \in H.\ e'.\mathsf{tid} \neq \tau\right\}
\end{array}
$$

Note that the atomic semantics of $\mathtt{lock}_\tau\ l$ returns an empty set when the lock l is already held by τ, thanks to the well-formedness condition on histories (see Fig. 7). Specifically, when the current machine state is $H_1 = H_\tau +\!\!+ H$ (where H contains no actions by τ) and τ then attempts to acquire the lock on l, then the resulting history $H_2 = H_\tau +\!\!+ \mathsf{L}(\tau, l) +\!\!+ H$ is well-formed and thus defined only if no other thread already holds the lock on l in H_τ. That is, H_2 is not well-formed ($\mathsf{wf}(H_2)$ does not hold) according to the definition in Fig. 7 when $l \in \mathsf{alocks}(H_\tau, \tau')$ for some τ'. Consequently, when some thread τ' already holds the lock on l, H_2 is not defined, rendering the $\{(H_1, H_2)\}$ set empty. Similarly, thanks to the well-formedness condition on histories, the atomic semantics of $\mathtt{unlock}_\tau\ l$ returns an empty set when the lock l is not already held by τ.

CISL_{RD} Atomic Soundness (Par. 8). Finally, in the technical appendix [Raad et al. 2022] we demonstrate that the CISL_{RD} atomic instructions are sound.

4.2 Generalising CISL_{RD}

Recall that the CISL_{RD} formalism in §4.1 (as with RacerD) cannot detect races such as that in Race1. This is because the definition of the race predicate simply requires that the set of locks held at the time of one of the conflicting accesses be empty, and thus fails to detect the race in Race1 where the conflicting access occur while their threads hold disjoint (but non-empty) sets of locks.

We next generalise the race predicate to account for races such as Race1. Note that we cannot naively update the race predicate to require that the set of locks held at the time of conflicting accesses be *disjoint* as this leads to *false positives*. To see this, consider the examples below:

1. $\mathtt{lock}_\tau\ l$;	6. $\mathtt{lock}_{\tau'}\ l'$;		1. $\mathtt{lock}_\tau\ l'$;	6. $\mathtt{lock}_{\tau'}\ l$;	
2. $\quad\mathtt{lock}_\tau\ l'$;	7. $\quad\mathtt{lock}_{\tau'}\ l$;		2. $\mathtt{lock}_\tau\ l$;	7. $\mathtt{lock}_{\tau'}\ l'$;	
3. $\quad\mathtt{unlock}_\tau\ l'$;	8. $\quad\mathtt{unlock}_{\tau'}\ l$;	(NoRace1)	3. $\mathtt{unlock}_\tau\ l'$;	8. $\mathtt{unlock}_{\tau'}\ l$;	(NoRace2)
4. $\quad\iota\colon [x] :=_\tau 1$;	9. $\quad\iota'\colon [x] :=_{\tau'} 2$;		4. $\quad\iota\colon [x] :=_\tau 1$;	9. $\quad\iota'\colon [x] :=_{\tau'} 2$;	
5. $\mathtt{unlock}_\tau\ l$;	10. $\mathtt{unlock}_{\tau'}\ l'$;		5. $\mathtt{unlock}_\tau\ l$;	10. $\mathtt{unlock}_{\tau'}\ l'$;	

In the case of NoRace1, although at the time of the conflicting accesses on x (at ι and ι') the corresponding threads (resp. τ and τ') hold disjoint sets of locks (resp. $\{l\}$ and $\{l'\}$), the two accesses do not race thanks to the synchronisation induced by the additional locks acquired in each thread (e.g. l' in the left thread). That is, even though these additional locks are released before the conflicting accesses and thus the locks held at the time of the conflicting accesses are disjoint, their mere acquisition induces additional synchronisation ('happens-before' ordering) that renders the program non-racy.[2] As such, reporting a race between ι and ι' would constitute a false positive.

[2]We encourage the reader to attempt to construct a history to witness a race in NoRace1 and note that this cannot be done.

Similarly, the synchronisation induced by the additional locks in NoRace2 (e.g. l' in the left thread) renders the programs non-racy. Note that NoRace2 is obtained from NoRace1 by swapping the order in which the two locks are acquired in each thread. That is, while the lock acquisitions in NoRace1 are *balanced* (well-nested), those in NoRace2 are not. Lock acquisition is balanced when acquired locks are released in the reverse acquisition order. Balanced locks are tantamount to `synchronized` blocks in Java, in that `lock` l; C; `unlock` l is commensurate with `synchronized`$(l)\{\mathsf{C}\}$. For simplicity, here we assume that the lock acquisitions are balanced as in RacerD (which allows lock acquisition only through (balanced) `synchronized` blocks). Nonetheless, it is straightforward to lift this assumption and generalise our formalism of CISL_{RD} yet again.

Generalising the race **Assertion.** We present our general race assertion in Fig. 9 and describe it shortly. Given history H and thread τ, a lock acquired by τ in H is *active* if it is *not released* subsequently by τ in H, i.e. is in $\mathsf{alocks}(H,\tau)$. Conversely, a lock acquired by τ in H is *passive* if it is *released* subsequently by τ in H, i.e. is in $\mathsf{syncs}(H,\tau)$ defined Fig. 9.

We compute the set of passive locks to account for the additional synchronisation induced by them as in NoRace1. More concretely, to avoid false positives such as that in NoRace1 while identifying races such as that in Race1, given τ and τ' and their respective histories $H{+\!\!+}e$ and $H'{+\!\!+}e'$ such that e and e' are conflicting, we identify a race between e and e' if the (active) locks held by τ prior to e are disjoint from *all* (both active and passive) locks of τ' or vice versa: $\mathsf{alocks}(H,\tau) \cap \mathsf{locks}(H',\tau')=\emptyset$ or $\mathsf{alocks}(H',\tau') \cap \mathsf{locks}(H,\tau)=\emptyset$, where $\mathsf{locks}(H,\tau) \triangleq \mathsf{alocks}(H,\tau) \cup \mathsf{syncs}(H,\tau)$. For instance, let $H=[1,2,3]$ and $H'=[6,7,8]$ respectively denote (partial) histories of τ and τ' in NoRace1, with n denoting the event associated with the instruction at line n. We then have $\mathsf{alocks}(H,\tau)=\{l\}$, $\mathsf{alocks}(H',\tau')=\{l'\}$, $\mathsf{syncs}(H,\tau)=\{l'\}$, $\mathsf{syncs}(H',\tau')=\{l\}$ and $\mathsf{locks}(H,\tau)=\mathsf{locks}(H',\tau')=\{l,l'\}$; thus $\mathsf{alocks}(H,\tau) \cap \mathsf{locks}(H',\tau')=\{l\}\neq\emptyset$ and $\mathsf{alocks}(H',\tau') \cap \mathsf{locks}(H,\tau)=\{l'\}\neq\emptyset$.

Finally, note that when computing the passive locks prior to an access e, we must only consider those passive locks that have been acquired *after* an active lock. To see this, consider a variant of NoRace1 in Race3 where the inner passive locks are promoted to the beginning of each thread. Observe that the accesses at $\mathtt{L}$ and $\mathtt{L}'$ race as witnessed by $H=[1,2,6,7,3,8,4,9]$. That is, unlike in NoRace1, the additional locks acquired by each thread at the beginning do not induce synchronisation when accessing x. Intuitively, this is because unlike in NoRace1, the passive locks are no longer preceded by an active lock, and thus when constructing a witness history as in H, they can always be acquired and released before the active locks.

1. $\mathtt{lock}_\tau\ l'$;	6. $\mathtt{lock}_{\tau'}\ l$;
2. $\mathtt{unlock}_\tau\ l'$;	7. $\mathtt{unlock}_{\tau'}\ l$;
3. $\mathtt{lock}_\tau\ l$;	8. $\mathtt{lock}_{\tau'}\ l'$;
4. $\mathtt{L}\colon [x] :=_\tau 1$;	9. $\mathtt{L}'\colon [x] :=_{\tau'} 2$;
5. $\mathtt{unlock}_\tau\ l$;	10. $\mathtt{unlock}_{\tau'}\ l'$;

(Race3)

To this end, given threads τ_1 and τ_2 and their respective histories $H_1 + e_1$ and $H_2 + e_2$ such that e_1 and e_2 are conflicting, we identify a race between e_1 and e_2 if 1) the active locks held by τ_1 prior to e_1 are disjoint from all locks of τ_2 or vice versa (as explained above); and 2) thread histories can be partitioned as $H_1{=}H_p + H_l$ and $H_2{=}H'_p + H'_l$, where H_p (resp. H'_p) is the maximal prefix of H_1 (resp. H_2) such that $\mathsf{alocks}(H_p,\tau_1)=\emptyset$ (resp. $\mathsf{alocks}(H'_p,\tau_2)=\emptyset$). We then construct a global history witnessing the race as $H_p + H'_p + H_l + H'_l + [e_1,e_2]$, as shown in Fig. 9.

Note that the witness history $H_p + H'_p + H_l + H'_l + [e_1,e_2]$ is always well-formed: 1) $\mathsf{alocks}(H_p,\tau_1) = \mathsf{alocks}(H'_p,\tau_2)=\emptyset$ (see partition) ensures that $H_p + H'_p$ is well-formed and that there are no locks held at the end of $H_p + H'_p$; and 2) $\mathsf{alocks}(H_l,\tau_1) \cap \mathsf{locks}(H'_l,\tau_2)=\emptyset$, and thus the (active or passive) locks acquired by τ_2 in H'_l are disjoint from those that are held by τ_1 at the end of H_l.

