

A Game Semantics of Concurrent Separation Logic

Paul-André Melliès^a Léo Stefanescu^b

^a *IRIF, CNRS, Université Paris Diderot*

^b *École Normale Supérieure de Lyon*

Abstract

In this paper, we develop a game-theoretic account of concurrent separation logic. To every execution trace of the Code confronted to the Environment, we associate a specification game where Eve plays for the Code, and Adam for the Environment. The purpose of Eve and Adam is to decompose every intermediate machine state of the execution trace into three pieces: one piece for the Code, one piece for the Environment, and one piece for the available shared resources. We establish the soundness of concurrent separation logic by interpreting every derivation tree of the logic as a winning strategy of this specification game.

1 Introduction

Concurrent separation logic (CSL) is an extension of Reynold’s separation logic [11] formulated by O’Hearn [9] to establish the correctness of concurrent imperative programs with shared memory and locks. This specification logic enables one to establish the good behavior of these programs in an elegant and modular way, thanks to the frame rule of separation logic. A sequent of concurrent separation logic

$$r_1 : P_1, \dots, r_n : P_n \vdash \{P\} C \{Q\}$$

consists of a Hoare triple $\{P\}C\{Q\}$ together with a context $\Gamma = r_1 : P_1, \dots, r_n : P_n$ which declares a number of *resource variables* r_k (or mutexes) together with the CSL formula P_k which they satisfy as invariant. The validity of the program logic relies on a soundness theorem, which states that the existence of a derivation tree in concurrent separation logic

$$\frac{\begin{array}{c} \pi \\ \vdots \end{array}}{r_1 : P_1, \dots, r_n : P_n \vdash \{P\} C \{Q\}}$$

ensures (1) that the concurrent program C will not produce any race condition at execution time, and (2) that the program C will transform every initial state satisfying P into a state satisfying Q when it terminates, *as long as* each resource r_k allocated in memory satisfies the CSL invariant P_k . The soundness of the logic was established by Brookes in his seminal papers on the trace semantics of concurrent

*This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

separation logic [4,5]. His soundness proof was the object of great attention in the community, and it was revisited in a number of different ways, either semantic [12], syntactic [1] or axiomatic [6] and formalised in proof assistants. One main technical challenge in all these proofs of soundness is to establish the validity of the concurrent rule:

$$\frac{\Gamma \vdash \{P_1\} C_1 \{Q_1\} \quad \Gamma \vdash \{P_2\} C_2 \{Q_2\}}{\Gamma \vdash \{P_1 * P_2\} C_1 \parallel C_2 \{Q_1 * Q_2\}} \quad \text{Concurrent Rule}$$

and of the frame rule:

$$\frac{\Gamma \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * R\} C \{Q * R\}} \quad \text{Frame Rule}$$

In this paper, we establish the validity of these two rules (and of CSL at large) based on a new approach inspired by game semantics, which relies on the observation that the derivation tree π of CSL defines a winning strategy $[\pi]$ in a specification game. As we will see, the specification game itself is derived from the execution of the code C and its interaction with the environment (called the frame) using locks on the shared memory. The specification game expresses the usual rely-and-guarantee conditions as winning conditions in an interactive game played between Eve (for the code) and Adam (for the frame).

In the semantic proofs of soundness, two notions of “state” are usually considered, besides the basic notion *memory state* which describes the state of the variables and of the heap: (1) the *machine states* which are used to describe the execution of the code, and in particular include information about the status of the locks, and (2) the *logical states* which include permissions and other information invisible at the execution level, but necessary to specify the states in the logic. In particular, the tensor product $*$ of separation logic requires information on the permissions, and it is thus defined on logical states, not on machine states. The starting point of the paper is the observation that there exists a third notion of state, which we call *separated state*, implicitly at work in all the semantic proofs of soundness. A separated state describes which part of the global (logical) state of the machine is handled by each component interacting in the course of the execution. It is defined as a triple $(\sigma_C, \sigma, \sigma_F)$ consisting of

- the logical state $\sigma_C \in \mathbf{LState}$ of the code,
- the logical state $\sigma_F \in \mathbf{LState}$ of the frame,
- a function $\sigma : \{r_1, \dots, r_n\} \rightarrow \mathbf{LState} + \{C, F\}$ which tells for every resource variable r whether it is locked and owned by the code, $\sigma(r) = C$, locked and owned by the frame, $\sigma(r) = F$, or available with logical state $\sigma(r) \in \mathbf{LState}$.

