ı=
`1

An Asynchronous Soundness Theorem for Concurrent Separation Logic

Paul-André Melliès
IRIF, CNRS & Université Paris Diderot

Léo Stefanesco
IRIF, Université Paris Diderot & CNRS

Abstract

Concurrent separation logic (CSL) is a specification logic for concurrent imperative programs with shared memory and locks. In this paper, we develop a concurrent and interactive account of the logic inspired by asynchronous game semantics. To every program C, we associate a pair of asynchronous transition systems $[\![C]\!]_S$ and $[\![C]\!]_L$ which describe the operational behavior of the Code when confronted to its Environment or Frame — both at the level of machine states (S) and of machine instructions and locks (L). We then establish that every derivation tree π of a judgment $\Gamma \vdash \{P\}C\{Q\}$ defines a winning and asynchronous strategy $[\![\pi]\!]_{Sep}$ with respect to both asynchronous semantics $[\![C]\!]_S$ and $[\![C]\!]_L$. From this, we deduce an asynchronous soundness theorem for CSL, which states that the canonical map $\mathscr{L} : [\![C]\!]_S \to [\![C]\!]_L$ from the stateful semantics $[\![C]\!]_S$ to the stateless semantics $[\![C]\!]_L$ satisfies a basic fibrational property. We advocate that this provides a clean and conceptual explanation for the usual soundness theorem of CSL, including the absence of data races.

CCS Concepts • **Theory of computation → Concurrency**; **Logic and verification**; **Separation logic**;

Keywords concurrent separation logic, asynchronous machine models, asynchronous game semantics, data races, stateful-to-stateless translation, separated states

ACM Reference Format:
Paul-André Melliès and Léo Stefanesco. 2018. An Asynchronous Soundness Theorem for Concurrent Separation Logic. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3209108.3209116

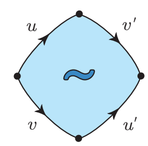
1 Introduction

A simple way to understand an imperative (possibly nondeterministic) program C is to interpret it as a binary relation $[C] \subseteq S \times S$ between machine states $s, s' \in S$. In that approach, the statement $s[C]s'$ indicates that one execution trace (at least) of the program C has initial state $s \in S$ and final state $s' \in S$. One practical advantage of this description is that the binary relation $[C]$ abstracts away from the execution traces of the program C, and only retains their initial and final states. However crude, this abstraction is generally sufficient to analyze the properties of sequential imperative programs, and to establish the soundness of Hoare logic. Unfortunately, the abstraction becomes too coarse when one decides to shift to concurrent imperative programs with shared memory and locks, and to establish the soundness of a specification logic like Concurrent Separation Logic (CSL). To that purpose, it has long been recognized that one needs a proper account of the execution traces of the program C, see Brookes [**?**]. In this paper, we go one step further, and advocate that the soundness theorem of CSL, and more specifically the absence of data races, is intrinsically related to the asynchronous structure of the execution paths of C. Inspired by asynchronous game semantics, we interpret every concurrent imperative program C as a pair of asynchronous graphs $[\![C]\!]_S$ and $[\![C]\!]_L$ related by an asynchronous graph homomorphism

$$\mathscr{L}_C \quad : \quad [\![C]\!]_S \longrightarrow [\![C]\!]_L \tag{1}$$

We start by recalling the notion of *asynchronous graph* [**? ?**] before discussing the relationship between time and space separation.

Asynchronous graphs A graph $G = (V, E, \partial^-, \partial^+)$ consists of a set V of vertices or nodes, a set of E of edges or transitions, and a source and a target function $\partial^-, \partial^+ : E \to V$. An *asynchronous graph* $(G, \diamond)$ is a graph G equipped with a binary relation $\diamond$ between paths $f, g : P \twoheadrightarrow Q$ of length 2, with the same source and target nodes. A pair (f, g) such that $f \diamond g$ is called a *permutation tile* and is depicted as a 2-dimensional tile between $f = u \cdot v'$ and $g = v \cdot u'$:



(2)

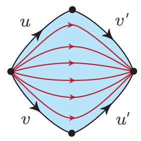
The intuition conveyed by such a permutation tile $u \cdot v' \diamond v \cdot u'$ is that the two transitions u and v are independent. For that reason, the two paths $u \cdot v'$ and $v \cdot u'$ may be seen as equivalent up to scheduling. The binary relation $\diamond$ is required to satisfy the following two axioms:
Axiom 1. The permutation relation $\diamond$ is symmetric, in the sense that $u \cdot v' \diamond v \cdot u'$ implies $v \cdot u' \diamond u \cdot v'$ for all transitions u, v, u', v'.
Axiom 2. In the situation below where $u \cdot w_1 \diamond v_1 \cdot u_1$ and $u \cdot w_2 \diamond v_2 \cdot u_2$, one has that $v_1 = v_2$ if and only if $w_1 = w_2$.

Two paths $f, g : M \twoheadrightarrow N$ of an asynchronous graph are equivalent modulo one permutation tile $h_1 \diamond h_2$ when f and g factor as $f = d \cdot h_1 \cdot e$ and $g = d \cdot h_2 \cdot e$ for two paths $d : M \twoheadrightarrow P$ and $e : Q \twoheadrightarrow N$. We write $f \sim g$ when the path $f : M \twoheadrightarrow N$ is equivalent to the path $g : M \twoheadrightarrow N$ modulo a number of permutation tiles. Note that the relation $\sim$ is is an equivalence relation, closed under composition.

Separation in space and time The 2-dimensional permutation tiles $f \diamond g$ provide a topological means to reflect the *temporal* nature of independence in concurrency theory. Every permutation tile (**??**) indicates that the two transitions u and v are independent in time: they may be equivalently executed in the sequential order $u \cdot v'$ or in the sequential order $v \cdot u'$. Although all the asynchronous graphs considered in this paper are discrete, it is enlightening to take the topological intuition of "homotopy" seriously, and to imagine that the path $u \cdot v'$ could be transformed "continuously" into the path

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209116

$v \cdot u'$ by a sequence of local deformations of the form



as it would be possible if one embedded our asynchronous graphs $(G, \diamond)$ in the topological framework of directed homotopy, see [?]. In the same spirit, we could replace our 2-dimensional graphs by higher-dimensional automata admitting n-dimensional cubes [?].

Interestingly, usually, the *temporal* independence of two transitions u and v is not primitive: it is a consequence of their *spatial* separation. In that respect, the idea of temporal independence may be seen as a layer of abstraction above the more concrete and machine-dependent idea of spatial separation. We illustrate this basic but important point by constructing an asynchronous graph $(G, \diamond_G)$ based on a very simple machine model, consisting of

- a countable set **Var** of variables, written $x, y, \ldots,$
- a countable set **Val** of values, written $v, w, \ldots,$
- a countable set $\mathbf{Loc} \subseteq \mathbf{Val}$ of memory locations, written ℓ.

A *memory state* $\mu = (s, h)$ of the machine is defined as a pair consisting of two partial functions

$$s : \mathbf{Var} \rightharpoonup_{fin} \mathbf{Val} \qquad h : \mathbf{Loc} \rightharpoonup_{fin} \mathbf{Val} \tag{3}$$

with finite domains, called the *stack* s and the *heap* h of the memory state μ. The instructions m of our machine are of three kinds:

$$x := v \qquad x := [\ell] \qquad [\ell] := x \tag{4}$$

where (1) the instruction $x := v$ assigns a value v to the variable x, (2) the instruction $x := [\ell]$ loads the value $h(\ell)$ at location ℓ and assigns it to the variable x, and (3) the instruction $[\ell] := x$ stores at location ℓ the current value $s(x)$ of the variable x. The asynchronous graph $(G, \diamond_G)$ is defined as follows. Its nodes are the memory states (**??**) of the machine, and its transitions are of the form

$$\begin{array}{lll} (s,h) \xrightarrow{x:=v} (s',h) & & \text{when } s' = s \otimes \{x \mapsto v\}, \\ (s,h) \xrightarrow{x:=[\ell]} (s',h) & & \text{when } h(\ell) \text{ is defined and} \\ & & s' = s \otimes \{x \mapsto h(\ell)\}, \\ (s,h) \xrightarrow{[\ell]:=x} (s,h') & & \text{when } s(x) \text{ is defined and} \\ & & h' = h \otimes \{\ell \mapsto s(x)\}. \end{array}$$

Here, we use the following convenient notation: given a partial function $f : X \rightharpoonup_{fin} Y$ with finite domain between two sets X and Y, and an element $y \in Y$, we write $f \otimes \{x \mapsto y\} : X \rightharpoonup_{fin} Y$ for the partial function with finite domain defined as

$$f \otimes \{x \mapsto y\} \quad : \quad x' \quad \mapsto \quad \begin{cases} f(x) & \text{when } x' \neq x, \\ y & \text{when } x' = x. \end{cases}$$

In order to define the permutation tiles of the asynchronous graph $(G, \diamond_G)$, one observes that every transition

$$u \quad : \quad (s,h) \xrightarrow{\quad m \quad} (s',h')$$

performed by an instruction m reads and writes on a specific area

$$\mathrm{rd}(u) \subseteq \mathbf{Var} + \mathbf{Loc} \qquad \mathrm{wr}(u) \subseteq \mathbf{Var} + \mathbf{Loc}$$

of the memory of the machine, which we shall call its *footprint*. This footprint may be computed from the instruction m performing the transition $u = (\mu, m, \mu')$ in the following way:

$$\boxed{\begin{array}{rclrcl} & & \mathrm{rd}(x := v) \; = \; \emptyset & & & \\ & & \mathrm{wr}(x := v) \; = \; \{x\} & & & \\ \mathrm{rd}(x := [\ell]) & = & \{\ell\} & \mathrm{rd}([\ell] := x) & = & \{x\} \\ \mathrm{wr}(x := [\ell]) & = & \{x\} & \mathrm{wr}([\ell] := x) & = & \{\ell\} \end{array}}$$