Example 4.3 (Race3). Let C and C′ denote the sequential programs of τ and τ' in Race3 above, respectively. Let $H_p \triangleq [\mathsf{L}(\tau,l'),\mathsf{U}(\tau,l')]$, $H'_p \triangleq [\mathsf{L}(\tau',l),\mathsf{U}(\tau',l)]$, $H_l \triangleq [\mathsf{L}(\tau,l)]$, $H'_l \triangleq [\mathsf{L}(\tau',l')]$, $w \triangleq \mathsf{W}(\mathtt{L},\tau,x)$, $w' \triangleq \mathsf{W}(\tau',\mathtt{L}',x)$, $H \triangleq H_p + H_l + [w]$ and $H' \triangleq H'_p + H'_l + [e']$. We then have:

$$\mathsf{race}(\mathtt{L}_1,\mathtt{L}_2,H) \overset{\text{def}}{\iff} \exists \tau_1,\tau_2,H_1,H_2,e_1,e_2,H_p,H_p',H_l,H_l'.\ \tau_1\mapsto H_1 \mathbin{+\!\!+} e_1 \mathbin{+\!\!+} - * \tau_2\mapsto H_2 \mathbin{+\!\!+} e_2 \mathbin{+\!\!+} - * e_1 \bowtie e_2 * \mathsf{lab}(e_1){=}\mathtt{L}_1 * \mathsf{lab}(e_2){=}\mathtt{L}_2 * \mathsf{partition}(H_1,\tau_1,H_p,H_l) * \mathsf{partition}(H_2,\tau_2,H_p',H_l') * H{=}H_p \mathbin{+\!\!+} H_p' \mathbin{+\!\!+} H_l \mathbin{+\!\!+} H_l' \mathbin{+\!\!+} [e_1,e_2] * \mathsf{alocks}(H_l,\tau_1)\cap\mathsf{locks}(H_l',\tau_2){=}\emptyset$$

$$\mathsf{partition}(H,\tau,H_1,H_2) \overset{\text{def}}{\iff} H{=}H_1 \mathbin{+\!\!+} H_2 \wedge \mathsf{alocks}(H_1,\tau){=}\emptyset \wedge \forall H_1'.\ H{=}H_1' \mathbin{+\!\!+} - \wedge \mathsf{alocks}(H_1',\tau){=}\emptyset \Rightarrow H_1' \subseteq H_1$$

$$\mathsf{locks}(H,\tau) \triangleq \mathsf{alocks}(H,\tau) \cup \mathsf{syncs}(H,\tau)$$

$$\mathsf{syncs}([],\tau) \triangleq \emptyset \qquad \mathsf{syncs}(e \mathbin{+\!\!+} H,\tau) \triangleq \begin{cases} x \cup \mathsf{syncs}(H,\tau) & \text{if } e = \mathsf{L}(\tau,x) \wedge \mathsf{U}(\tau,x) \in H \\ \mathsf{syncs}(H,\tau) & \text{otherwise} \end{cases}$$

Fig. 9. The general race assertion; the highlighted text denotes the extensions generalising race on p. 14

$$\dfrac{\dfrac{(\text{derived via Seq, Atom and CISL}_{\text{RD}}\text{ axioms on p. 14})}{[\tau\mapsto[]]\ \mathtt{C}_1\ [ok\colon \tau\mapsto H]} \quad \dfrac{(\text{derived via Seq, Atom and CISL}_{\text{RD}}\text{ axioms on p. 14})}{[\tau'\mapsto[]]\ \mathtt{C}_2\ [ok\colon \tau'\mapsto H']}}{[\tau\mapsto[] * \tau'\mapsto[]]\ \mathtt{C}\ ||\ \mathtt{C}'\ [ok\colon \tau\mapsto H * \tau'\mapsto H']}\ \textsc{Par}}{[\tau\mapsto[] * \tau'\mapsto[]]\ \mathtt{C}\ ||\ \mathtt{C}'\ \left[ok\colon (\tau\mapsto H * \tau'\mapsto H') \wedge \mathsf{race}(\mathtt{L},\mathtt{L}',H_p \mathbin{+\!\!+} H_p' \mathbin{+\!\!+} H_l \mathbin{+\!\!+} H_l' \mathbin{+\!\!+} [w,w'])\right]}\ \textsc{Cons}$$

5 CISL$_{\text{DD}}$: CISL FOR DEADLOCK DETECTION

As with races, deadlocks are *global* errors (due to two or more threads). We present CISL$_{\text{DD}}$ for detecting data-agnostic deadlocks. To this end, we follow the approach of Brotherston et al. [2021] and assume that all program conditions are *non-deterministic*. We compare CISL$_{\text{DD}}$ to the work of [Brotherston et al. 2021] and show how the CISL$_{\text{DD}}$ deadlock assertion is more expressive in that it can produce a history *witnessing* the deadlock. As in CISL$_{\text{RD}}$, we do not treat deadlocks as errors; instead, we compute a sequential history of each thread in isolation, combine them using Par, and examine them to detect deadlocks and construct a concurrent history witnessing the deadlock.

Deadlocks and Critical Pairs. As with races, deadlocks may only manifest under certain interleavings. To see this, consider the example shown in DL, where $\mathtt{C}_{\text{ll}}$, $\mathtt{C}'_{\text{ll}}$ denote arbitrary code without lock/unlock instructions. The program in DL induces several (abbreviated) histories, including $H{=}[1,5]$ and $H'{=}[1,2,3,4,5,6,7,8]$. However, the deadlock between lines 1, 2 and lines 5, 6 only manifest in H and not H'. That is, after τ and τ' respectively execute lines 1 and 5, then neither thread can proceed further as they each need to acquire a lock already held by the other. Specifically, note that as part of the history we do not record the instructions associated with $\mathtt{C}_{\text{ll}}/\mathtt{C}'_{\text{ll}}$: as we discuss below, the only instructions relevant to deadlocks are locks and unlocks, and thus the instructions in $\mathtt{C}_{\text{ll}}$, $\mathtt{C}'_{\text{ll}}$ are immaterial. In particular, since all conditions are non-deterministic, threads cannot affect the control flow of each other; as such, we do ignore memory accesses (read/write instructions) altogether (unlike in races).

1. $\mathtt{L}_1\colon \mathtt{lock}_\tau\ l$;	5. $\mathtt{L}_5\colon \mathtt{lock}_{\tau'}\ l'$;
2. $\mathtt{L}_2\colon \mathtt{lock}_\tau\ l'$;	6. $\mathtt{L}_6\colon \mathtt{lock}_{\tau'}\ l$;
$\mathtt{C}_{\text{ll}}$	$\mathtt{C}'_{\text{ll}}$
3. $\mathtt{unlock}_\tau\ l'$;	7. $\mathtt{unlock}_{\tau'}\ l$;
4. $\mathtt{unlock}_\tau\ l$;	8. $\mathtt{unlock}_{\tau'}\ l'$;

(DL)

The notion of deadlocks is captured by Brotherston et al. [2021] in terms of *critical pairs*. Specifically, a critical pair of a thread τ is a pair (A,l) such that in some execution τ attempts to acquire a lock l while already holding the set of locks (i.e. active locks) A. For instance, after executing lines 1 and 5 in DL, the critical pairs of τ and τ' are given by $(l',\{l\})$ and $(l,\{l'\})$, respectively. Two threads τ_1 and τ_2 deadlock iff they respectively yield critical pairs $c_1{=}(l_1,A_1)$ and $c_2{=}(l_2,A_2)$ such that $l_1\in A_2$, $l_2\in A_1$ and $A_1\cap A_2{=}\emptyset$, as is the case for $(l',\{l\})$ and $(l,\{l'\})$ of τ and τ' in DL. This intuition can be generalised to n threads: threads $\tau_1,\cdots,\tau_n$ with critical pairs $(l_1,A_1),\cdots,(l_n,A_n)$ deadlock iff for all $i\in\{1\cdots n\}$, $l_i\in\bigcup_{j\neq i}A_j$ and $A_i\cap\bigcup_{j\neq i}A_j=\emptyset$.

$$H \in \textsc{Hist} \triangleq \left\{E \in \textsc{Seq}\langle\textsc{Event}\rangle \;\middle|\; \mathsf{wf}(E)\right\} \qquad e \in \textsc{Event} \triangleq \left\{\mathsf{L}(\iota,\tau,x), \mathsf{U}(\tau,x) \;\middle|\; \iota \in \textsc{Label} \wedge \tau \in \textsc{Tid} \wedge x \in \textsc{Loc}\right\}$$
$$s \in \textsc{State}_{\mathrm{DD}} \triangleq \left\{f \in \textsc{Tid} \xrightarrow{\text{fin}} \textsc{Hist} \;\middle|\; \forall \tau, H, e.\ f(\tau){=}H \wedge e \in H \Rightarrow \mathsf{tid}(e){=}\tau\right\} \qquad m \in \textsc{MState}_{\mathrm{DD}} \triangleq \textsc{Hist}$$

Fig. 10. The $\mathrm{CISL}_{\mathrm{DD}}$ Model Domain, where the $\mathsf{wf}(.)$ definition is analogous to that in Fig. 7

We can capture the notion of critical pairs using our *histories* as defined in §4. For instance, the sequential history of τ in DL is $\tau \mapsto H$ with $H = [\mathsf{L}(\iota_1,\tau,l), \mathsf{L}(\iota_2,\tau,l'), \mathsf{U}(\tau,l'), \mathsf{U}(\tau,l)]$, which yields the critical pair c_1: H can be decomposed as $H_1 +\!\!+ [\mathsf{L}(\iota_2,\tau,l')] +\!\!+ -$, with $H_1 = [\mathsf{L}(\iota_1,\tau,l)]$ and the τ active locks in H_1 is given by A_1 ($\mathsf{alocks}(H_1,\tau){=}A_1$); i.e. τ attempts to acquire l' (via $\mathsf{L}(\iota_2,\tau,l')$) while holding the locks in A_1. Analogously, the sequential history of τ' yields the critical pair c_2.