This leads us to a “span”

$$\text{machine states} \xleftarrow{\text{refines}} \text{separated states} \xrightarrow{\text{refines}} \text{logical states} \quad (1)$$

where the two notions of machine state and of logical state are “refined” by the notion of separated state, which conveys information about locks (as machine states) and about permissions (as logical states). Namely, every separated state

$$(\sigma_C, \sigma, \sigma_F) \in \mathbf{SState}$$

refines the logical state $\bigotimes(\sigma_C, \sigma, \sigma_F)$ defined by the separation tensor product

$$\bigotimes(\sigma_C, \sigma, \sigma_F) \stackrel{\text{def}}{=} \sigma_C * \left\{ \bigotimes_{r \in \text{dom}(\sigma)} \sigma(r) \right\} * \sigma_F \quad (2)$$

where $\text{dom}(\sigma)$ denotes the set of resources available in σ , in the sense that $\sigma(r) \neq C, F$. Similarly, every separated state $(\sigma_C, \sigma, \sigma_F)$ refines a machine state (μ, L) defined as the memory state μ underlying the logical state (2) just constructed, plus the set of locked resources $L = \text{dom}_C(\sigma) \uplus \text{dom}_F(\sigma)$, see §8 for details. In the same way as the notion of logical state is necessary to define the tensor product $*$ of separation logic, and thus to specify the states, the shift from machine states to separated states is necessary to specify the code, and the way it interacts with its environment and with its resources. Our point here is that the formulas P and Q of separation logic in a Hoare triple $\Gamma \vdash \{P\} C \{Q\}$ do not specify the logical state $\sigma = \bigotimes(\sigma_C, \sigma, \sigma_F) \in \mathbf{LState}$ of the machine itself, but the *fragment* σ_C of this logical state σ owned by the code C at the beginning and at the end of the execution. The notion of separated state is thus at the very heart of the very concept of Hoare triple in separation logic.

We follow the following track in the paper. After discussing the related work, we formulate the two notions of machine states and of machine instructions in §3. This enables us to define the notion of execution traces on machine states in §4 and a number of algebraic operations on them. The trace semantics of concurrent programs, and their interpretation as transition systems, is then formulated in §5 and §6. Once the notion of machine state has been used to describe the trace semantics of the language, we move to the logical side of the span, and formulate the notions of logical state in §7 and the notion of separated state in §8. In §10, we explain how to associate to every execution trace t a specification game played on the paths of the graph of separated states, which is defined in §9. The moves of those games express the ownership discipline enforced by separation logic, and in particular the discipline associated to the locks in concurrent separation logic. Finally, we show in §11 that CSL is sound by proving that every derivation tree of the logic defines a strategy, which lifts each step of the Code of an execution trace into the graph of separated states.

2 Related Work

Several proofs of soundness have already been given for concurrent separation logic. The first proof of correctness was designed by Brookes in [4,5] using semantic ideas. In his proof, every program C is interpreted as a set of “action traces”, defined as finite or infinite sequences of “actions” that look like:

`read 71 from x, read 36 from y, acquire lock r, ...`

An interesting feature of the model is that these action traces do not mention (at least explicitly) the machine states produced by the Code at execution time. The environment is taken into account through the existence of *non sequentially consistent traces* such as

`write 89 in x, read 14 from x`

in the model. The idea is that the Environment presumably changed the value of the variable x between the two actions of the Code. Separation in the logic enables one to decompose actions traces into *local computations*, in order to reflect the program's subjective view of the execution.

Vafeiadis gave another proof of correctness [12] based on more directly operational intuitions. In his proof, the Code is interpreted as a *transition system* whose vertices are pairs (C, σ) consisting of the Code C and of the state σ of the memory, and where edges are execution steps. The core of the soundness proof is that each step of the execution preserves a decomposition of the heap into three parts, which correspond respectively to the Code, the resources, and the Frame. The proof is done by induction on the derivation tree π establishing the triple $\Gamma \vdash \{P\} C \{Q\}$ in concurrent separation logic. The idea of using separated states thus comes from Vafeiadis' proof, which is the closest to ours. One difference, however, besides the game-theoretic point of view we develop, is that we have a more *intensional* description of separated states, provided by the function σ which tracks the states of each of the available locks.