Now, suppose given two transitions $u : \mu \longrightarrow \mu_1$ and $v : \mu \longrightarrow \mu_2$ starting from the same memory state μ in the graph G. The two transitions u and v are declared *independent* when

$$(\mathrm{rd}(u) \cup \mathrm{wr}(u)) \cap \mathrm{wr}(v) = \emptyset \quad \text{and} \quad \mathrm{wr}(u) \cap (\mathrm{rd}(v) \cup \mathrm{wr}(v)) = \emptyset.$$

Note that the independence of the transitions u and v is a *consequence* of their spatial separation. It is not difficult to see that for every pair of such independent transitions

$$u \;:\; \mu_1 \xrightarrow{\quad m_1 \quad} \mu_2 \qquad\qquad v \;:\; \mu_2 \xrightarrow{\quad m_2 \quad} \mu_3$$

there exists a unique memory state μ_2' such that

$$u' \;:\; \mu_2' \xrightarrow{\quad m_1 \quad} \mu_3 \qquad\qquad v' \;:\; \mu_1 \xrightarrow{\quad m_2 \quad} \mu_2'$$

are transitions of the graph G. In that case, we say that u' is the residual of u after v and, symmetrically, that v' is the residual of v after u. This basic confluence property leads us to the following definition. A permutation tile of the form (**??**)

$$u \cdot v' \quad \diamond_G \quad v \cdot u'$$

in the asynchronous graph $(G, \diamond_G)$ is defined as a pair of independent transitions u and v where the transition u' is defined as the residual of u after v, and the transition v' is defined as the residual of v after u. It is not difficult to see that the graph $G = (V, E)$ of memory states and transitions between them, together with the notion of permutation tile $u \cdot v' \diamond_G v \cdot u'$ just defined, satisfy the axioms required of an asynchronous graph $(G, \diamond_G)$.

Stateful vs. stateless semantics Along the *stateful* description of the machine provided by the asynchronous graph $(G, \diamond_G)$, comes a *stateless* description of the same machine, conveyed this time by an asynchronous graph $(H, \diamond_H)$ where only the instructions are considered, not their action on the machine states. Accordingly, the graph H has a single node $*$ and a transition

$$a \quad : \quad * \xrightarrow{\quad m \quad} *$$

for each instruction m of the machine displayed in (**??**) parametrized by $x \in \mathbf{Var}$, $v \in \mathbf{Val}$ and $\ell \in \mathbf{Loc}$. The graph H is moreover equipped with a permutation tile

$$a \cdot b' \diamond_H b \cdot a'$$

for every pair $a = a'$ and $b = b'$ of instructions of the machine. The two asynchronous transition graphs $(G, \diamond_G)$ and $(H, \diamond_H)$ are related by an asynchronous graph homomorphism

$$\mathscr{L} \quad : \quad (G, \diamond_G) \;\longrightarrow\; (H, \diamond_H) \tag{5}$$

which maps every memory state μ to the node $*$, and every instruction to itself. We recall the definition of such a homomorphism:

Definition 1.1 (homomorphism)**.** An asynchronous graph homomorphism

$$\mathscr{F} \quad : \quad (G, \diamond_G) \;\longrightarrow\; (H, \diamond_H) \tag{6}$$

is a graph homomorphism $\mathscr{F} : G \to H$ between the underlying graphs, such that

$$u \cdot v' \diamond_G v \cdot u' \quad \Rightarrow \quad \mathscr{F}(u) \cdot \mathscr{F}(v') \diamond_H \mathscr{F}(v) \cdot \mathscr{F}(u')$$

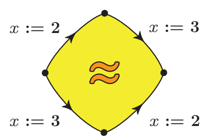
for all transitions u, u', v, v' of the asynchronous graph G.

Note that, in that situation, one has

$$f \sim g \quad \Rightarrow \quad \mathscr{L}(f) \approx \mathscr{L}(g)$$

for all paths $f, g : M \twoheadrightarrow N$ in G, where $\approx$ denotes the permutation equivalence in the asynchronous graph $(H, \diamond_H)$.

Data races as topological obstructions The reason for the liberal definition of $\diamond_H$ is that nothing should forbid two instructions m_1 and m_2 to commute at the *stateless* level of abstraction. By way of illustration, there exists a permutation tile in H (depicted below in light yellow) which permutes the two instructions $x := 2$ and $x := 3$ in the following way:



(7)

This permutation tile (**??**) should be understood as a basic example of *data race* in the machine, where the two instructions $x := 2$ and $x := 3$ compete for the same variable x. As a matter of fact, one key observation and guiding idea of the paper is that such a data race may be detected by the fact that it defines a permutation tile in the stateless semantics $(H, \diamond_H)$ which does *not* lift along $\mathscr{L}$ to a permutation tile in the stateful semantics $(G, \diamond_G)$. This line of thought leads us to the following definitions of 1-fibration and 2-fibration.

Definition 1.2 (1-fibration)**.** An asynchronous graph homomorphism $\mathscr{F} : (G, \diamond_G) \to (H, \diamond_H)$ is called a 1-fibration when for every node x of G and transitions $v : \mathscr{F}(x) \to z$, there exists a transition $u : x \to y$ such that $\mathscr{F}(u) = v$.

Definition 1.3 (2-fibration)**.** An asynchronous graph homomorphism $\mathscr{F} : (G, \diamond_G) \to (H, \diamond_H)$ is called a 2-fibration when for every pair of transitions u and v' defining a path $u \cdot v'$ of length 2 in G and for every permutation tile

$$\mathscr{F}(u) \cdot \mathscr{F}(v') \diamond_H b \cdot a'$$

in H, there exists a pair of transitions v and u' in G such that

$$u \cdot v' \diamond_G v \cdot u' \quad \text{and} \quad \mathscr{F}(v) = b \quad \text{and} \quad \mathscr{F}(u') = a'.$$

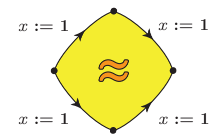
Coming back to our construction, our point is that the asynchronous graph homomorphism $\mathscr{L}$ defined in (**??**) is *not* a 2-fibration because of the presence of data races such as (**??**) in the stateless semantics. Typically, any sequence of transitions in $(G, \diamond_G)$

$$\mu_1 \xrightarrow{x:=2} \mu_2 \xrightarrow{x:=3} \mu_3 \tag{8}$$

mapped by $\mathscr{L}$ to the upward border $* \xrightarrow{x:=2} * \xrightarrow{x:=3} *$ of the permutation tile (**??**) in the asynchronous graph $(H, \diamond_H)$ satisfies $\mu_2(x) = 2$ and $\mu_3(x) = 3$. For that reason, there exists no way to lift the permutation tile (**??**) along $\mathscr{L}$ and to permute the sequence of instructions (**??**) accordingly in $(G, \diamond_G)$ as follows:

$$\mu_1 \xrightarrow{x:=3} \mu_2' \xrightarrow{x:=2} \mu_3 \tag{9}$$

because this would mean that $\mu_3(x) = 2$, and this would contradict the fact that $\mu_3(x) = 3$. More generally, every data race in the machine may be detected as a topological obstruction to the fact that the stateful-to-stateless homomorphism $\mathscr{L}$ is a 2-fibration. Note that, in the same way but for different reasons, the data race between the two instructions $x := 1$ described by the permutation tile in H below



(10)

does *not* lift along $\mathscr{L}$ to a permutation tile in G. Indeed, the instruction $x := 1 : \mu \to \mu'$ starting from any memory state μ has the nontrivial footprint $\mathrm{wr}(x := 1) = \{x\}$, and is thus not independent of itself in the asynchronous graph $(G, \diamond_G)$.

An asynchronous semantics of code The machine just considered is a very elementary toy model, which can be easily extended with locks and with memory allocation and deallocation. Also, more than in the machine itself, we are interested in the asynchronous description of the code C we want to analyse. We thus need to explain how we shift from the machine to the code. Interestingly, the story remains essentially the same. To every program C, we associate a stateful interpretation $[\![C]\!]_S$ and a stateless interpretation $[\![C]\!]_L$ which reflect the interactive behavior of the program C when confronted to its Environment, called Frame in that context. The two interpretations $[\![C]\!]_S$ and $[\![C]\!]_L$ are formulated as *asynchronous transition systems* (ATS) related by a homomorphism

$$\mathscr{L}_C \quad : \quad [\![C]\!]_S \longrightarrow [\![C]\!]_L \tag{11}$$

mentioned in (**??**) which plays the same role for the code C as the homomorphism (**??**) for the machine model. The two ATSs $[\![C]\!]_S$ and $[\![C]\!]_L$ are defined uniformly by structural induction on the program C. Their construction — and more specifically the interpretation of the parallel product $C_1 \parallel C_2$ — requires to develop a number of new techniques, in particular an asynchronous parallel product of two ATSs based on the same machine model.

The asynchronous soundness theorem As in the case of the machine model, the data races produced by the program C will be detected as *obstructions* to the fact that $\mathscr{L}_C$ is a 2-fibration. Typically, the program C defined as $x := 2\,; x := 3$ is data-race-free because the permutation tile (**??**) does not appear in the stateless semantics $[\![C]\!]_L$, while the program C' defined as $x := 2 \parallel x := 3$ produces a data race reflected by the fact that the permutation tile (**??**) appears in the stateless interpretation $[\![C']\!]_L$ and cannot be lifted along $\mathscr{L}$ to the stateful interpretation $[\![C']\!]_S$.