We next present our $\mathrm{CISL}_{\mathrm{DD}}$ formalism and show how we use it to detect deadlocks using a deadlock assertion. Our formalism is inspired by that of Brotherston et al. [2021] and uses their notion of critical pairs. We then demonstrate how we can strengthen our deadlock assertion to go beyond the critical pairs of Brotherston et al. [2021] and produce a history *witnessing* the deadlock.

$\mathrm{CISL}_{\mathrm{DD}}$ Atomic Commands (Par. 1). As before, we assume a set of (shared) memory locations ranged over by x, y, z. We instantiate $\mathrm{CISL}_{\mathrm{DD}}$ with a simple set of atomic commands comprising constructs for acquiring and releasing locks, assignment and accessing the memory:

$$\textsc{Atom}_{\mathrm{DD}} \ni \mathsf{a} ::= \iota\colon \mathsf{lock}_\tau\ x \mid \mathsf{unlock}_\tau\ y \mid a := \mathsf{e} \mid a := [x] \mid [x] := a$$

To identify and locate deadlocks, we annotate each lock instruction with a label, ι, and each lock/unlock with a thread id, τ. Since conditions are non-deterministic, threads cannot affect the control flow of one another via (reading) in-memory values. As such, assignments and memory accesses have no bearing on deadlocks and thus need not be annotated. Similarly, when reporting a deadlock, we identify the culprit lock instructions, and thus do not label unlock instructions. As in $\mathrm{CISL}_{\mathrm{DD}}$ we assume that a $\mathrm{CISL}_{\mathrm{DD}}$ program is *well-formed*: it meets conditions (1)–(3) on p. 13.

$\mathrm{CISL}_{\mathrm{DD}}$ State PCM (Par. 2) and Error Conditions (Par. 3). The $\mathrm{CISL}_{\mathrm{DD}}$ PCM is $(\textsc{State}_{\mathrm{DD}}, \uplus, \emptyset)$, where $\textsc{State}_{\mathrm{DD}}$ is defined in Fig. 10, $\uplus$ denotes disjoint function union and $\emptyset$ denotes a map with an empty domain. As before, we write $\tau \mapsto H$ for $p{=}\{\tau \mapsto H\}$; the $\mathrm{CISL}_{\mathrm{DD}}$ functions alocks, syncs and locks are defined analogously to those of $\mathrm{CISL}_{\mathrm{RD}}$ in Figures 7 and 9. As discussed, we do not treat deadlocks as errors and thus define the $\mathrm{CISL}_{\mathrm{DD}}$ erroneous exit conditions as: $\textsc{ErExit}_{\mathrm{DD}} \triangleq \emptyset$.

$\mathrm{CISL}_{\mathrm{DD}}$ Axioms (Par. 4). The set of $\mathrm{CISL}_{\mathrm{DD}}$ axioms, $\textsc{Axiom}_{\mathrm{DD}}$, is defined below, where assignment and memory accesses require no resource: they have no bearing on deadlocks (recall that we prohibit deterministic conditions) and thus are not included in the history. By contrast, lock and unlock instructions extend their thread history with an associated event. As before, when τ already holds the lock on l (i.e. $l \in \mathsf{alocks}(H,\tau)$), then the resulting history $H +\!\!+ \mathsf{L}(\tau,l)$ is not well-formed, and thus $\tau \mapsto H +\!\!+ \mathsf{L}(\tau,l)$ in the postcondition of lock would render the triple vacuously valid.

DD-Lock
$[\tau \mapsto H]\ \iota\colon \mathsf{lock}_\tau\ l\ [ok\colon \tau \mapsto H +\!\!+ \mathsf{L}(\iota,\tau,l)]$

DD-Unlock
$[\tau \mapsto H]\ \mathsf{unlock}_\tau\ l\ [ok\colon \tau \mapsto H +\!\!+ \mathsf{U}(\tau,l)]$

DD-Assign
$[\mathsf{emp}]\ a := \mathsf{e}\ [ok\colon \mathsf{emp}]$

DD-Read
$[\mathsf{emp}]\ a := [x]\ [ok\colon \mathsf{emp}]$

DD-Write
$[\mathsf{emp}]\ [x] := a\ [ok\colon \mathsf{emp}]$

A Simple Deadlock Assertion. Recall that we can formulate a deadlock between two threads in terms of their critical pairs which in turn can be derived from the sequential thread histories. This is formulated in the $\mathsf{DL}(l_1,A_1,l_2,A_2)$ assertion below:

$$\mathsf{DL}(l_1,A_1,l_2,A_2) \overset{\text{def}}{\Longleftrightarrow} \exists \tau_1,\tau_2,H_1,H_2.\ \tau_1 \mapsto H_1 +\!\!+ \mathsf{L}(-,\tau_1,l_1) +\!\!+ - * \mathsf{alocks}(H_1,\tau_1){=}A_1 * l_2 \in A_1$$
$$* \tau_2 \mapsto H_2 +\!\!+ \mathsf{L}(-,\tau_2,l_2) +\!\!+ - * \mathsf{alocks}(H_2,\tau_2){=}A_2 * l_1 \in A_2 * A_1 \cap A_2 = \emptyset$$

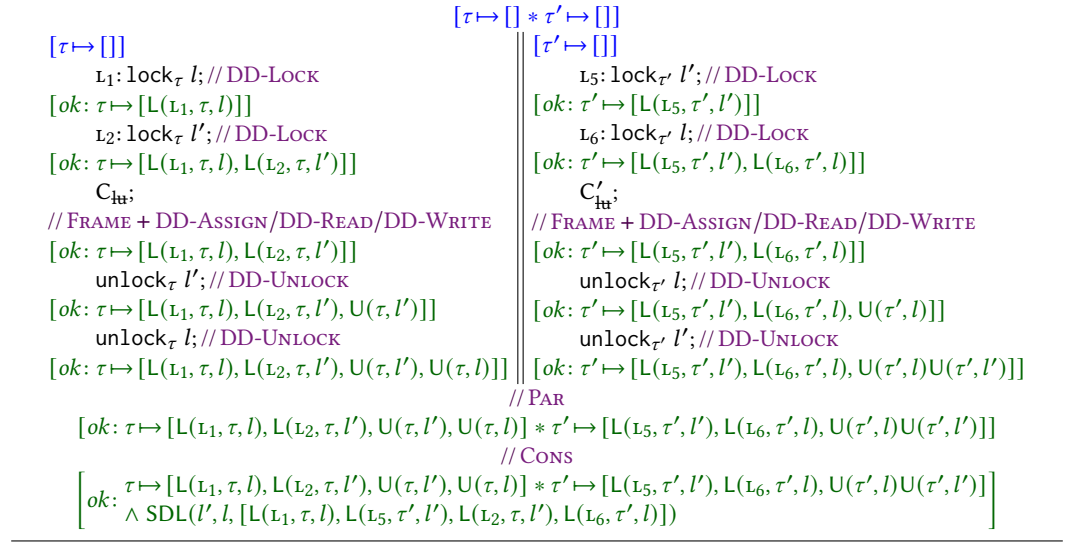


Fig. 11. A proof sketch of the deadlock in DL

That is, a deadlock exists between τ_1, τ_2 when they respectively yield critical pairs (l_1, A_1) and (l_1, A_1), and their respective critical pairs clash as described above. Specifically, a thread τ_1 attempts to lock l_1 after having locked l_2, while another thread τ_2 attempts to lock l_2 after having locked l_1, thus leading to a deadlock. The DL assertion corresponds to the deadlock reporting mechanism of Brotherston et al. [2021] using critical pairs; as discussed, this can be generalised to n threads.

A More Expressive Deadlock Assertion. Note that the DL assertion above (and that of Brotherston et al. [2021]) merely reports the *presence* of a deadlock, rather than *how* a deadlock may be encountered. Specifically, DL does not reflect how one may construct a history witnessing the deadlock. However, thanks to the sequential histories recorded for each thread in CISL_{DD}, we can indeed strengthen the DL assertion as follows to record a history (trace) H *witnessing* the deadlock, where the highlighted sections denote the extensions strengthening DL:

$$\begin{aligned}\mathsf{SDL}(l_1,l_2,H) \overset{\text{def}}{\iff} \exists \tau_1,\tau_2,H_1',H_2',H_1,H_2,e_1,e_2,A_1,A_2,P_2.\\ \tau_1 \mapsto H_1' +\!\!+ H_1 +\!\!+ e_1 +\!\!+ - * e_1 = \mathsf{L}(-,\tau_1,l_1) * \mathsf{alocks}(H_1,\tau_1) = A_1 * l_2 \in A_1\\ *\tau_2 \mapsto H_2' +\!\!+ H_2 +\!\!+ e_2 +\!\!+ - * e_2 = \mathsf{L}(-,\tau_2,l_2) * \mathsf{alocks}(H_2,\tau_2) = A_2 * l_1 \in A_2 * \mathsf{locks}(H_2,\tau_2) = P_2\\ *\mathsf{alocks}(H_1',\tau_1) = \mathsf{alocks}(H_2',\tau_2) = \emptyset * A_1 \cap P_2 = \emptyset * H = H_1' +\!\!+ H_2' +\!\!+ H_1 +\!\!+ H_2 +\!\!+ e_1 +\!\!+ e_2\end{aligned}$$

The H_1' and H_2' prefixes allow the two deadlocking threads to acquire the same lock so long as they release it before the deadlocking accesses ($\mathsf{alocks}(H_1',\tau_1) = \mathsf{alocks}(H_2',\tau_2) = \emptyset$).