In contrast to the semantic proofs mentioned above, Balabonski, Pottier and Protzenko [1] developed a purely syntactic proof of correctness for Mezzo, a functional language equipped with a type-and-capability system based on concurrent separation logic. The soundness of the logic follows in their approach from a *progress* and a *preservation* theorem on the type system of Mezzo.

Our focus in this work is to develop a game-theoretic approach to concurrent separation logic. For that reason, we prefer to keep the logic as well as the concurrent language fairly simple and concrete. In particular, we do not consider more recent, sophisticated and axiomatic versions of the logic, like Iris [7,8].

3 Machine states and machine instructions

The purpose of this section is to introduce the notions of *machine state* and of *machine instruction* which will be used all along 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, **Loc** = \mathbb{N} and **Val** = \mathbb{Z} .

Definition 3.1 (Memory state) A memory state μ is a pair (s, h) of partial functions with finite domains $s : \mathbf{Var} \rightarrow_{\text{fn}} \mathbf{Val}$ and $h : \mathbf{Loc} \rightarrow_{\text{fn}} \mathbf{Val}$ called the stack s and the heap h of the memory state μ . The set of memory states is denoted **State**. The domains of the partial function s and of h are noted $\text{vdom}(\mu)$ and $\text{hdom}(\mu)$ respectively, and we write $\text{dom}(\mu)$ for their disjoint union.

Definition 3.2 (Machine state) A machine state $\mathfrak{s} = (\mu, L)$ is a pair consisting of a memory state μ and of a subset of resources $L \subseteq \mathbf{LockName}$, called the lock state, which describes the subset of locked resources in \mathfrak{s} . The set of machine states is denoted **MState**.

A *machine step* is defined as a labelled transition between machine states, which can be of two different kinds:

$$\mathfrak{s} \xrightarrow{m} \mathfrak{s}' \qquad \mathfrak{s} \rightsquigarrow^m \mathfrak{s}'$$

depending on whether the instruction $m \in \mathbf{Instr}$ has been executed successfully (on the left) or it has produced a runtime error (on the right). We write $m : \mathfrak{s} \rightsquigarrow \mathfrak{s}'$ when we do not want to specify whether the instruction has produced a runtime error. The *machine instructions* which label the machine steps are defined below:

$$m ::= x := E \mid x := [E] \mid [E] := E' \mid \mathbf{nop} \mid x := \mathbf{alloc}(E) \mid \mathbf{dispose}(E) \mid P(r) \mid V(r)$$

where $x \in \mathbf{Var}$ is a variable, $r \in \mathbf{LockName}$ is a resource variable, and E, E' are arithmetic expressions with variables. Typically, the instruction $x := E$ assigns to the variable x the value $E(\mu)$ of the expression E in the memory state μ , the instruction $P(r)$ locks the resource variable r when it is available, while the instruction $V(r)$ releases it when it is locked, as described below:

$$\frac{E(\mu) = v}{(\mu, L) \rightsquigarrow^{x := E} (\mu[x \mapsto v], L)} \quad \frac{r \notin L}{(\mu, L) \rightsquigarrow^{P(r)} (\mu, L \uplus \{r\})} \quad \frac{r \notin L}{(\mu, L \uplus \{r\}) \rightsquigarrow^{V(r)} (\mu, L)}$$

Thanks to the inclusion $\mathbf{Loc} \subseteq \mathbf{Val}$, an expression E may also denote a location. In that case, $[E]$ refers to the value of the location E in memory. The instruction \mathbf{nop} (for no-operation) does not alter the logical state, while $x := \mathbf{alloc}(E)$ allocates (in a non-deterministic way) some memory space on the heap, initializes it with the value of the expression E , and returns the address of the location to the variable x , while $\mathbf{dispose}(E)$ deallocates the location with address E .

It will be convenient in the sequel to write $\text{lock}^+(m)$ for the set of locks which are taken by an instruction m , that is, $\text{lock}^+(m) = \{r\}$ if $m = P(r)$ and $\text{lock}^+(m) = \emptyset$ otherwise; $\text{lock}^-(m)$ is the set of locks which are released by the instruction m , that is, $\text{lock}^-(m) = \{r\}$ if $m = V(r)$ and $\text{lock}^-(m) = \emptyset$ otherwise.