In the present paper, we carry on our game-theoretic investigation of Concurrent Separation Logic (CSL) initiated in [**?**] and establish that *well-specified* programs are data-race-free. We achieve this by interpreting every derivation tree

$$\frac{\vdots\ \pi}{\Gamma \vdash \{P\}C\{Q\}} \tag{12}$$

of CSL as an asynchronous strategy $[\![\pi]\!]_{Sep}$ playing on the asynchronous game of separated states. Our asynchronous version of the Soundness Theorem is then formulated in the following fibrational way. Suppose that a code C comes equipped with a proof of the Hoare triple $\Gamma \vdash \{P\}C\{Q\}$ in CSL, and consider the asynchronous subgraph $[\![\{P\}C]\!]_S^\tau$ obtained by restricting $[\![C]\!]_S$ to the nodes reachable from an initial node satisfying the precondition P. In that situation, we establish (see Thm. **??** in §**??** for details) that

Asynchronous Soundness Theorem. *The stateful-to-stateless homomorphism $\mathscr{L}_C : [\![C]\!]_S \to [\![C]\!]_L$ is a 2-fibration when restricted to the asynchronous subgraph $[\![\{P\}C]\!]^{\tau}_{S}$.*

The 2-fibrational property is conceptually new and provides the first structural explanation for the absence of data races in concurrent programs specified by CSL.

Related works Stephen Brookes established the first proof of soundness of CSL in [**?**], using a stateless trace semantics similar to $[\![C]\!]_L$ for the concurrent imperative programs. More recently, Viktor Vafeiadis [**?**] gave a new proof of soundness, based this time on a stateful operational semantics, similar to $[\![C]\!]_S$. Our approach can be seen as unifying the two schools of semantics, by revealing the asynchronous graph morphism (**??**) between them. Also, one main benefit of our asynchronous approach is that we can directly describe and analyze the concurrent execution of two instructions.

In the same way as we do here, Jonathan Hayman and Glynn Winskel [**?**] establish the soundness of CSL in a "truly concurrent" setting. They interpret programs as Petri nets, where the interference of the environment is modeled by adding events to the Petri net. In contrast to our work, precision of the invariants is necessary for their semantics to work whereas [**?**] has shown that precision is only needed in order to interpret properly the conjunction rule.

We give in [**?**] a game-theoretic interpretation of CSL, where every Hoare triple is interpreted as a *game* between Adam and Eve, and every derivation tree π as a *winning strategy* for Eve in that game. Every program is interpreted there as a set of purely sequential traces. For that reason, it is not possible to establish in this framework the absence of data races, at least in a nice and conceptual way. One main achievement of the paper is thus to define a properly asynchronous game semantics of CSL, and to derive for the first time the absence of data races from purely semantic considerations on the model.

Synopsis of the paper After the machine states and instructions are described in §**??**, we construct in §**??** the two asynchronous graphs $\pitchfork_S$ and $\pitchfork_L$ defining our stateful and stateless machine models. We then explain in §**??** how to interpret every code C as a pair $[\![C]\!]_S$ and $[\![C]\!]_L$ of asynchronous transition systems (ATS) with respective machine models $\pitchfork_S$ and $\pitchfork_L$. Once the notions of logical state and of separated state are recalled in §**??** and in §**??**, we explain in §**??** how to interpret every proof π of CSL as an asynchronous strategy $[\![\pi]\!]_{Sep}$ playing on the machine model $\pitchfork_{Sep}$ of separated states. From this, we establish our asynchronous soundness theorem in §**??**, and conclude in §**??**.

2 Machine states and machine instructions

We introduce below the notions of *machine state* and of *machine instruction* which will be used throughout the paper. We suppose given countable sets **Var** of *variable names*, **Val** of values, **Loc** $\subseteq$ **Val** of *memory locations*, and **LockName** of *resources*. In practice, we consider the case where **Loc** $= \mathbb{N}$ and **Val** $= \mathbb{Z}$.

Definition 2.1 (*Memory states*)**.** A *memory state* μ is a pair (s, h) of partial functions with finite domains $s : \mathbf{Var} \rightharpoonup_{fin} \mathbf{Val}$ and $h : \mathbf{Loc} \rightharpoonup_{fin} \mathbf{Val}$ called the *stack s* and the *heap h* of the memory state μ. The set of memory states is denoted by **State**. The domains of the partial function s and of h are denoted by $\mathrm{vdom}(\mu)$ and $\mathrm{hdom}(\mu)$ respectively, and we write $\mathrm{dom}(\mu)$ for their disjoint union.

Definition 2.2 (*Machine states*)**.** A *machine state* is either a pair $\mathbb{s} = (\mu, L)$ consisting of a memory state μ and a subset of resources $L \subseteq$ **LockName**, called the *lock state*, which describes the subset of locked resources in $\mathbb{s}$; or an error state $\lightning$. The set of machine states is denoted by **MState**. Formally:

$$\mathbf{MState} = \mathbf{State} \times \wp(\mathbf{LockName}) + \{\lightning\}$$

A machine step is defined as a labeled transition between machine states. There are two kinds of transitions:

$$(\mu, L) \stackrel{m}{\rightsquigarrow} (\mu', L') \qquad\qquad (\mu, L) \stackrel{m}{\rightsquigarrow} \lightning \qquad (13)$$

depending on whether the instruction $m \in$ **Instr** has been executed successfully (on the left) or has produced a runtime error (on the right). In particular, $\lightning$ has no successor. The machine instructions $m \in$ **Instr** which label the machine steps are of the following form:

$$\begin{aligned} m ::= {} & x := E \mid x := [E] \mid [E] := E' \mid \mathtt{nop} \\ & \mid x := \mathtt{alloc}(E, \ell) \mid \mathtt{dispose}(E) \mid P(r) \mid V(r) \end{aligned}$$

where $x \in$ **Var** is a variable, $r \in$ **LockName** is a resource name, ℓ is a location, and E, E' are arithmetic expressions, possibly with "*free*" variables in **Var**. For example, the instruction $x := E$ executed in a machine state $\mathbb{s} = (\mu, L)$ assigns to the variable x the value $E(\mu) \in$ **Val** when the value of the expression E can be evaluated in the memory state μ, and produces the runtime error $\lightning$ otherwise. The instruction $P(r)$ acquires the resource variable r when it is available, while the instruction $V(r)$ releases it when r is locked, as described below:

$$\frac{E(\mu) = v}{(\mu, L) \stackrel{x:=E}{\rightsquigarrow} (\mu[x \mapsto v], L)} \qquad \frac{E(\mu) \text{ not defined}}{(\mu, L) \stackrel{x:=E}{\rightsquigarrow} \lightning}$$

$$\frac{r \notin L}{(\mu, L) \stackrel{P(r)}{\rightsquigarrow} (\mu, L \uplus \{r\})} \qquad \frac{r \notin L}{(\mu, L \uplus \{r\}) \stackrel{V(r)}{\rightsquigarrow} (\mu, L)}$$

The inclusion **Loc** $\subseteq$ **Val** means that an expression E may also denote a location. In that case, $[E]$ refers to the value stored at location E in the heap. The instruction $x := \mathtt{alloc}(E, \ell)$ allocates some memory space on the heap at address $\ell \in$ **Loc**, initializes it with the value of the expression E, and assigns the address ℓ to the variable $x \in$ **Var** if *location* was free, otherwise there is no transition. $\mathtt{dispose}(E)$ deallocates the location denoted by E when it is allocated, and returns $\lightning$ otherwise. Finally, the instruction nop (for no-operation) does not alter the state.

3 Asynchronous Machine Models

As explained in the introduction, *machine models* are described using asynchronous graphs. Since we consider *stateful* as well as *stateless* descriptions of the machine and of the code, we will consider two kinds of machine models, organized into a pair of asynchronous graphs: the *stateful model* $\pitchfork_S$ based on machine states, and the *stateless model* $\pitchfork_L$ based on locks. Their tiles will be defined using the notion of footprint, which summarizes which area of the state (memory, locks) an instruction relies on, and how it uses it. In both cases, we write $\mathrm{footprint}_s(m)$ for the footprint of an instruction m in state s, omitting the subscript when it is clear from the context. Our machine models $\pitchfork_S$ and $\pitchfork_L$ are parameterized over the finite set *Locks* $\subseteq$ **LockName** of locks, or resources, which are considered well-defined. We sometimes write $\pitchfork_S(Locks)$ or $\pitchfork_L(Locks)$ to make it explicit.

The stateful model A machine state footprint

$$\rho \in \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(\mathit{Locks}) \times \wp(\mathbf{Loc})$$

is, made of: (i) $\mathrm{rd}(\rho)$, the part of the memory that is *read*, (ii) $\mathrm{wr}(\rho)$, the part of the memory that is *written*, (iii) $\mathrm{lock}(\rho)$, the locks that are *touched*, and (iv) $\mathrm{mem}(\rho)$ the addresses that are *allocated* or *deallocated*. Two footprints ρ and ρ' are declared *independent* when:

$$\begin{array}{lcl} (\,\mathrm{rd}(\rho\ ) \cup \mathrm{wr}(\rho\ )\,) \cap \mathrm{wr}(\rho') &=& \emptyset \qquad \mathrm{lock}(\rho) \cap \mathrm{lock}(\rho') = \emptyset \\ (\,\mathrm{rd}(\rho') \cup \mathrm{wr}(\rho')\,) \cap \mathrm{wr}(\rho\ ) &=& \emptyset \qquad \mathrm{mem}(\rho) \cap \mathrm{mem}(\rho') = \emptyset \end{array}$$

The stateful model $\pm_S$ is the following asynchronous graph: its nodes are the machine states in **MState**, its transitions are of the form

$$(\mu, L) \xrightarrow{m} (\mu', L') \quad \text{or} \quad (\mu, L) \xrightarrow{m} \lightning$$

corresponding to the machine steps, defined in §??. The asynchronous tiles of $\pm_S$ are the squares of the form

$$\mathfrak{s} \xrightarrow{m} \mathfrak{s}_1 \xrightarrow{m'} \mathfrak{s}' \quad \sim \quad \mathfrak{s} \xrightarrow{m'} \mathfrak{s}_2 \xrightarrow{m} \mathfrak{s}'$$

where their footprints are independent in the sense above.