1. τ_1: lock$_\tau$ x; ‖ 4. τ_2: lock$_\tau$ y;
2. τ_1: lock$_\tau$ z; ‖ 5. τ_2: lock$_\tau$ z;
3. τ_1: lock$_\tau$ y; ‖ 6. unlock$_{\tau_2}$ z;
‖ 7. τ_2: lock$_\tau$ x;

(DL1)

The $A_1 \cap P_2 = \emptyset$ ensures that the histories of τ_1, τ_2 can be combined into a well-formed global history. That is, (without loss of generality) we require that the locks (active or passive) acquired by τ_2 (in H_2) on the way to acquiring l_2 (via e_2) do not intersect with the active locks held by τ_1 thus far; were this not the case and τ_2 attempted to acquire a lock already held by τ_1, then appending H_2 after H_1 would not yield a well-formed history. To see this, consider DL1 and let $H_1'=H_2'=[]$, $H_1=[1,2]$, $H_2=[4,5,6]$, and let e_1 and e_2 be the events of locks on lines 3 and 7, respectively. We then have $A_1=\mathsf{alocks}(H_1,\tau_1)=\{x,z\}$ and $P_2=\mathsf{locks}(H_2,\tau_2)=\{y,z\}$, rendering the combined history $H = H_1' +\!\!+ H_2' +\!\!+ H_1 +\!\!+ H_2 +\!\!+ e_1 +\!\!+ e_2$ *not well-formed*, as τ_2 attempts to acquire z (line 5 in H_2) which is already held by τ_1 (line 2 in H_1).

Example 5.1. We present a proof sketch of the deadlock in DL in Fig. 11. Note that since $C_{\not\text{lu}}$ (resp. $C'_{\not\text{lu}}$) contain no luck/unlock instructions, and the remaining CISL_{DD} instructions (assignment, read and write) do not alter the history, it is straightforward to show that after executing $C_{\not\text{lu}}$ (resp. $C'_{\not\text{lu}}$) the sequential thread histories remain unchanged.

The SDL versus DL Assertion. Note that the SDL assertion is stronger (stricter) than DL, in that SDL may fail to catch deadlocks that DL can catch. To see this, consider the program in DL2, where the deadlock between lines 4, 5, 9 and 10 can be captured by DL as $\text{DL}(y, \{z, x\}, x, \{w, y\})$, but not by SDL. However, our aim with SDL is not to detect more deadlocks, but rather to provide a more expressive assertion that provides the user with a trace witnessing the deadlock. This is indeed a tradeoff: while DL can identify all potential deadlocks, it does not describe how the deadlock may arise (e.g. via a witness trace), and it is not always immediately obvious how the deadlock may be encountered, which can be crucial when fixing the deadlock. By contrast, SDL provides a witness trace that may aid the deadlock fixing process, albeit at the cost of missing some deadlocks. Note that given the under-approximate nature of CISL, it is sound to miss some deadlocks in the case of SDL.

1. τ_1: $\texttt{lock}_\tau\ z$;	6. τ_2: $\texttt{lock}_\tau\ w$;
2. τ_1: $\texttt{lock}_\tau\ y$;	7. τ_2: $\texttt{lock}_\tau\ x$;
3. $\texttt{unlock}_{\tau_1}\ y$;	8. $\texttt{unlock}_{\tau_2}\ x$;
4. τ_1: $\texttt{lock}_\tau\ x$;	9. τ_2: $\texttt{lock}_\tau\ y$;
5. τ_1: $\texttt{lock}_\tau\ y$;	10. τ_2: $\texttt{lock}_\tau\ x$;

(DL2)

Machine States (Par. 5) and Erasure (Par. 7). As in CISL_{RD}, a *$CISL_{DD}$ machine state*, $m \in \text{MSTATE}_{\text{DD}}$ in Fig. 10, records the global execution history, obtained by combining the thread histories (in STATE_{DD}) into a well-formed history via the *$CISL_{DD}$ erasure function* below, where $H|_\tau$ is as defined in §4:

$$\lfloor s \rfloor_{\text{DD}} \triangleq \left\{ H \;\middle|\; \forall \tau.\ s(\tau) = H' \Rightarrow H|_\tau = H' \right\}$$

CISL_{DD} Atomic Semantics (Par. 6). The CISL_{DD} atomic semantics is defined below, where (as in CISL_{RD}) each instruction of τ extends the current history (machine state) H_g by inserting an associated event e in H_g such that e is the last event of τ in the extended history, and the extended history is well-formed (i.e. is in HIST). As before, the atomic semantics of $\text{L}: \texttt{lock}_\tau\ l$ (resp. $\texttt{unlock}_\tau\ l$) returns an empty set when the lock l is already held (resp. not held) by τ, thanks to the well-formedness condition on histories (see Fig. 10).

$$\begin{aligned}
[\![\text{L}: \texttt{lock}_\tau\ l]\!]_{\text{A}} ok &\triangleq \left\{ (H_\tau +\!\!+ H, H_\tau +\!\!+ e +\!\!+ H) \in \text{HIST}^2 \;\middle|\; e{=}\text{L}(\text{L}, \tau, l) \wedge \forall e' \in H.\ e'.\text{tid} \neq \tau \right\} \\
[\![\texttt{unlock}_\tau\ l]\!]_{\text{A}} ok &\triangleq \left\{ (H_\tau +\!\!+ H, H_\tau +\!\!+ e +\!\!+ H) \in \text{HIST}^2 \;\middle|\; e{=}\text{U}(\tau, l) \wedge \forall e' \in H.\ e'.\text{tid} \neq \tau \right\} \\
[\![a := \text{e}]\!]_{\text{A}} ok = [\![a := [x]]\!]_{\text{A}} &= [\![[x] := a]\!]_{\text{A}} \triangleq \left\{ (H, H) \;\middle|\; H \in \text{HIST} \right\}
\end{aligned}$$

CISL_{DD} Atomic Soundness (Par. 8). In the technical appendix [Raad et al. 2022] we demonstrate that the CISL_{DD} atomic instructions are sound.

6 GENERALISING CISL TO PCMS WITH INTERFERENCE

Interference. Our notion of views thus far (Def. 3.4) describes abstract states *constructed* in such a way that are *stable*, i.e. immune to *interference* (modification) from concurrent threads. For instance, let τ and τ' denote two concurrent threads in CISL_{DC}, with their current states respectively given by s and s', such that their composition $(s \circ_{\text{DC}} s')$ is defined (otherwise they cannot run concurrently). From CISL_{DC} axioms we then know τ (resp. τ') can only modify an entry in s (resp. s') for which it has full permission (π=1), which (by definition of $\circ_{\text{DC}}$) cannot be in the domain of s' (resp. s). As such, τ and τ' cannot modify the states (and by lifting the views) of one another.

In our CISL instantiations so far, views are stable *by construction*. Although stable-by-construction views are simpler, and are indeed sufficient for our CISL instantiations so far, they are not strictly necessary. In particular, in reasoning frameworks with finer-grained permissions such as those of

$$\textsc{FrameInter}\ \frac{\vdash [p]\ \mathrm{C}\ [\epsilon : q] \qquad \mathsf{stable}(r)}{\vdash [p * r]\ \mathrm{C}\ [\epsilon : q * r]} \qquad \textsc{ParInter}\ \frac{(\mathsf{stable}(p_1, q_2) \vee \mathsf{stable}(p_2, q_1)) \qquad \vdash [p_i]\ \mathrm{C}_i\ [ok : q_i] \quad \text{for all } i \in \{1, 2\}}{\vdash [p_1 * p_2]\ \mathrm{C}_1 \,||\, \mathrm{C}_2\ [ok : q_1 * q_2]}$$

Fig. 12. Generalised rules of frame and parallel composition in the presence of interference

[Jung et al. 2015; Dinsdale-Young et al. 2010], views need not be stable by construction, so long as they are stable with respect to *interference* from other threads. Accordingly, the rules of frame and parallel composition are adapted to admit views that are stable with respect to interference. We can analogously generalise CISL to allow for interference. An interference relation describes how the states of one thread can be modified by other threads in the environment. As states are supplied as a CISL parameter, CISL is also parametric in their associated interference. In all our examples thus far, interference can be denoted by the *identity relation* as threads do not interfere by construction. Later in §7 we present a CISL instance with a non-identity interference relation.

Parameter 9 (Interference). Assume an interference relation $\mathcal{I} \subseteq \textsc{State} \times \textsc{State}$ such that:

- $\mathcal{I}$ is reflexive and transitive;
- for all s, s_1, s_2, s', if $s{=}s_1 \circ s_2$ and $(s, s') \in \mathcal{I}$, then there exist s_1', s_2' such that $s'{=}s_1' \circ s_2'$ and $(s_1, s_1'), (s_2, s_2') \in \mathcal{I}$; and
- for all $s_0 \in \textsc{State}^0$ and $s \in \textsc{State}$, if $(s, s_0) \in \mathcal{I}$, then $s \in \textsc{State}^0$.

Example 6.1. We illustrate how interference can be used to construct views by revisiting the states in $\mathrm{CISL}_{\mathrm{DC}}$ (Example 3.3). Specifically, rather than modelling states as maps with non-zero permissions, we can additionally allow zero permissions: $\textsc{State}^{\mathsf{z}}_{\mathrm{DC}} \triangleq \textsc{Var} \xrightarrow{\text{fin}} (\textsc{Val} \times [0, 1]) \cup (\textsc{Loc} \xrightarrow{\text{fin}} \textsc{Val} \uplus \{\bot\} \times [0, 1])$, with composition $\circ_{\mathrm{DC}}$ and units $\textsc{State}^0_{\mathrm{DC}}$ defined as before. The zero permission on k denotes the absence of ownership on k and confers no resources; as such, the $\mathrm{CISL}_{\mathrm{DC}}$ axioms remain unchanged. However, the interference is no longer the identity relation as we should account for other threads modifying those entries for which the current thread has zero permission: $\mathcal{I} \triangleq \left\{(s, s') \,\middle|\, \forall k.\ s(k) = (-, \pi) \wedge \pi > 0 \Rightarrow s(k) = s'(k)\right\}$. That is, concurrent threads may modify those entries that are not (partially) owned by the current thread.