4 Execution traces

Now that the notion of machine state has been introduced, the next step towards the interpretation of programs is to define the notion of execution trace, with two kinds of transitions: the even transition “played” by the Code, and the odd transitions “played” by the Environment.

Definition 4.1 (Traces) *A trace t is a sequence of machine states*

$$\mathfrak{s}_1 \xrightarrow{\text{env}} \mathfrak{s}_2 \xrightarrow{m_1} \mathfrak{s}_3 \xrightarrow{\text{env}} \dots \xrightarrow{\text{env}} \mathfrak{s}_{2p} \xrightarrow{m_p} \mathfrak{s}_{2p+1} \xrightarrow{\text{env}} \mathfrak{s}_{2p+2}$$

whose even transitions

$$\mathfrak{s}_{2k} \xrightarrow{m_k} \mathfrak{s}_{2k+1} \quad 1 \leq k \leq p$$

are labelled by an instruction $m_k \in \mathbf{Instr}$ such that $\mathfrak{s}_{2k} \rightsquigarrow^{m_k} \mathfrak{s}_{2k+1}$ and whose last transition is played by the environment. The set of traces is denoted by \mathbf{Traces} .

We write $\partial_0 t = \mathfrak{s}_1$ and $\partial_1 t = \mathfrak{s}_{2p+2}$ for the initial and the final states of a trace $t \in \mathbf{Traces}$, respectively. The length $\text{len}(t) = p$ is defined as the number of Code transitions in the trace, and

$$t[k] = \mathfrak{s}_{2k} \xrightarrow{m_k} \mathfrak{s}_{2k+1}$$

denotes the k -th even transition of the trace t , for $1 \leq k \leq \text{len}(t)$. Observe that a trace t always starts and stops by an Environment transition, and that its number of

transitions is equal to $2 \times \text{len}(t) + 1$. We point out the following fact which we will often use in our proofs and constructions:

Proposition 4.2 *A trace $t \in \mathbf{Traces}$ is characterized by its initial state $\partial_0 t$ and by its final state $\partial_1 t$, together with the sequence of Code transitions $t[k]$ for $1 \leq k \leq \text{len}(t)$.*

We introduce now a number of important algebraic constructions on execution traces, whose purpose is to reflect at the level of traces the sequential and parallel composition of programs.

Definition 4.3 (Sequential composition) *Given two traces $t_1, t_2 \in \mathbf{Traces}$ such that $\partial_1(t_1) = \partial_0(t_2)$, one defines $t_1 \cdot t_2 \in \mathbf{Traces}$ as the trace of length $\text{len}(t_1) + \text{len}(t_2)$ with initial state $\partial_0(t_1)$ and final state $\partial_1(t_2)$, and with even transitions defined as*

$$(t_1 \cdot t_2)[k] = \begin{cases} t_1[k] & \text{if } 1 \leq k \leq p, \\ t_2[k - p] & \text{if } p + 1 \leq k \leq p + q. \end{cases}$$

Definition 4.4 (Restriction) *Let \mathbf{Traces}_p denote the set of traces of length p . Every increasing function $f : \{1, \dots, p\} \rightarrow \{1, \dots, q\}$ induces a restriction function*

$$f^* : \mathbf{Traces}_q \longrightarrow \mathbf{Traces}_p$$

which transports a trace t of length q to a coinital and cofinal trace $f^(t)$ of length p*

$$\partial_0 f^*(t) = \partial_0 t \quad \partial_1 f^*(t) = \partial_1 t$$

defined by the instructions $f^(t)[k] = t[f(k)]$ for $1 \leq k \leq p$.*

Definition 4.5 (Shuffle) *A shuffle of two natural numbers $p \in \mathbb{N}$ and $q \in \mathbb{N}$ is a monotone bijection $\omega : \{1, \dots, p\} \uplus \{1, \dots, q\} \rightarrow \{1, \dots, p + q\}$. The set of shuffles of p and q is denoted $\mathbf{Shuffles}(p, q)$.*

Every shuffle $\omega \in \mathbf{Shuffles}(p, q)$ induces a pair of increasing functions

$$\omega_1 : \{1, \dots, p\} \rightarrow \{1, \dots, p + q\} \quad \text{and} \quad \omega_2 : \{1, \dots, q\} \rightarrow \{1, \dots, p + q\}$$

defined by restricting ω to $\{1, \dots, p\}$ and to $\{1, \dots, q\}$, respectively. From this follows immediately that