The stateless model A lock footprint

$$\rho \in \wp(\mathit{Locks}) \times \wp(\mathbf{Loc})$$

is made of a set of locks $\mathrm{lock}(\rho)$ and a set of locations $\mathrm{mem}(\rho)$. Two such footprints are *independent* when their sets are componentwise disjoint. The stateless model $\pm_L$ is defined in the following way: its nodes are the subsets of *Locks*, and its transitions are all the edges of the form (note the non-determinism)

$$\begin{array}{lll} L \xrightarrow{P(r)} L \uplus \{r\} & L \xrightarrow{\texttt{alloc}(\ell)} L & L \xrightarrow{\tau} L \\ L \uplus \{r\} \xrightarrow{V(r)} L & L \xrightarrow{\texttt{dispose}(\ell)} L & L \xrightarrow{m} \lightning \end{array}$$

where m is a *lock instruction* of the form:

$$P(r) \mid V(r) \mid \texttt{alloc}(\ell) \mid \texttt{dispose}(\ell) \mid \tau$$

for $\ell \in \mathbf{Loc}$ and $r \in \mathit{Locks}$. The purpose of these transitions is to extract from each instruction of the machine its synchronization behavior. An important special case, the transition τ represents the absence of any synchronization mechanism in an instruction like $x := E$, $x := [E]$ or $[E] := E'$. The asynchronous tiles of $\pm_L$ are the squares of the form

$$L \xrightarrow{x} L_1 \xrightarrow{y} L' \quad \sim \quad L \xrightarrow{y} L_2 \xrightarrow{x} L'$$

when the lock footprints of x and y are independent. It is worth noting that L' may be equal to $\lightning$ in such an asynchronous tile. Note that the asynchronous graph $\pm_L$ is more liberal than $\pm_S$ about which footprints commute, because it only takes into account the locks as well as the allocated and deallocated locations. As explained in the introduction, this mismatch enables us to detect *data races* in the machine as well as in the code.

Remark The last component $\mathrm{mem}(\rho)$ in the machine state footprint as well as in the lock footprint enables us to forbid a deallocation followed by an allocation to happen at the same address without some kind of synchronization, both at the stateful and stateless level. This is consistent with practice, since the malloc implementation would typically synchronize its accesses to the free-list(s) of the different threads.

4 Asynchronous Semantics of Code

In this section, we associate to every program C a pair of asynchronous transition systems $[\![C]\!]_S$ and $[\![C]\!]_L$ over the machine models $\pm_S$ and $\pm_L$ introduced in the previous section. The first interpretation $[\![C]\!]_S$ is *stateful* and describes how each instruction of the program C acts on the memory states and on the locks. The second interpretation $[\![C]\!]_L$ is *stateless* and only remembers the action of the instructions on the locks.

4.1 Asynchronous transition systems (ATSs)

Asynchronous transition systems (ATSs) are specific asynchronous graphs where every transition is either executed by Code or by Frame. We thus start by introducing the following notion:

Definition 4.1 (Asynchronous graph with polarities)**.** An asynchronous graph with polarities is an asynchronous graph $(G, \diamond_G)$ where every transition is assigned a *polarity* Code or Frame. One requires that in every permutation tile $u \cdot v' \diamond_G v \cdot u'$, the two transitions u and u' (symmetrically v and v') have the same polarity.

A path in an asynchronous graph G with polarities is called *Code-proper* when it contains (at least) one Code transition. A node x is called initial in G when there are no Code-proper incoming paths into x, and final when there are no Code-proper outgoing paths from x. The sets of initial and final nodes in G are denoted $\partial_0 G$ and $\partial_1 G$, respectively. The graph G is called *Code-acyclic* when there are no Code-proper cycles, that is, every cycle of the graph G contains only Frame transitions. A set S of nodes of a graph is *forward-closed* when $x \in S$ and $x \to y$ implies that $y \in S$.

Definition 4.2 (ATS)**.** An asynchronous transition system (ATS) is a Code-acyclic asynchronous graph with polarities $(G, \diamond_G)$ equipped with a forward-closed subset $|G| \subseteq \partial_1(G)$ of final nodes. A final node in $|G|$ is called a returning node of the ATS.

Definition 4.3. An *ATS* with machine model $(\pm, \diamond)$ is defined as an ATS $(G, |G|)$ equipped with an asynchronous graph homomorphism

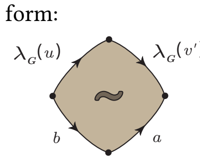
$$\lambda_G \quad : \quad (G, \diamond_G) \longrightarrow (\pm, \diamond)$$

One requires moreover that

1. the map λ_G defines a bijection between the set $\partial_0 G$ of initial nodes and the set of nodes of $\pm$, and an injection from the set $|G|$ of returning nodes into the set of nodes of $\pm$.

2. the map λ_G is a Frame 1-fibration, in the sense that for every transition $v : \lambda_G(x) \to z$ in the machine model $\pm$, there exists a unique Frame transition $u : x \to y$ in G such that $\lambda_G(u) = v : \lambda_G(x) \to \lambda_G(y)$,

3. the map λ_G is a Code-Frame and Frame-Frame 2-fibration, in the sense that for every sequence of transitions $x \xrightarrow{u} y \xrightarrow{v'} z$ in G where $v' : y \to z$ is a Frame transition, and for every permutation tile in $\pm$ of the form:



(14)

there exists a sequence of transitions $x \xrightarrow{v} y' \xrightarrow{u'} z$ and a permutation tile $u \cdot v' \diamond_G v \cdot u'$ in G transported by λ_G to the permutation tile (??) in the sense that

$$\lambda_G(x) \xrightarrow{\lambda_G(v)} \cdot \xrightarrow{\lambda_G(u')} \lambda_G(z) \quad = \quad \lambda_G(x) \xrightarrow{b} \cdot \xrightarrow{a} \lambda_G(z)$$

Notation We often find convenient to label the transitions $u : x \longrightarrow y$ in G with the instruction or lock instruction m which labels the transition $\lambda_G(u)$ in the underlying asynchronous graph $\pm_S$ or $\pm_L$. We also write $m : C$ or $m : F$ to mean that the transition $u : x \longrightarrow y$ has the polarity Code or Frame in G, respectively.

4.2 Basic constructions on ATSs

The asynchronous interpretations $[\![C]\!]_S$ and $[\![C]\!]_L$ of the program C are performed by structural induction, using a number of primitive operations on ATSs defined below. Note that whenever a construction makes some nodes unreachable from the initial nodes, they are implicitly removed.

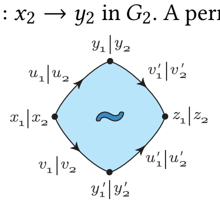
Sum The sum of two ATSs G_1 and G_2 with same machine model $\pm$, written $G_1 \oplus G_2$, is the disjoint union of the two asynchronous graphs G_1 and G_2, where we identify their respective initial and returning states together, when they have the same image under λ_{G_1} and λ_{G_2}. This means that for the case of the returning states, there are three cases. If they both have returning states, we identify $\partial_1(G_1)$ with $\partial_1(G_2)$; if only one of G_1 and G_2 has returning states, we keep this one as our returning states; otherwise the juxtaposition has no returning states.

Sequential composition The sequential composition $G; G'$ of two ATSs G and G' is the disjoint union of G and G' where we identify the returning nodes of G and the initial nodes of G' with the same underlying image under λ_G and $\lambda_{G'}$. Because we remove the inaccessible nodes, when G has no returning nodes, $G; G' = G$.

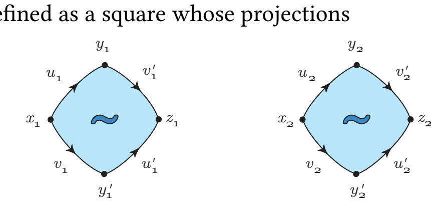
Parallel product The parallel product $G_1 \parallel G_2$ of two ATSs G_1 and G_2 over the same machine model $\pm$ is defined as follows. The nodes of $G_1 \parallel G_2$ are the pairs of nodes $x_1 \mid x_2 \in G_1 \times G_2$ such that $\lambda_{G_1}(x_1) = \lambda_{G_2}(x_2)$ and $\lambda_{G_1 \parallel G_2}(x_1, x_2)$ is defined to be that common value. The transitions of $G_1 \parallel G_2$ are of three kinds:

1. the Code transitions $x_1|x_2 \xrightarrow{m:C} x_1'|x_2'$ where
$$x_1 \xrightarrow{m:C} x_1' \text{ in } G_1 \qquad \text{and} \qquad x_2 \xrightarrow{m:F} x_2' \text{ in } G_2.$$
2. the Code transitions $x_1|x_2 \xrightarrow{m:C} x_1'|x_2'$ where
$$x_1 \xrightarrow{m:F} x_1' \text{ in } G_1 \qquad \text{and} \qquad x_2 \xrightarrow{m:C} x_2' \text{ in } G_2.$$
3. the Frame transitions $x_1|x_2 \xrightarrow{m:F} x_1'|x_2'$ where
$$x_1 \xrightarrow{m:F} x_1' \text{ in } G_1 \qquad \text{and} \qquad x_2 \xrightarrow{m:F} x_2' \text{ in } G_2.$$

Note that every transition $u : x_1|x_2 \longrightarrow y_1|y_2$ in the graph $G_1 \parallel G_2$ is a pair $u = (u_1, u_2)$ also written $u = u_1|u_2$ of a transition $u_1 : x_1 \longrightarrow y_1$ in G_1 and $u_2 : x_2 \longrightarrow y_2$ in G_2. A permutation tile in $G_1|G_2$



is then defined as a square whose projections



define permutation tiles in $(G_1, \diamond_1)$ and $(G_2, \diamond_2)$, respectively. Finally, the returning nodes $x_1|x_2 \in |G_1||G_2|$ are defined as the pairs $x_1|x_2$ of returning nodes $x_1 \in |G_1|$ and $x_2 \in |G_2|$.