We next define the notion of *stable* views as those views that are invariant under interference.

Definition 6.2 (Stability). Given the interference relation $\mathcal{I}$ (Par. 9), a view p is *stable*, written $\mathsf{stable}(p)$, iff $\mathcal{I}^{-1}(p) \subseteq p$, where $\mathcal{I}^{-1}(p) \triangleq \bigcup_{s \in p} \mathcal{I}^{-1}(s)$ and $\mathcal{I}^{-1}(s) \triangleq \left\{s' \,\middle|\, \exists s \in p.\ (s', s) \in \mathcal{I}\right\}$.

Note that unlike in correctness settings where stability is defined in the forward direction on $\mathcal{I}$, namely $\mathsf{stable}(p) \iff \mathcal{I}(p) \subseteq p$ with $\mathcal{I}(s) \triangleq \left\{s' \,\middle|\, (s, s') \in \mathcal{I}\right\}$, in the incorrectness setting of CISL stability is defined in the backward direction via $\mathcal{I}^{-1}$. To see why, let $p \triangleq x \overset{1}{\mapsto} 2$, $q \triangleq x \overset{1}{\mapsto} 3$, $r \triangleq x \overset{0}{\mapsto} 3$ and $\mathrm{C} \triangleq x := 3$. Using the DC-Assign axiom we obtain $[p]\ \mathrm{C}\ [ok : q]$. Subsequently, we can apply Frame to get $[p * r]\ \mathrm{C}\ [ok : q * r]$, i.e. $[\mathsf{false}]\ \mathrm{C}\ [ok : q * r]$. The resulting triple is *invalid*: it states that each state in $q * r$ is reachable from some state in the empty state set false! Similarly, using Skip we have $[r]\mathtt{skip}[ok : r]$ and we can apply Par to get $[\mathsf{false}]\ \mathrm{C} \,||\, \mathtt{skip}\ [ok : q * r]$!

In other words, when interference is a non-identity relation, we jeopardise the soundness of Frame and Par. In the example above, as the frame r holds zero permission on x, it should anticipate the environment to modify it. That is, r should be (but it is not) *stable* under $\mathcal{I}$ as defined in Example 6.1. As such, even though r is compatible with q ($q{*}r$ is defined), when x is modified by the environment, this change is not anticipated by r and thus $p * r \equiv \mathsf{false}$. By contrast, $r' \triangleq \exists v.\ x \overset{0}{\mapsto} v$ is stable, resulting in valid triples $[p * r']\ \mathrm{C}\ [ok : q * r']$ and $[p * r']\ \mathrm{C} \,||\, \mathtt{skip}\ [ok : q * r']$.

As such, we adapt the Frame and Par rules as shown in Fig. 12, where we write $\mathsf{stable}(p_1, p_2)$ as a shorthand for $\mathsf{stable}(p_1) \wedge \mathsf{stable}(p_2)$. Note that in ParInter it is sufficient for either p_1, q_2 or p_2, q_1 to be stable. Intuitively, this is because it suffices to show $[p_1 * p_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [ok\colon q_1 * q_2]$ holds for *one interleaving* of $\mathsf{C}_1 \,||\, \mathsf{C}_2$. In particular, $\mathsf{C}_1; \mathsf{C}_2$ is a valid interleaving of $\mathsf{C}_1 \,||\, \mathsf{C}_2$, and so long as we have $\mathsf{stable}(p_2, q_1)$ and $[p_i]\ \mathsf{C}_i\ [ok\colon q_i]$ for $i \in \{1, 2\}$, we can derive:

$$\dfrac{\dfrac{\dfrac{[p_1]\ \mathsf{C}_1\ [ok\colon q_1] \quad \mathsf{stable}(p_2)}{[p_1 * p_2]\ \mathsf{C}_1\ [ok\colon q_1 * p_2]}\ \textsc{FrameInter} \qquad \dfrac{[p_2]\ \mathsf{C}_2\ [ok\colon q_2] \quad \mathsf{stable}(q_1)}{[q_1 * p_2]\ \mathsf{C}_1\ [ok\colon q_1 * q_2]}\ \textsc{FrameInter}}{[p_1 * p_2]\ \mathsf{C}_1; \mathsf{C}_2\ [ok\colon q_1 * q_2]}\ \textsc{Seq}}{[p_1 * p_2]\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ [ok\colon q_1 * q_2]}\ \textsc{ParSeq}$$

Analogously, $\mathsf{stable}(p_1, q_2)$ allows us to derive $[p_1 * p_2]\ \mathsf{C}_2; \mathsf{C}_1\ [ok\colon q_1 * q_2]$ as another interleaving of $\mathsf{C}_1 \,||\, \mathsf{C}_2$. Lastly, we generalise axiom soundness accordingly to allow for interference: atomic commands may modify compatible frames (of concurrent threads) up to their interference.

Parameter 10 (General axiom soundness). Assume for all $(p, l, \epsilon, q) \in \textsc{Axiom}$ the following holds:

$$\forall s \in \textsc{State},\ m_q \in \lfloor q * \{s\} \rfloor.\ \exists m_p \in \lfloor p * \mathcal{I}^{-1}(s) \rfloor.\ (m_p, m_q) \in [\![l]\!]\epsilon$$

7 $\mathsf{CISL}_{\mathsf{SV}}$: CISL FOR SHARED CONCURRENCY WITH RESOURCE SUBVARIANTS

We next develop $\mathsf{CISL}_{\mathsf{SV}}$ as an instance of CISL with non-identity interference. $\mathsf{CISL}_{\mathsf{SV}}$ is the under-approximate analogue of concurrent separation logic (CSL) [O'Hearn 2004], used for reasoning about shared concurrency with resource invariants. In the correctness setting of CSL one must show that a shared resource $\mathbf{r}$ always satisfies an *invariant* that *over-approximates* the possible states of $\mathbf{r}$, after it has been accessed (within a critical section) an arbitrary number for times. However, in the incorrectness setting of CISL such over-approximation is of little use. Instead, we appeal to a resource *subvariant* that *under-approximates* the possible states of $\mathbf{r}$. Specifically, a subvariant $\mathcal{S}$ associated with resource $\mathbf{r}$ is a map from natural numbers to states, with $\mathcal{S}(n)$ under-approximating the possible states of $\mathbf{r}$ after it has been accessed n times; i.e. the initial state of $\mathbf{r}$ is given by $\mathcal{S}(0)$.

$\mathsf{CISL}_{\mathsf{SV}}$ Atomic Commands (Par. 1). $\mathsf{CISL}_{\mathsf{SV}}$ atomics include the heap-manipulating constructs of $\mathsf{CISL}_{\mathsf{DC}}$, as well as commands for accessing shared resources: 1) $\mathtt{acq}_\tau\ \mathbf{r}$ for acquiring the shared resource $\mathbf{r} \in \textsc{RId}$ within a critical section of thread τ; and 2) $\mathtt{rel}_\tau\ \mathbf{r}$ for releasing the shared resource $\mathbf{r}$ held by τ. As in CSL, we assume shared resources are declared sequentially before forking parallel threads: we focus on programs of the form $\mathtt{resource}\ \mathbf{r}_1 \cdots \mathbf{r}_k; \mathsf{C}_1 \,||\, \cdots \,||\, \mathsf{C}_n$. As noted in [O'Hearn 2004], it is possible to consider nested resource declarations and parallel compositions; however, for brevity we focus on this restricted form. We write $\mathtt{with}_\tau\ \mathbf{r}\ \mathtt{do}\ \mathsf{C}$ as a shorthand for $\mathtt{acq}_\tau\ \mathbf{r}; \mathsf{C}; \mathtt{rel}_\tau\ \mathbf{r}$.

$$\textsc{Atom}_{\mathsf{SV}} \ni \mathsf{a} ::= x := v \mid x := \mathtt{alloc}() \mid \textsc{l}\colon \mathtt{free}(x) \mid \textsc{l}\colon x := [y] \mid \textsc{l}\colon [x] := y \mid \mathtt{acq}_\tau\ \mathbf{r} \mid \mathtt{rel}_\tau\ \mathbf{r}$$

In what follows we first present the $\mathsf{CISL}_{\mathsf{SV}}$ axioms using an intuitive description of $\mathsf{CISL}_{\mathsf{SV}}$ states, and then present the formal definition of $\mathsf{CISL}_{\mathsf{SV}}$ states and their interference.

$\mathsf{CISL}_{\mathsf{SV}}$ Atomic Axioms. We present the $\mathsf{CISL}_{\mathsf{SV}}$ axioms in Fig. 13. The $\mathsf{res}^{\mathbf{r}}_{\mathcal{S}}(\tau\colon n)$ (defined shortly below) in the precondition of SV-Acq describes the resources necessary for τ to acquire $\mathbf{r}$ with subvariant $\mathcal{S}$, where n denotes the *contribution* of τ on $\mathbf{r}$: the number of times τ has accessed $\mathbf{r}$. That is, $\mathsf{res}^{\mathbf{r}}_{\mathcal{S}}(\tau\colon n)$ reflects the contribution of τ only, and not that of other threads; as such, the total contribution on $\mathbf{r}$ is unknown and may be any value $m \geq n$, as captured by the disjunction in the SV-Acq postcondition. Given the total contribution $m \geq n$, once τ acquires $\mathbf{r}$, it claims the resources of $\mathbf{r}$ ($\mathcal{S}(m)$) and changes $\mathsf{res}^{\mathbf{r}}_{\mathcal{S}}(\tau\colon n)$ for $\mathsf{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau\colon n, m)$. That is, once τ acquires $\mathbf{r}$, others can no longer access $\mathbf{r}$ and thus its total contribution m remains unchanged and can be reflected in $\mathsf{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau\colon n, m)$.