Proposition 4.6 *Every shuffle $\omega \in \mathbf{Shuffles}(p, q)$ induces a function*

$$\omega^* : \mathbf{Traces}_{p+q} \longrightarrow \mathbf{Traces}_p \times \mathbf{Traces}_q$$

which transports a trace t of length $p + q$ to the pair $(\omega_1^(t), \omega_2^*(t)) \in \mathbf{Traces}_p \times \mathbf{Traces}_q$.*

Definition 4.7 *The parallel composition $t_1 \parallel t_2$ is the set of traces $t \in \mathbf{Traces}$ such that $\omega^*(t) = (t_1, t_2)$ for some shuffle $\omega \in \mathbf{Shuffles}(\text{len}(t_1), \text{len}(t_2))$.*

Note that every trace t in $t_1 \parallel t_2$ satisfies $\text{len}(t) = \text{len}(t_1) + \text{len}(t_2)$ and more importantly, that the parallel composition $t_1 \parallel t_2$ of two traces t_1 and t_2 is empty whenever the two traces t_1 and t_2 are not coinital and cofinal.

The purpose of our last construction $\text{hide}[r]$ is to “hide” the name of a resource variable $r \in \mathbf{LockName}$ in an execution trace.

Definition 4.8 *The function $\text{hide}[r] : \mathbf{Traces} \rightarrow \mathbf{Traces}$ transforms every trace by applying the function*

$$(\mu, L) \longmapsto (\mu, L \setminus \{r\}) \quad : \quad \mathbf{MState} \longrightarrow \mathbf{MState}$$

to each machine state of the original trace, and the function

$$m \mapsto \begin{cases} \text{nop} & \text{if } m = P(r) \text{ or } V(r) \\ m & \text{otherwise} \end{cases} : \mathbf{Instr} \longrightarrow \mathbf{Instr}$$

to the instructions of the trace.

5 Transition Systems

At this stage, we are ready to introduce the notion of transition system which we will use in order to describe the traces generated by a program of our concurrent language. Among these execution traces, one wishes to distinguish (1) the traces which terminate and return from (2) the other traces which are not yet finished or terminate and abort. This leads us to the following definition of transition system:

Definition 5.1 (Transition Systems) *A transition system $\mathbf{T} = (T, |T|)$ is a set of traces $T \subseteq \mathbf{Traces}$ closed under prefix, together with a subset $|T| \subseteq T$, whose traces are said to return.*

We explain below how to lift to transition systems the algebraic operations defined on traces in the previous section §4.

Definition 5.2 *The sequential composition of two transition systems \mathbf{T} and \mathbf{T}' , is defined as the transition system $\mathbf{T}; \mathbf{T}'$ below:*

$$\begin{aligned} T; T' &= T \cup \{t \cdot t' \mid t \in |T|, t' \in T' \text{ and } \partial_1 t = \partial_0 t'\} \\ |T; T'| &= \{t \cdot t' \mid t \in |T|, t' \in |T'| \text{ and } \partial_1 t = \partial_0 t'\} \end{aligned}$$

Definition 5.3 *The parallel composition of two transition systems \mathbf{T} and \mathbf{T}' , is defined as the transition system $\mathbf{T} \parallel \mathbf{T}'$ below:*

$$T_1 \parallel T_2 = \bigcup_{t_i \in T_i} t_1 \parallel t_2 \quad |T_1 \parallel T_2| = \bigcup_{t_i \in |T_i|} t_1 \parallel t_2$$

Definition 5.4 *The transition system $\text{hide}[r](\mathbf{T})$ associated to a transition system \mathbf{T} and to a lock $r \in \mathbf{LockName}$ is defined as follows:*

$$\text{hide}[r](T) = \{\text{hide}[r](t) \mid t \in T\} \quad |\text{hide}[r](T)| = \{\text{hide}[r](t) \mid t \in |T|\}.$$