The parallel product of G_1 and G_2 is *asynchronous* in the sense that every Code transition in $G_1 \parallel G_2$ is a Code transition performed by G_1 and seen as a Frame transition by G_2, or symmetrically, a Code transition performed by G_2 and seen as a Frame transition by G_1. In particular, by definition, the two components G_1 and G_2 never execute (or "fire") a Code transition simultaneously in $G_1 \parallel G_2$. At the level of permutation tiles, a Code transition $u_1|u_2 : x_1|x_2 \longrightarrow y_1|y_2$ performed in $G_1 \parallel G_2$ by the component G_1 and a Code transition $v_1'|v_2' : y_1|y_2 \longrightarrow z_1|z_2$ performed in $G_1 \parallel G_2$ by the component G_2 define a permutation tile precisely when the transitions $\lambda_{G_1 \parallel G_2}(u_1|u_2) = \lambda_{G_1}(u_1) = \lambda_{G_2}(u_2)$ and $\lambda_{G_1 \parallel G_2}(v_1'|v_2') = \lambda_{G_1}(v_1') = \lambda_{G_2}(v_2')$ define a permutation tile in the underlying machine model $\pm$. As a matter of fact, one purpose of the machine model $(\pm, \diamond)$ is precisely to provide that piece of information necessary to construct the parallel product of G_1 and G_2.

Resource hiding In order to interpret the resource introduction construct `resource` r `do` C, we introduce a *hiding* operator $\mathsf{hide}[r]$ on ATSs which hides the new resource r, similarly to the operator ν in the π-calculus. Formally, if G is an ATS over $\pm(Locks \uplus \{r\})$, then $\mathsf{hide}[r](G)$ is the ATS over $\pm(Locks)$ where: (1) the resource r has been removed from the sets of locked resources of all states, (2) the Code transitions $P(r)$ and $V(r)$ are replaced with nops, (3) the Frame transitions $P(r)$ and $V(r)$ are removed from the graph G, and (4) the remaining permutation tiles are preserved. Moreover, we only keep as initial and returning states the initial and returning states x of G such that the resource r is not held in $\lambda_G(x)$.

Critical sections Dually, inside critical sections, we need to "lift" ATSs over some set *Locks* of locks to ATSs over $Locks \uplus \{r\}$. This can be done naturally in this case because we know that, during the critical section, the resource r is held by the Code. Formally, $\mathsf{when}[r](G)$ has the same underlying asynchronous graph as G, where $\lambda' := \lambda_{\mathsf{when}[r](G)}$ is defined by:

$$\begin{array}{lcll} \lambda'(x) & := & L \uplus \{r\} & \text{if } \lambda_G(x) = L \\ \lambda'(x) & := & (\mu, L \uplus \{r\}) & \text{if } \lambda_G(x) = (\mu, L). \end{array}$$

This does not define an ATS yet, for condition¬**3** is not satisfied: there are not enough Environment transitions. This is why we must freely add Frame transitions and new nodes to make it an ATS. The returning nodes are defined to be the same as G.

Other constructions on ATSs Given an ATS G and a Boolean formula B, we define $\mathsf{whentrue}[B](G)$ as the graph G where, among the Code transitions out of initial nodes, we only keep those where B holds on $\lambda_G(x)$. Then, we remove the nodes made unreachable by this edge removal. Similarly, we define $\mathsf{whenfalse}[B]$ for when B does not hold. Finally, $\mathsf{whenabort}[B]$ is the graph with transitions from the initial states where B errors out, because it tries to read undefined variables, to $\lightning$. Note that in the case of the $[\![C]\!]_L$ semantics, since the nodes do not contain information on the state, the first two constructions above are the identity. This means that we sometimes consider impossible branches.

4.3 Asynchronous semantics of the code

We explain how to give a semantics to any code C as an ATS $[\![C]\!]$, by induction on its structure. This lets us build $[\![C]\!]_S$ and $[\![C]\!]_L$ in the same way. First, we give the syntax of our imperative concurrent language, which we borrow from [? ?].

$$
\begin{aligned}
B &::= \mathbf{true} \mid \mathbf{false} \mid B \wedge B' \mid B \vee B' \mid E = E' \\
E &::= 0 \mid 1 \mid \ldots \mid x \mid E + E' \mid E * E' \\
C &::= x := E \mid x := [E] \mid [E] := E' \mid C; C' \mid C_1 \parallel C_2 \mid \texttt{skip} \\
&\quad \mid \texttt{while}\, B\, \texttt{do}\, C \mid \texttt{resource}\, r\, \texttt{do}\, C \mid \texttt{with}\, r\, \texttt{when}\, B\, \texttt{do}\, C \\
&\quad \mid \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \mid x := \texttt{malloc}(E) \mid \texttt{dispose}(E)
\end{aligned}
$$

Semantics of instructions To every instruction $m \in \mathbf{Instr}$, we associate the ATS $[\![m]\!]$ with machine model $\mathbb{M}$ defined as two copies $\mathbb{M}_0 + \mathbb{M}_1$ (called *source* and *target*) of the asynchronous graph $\mathbb{M}$. Every transition in $\mathbb{M}_0 + \mathbb{M}_1$ is assigned the Frame polarity. To this, one adds a Code transition $x_0 \to y_1$ for every transition of the form (??) labeled by m in the small step semantics. Here, x_0 and y_1 are the nodes x and y of $\mathbb{M}$ taken in the source and target components $\mathbb{M}_0$ and $\mathbb{M}_1$ of $[\![m]\!]$, respectively. The transition $x_0 \to y_1$ is mapped by $\lambda_{[\![m]\!]}$ to the transition associated to the small step transition (??) in $\mathbb{M} = \mathbb{M}_S$ or $\mathbb{M} = \mathbb{M}_L$. Finally, one adds a Code-Frame permutation tile in $[\![m]\!]$ for each Code-Frame permutation tile in $\mathbb{M}$, in such a way that $\lambda_{[\![m]\!]} : [\![m]\!] \to \mathbb{M}$ defines a Code-Frame 2-fibration.

Leaf codes For leaf codes that correspond to instructions (all, except for `malloc`), their semantics is the same as that of the instruction. For $\texttt{malloc}(E)$, we take the non-deterministic union of all the $\texttt{alloc}(E, \ell)$:

$$[\![\texttt{malloc}(E)]\!] \quad := \quad \bigoplus_{\ell \in \mathbf{Loc}} [\![\texttt{alloc}(E, \ell)]\!]$$

Conditionals Conditional branching is interpreted as

$$
\begin{aligned}
[\![\texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C_1]\!] = {} & \mathsf{whentrue}[B]([\![\mathsf{nop}]\!])\,;[\![C_1]\!] \\
& \oplus \mathsf{whenfalse}[B]([\![\mathsf{nop}]\!])\,;[\![C_2]\!] \\
& \oplus \mathsf{whenabort}[B]
\end{aligned}
$$

The nops are needed because the environment can interfere between the evaluation of B and the beginning of the execution of C_i.

Sequential and parallel compositions We use the sequential and parallel product of ATSs with machine models defined in §??, in the following way:

$$[\![C_1 \| C_2]\!] \;=\; [\![C_1]\!] \| [\![C_2]\!] \qquad [\![C_1; C_2]\!] \;=\; [\![C_1]\!]; [\![C_2]\!].$$

Resource introduction The interpretation of $\texttt{resource}\, r\, \texttt{do}\, C$ is defined as

$$[\![\texttt{resource}\, r\, \mathsf{do}\, C]\!] \quad = \quad \mathsf{hide}[r]([\![C]\!])$$

Critical sections The semantics $[\![\texttt{with}\, r\, \mathsf{when}\, B\, \mathsf{do}\, C]\!]$ is defined using the sequential composition above and whentrue:

$$\mathsf{whentrue}[B]\Big([\![P(r)]\!]; \mathrm{when}[r]([\![C]\!]); [\![V(r)]\!]\Big) \oplus \mathsf{whenabort}[B].$$

Loops For loops, the interpretation of $C' = \texttt{while}\, B\, \mathsf{do}\, C$ is defined as the (possibly infinite) least fixpoint of the function F:

$$
\begin{aligned}
F(G) = {} & \mathsf{whentrue}[B]([\![\mathsf{nop}]\!]); [\![C]\!]; G \oplus \mathsf{whenfalse}[B]([\![\mathsf{nop}]\!]) \\
& \oplus \mathsf{whenabort}[B].
\end{aligned}
$$

Remark The map λ_G is a 2-fibration for Code-Frame and Frame-Frame permutations, but not for Code-Code permutations in general. Consider for instance the interpretation of the program

$$C = \texttt{resource}\, r\, \mathsf{do}\, \big\{ (P(r); V(r)) \parallel (P(r); V(r)) \big\}$$

where $r \in \mathbf{LockName}$ is a resource name. Since the resource introduction performed by $\texttt{resource}\, r\, \mathsf{do}\, C$ is interpreted by hiding the resource r, the two instructions $P(r)$ and $V(r)$ are both transformed in nops instructions. However the two nop*s do not* form a tile! Another example is, of course, the sequential composition $C_1; C_2$ of two codes C_1 and C_2.