SV-Acq
$$\left[\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)\right]\mathtt{acq}_\tau\ \mathbf{r}\left[ok{:}\bigvee_{m\geq n}(\mathcal{S}(m)*\mathrm{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n,m))\right]$$

SV-Rel
$$\left[\bigvee_{m\geq n}(\mathcal{S}(m{+}1)*\mathrm{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n,m))\right]\mathtt{rel}_\tau\ \mathbf{r}\left[ok{:}\ \mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n{+}1)\right]$$

SV-Acq-G
$$\left[\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(m)\right]\mathtt{acq}_\tau\ \mathbf{r}\left[ok{:}\ \mathcal{S}(m)*\mathrm{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau,m)\right]$$

SV-Rel-G
$$\left[\mathcal{S}(m{+}1)*\mathrm{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau,m)\right]\mathtt{rel}_\tau\ \mathbf{r}\left[ok{:}\ \mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(m{+}1)\right]$$

SV-CS
$$\frac{\forall m\geq n.\ [p*\mathcal{S}(m)]\ \mathrm{C}\ [ok{:}\,q*\mathcal{S}(m{+}1)]\quad \mathrm{stable}(p,q)}{\left[p*\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)\right]\mathtt{with}_\tau\ \mathbf{r}\ \mathtt{do}\ \mathrm{C}\left[ok{:}\,q*\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n{+}1)\right]}$$

SV-CS-G
$$\frac{[p*\mathcal{S}(n)]\ \mathrm{C}\ [ok{:}\,q*\mathcal{S}(n{+}1)]}{\left[p*\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(n)\right]\mathtt{with}_\tau\ \mathbf{r}\ \mathtt{do}\ \mathrm{C}\left[ok{:}\,q*\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(n{+}1)\right]}$$

Fig. 13. $\mathrm{CISL}_{\mathrm{SV}}$ axioms (excerpt); SV-CS and SV-CS-G are derived from SV-Acq, SV-Rel and CISL proof rules

Upon successful acquisition of $\mathbf{r}$, τ enters a critical section and may freely modify $\mathcal{S}(m)$. However, prior to exiting the critical section, it must re-establish the subvariant for its incremented contribution, namely $\mathcal{S}(m{+}1)$, as shown in the precondition of SV-Rel. Once τ releases $\mathbf{r}$, it relinquishes $\mathcal{S}(m{+}1)$ and increments its contribution to $n{+}1$ by changing $\mathrm{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n,m)$ for $\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n{+}1)$.

While $\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)$ describes the resources needed for τ to acquire $\mathbf{r}$ and thus denotes a *τ-local* view, the $\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(n)$ describes those needed for *all threads*, denoting a *global* view. That is, $\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(n)$ grants full permission on $\mathbf{r}$, the environment cannot interfere with $\mathbf{r}$ and thus we can stably reflect the total contribution of all threads (m). Similarly, $\mathrm{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau,m)$ denotes a global analogue of $\mathrm{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n,m)$. Note that a global view may always be split to its constituent local views via the following equivalence:

$$\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(k)\Leftrightarrow\exists k_1\cdots k_n.\ k=\sum k_i*\mathop{\Large\ast}_{\tau_i\in\mathrm{TID}}\mathrm{res}^{\mathbf{r}}_{\mathcal{S}}(\tau_i{:}\,k_i)\qquad\text{(SUBV-SPLIT)}$$

We use the global views in SV-Acq-G and SV-Rel-G which denote global analogues of the local SV-Acq and SV-Rel rules. In contrast to the local, in the global rules we know the precise total contribution m and thus no longer need the outer disjunct ($\vee_{m\geq n}$).

The SV-CS for executing C within a critical section of $\mathbf{r}$ can be derived using SV-Acq, SV-Rel and other CISL rules, where p, q denote additional resources needed for executing C, provided that they are stable (as they will be framed off when applying SV-Acq and SV-Rel, respectively). The remaining axioms of $\mathrm{CISL}_{\mathrm{SV}}$ (e.g. for $\mathtt{free}(x)$) are the same as their $\mathrm{CISL}_{\mathrm{DC}}$ counterparts in Fig. 4.

Example 7.1. Assume that location x is shared within resource $\mathbf{r}$ (i.e. can only be accessed within a critical section), and let $\mathtt{disp}_\tau\ x$ be defined as follows to dispose (free) x atomically:

$$\mathtt{disp}_\tau\ x\triangleq\mathtt{with}_\tau\ \mathbf{r}\ \mathtt{do}\ \ C_1+C_2\quad C_1\triangleq\mathtt{assume}(\neg\mathtt{flag});\ \mathtt{free}(x);\ \mathtt{flag}:=1\quad C_2\triangleq\mathtt{assume}(\mathtt{flag});\mathtt{skip}$$

That is, if $\mathtt{flag}$ is set, then x is already deallocated and $\mathtt{disp}_\tau\ x$ does nothing; otherwise x is deallocated and $\mathtt{flag}$ is set. Let $\mathcal{S}(0)\triangleq\mathtt{flag}\mapsto 0*x\mapsto-$ and $\mathcal{S}(n)\triangleq\mathtt{flag}\mapsto 1*x\not\mapsto$ for $n\geq 1$. We present a proof derivation of $\mathtt{disp}_{\tau_1}\ x\ ||\ \mathtt{disp}_{\tau_1}\ x$ in Fig. 14 (top), where we assume $\mathrm{TID}=\{\tau_1,\tau_2\}$. Note that we use the SUBV-SPLIT equivalence together with the CISL CONS rule to split the global view into thread-local ones and subsequently pass them on to respective threads via PAR.

Example 7.2. Let $\mathtt{disp}_\tau\ x$ and $\mathcal{S}$ be as in Example 7.1, and $\mathrm{C}\triangleq\mathtt{disp}_{\tau_1}\ x;\mathrm{C}_3\ ||\ \mathrm{C}'$ for an arbitrary C' and with $\mathrm{C}_3\triangleq\mathtt{with}_{\tau_1}\ \mathbf{r}\ \mathtt{do}\ \ \mathrm{L}{:}\,\mathtt{free}(x)$. There is a local memory safety error at L: after executing $\mathtt{disp}_{\tau_1}\ x$, the location at x is guaranteed to be deallocated and thus the call to $\mathtt{free}(x)$ at L causes a memory safety error. We present a CISL proof of this error using PAREr in Fig. 14 (middle).

$\mathrm{CISL}_{\mathrm{SV}}$ States. A $\mathrm{CISL}_{\mathrm{SV}}$ *state* is a triple $s=(\mathbf{l},\mathbf{p},\rho)$, where $\mathbf{l}\in\mathrm{LSTATE}$ is a *local state*, $\mathbf{p}\in\mathrm{PERM}$ is a *permission* and $\rho\in\mathrm{RMAP}$ is a *shared resource map*, provided that s is *well-formed* (defined below).

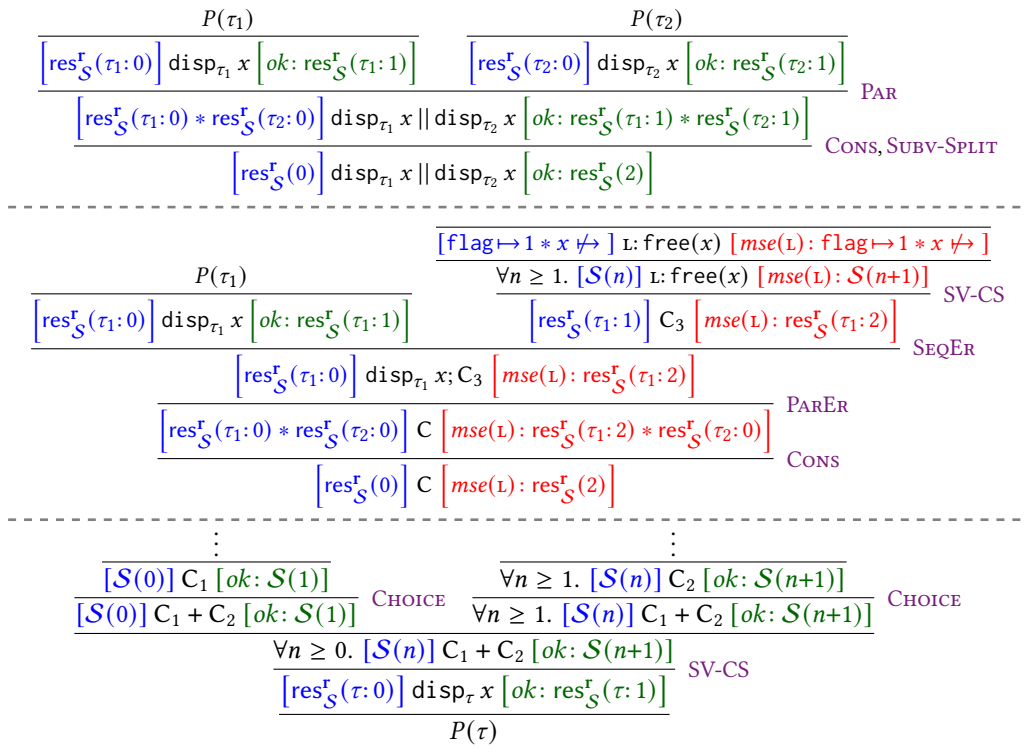


Fig. 14. Proof derivations of Example 7.1 (top) and Example 7.2 (middle), with $P(\tau)$ as defined at the bottom

Intuitively, a local state denotes the resources owned locally by a thread and is given by a PCM. The PCM choice, $(\textsc{LState}, \circ_{\mathrm{l}}, \textsc{LState}^0)$, can be supplied as a parameter to $\mathrm{CISL}_{\mathrm{SV}}$. For clarity, here we instantiate it with the $\mathrm{CISL}_{\mathrm{DC}}$ PCM in §3: $\textsc{LState} \triangleq \textsc{State}_{\mathrm{DC}}$, $\circ_{\mathrm{l}} \triangleq \circ_{\mathrm{DC}}$ and $\textsc{LState}^0 \triangleq \textsc{LState}^0$.