Note that every instruction $m \in \mathbf{Instr}$ induces a transition system $\llbracket m \rrbracket$ defined in the following way:

$$\begin{aligned} \llbracket m \rrbracket &= \{s_1 \xrightarrow{\text{env}} s_2 \xrightarrow{m} s_3 \xrightarrow{\text{env}} s_4 \mid s_2 \rightsquigarrow^m s_3\} \\ |\llbracket m \rrbracket| &= \{s_1 \xrightarrow{\text{env}} s_2 \xrightarrow{m} s_3 \xrightarrow{\text{env}} s_4 \mid s_2 \rightsquigarrow^m s_3\} \end{aligned}$$

The intuition is that the program interpreted by $\llbracket m \rrbracket$ executes the instruction m after the environment has made the transition $s_1 \xrightarrow{\text{env}} s_2$ and returns when the machine step $s_2 \xrightarrow{m} s_3$ is succesful, and does not abort. The following algebraic operation on transition systems reflects the computational situation of a program taking a lock r before executing, and releasing the lock r in case the program returns.

Definition 5.5 *The transition system $\text{inside}[r](\mathbf{T})$ associated to a transition sys-*

tem \mathbf{T} and to a lock $r \in \mathbf{LockName}$ is defined as follows:

$$\text{inside}[r](\mathbf{T}) = \llbracket P(r) \rrbracket; \mathbf{T}; \llbracket V(r) \rrbracket.$$

The following operation on transition systems will enable us to interpret conditional branching on concurrent programs.

Definition 5.6 *The transition system $\text{whentru}[P](\mathbf{T})$ associated to a transition system $\mathbf{T} = (T, |T|)$ and a predicate $P : \mathbf{MState} \rightarrow \{\mathbf{true}, \mathbf{false}, \mathbf{abort}\}$ on memory states is defined as follows:*

$$\begin{aligned} \text{whentru}[P](\mathbf{T}) &= \{t \in T \mid P(\bar{\partial}_0 t) = \mathbf{true}\} \\ |\text{whentru}[P](\mathbf{T})| &= \{t \in |T| \mid P(\bar{\partial}_0 t) = \mathbf{true}\} \end{aligned}$$

where $\bar{\partial}_0 t = \mathfrak{s}_2$ denotes the first state played by Code in the trace t .

The transition system $\text{whenfalse}[P](\mathbf{T})$ is defined similarly, by replacing **true** by **false** in the definition. A subtle but important aspect of the interpretation of conditional branching in the language is that the evaluation of a boolean expression B may not succeed, typically because one of its variables $x \in \mathbf{Var}$ is not allocated. In that case, the evaluation produces an exception which is then handled by the operating system. This **abort** case is handled in our trace semantics by the definition of a dedicated transition system called $\text{whenabort}[P, C]$, whose construction is detailed in the Appendix.

6 Trace semantics of the concurrent language

Now that we have defined the basic operations on transition systems, we are ready to define the operational and interactive semantics of our concurrent language. The language is constructed with Boolean expressions B , arithmetic expressions E and commands C , using the grammar below:

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

The parallel composition operator $C_1 \parallel C_2$ enables the two programs C_1 and C_2 to interact concurrently through *mutexes* called *resources*. A resource r is declared using **resource** r and acquired using **with** r **when** B **do** C , which waits for the Boolean expression B to be true in order to proceed. Of course, a mutex can be held by at most one execution thread at any one time.

In the semantic approach we are following, every command C is translated into a transition system $\llbracket C \rrbracket$ which describes the possible interactive executions of C , and whether they return.

$$\text{Code } C \xrightarrow{\text{translation}} \text{Transition system } \llbracket C \rrbracket$$

The interpretation $\llbracket C \rrbracket$ is defined by structural induction on the syntax of the

command C . To each leaf node C , one associates an instruction $m \in \mathbf{Instr}$

$$x := E \mid x := [E] \mid [E] := [E'] \mid \mathbf{nop} \mid x := \mathbf{alloc}(E) \mid \mathbf{dispose}(E)$$

which defines the transition system $\llbracket C \rrbracket \stackrel{\text{def}}{=} \llbracket m \rrbracket$. The semantics of non-leaf commands is then defined using the algebraic operations on transition systems introduced in §5:

$$\llbracket C \parallel C' \rrbracket \stackrel{\text{def}}{=} \llbracket C \rrbracket \parallel \llbracket C' \rrbracket, \quad \llbracket C; C' \rrbracket \stackrel{\text{def}}{=} \llbracket C \rrbracket; \llbracket C' \rrbracket,$$