4.4 Comparing the stateful and the stateless semantics

We construct a category of ATSs with machine models, in the following way. A *morphism* between ATSs with machine models

$$\lambda_{G_1} : G_1 \longrightarrow \mathbb{M}_1 \qquad \lambda_{G_2} : G_2 \longrightarrow \mathbb{M}_2$$

is a pair of asynchronous graph morphisms $\mathscr{F} : \mathbb{M}_1 \to \mathbb{M}_2$ and $\mathscr{G} : G_1 \to G_2$ such that the diagram below commutes:

$$
\begin{array}{ccc}
G_1 & \xrightarrow{\quad \mathscr{G} \quad} & G_2 \\
{\scriptstyle \lambda_{G_1}} \big\downarrow & & \big\downarrow {\scriptstyle \lambda_{G_2}} \\
\mathbb{M}_1 & \xrightarrow{\quad \mathscr{F} \quad} & \mathbb{M}_2
\end{array}
$$

One requires moreover that $\mathscr{G}$ send initial (resp. returning) nodes of G_1 to initial (resp. returning) nodes of G_2. This defines a category noted **ATS**. Let $\mathscr{F} : \mathbb{M}_S \to \mathbb{M}_L$ denote the asynchronous graph morphism which transports every machine state $\mathfrak{s} = (\mu, L)$ to the underlying subset $L \subseteq Locks$ of locks held in $\mathfrak{s}$. Every instruction $m \in \mathbf{Instr}$ comes equipped with an ATS morphism

$$\mathscr{L}_m = (\mathscr{F}, \mathscr{G}_m) \quad : \quad [\![m]\!]_S \longrightarrow [\![m]\!]_L$$

where the asynchronous graph morphism $\mathscr{G}_m$ is defined as

$$(\mu, L) \xrightarrow{m} (\mu', L') \quad \longmapsto \quad L \xrightarrow{m} L'$$

Because the stateful and stateless interpretations $[\![-]\!]_S$ and $[\![-]\!]_L$ are defined using the same functorial operations over $\mathscr{F}$, we can associate to every code C a morphism of **ATS**

$$\mathscr{L}_C = (\mathscr{F}, \mathscr{G}_C) \quad : \quad [\![C]\!]_S \longrightarrow [\![C]\!]_L$$

starting from the family of morphisms $\mathscr{L}_m$ associated to instructions. Note that this morphism $\mathscr{L}_C = (\mathscr{F}, \mathscr{G}_C)$ living in the category **ATS** plays a fundamental role in the present work, since our asynchronous refinement of the original Soundness Theorem for CSL relies on it, see §?? for details.

5 Logical States

As discussed in [?], reasoning about concurrent programs in separation logic requires to introduce an appropriate notion of *logical state*, including information about permissions. The version of concurrent separation logic we consider is almost the same as its original formulation by O'Hearn and Brookes [? ?]. One difference is that we benefit from the work of Bornat, Calcagno, O'Hearn, Parkinson and Yang in [? ? ?] and use permissions p and the predicate $\mathrm{Own}_p(x)$ in order to handle the heap as well as variables in the stack. We suppose given an arbitrary partial cancellative commutative monoid **Perm** which we call the permission monoid, following [?]. The element $\top$ will be used as the permission required for a program to write somewhere in memory. We thus require that $\top$ does not admit any multiples, ie. $\forall x \in \mathbf{Perm}$, $\top \cdot x$ is not defined.

The intuition (which we will need to turn into a theorem) is that we prevent in this way concurrent mutation and observation of the same location, that is, data races. The set **LState** of *logical states* is defined in much the same way as the set **State** of memory states, with the addition of permissions:

$$\mathbf{LState} \;=\; (\mathbf{Var} \rightharpoonup_{fin} (\mathbf{Val} \times \mathbf{Perm})) \times (\mathbf{Loc} \rightharpoonup_{fin} (\mathbf{Val} \times \mathbf{Perm}))$$

The main benefit of permissions is that they enable us to define a *separation product* $\sigma * \sigma'$ between two logical states σ and σ', which generalizes the disjoint union. When it is defined, the logical state $\sigma * \sigma'$ is defined as a partial function with domain

$$\mathrm{dom}(\sigma * \sigma) = \mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma')$$

in the following way: for $a \in \mathbf{Var} \amalg \mathbf{Loc}$,

$$\sigma * \sigma'(a) = \begin{cases} \sigma(a) & \text{if } a \in \mathrm{dom}(\sigma) \setminus \mathrm{dom}(\sigma') \\ \sigma'(a) & \text{if } a \in \mathrm{dom}(\sigma') \setminus \mathrm{dom}(\sigma) \\ (v, p \cdot p') & \text{if } \sigma(a) = (v,p) \text{ and } \sigma'(a) = (v,p') \end{cases}$$

The separation product $\sigma * \sigma'$ of the two logical states σ and σ' is not defined otherwise. In particular, the memory states underlying σ and σ' agree on the values of the shared variables and heap locations when the separation product is well defined. The syntax and the semantics of the *formulas* of Concurrent Separation Logic is the same as in Separation Logic. The grammar of formulas is:

$$\begin{aligned} P, Q, R, J ::= {} & \mathbf{emp} \mid \mathbf{true} \mid \mathbf{false} \mid P \vee Q \mid P \wedge Q \mid \neg P \mid \ \forall v.P \mid \ \exists v.P \\ & \mid P * Q \mid v \overset{p}{\mapsto} w \mid \mathrm{Own}_p(x) \mid E_1 = E_2 \end{aligned}$$

where $x \in \mathbf{Var}, p \in \mathbf{Perm}, v, w \in \mathbf{Val}$. Given a logical state $\sigma = (s, h)$ consisting of a logical stack s and of a logical heap h, the semantics of the formulas, expressed as the predicate $\sigma \models P$, is standard:

$$\begin{aligned} \sigma \models v \overset{p}{\mapsto} w &\iff v \in \mathbf{Loc} \wedge s = \emptyset \wedge h = [v \mapsto (w,p)] \\ \sigma \models \mathrm{Own}_p(x) &\iff \exists v \in \mathbf{Val}, s = [x \mapsto (v,p)] \wedge h = \emptyset \\ \sigma \models E_1 = E_2 &\iff [\![E_1]\!] = [\![E_2]\!] \wedge \mathrm{fv}(E_1 = E_2) \subseteq \mathrm{vdom}(s) \\ \sigma \models P \wedge Q &\iff \sigma \models P \text{ and } \sigma \models Q \\ \sigma \models P * Q &\iff \exists \sigma_1 \sigma_2,\ \sigma = \sigma_1 * \sigma_2 \text{ and } \sigma_1 \models P \text{ and } \sigma_2 \models Q. \end{aligned}$$

The *proof system* underlying concurrent separation logic is a sequent calculus, whose sequents are *Hoare triples* of the form

$$\Gamma \vdash \{P\} C \{Q\}$$

where $C \in \mathbf{Code}$, P, Q are predicates, and Γ is a context, defined as a partial function with finite domain from the set **LockName** of resource variables to predicates. Intuitively, the context $\Gamma = r_1 : J_1, \ldots, r_k : J_k$ describes the *invariant* J_i satisfied by the resource variable r_i. The purpose of these resources is to describe the fragments of memory shared between the various threads during the execution.

The more interesting inference rules of CSL are given in Figure **??**. The inference rule Res associated to `resource` r `do` C moves a piece of logical state which is owned by the Code into the shared context Γ, which means that it can be accessed concurrently inside the code C. However, the access to that piece of state is mediated by the `with` construct, which grants temporary access under the condition that one must give it back (rule With). Note that the rule With has the side condition $P \Rightarrow \mathrm{def}(B)$. This means that if P is true in some logical state, then it implies, for each free variable x of B, that there exists some permission p such that $\mathrm{Own}_p(x)$ holds.

6 The machine model of separated states

We recall the notion of *separated state* formulated in [?] whose purpose is to separate the logical memory state into one region controlled by the Code, one region controlled by the Frame, and one independent region for each unlocked resource. In order to define the notion, we suppose given a finite set $Locks \subseteq \mathbf{LockName}$ of resource variables, or locks.

Definition 6.1. A separated state is a triple

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \in \mathbf{LState} \times (Locks \to \mathbf{LState} + \{C, F\}) \times \mathbf{LState}$$

such that the logical state below is defined:

$$\sigma_C \,*\, \Big\{ \underset{r \in \mathrm{dom}(\boldsymbol{\sigma})}{\circledast} \boldsymbol{\sigma}(r) \Big\} \,*\, \sigma_F \quad \in \quad \mathbf{LState} \tag{15}$$

where

$$\begin{aligned} \mathrm{dom}(\boldsymbol{\sigma}) &= \{\, r \in Locks \mid \boldsymbol{\sigma}(r) \in \mathbf{LState} \,\}, \\ \mathrm{dom_C}(\boldsymbol{\sigma}) &= \{\, r \in Locks \mid \boldsymbol{\sigma}(r) = C \,\}, \\ \mathrm{dom_F}(\boldsymbol{\sigma}) &= \{\, r \in Locks \mid \boldsymbol{\sigma}(r) = F \,\}. \end{aligned}$$

We say that a separated state $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ combines into a machine state $\mathfrak{s} = (\mu, L)$ precisely when $L = \mathrm{dom_C}(\boldsymbol{\sigma}) \uplus \mathrm{dom_F}(\boldsymbol{\sigma})$ and when the function $U : \mathbf{LState} \to \mathbf{State}$ which forgets the permissions transports the logical state (**??**) into the memory state $\mu \in \mathbf{State}$. Note that, by definition, every separated state $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ combines into a unique machine state, which we write for concision

$$(\mu, L) \quad = \quad \circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F). \tag{16}$$

Interestingly, the notion of separated state comes with the same notion of footprint as the machine states, defined as elements of

$$\rho \in \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(Locks) \times \wp(\mathbf{Loc}).$$

which describes the footprint of a transition by Eve or Adam.