Permissions are used for accessing shared resources and are defined as $\textsc{Perm} \triangleq (\textsc{Rid} \times \textsc{Tid}) \to \mathbb{N}$. Intuitively, given a permission $\mathbf{p}$, a shared resource $\mathbf{r}$ and a thread τ, the $\mathbf{p}(\mathbf{r}, \tau)$ is held locally by τ and grants it the permission to access $\mathbf{r}$ within a critical section. Moreover, when $\mathbf{p}(\mathbf{r}, \tau) = n$, then n denotes the contribution of τ on $\mathbf{r}$: the number of times τ has accessed $\mathbf{r}$.

The set of resource maps is: $\rho \in \textsc{RMap} \triangleq \textsc{Rid} \xrightarrow{\text{fin}} (\textsc{Tid} \uplus \{\bot\}) \times \textsc{SubV} \times \textsc{TMap}$ and describes shared resources. Specifically, a resource map ρ associates a resource $\mathbf{r}$ with a triple of the form $(o, \mathcal{S}, t)$, where $o \in \textsc{Tid} \uplus \{\bot\}$ denotes the *owner*; $\mathcal{S} \in \textsc{SubV} \triangleq \mathbb{N} \to \mathcal{P}(\textsc{LState})$ denotes the subvariant; and $t \in \textsc{TMap}$ denotes a *thread map*. Intuitively, $o = \tau \in \textsc{Tid}$ denotes that $\mathbf{r}$ is currently being accessed within a critical section by τ (and thus τ owns the resources of $\mathbf{r}$). Conversely, $o = \bot$ denotes that $\mathbf{r}$ is not currently being accessed. The $\mathcal{S}$ denotes the resource subvariant as described above, with $\mathcal{S}(k)$ denoting the possible resources associated with $\mathbf{r}$ after it has been accessed k times. Lastly, the set of thread maps is $t \in \textsc{TMap} \triangleq \textsc{Tid} \to \mathbb{N}$. Intuitively, when t is associated with $\mathbf{r}$, $t(\tau)$ denotes the contribution of τ to $\mathbf{r}$ (i.e. the number of times τ has accessed $\mathbf{r}$). As such, as we formalise below, $t(\tau)$ and the permission $\mathbf{p}(\mathbf{r}, \tau)$ held locally by τ on $\mathbf{r}$ must agree. When accessing $\mathbf{r}$ within a critical section, τ can then use $\mathbf{p}(\mathbf{r}, \tau)$ to increment $\mathbf{p}(\mathbf{r}, \tau)$ and $t(\tau)$ *in tandem*, reflecting its added contribution. Given a thread map t, we write $\mathit{count}\,(t)$ for $\sum_{\tau \in dom(t)} t(\tau)$. Given a resource $\mathbf{r}$ and resource map ρ with $\rho(\mathbf{r}) = (-, -, t)$, we define $\mathit{count}\,(\rho, \mathbf{r}) \triangleq \mathit{count}\,(t)$ and $\mathit{count}\,(\rho, \mathbf{r}, \tau) \triangleq t(\tau)$.

Note that $\mathit{count}\,(\rho, \mathbf{r})$ denotes the *total* number of times $\mathbf{r}$ has been accessed (by all threads). As such, when $\mathit{count}\,(\rho, \mathbf{r}) = n{:}\ 1)$ if $\rho(\mathbf{r}) = (\bot, \mathcal{S}, -)$ (i.e. $\mathbf{r}$ is not being accessed), then the resources of

$\mathbf{r}$ are unclaimed and are given by $\mathcal{S}(n)$; and 2) if $\rho(\mathbf{r}) = (\tau, \mathcal{S}, -)$, then τ has acquired the resources of $\mathbf{r}$ ($\mathcal{S}(n)$) and transferred them to its local state; upon exiting the critical section, τ increments its contribution and releases the $\mathbf{r}$ resources, now given by $\mathcal{S}(n{+}1)$, returning them to the shared state.

A triple $(\mathbf{l}, \mathbf{p}, \rho)$ is *well-formed* if 1) the composition of the local state $\mathbf{l}$ and the resource of each $\mathbf{r}$ in ρ is defined; and 2) the contribution of $\mathbf{p}$ for each $(\mathbf{r}, \tau)$ agrees with its counterpart in ρ. We thus define the set of CISL_{SV} states as follows, where $\mathcal{L} \triangleq \left\{\mathcal{S}(n) \;\middle|\; \exists \mathbf{r}.\ \rho(\mathbf{r}) = (\bot, \mathcal{S}, t) \wedge \mathit{count}\,(\rho, \mathbf{r}) = n\right\}$:

$$\textsc{State}_{\text{SV}} \triangleq \left\{(\mathbf{l}, \mathbf{p}, \rho) \in \textsc{LState} \times \textsc{Perm} \times \textsc{RMap} \;\middle|\; \{\mathbf{l}\} * \mathop{\Large *}_{L_i \in \mathcal{L}} L_i \neq \emptyset \wedge \forall \mathbf{r}, \tau, n.\ \mathbf{p}(\mathbf{r}, \tau) = n \Rightarrow \mathit{count}\,(\rho, \mathbf{r}, \tau) = n\right\}$$

The *CISL_{SV} state composition* is defined component-wise as $\circ_{\text{SV}} \triangleq (\circ_{\mathbf{l}}, \uplus, \circ_{=})$, where $\rho_1 \circ_{=} \rho_2 = \rho_1$ if $\rho_1 = \rho_2$ and is otherwise undefined. The *CISL_{SV}* unit set is $\left\{(\mathbf{l}_0, \emptyset, \rho) \;\middle|\; \mathbf{l}_0 \in \textsc{LState}^0 \wedge \rho \in \textsc{RMap}\right\}$.

CISL_{SV} Interference. The CISL_{SV} interference is $\mathcal{I} \triangleq (\textsc{State}_{\text{SV}} \times \textsc{State}_{\text{SV}}) \cap (\mathcal{I}_a \cup \mathcal{I}_r)^*$, where $(\mathcal{I}_a \cup \mathcal{I}_r)^*$ denotes the transitive closure of $(\mathcal{I}_a \cup \mathcal{I}_r)$, and:

$$\mathcal{I}_a \triangleq \left\{\left((\mathbf{l}, \mathbf{p}, \rho), (\mathbf{l}, \mathbf{p}, \rho')\right) \middle| \exists \mathbf{r}, \mathcal{S}, t, \tau.\ \rho(\mathbf{r}) = (\bot, \mathcal{S}, t) \wedge (\mathbf{r}, \tau) \notin \mathit{dom}(\mathbf{p}) \wedge \rho' = \rho[\mathbf{r} \mapsto (\tau, \mathcal{S}, t)]\right\}$$
$$\mathcal{I}_r \triangleq \left\{\left((\mathbf{l}, \mathbf{p}, \rho), (\mathbf{l}, \mathbf{p}, \rho')\right) \middle| \exists \mathbf{r}, \mathcal{S}, t, \tau.\ \rho(\mathbf{r}) = (\tau, \mathcal{S}, t) \wedge (\mathbf{r}, \tau) \notin \mathit{dom}(\mathbf{p}) \wedge \rho' = \rho[\mathbf{r} \mapsto (\bot, \mathcal{S}, t[\tau \mapsto t(\tau){+}1])]\right\}$$

The $\mathcal{I}_a$ describes how a concurrent thread τ may modify a state $(\mathbf{l}, \mathbf{p}, \rho)$ of the current thread to $(\mathbf{l}, \mathbf{p}, \rho')$ when *acquiring* $\mathbf{r}$ (entering a critical section on $\mathbf{r}$). Note that $(\mathbf{r}, \tau) \notin \mathit{dom}(\mathbf{p})$ ensures the current thread holds no permission on $(\mathbf{r}, \tau)$, i.e. τ is a thread in the environment. Moreover, τ does not alter the local state ($\mathbf{l}$) and permission ($\mathbf{p}$) of the current thread, and its modification is limited to the shared state ρ. Specifically, τ acquires $\mathbf{r}$ by first checking it is not currently being accessed (the first component of $\rho(\mathbf{r})$ is $\bot$) and subsequently changing it to reflect it is being accessed by τ.

Dually, $\mathcal{I}_r$ describes how τ may modify a CISL_{SV} state when *releasing* $\mathbf{r}$ (exiting a critical section on $\mathbf{r}$). As before, the changes are limited to the shared state, whereby the first component of $\rho(\mathbf{r})$ is updated from τ (the releasing thread) to $\bot$, denoting that $\mathbf{r}$ is no longer being accessed. Moreover, the thread map of $\mathbf{r}$ (t) is updated to increment the number of accesses on $\mathbf{r}$ by τ.