Definition 6.2. The machine model of separated states $\pm_{Sep}$ is the asynchronous graph whose nodes are the separated states and whose edges are either Adam or Eve transitions:

- Eve transitions are of the form

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \xrightarrow{\ m : C\ } (\sigma'_C, \boldsymbol{\sigma}', \sigma_F)$$

where $m \in \mathbf{Instr}$ is an instruction such that

$$\circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \overset{m}{\rightsquigarrow} \circledast(\sigma'_C, \boldsymbol{\sigma}', \sigma_F)$$

and such that the following conditions are satisfied:

$$\begin{gathered} \forall \ell \notin \mathrm{wr}(m),\ \sigma_C(\ell) = \sigma'_C(\ell) \qquad \mathrm{wr}(m) \cup \mathrm{rd}(m) \subseteq \mathrm{dom}(\sigma_C) \\ \mathrm{lock}(m) \subseteq \mathrm{dom}(\boldsymbol{\sigma}) \cup \mathrm{dom_C}(\boldsymbol{\sigma}) \qquad \forall r \notin \mathrm{lock}(m),\ \boldsymbol{\sigma}(r) = \boldsymbol{\sigma}'(r). \end{gathered}$$

- Adam moves of the form

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \xrightarrow{\ m : F\ } (\sigma_C, \boldsymbol{\sigma}', \sigma'_F)$$

where $m \in \mathbf{Instr}$ is an instruction, such that

$$\circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \overset{m}{\rightsquigarrow} \circledast(\sigma_C, \boldsymbol{\sigma}', \sigma'_F)$$

and such that the following conditions are satisfied:

$$\begin{gathered} \forall \ell \notin \mathrm{wr}(m),\ \sigma_F(\ell) = \sigma'_F(\ell) \qquad \mathrm{wr}(m) \cup \mathrm{rd}(m) \subseteq \mathrm{dom}(\sigma_F) \\ \mathrm{lock}(m) \subseteq \mathrm{dom}(\boldsymbol{\sigma}) \cup \mathrm{dom_F}(\boldsymbol{\sigma}) \qquad \forall r \notin \mathrm{lock}(m),\ \boldsymbol{\sigma}(r) = \boldsymbol{\sigma}'(r). \end{gathered}$$

The permutation tiles of the asynchronous graph $\pm_{Sep}$ are defined in the expected way. The resulting definition of the machine model $\pm_{Sep}$ of separated states ensures that the operation (**??**) defines a morphism $\circledast : \pm_{Sep} \to \pm_S$ of asynchronous graphs from $\pm_{Sep}$ to the stateful model $\pm_S$.

$$\frac{}{\Gamma \vdash \{E \mapsto -\}[E] := E'\{E \mapsto E'\}}\ \textsc{Store} \qquad \frac{\Gamma \vdash \{P\}C\{Q\}}{\Gamma \vdash \{P * R\}C\{Q * R\}}\ \textsc{Frame} \qquad \frac{\Gamma, r : J \vdash \{P\}C\{Q\}}{\Gamma \vdash \{P * J\}\texttt{resource}\ r\ \texttt{do}\ C\{Q * J\}}\ \textsc{Res}$$

$$\frac{\Gamma \vdash \{P_1\}C\{Q_1\} \quad \Gamma \vdash \{P_2\}C\{Q_2\}}{\Gamma \vdash \{P_1 \vee P_2\}C\{Q_1 \vee Q_2\}}\ \textsc{Disj} \qquad \frac{\Gamma \vdash \{P_1\}C_1\{Q_1\} \quad \Gamma \vdash \{P_2\}C_2\{Q_2\}}{\Gamma \vdash \{P_1 * P_2\}C_1 \parallel C_2\{Q_1 * Q_2\}}\ \textsc{Par} \qquad \frac{P \Rightarrow \mathrm{def}(B) \quad \Gamma \vdash \{(P * J) \wedge B\}C\{Q * J\}}{\Gamma, r : J \vdash \{P\}\texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\{Q\}}\ \textsc{With}$$

Figure 1. Inference rules of Concurrent Separation Logic

7 An asynchronous semantics of proofs

In this section, we interpret derivation trees (or proofs) of CSL in our asynchronous semantics. In the same way as we did for the Code in §??, we interpret every proof π of a Hoare triple $\Gamma \vdash \{P\}C\{Q\}$ as an asynchronous transition system (ATS, Definition ??). The underlying asynchronous machine model is $\pm_{Sep}$, the graph of separated states. As in the previous case, our ATSs have the 1-fibration property, and moreover the initial states are all the states that satisfy P, and all the final states satisfy Q. The interpretation $[\![\pi]\!]_{Sep}$ also satisfies that the second component $\boldsymbol{\sigma}$ of all its nodes of $[\![\pi]\!]_{Sep}$ satisfies the invariants of Γ pointwise. In order to define the interpretation of a proof π by induction on its structure, we start by defining a small number of new constructions on ATSs.

The parallel product with separated states In order to define the parallel product $G_1 \| G_2$ of two ATSs on the model $\pm_{Sep}$ of separated states, we need to adapt the compatibility condition given by the equality $\lambda_G(x_1) = \lambda_G(x_2)$ in the case of the stateful and stateless models $\pm_S$ and $\pm_L$. In the case of $\pm_{Sep}$, two nodes of G_1 and G_2 should be declared compatible when they describe the two "subjective" (and dual) views of the same situation, provided in this case by a separated state of $G_1 \| G_2$. This leads us to the notion of *three-party* separated state, defined as a tuple $(\sigma_1, \sigma_2, \boldsymbol{\sigma}^*, \sigma_F)$ where $\sigma_1, \sigma_2, \sigma_F \in \mathbf{LState}$ are logical states, where $\boldsymbol{\sigma}^* : Locks \to \mathbf{LState} + \{C_1, C_2, F\}$, and where the product $\circledast(\sigma_1, \sigma_2, \boldsymbol{\sigma}^*, \sigma_F)$ immediately adapted from (??) is well-defined.

We define three functions on these new separated states: the "objective" projection, which corresponds to the view of the whole program $C_1 \parallel C_2$, is defined by:

$$(\sigma_1, \sigma_2, \boldsymbol{\sigma}^*, \sigma_C) \mapsto (\sigma_1 * \sigma_2, \boldsymbol{\sigma}^*[C_i \mapsto C], \sigma_C)$$

the left and right "subjective" projections

$$\lambda_1 : (\sigma_1, \sigma_2, \boldsymbol{\sigma}^*, \sigma_C) \mapsto (\sigma_1, \boldsymbol{\sigma}^*[C_1 \mapsto C, C_2 \mapsto F], \sigma_C * \sigma_2)$$
$$\lambda_2 : (\sigma_1, \sigma_2, \boldsymbol{\sigma}^*, \sigma_C) \mapsto (\sigma_2, \boldsymbol{\sigma}^*[C_1 \mapsto F, C_2 \mapsto C], \sigma_C * \sigma_1)$$

which give the state of the program from the point of view of one of the programs in parallel. This leads us to the following definition:

Definition 7.1. Two separated states $x_1, x_2 \in \mathbf{SState}$ are *compatible* when there exists a three-party separated state y such that $\lambda_1(y) = x_1$ and $\lambda_2(y) = x_2$. Note that the three-party separated state y is unique in that case.

Framing To handle framing (Frame rule), we need to be able to move a piece of (logical) heap from the Frame side to the Code side. First, we define the framing of a logical state σ_R. Given an ATS G over $\pm_{Sep}$, we define frame$[\sigma_R](G)$ pointwise with:

$$\lambda'(x) = \begin{cases} (\sigma_C * \sigma_R, \boldsymbol{\sigma}, \sigma_F) & \text{if } \lambda(x) = (\sigma_C, \boldsymbol{\sigma}, \sigma_F * \sigma_R) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Given such a graph G and a predicate R, we define frame$[R](G)$ as the following union of ATSs

$$\mathrm{frame}[R](G) \quad = \quad \bigcup_{\sigma_R \vDash R} \mathrm{frame}[\sigma_R](G)$$

Resource introduction To give a semantics to the resource introduction rule, we need to extend the hiding operator to separated states. More precisely, the action of hide$[r]$, in addition to replacing $P(r)$ and $V(r)$ with nop, is:

$$(\sigma_C, \boldsymbol{\sigma} \uplus [r \mapsto \sigma], \sigma_F) \quad \mapsto \quad (\sigma_C * \sigma, \boldsymbol{\sigma}, \sigma_F)$$
$$(\sigma_C, \boldsymbol{\sigma} \uplus [r \mapsto C \text{ or } F], \sigma_F) \quad \mapsto \quad (\sigma_C, \boldsymbol{\sigma}, \sigma_F)$$

Critical sections Similarly, we extend the definition of when$[r](G)$ to separated states: to the same underlying graph we associate the asynchronous morphism λ' defined as

$$\lambda'(x) = (\sigma_C, \boldsymbol{\sigma} \uplus [r \mapsto C], \sigma_F) \quad \text{if } \lambda_G(x) = (\sigma_C, \boldsymbol{\sigma}, \sigma_F)$$

We also need to take and release locks in the semantics of the proofs: the ATS acquire$[r]$ is defined by its Eve moves:

$$(\sigma_C, \boldsymbol{\sigma} \uplus [r \mapsto \sigma], \sigma_F) \xrightarrow{P(r):C} (\sigma_C * \sigma, \boldsymbol{\sigma} \uplus [r \mapsto C], \sigma_F)$$

and release$[r]$ by (for all $\sigma \in \mathbf{LState}$ satisfying r's invariant in Γ):

$$(\sigma_C * \sigma, \boldsymbol{\sigma} \uplus [r \mapsto C], \sigma_F) \xrightarrow{V(r):C} (\sigma_C, \boldsymbol{\sigma} \uplus [r \mapsto \sigma], \sigma_F).$$

Machine instructions The rules that correspond to machine instructions $m \in \mathbf{Instr}$ (such as Load) are interpreted in the obvious way, always preserving the permission associated to affected locations.

Semantics of proofs

$$\left[\!\!\left[\frac{\overset{\vdots\, \pi_1}{\Gamma \vdash \{P\}C_1\{Q\}} \quad \overset{\vdots\, \pi_2}{\Gamma \vdash \{Q\}C_2\{R\}}}{\Gamma \vdash \{P\}C_1; C_2\{R\}} \right]\!\!\right]_{Sep} \quad = \quad [\![\pi_1]\!]_{Sep}\, ;\, [\![\pi_2]\!]_{Sep}$$

For the parallel product rule Par, we use the parallel product of ATSs using the above notion of *compatibility*:

$$\left[\!\!\left[\frac{\overset{\vdots\, \pi_1}{\Gamma \vdash \{P_1\}C_1\{Q_1\}} \quad \overset{\vdots\, \pi_2}{\Gamma \vdash \{P_2\}C_2\{Q_2\}}}{\Gamma \vdash \{P_1 * P_2\}C_1 \parallel C_2\{Q_1 * Q_2\}} \right]\!\!\right]_{Sep} \quad = \quad [\![\pi_1]\!]_{Sep} \parallel [\![\pi_2]\!]_{Sep}$$

$$\left[\!\!\left[\frac{\overset{\vdots\, \pi}{\Gamma, r : J \vdash \{(P * J) \wedge B\}C\{Q * J\}}}{\Gamma, r : J \vdash \{P\}\texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\{Q\}} \right]\!\!\right]_{Sep} \quad =$$
$$\mathrm{whentrue}[B]\big(\mathrm{acquire}[r]\big)\, ;\, \mathrm{when}[r]\big([\![\pi]\!]_{Sep}\big)\, ;\, \mathrm{release}[r]$$

$$\left[\!\!\left[\frac{\overset{\vdots\, \pi_1}{\Gamma \vdash \{P_1\}C\{Q_1\}} \quad \overset{\vdots\, \pi_2}{\Gamma \vdash \{P_2\}C\{Q_2\}}}{\Gamma \vdash \{P_1 \vee P_2\}C\{Q_1 \vee Q_2\}} \right]\!\!\right]_{Sep} \quad = \quad [\![\pi_1]\!]_{Sep} \cup [\![\pi_2]\!]_{Sep}$$

The semantics of FRAME and RES are defined in the obvious way using frame$[R]$ and hide$[r]$; and the semantics of IF and WHILE are interpreted the same way as for the code. One can give a semantics to the proof rule for conjunction, which requires precision of the invariants.

8 An asynchronous soundness theorem

At this stage, we are ready to state our soundness theorem for Concurrent Separation Logic. We start by observing that every proof π in CSL of a Hoare triple of the form $\Gamma \vdash \{P\}C\{Q\}$ comes equipped with a morphism of asynchronous graphs

$$\mathscr{S}_\pi \quad : \quad [\![\pi]\!]_{Sep} \longrightarrow [\![C]\!]_S$$

which makes the diagram below commute

$$\begin{array}{ccc} [\![\pi]\!]_{Sep} & \xrightarrow{\quad \mathscr{S}_\pi \quad} & [\![C]\!]_S \\ \lambda_\pi \downarrow & & \downarrow \lambda_C \\ \pitchfork_{Sep} & \xrightarrow{\quad \circledast \quad} & \pitchfork_S \end{array}$$

The morphism $\mathscr{S}_\pi$ thus defines a morphism of ATS which relates the interpretation of the proof π with the stateful interpretation of C. For every such proof π of a Hoare triple $\Gamma \vdash \{P\}C\{Q\}$, we write

$$\mathscr{L}_\pi \quad : \quad [\![\pi]\!]_{Sep} \quad \longrightarrow \quad [\![C]\!]_L$$

for the composite $\mathscr{L}_\pi = \mathscr{L}_C \circ \mathscr{S}_\pi$ below:

$$[\![\pi]\!]_{Sep} \xrightarrow{\quad \mathscr{S}_\pi \quad} [\![C]\!]_S \xrightarrow{\quad \mathscr{L}_C \quad} [\![C]\!]_L$$

Our soundness theorem follows from two properties of the asynchronous strategy $[\![\pi]\!]_{Sep}$ associated to a CSL proof tree π. The first property (*1-soundness*) implies that a well-specified program does not crash during a *valid execution*, that is, an execution which starts from a state satisfying the precondition P and where the Frame performs only legal transitions. The second property (*2-soundness*) implies that such a program does not encounter any data race.

Theorem 8.1 (1-soundness)**.** *$\mathscr{S}_\pi$ is a Code 1-fibration.*

A *Code 1-fibration* is a 1-fibration (Def. **??**) where we only ask that Code transitions can be lifted, similarly to axiom **3** of Def. **??**. This lifting property reflects the fact that the strategy $[\![\pi]\!]_{Sep}$ interpreting the proof π is *winning*, in the sense that every transition performed by the Code on machine states can be lifted (and thus logically justified) by the strategy into a transition between separated states, see [**?**] for a discussion. This implies in particular that *well specified programs do not go wrong*, because the error state $\lightning$ cannot be lifted to a separated state. The next statement is of a different nature: it says that the strategy $[\![\pi]\!]_{Sep}$ adapts at the *separated* level to the possible reorderings of scheduling performed at the *stateless* level:

Theorem 8.2 (2-soundness)**.** *$\mathscr{L}_\pi$ is a 2-fibration.*

This property implies (in particular) that valid executions of C never produce data races. More deeply, it says that two executions which are equivalent modulo $\approx$ at the stateless level, in the sense that they behave in the same way with respect to the locks (each thread acquires and releases each lock in the same order), are also equivalent modulo $\sim$ at the stateful level. To make this statement formal, consider a well-specified program $\emptyset \vdash \{P\}C\{Q\}$ and define $[\![\{P\}C]\!]_S^\tau$ as the subgraph of $[\![C]\!]_S$ obtained by removing every Frame transition, and keeping only the states which can be reached from an initial node satisfying P. We are interested in the morphism

$$\mathscr{L}_C^P \quad : \quad [\![\{P\}C]\!]_S^\tau \quad \longrightarrow \quad [\![C]\!]_L$$

obtained by restricting $\mathscr{L}_C$ to the asynchronous subgraph $[\![\{P\}C]\!]_S^\tau$ of $[\![C]\!]_S$. This enables us to establish the soundness theorem:

Theorem 8.3 (Soundness)**.** *$\mathscr{L}_C^P$ is a 2-fibration.*

9 Conclusion and future works

For the first time, we devise and establish a properly asynchronous version of the Soundness Theorem for Concurrent Separation Logic (CSL). In our formulation, the absence of data races follows from a more fundamental lifting property of scheduling along the stateful-to-stateless translation $[\![C]\!]_S \to [\![C]\!]_L$. The proof of the theorem itself is original in design, and relies on the construction of an asynchronous game semantics of CSL, building on the foundations set in [**?**]. In future work, we wish to adapt this asynchronous semantics of CSL to weak memory models [**? ? ? ? ?**] and to distributed algorithms [**?**]. Another extension would be extending our version of the soundness theorem to a higher-order and axiomatic setting like Iris [**?**]. Also, now that the asynchronous soundness theorem has been established by semantic means, a nice and instructive challenge will be to prove it again using purely syntactic techniques, in the line adopted for Mezzo [**?**].

Acknowledgments The authors are grateful to Richard Bornat, Stephen Brookes, Tony Hoare, François Pottier and Viktor Vafeiadis for discussions at an early stage of this work.

References

[] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2014. Type Soundness and Race Freedom for Mezzo. In *FLOPS*.
[] Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *POPL*.
[] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. 2006. Variables as Resource in Separation Logic. *ENTCS* 155 (2006).
[] Gérard Boudol and Gustavo Petri. 2009. Relaxed Memory Models: An Operational Approach *(POPL)*.
[] Stephen Brookes. 2004. A semantics for concurrent separation logic. In *CONCUR*.
[] Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. 2016. *Directed Algebraic Topology and Concurrency*. Springer.
[] Alexey Gotsman, Josh Berdine, and Byron Cook. 2011. Precision and the Conjunction Rule in Concurrent Separation Logic. MFPS.
[] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. ’Cause I’m strong enough: reasoning about consistency choices in distributed systems. In *POPL*.
[] Jonathan Hayman and Glynn Winskel. 2008. Independence and Concurrent Separation Logic. *LMCS* (2008).
[] 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 *POPL*.
[] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*.
[] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*.
[] Ryan Kavanagh and Stephen Brookes. 2017. A Denotational Semantics for SPARC TSO *(MFPS)*.
[] Paul-André Melliès and Samuel Mimram. 2007. Asynchronous Games: Innocence Without Alternation. In *CONCUR*.
[] Paul-André Melliès. 2017. *Une étude micrologique de la négation*. HDR.
[] Paul-André Melliès and Léo Stefanesco. 2017. A Game Semantics for Concurrent Separation Logic. In *MFPS*.
[] Peter W. O’Hearn. 2007. Resources, Concurrency, and Local Reasoning. *TCS* 375 (2007).
[] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. 2006. Variables as Resource in Hoare Logics. In *LICS*.
[] Vaughn Pratt. 1991. Modeling Concurrency with Geometry *(POPL)*.
[] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. ESOP.
[] Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *ENTCS* 276 (2011).