CISL_{SV} Stable Views. Recall that the CISL_{SV} axioms in Fig. 13 use the stable views below:

$$\begin{aligned}
\text{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n) &\triangleq \left\{(\mathbf{l}_0, \mathbf{p}, \rho) \;\middle|\; \mathbf{l}_0 \in \textsc{LState}^0 \wedge \exists o, t.\ \rho(\mathbf{r}) = (o, \mathcal{S}, t) \wedge o \neq \tau \wedge n = t(\tau) \wedge \mathbf{p} = [(\mathbf{r}, \tau) \mapsto n]\right\}\\
\text{res}^{\mathbf{r}}_{\mathcal{S}}(m) &\triangleq \left\{(\mathbf{l}_0, \mathbf{p}, \rho) \;\middle|\; \mathbf{l}_0 \in \textsc{LState}^0 \wedge \exists t.\ \rho(\mathbf{r}) = (\bot, \mathcal{S}, t) \wedge m = \mathit{count}\,(t) \wedge \mathbf{p} = \uplus_{\tau \in \text{TID}}[(\mathbf{r}, \tau) \mapsto t(\tau)]\right\}\\
\text{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n, m) &\triangleq \left\{(\mathbf{l}_0, \mathbf{p}, \rho) \;\middle|\; \mathbf{l}_0 \in \textsc{LState}^0 \wedge \exists t.\ \rho(\mathbf{r}) = (\tau, \mathcal{S}, t) \wedge n = t(\tau) \wedge \mathbf{p} = [(\mathbf{r}, \tau) \mapsto n] \wedge m = \mathit{count}\,(t)\right\}\\
\text{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau, m) &\triangleq \left\{(\mathbf{l}_0, \mathbf{p}, \rho) \;\middle|\; \mathbf{l}_0 \in \textsc{LState}^0 \wedge \exists t.\ \rho(\mathbf{r}) = (\tau, \mathcal{S}, t) \wedge m = \mathit{count}\,(t) \wedge \mathbf{p} = \uplus_{\tau \in \text{TID}}[(\mathbf{r}, \tau) \mapsto t(\tau)]\right\}
\end{aligned}$$

The $\text{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)$ describes the resources necessary for τ to acquire $\mathbf{r}$ (as in the precondition of SV-Acq): no local resources are needed ($\mathbf{l}_0$), τ has not already acquired $\mathbf{r}$ ($o \neq \tau$) and the current state holds the permission for τ to access $\mathbf{r}$ with matching contributions ($n = t(\tau) \wedge \mathbf{p} = [(\mathbf{r}, \tau) \mapsto n]$). Note that $\text{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)$ does not require $o = \bot$ and thus allows for the possibility that another thread $\tau' \neq \tau$ may be currently accessing $\mathbf{r}$ (i.e. allows for $o = \tau'$). Were we to stipulate $o = \bot$, then $\text{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)$ would no longer be stable, as another thread τ' could release $\mathbf{r}$ and change o from τ' to $\bot$ as per $\mathcal{I}_r$.

Analogously, $\text{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)$ only reflects the contribution of τ, and not that of other threads (as this could change as per $\mathcal{I}_r$). Once τ acquires $\mathbf{r}$, it changes $\text{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)$ for $\text{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n, m)$ by updating o to τ, where $m \geq n$ is the total contribution on $\mathbf{r}$. As such, once τ acquires $\mathbf{r}$, threads can no longer access $\mathbf{r}$ and thus its total contribution remains unchanged and can be *stably* reflected in $\text{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n, m)$.

Recall that $\text{res}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n)$ denotes a τ-local view, while $\text{res}^{\mathbf{r}}_{\mathcal{S}}(n)$ denotes a global view. As such, the current state in $\text{res}^{\mathbf{r}}_{\mathcal{S}}(n)$ must hold the permissions of all threads ($\mathbf{p} = \uplus_{\tau \in \text{TID}}[(\mathbf{r}, \tau) \mapsto t(\tau)]$). Moreover, since the current state holds all thread permissions, the environment cannot interfere and thus we can stably 1) stipulate that $\mathbf{r}$ not be currently accessed ($o = \bot$); and 2) reflect the total contribution of all threads ($m = \mathit{count}\,(t)$). The $\text{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau, m)$ is defined analogously to $\text{cs}^{\mathbf{r}}_{\mathcal{S}}(\tau{:}\,n, m)$.

The remaining CISL_{SV} axioms (e.g. for $\texttt{free}(x)$) are as in Fig. 4, provided that we lift CISL_{DC} views to CISL_{SV}. Specifically, in CISL_{SV} we write $\texttt{emp}$ and $x \overset{\pi}{\mapsto} v$ for the stable views $\{(\emptyset, \emptyset, -)\}$ and $\{([x \mapsto (v, \pi)], \emptyset, -)\}$, respectively. We then write $x \mapsto v$ for $x \overset{1}{\mapsto} v$, and so forth.

CISL_{SV} Soundness. We define the remaining CISL_{SV} parameters (Parameters 5 to 7) in the technical appendix [Raad et al. 2022], proving that CISL_{SV} atomic instructions are sound (Par. 8).

8 CONCLUSIONS AND RELATED WORK

This work builds on CSL (concurrent separation logic) [O'Hearn 2004], IL (incorrectness logic) [O'Hearn 2019], and ISL (incorrectness separation logic) [Raad et al. 2020].

CSL spawned a wealth of prior research; see the survey article of Brookes and O'Hearn [2016] for an account of work in the area up to 2016. All of this prior work has focussed on over-approximation, used for proving the absence of violations of safety properties. The impact of this work has remained, however, largely academic. In contrast, the two under-approximate static analyses [Blackshear et al. 2018; Brotherston et al. 2021]—both intuitively (but not formally) related to separation logic—have already had a great deal of real-world impact beyond the academic subject area. These analyses share some of the pre-formal intuitions with CSL: both systems are based on compositional and thread-modular reasoning—the under-approximate analyses just use such modular reasoning to *find* bugs rather than exclude them. This commonality is unsurprising, given that—as O'Hearn [2018] explains—CSL-based ideas were in fact the genesis of RacerD, even if formally the connection was tenuous. In this paper, we bring these threads back together, by showing that the two analyses can in fact be understood in terms of a concurrent separation logic, but an under-approximate one, CISL, rather than the original CSL or one of its other successors.

IL and ISL are very young. The ISL paper [Raad et al. 2020] makes a partial but not full connection to an in-production static analyser, Pulse. The present paper makes a fuller connection to different analyses which have already proven effective in practice, and which also rely crucially on the compositional nature of reasoning as formalised in IL and CISL. This new account of recent, novel analysers adds significantly to the known ability of IL to describe legacy testing and symbolic execution reasoning techniques.

There is a body of work on static concurrency analysis which often is thread-modular (but not always compositional), exemplified by such papers as [Gotsman et al. 2007; Berdine et al. 2008; Li et al. 2019]. Noteworthy advances have been made on automated reasoning about concurrent programs in these works and, even if the technical expression of the ideas has tended to use over-approximation, it makes sense to consider in these and many other cases whether under-approximate variants would be possible or useful.

There is a fairly well-developed tradition of dynamic analysis techniques for concurrency, some of which has crossed over to have industrial impact. Prominent examples include probabilistic concurrency testing [Burckhardt et al. 2010], context-bounded model checking [Qadeer and Rehof 2005], and dynamic race detection as exemplified by Google's Thread Sanitizer [Serebryany and Iskhodzhanov 2009]. It is not yet clear whether CISL has anything to offer these techniques directly, but their insights might be transferred to static under-approximate concurrency analysis, and merging with CISL could conceivably open doors to more compositional and scalable techniques which are fast enough for deployment in code review on pull requests.

Finally, in addition to CSL, the Owicki-Gries [Owicki and Gries 1976] and Rely-Guarantee [Jones 1983] proof methods produced fundamental insights into reasoning about safety properties for concurrency, which were particularly well-suited to programs with more interference and less resource separation. We wonder whether under-approximate variants of these theories exist with

similarly compelling intuitive bases, and whether such under-approximation might open yet further avenues for automated analyses of concurrent programs.

ACKNOWLEDGMENTS

We thank the POPL 2022 reviewers and Viktor Vafeiadis for their valuable feedback. This research was supported in part by a UKRI Future Leaders Fellowship [grant number MR/V024299/1] and a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

REFERENCES

Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. 2008. Thread Quantification for Concurrent Shape Analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5123)*, Aarti Gupta and Sharad Malik (Eds.). Springer, 399–413. https://doi.org/10.1007/978-3-540-70545-1_37

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276514

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65. https://dl.acm.org/citation.cfm?id=2984457

James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max Kanovich. 2021. A Compositional Deadlock Detector for Android Java. In *Proceedings of ASE-36*. ACM. http://www0.cs.ucl.ac.uk/staff/J.Brotherston/ASE21/deadlocks.pdf

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve (Eds.). ACM, 167–178. https://doi.org/10.1145/1736020.1736040

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/2429069.2429104

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290370

Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. 2007. Thread-Modular Shape Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 266–277. https://doi.org/10.1145/1250734.1250765

C. 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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Yanze Li, Bozhen Liu, and Jeff Huang. 2019. SWORD: a scalable whole program race detector for Java. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 75–78. https://doi.org/10.1109/ICSE-Companion.2019.00042

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–310.

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67.

Peter W. O'Hearn. 2018. Experience Developing and Deploying Concurrency Analysis at Facebook. In *Static Analysis*, Andreas Podelski (Ed.). Springer International Publishing, Cham, 56–70.

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. http://doi.acm.org/10.1145/3371078

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (01 Dec 1976), 319–340. https://doi.org/10.1007/BF00268134

Matthew Parkinson. 2010. The Next 700 Separation Logics. In *Verified Software: Theories, Tools, Experiments*, Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–182.

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 225–252.

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter O'Hearn. 2022. Technical Appendix. https://www.soundandcomplete.org/papers/POPL2022/CISL/appendix.pdf

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 710–735.

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. 62–71.

Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–271.