$$\llbracket \mathbf{resource } r \text{ do } C \rrbracket \stackrel{\text{def}}{=} \mathbf{hide}[r](\llbracket C \rrbracket),$$

$$\llbracket \mathbf{with } r \text{ when } B \text{ do } C \rrbracket \stackrel{\text{def}}{=} \mathbf{whentrue}[B](\mathbf{inside}[r](\llbracket C \rrbracket)) \cup \mathbf{whenabort}[B, C']$$

where $C' = \mathbf{with } r \text{ when } B \text{ do } C$ in the last part of the definition, and finally

$$\begin{aligned} \llbracket \mathbf{if } B \text{ then } C_1 \text{ else } C_2 \rrbracket &\stackrel{\text{def}}{=} \mathbf{whentrue}[B](\llbracket \mathbf{nop} \rrbracket); \llbracket C_1 \rrbracket \cup \mathbf{whenfalse}[B](\llbracket \mathbf{nop} \rrbracket); \llbracket C_2 \rrbracket \\ &\cup \mathbf{whenabort}[B, \mathbf{if } B \text{ then } C_1 \text{ else } C_2], \end{aligned}$$

and the while loop

$$\llbracket \mathbf{while } B \text{ do } C \rrbracket \stackrel{\text{def}}{=} \bigcup_{n \geq 0} F^n(\emptyset)$$

is defined as the least fixpoint of the continuous function $F : \mathbf{Trans} \rightarrow \mathbf{Trans}$ below:

$$\begin{aligned} F(\mathbf{T}) &= \mathbf{whentrue}[B](\llbracket \mathbf{nop} \rrbracket); \llbracket C \rrbracket; \mathbf{T} \cup \mathbf{whenfalse}[B](\llbracket \mathbf{nop} \rrbracket) \cup \\ &\mathbf{whenabort}[B, \mathbf{while } B \text{ do } C]. \end{aligned}$$

7 Logical States

As we explained in the introduction, reasoning about concurrent programs in separation logic requires introducing an appropriate notion of *logical state*, including information about permissions. The version of concurrent separation logic we consider is almost the same as in its original formulation by O'Hearn and Brookes [9,4]. One difference is that we benefit from the work in [2,3,10] and use the permissions and the $\mathbf{Own}_p(x)$ in order to handle the heap as well as variables in the stack. So, we suppose given an arbitrary partial cancellative commutative monoid \mathbf{Perm} that we call the *permission monoid*, following [2]. We require that the permission monoid contains a distinguished element \top which does not admit any multiple, ie. $\forall x \in \mathbf{Perm}, \top \cdot x$ is not defined. The idea is that the permission \top is required for a program to write somewhere in memory. The property above ensures that a piece of state cannot be written and accessed (with a read or a write) at the same time by two concurrent programs, and therefore, that there is memory safety and no data race in the semantics. The set \mathbf{LState} of logical states is defined in a similar way as the set \mathbf{State} of memory states, with the addition of permissions:

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

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

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

$$\begin{aligned}
 \sigma \models \text{Own}_p(x) &\iff \exists v \in \mathbf{Val}, \sigma(x) = (v, p) \\
 \sigma \models E_1 = E_2 &\iff \llbracket E_1 \rrbracket = \llbracket E_2 \rrbracket \wedge \text{fv}(E_1 = E_2) \subseteq \text{vdom}(h) \\
 \sigma \models P \Rightarrow Q &\iff (\sigma \models P) \Rightarrow (\sigma \models Q) \\
 \sigma \models P \wedge Q &\iff \sigma \models P \text{ et } \sigma \models Q \\
 \sigma \models P * Q &\iff \exists \sigma_1 \sigma_2, \sigma = \sigma_1 * \sigma_2 \text{ et } \sigma_1 \models P \text{ et } \sigma_2 \models Q
 \end{aligned}$$

Figure 1. Semantics of the predicates of concurrent separation logic

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

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

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

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

The semantics of the formulas is expressed as the satisfaction predicate $\sigma \models P$ defined in Figure 1. The proof system underlying concurrent separation logic is a sequent calculus on sequents defined as 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 $\Gamma = r_1 : J_1, \dots, r_k : J_k$ describes the invariant J_i satisfied by the resource variable r_i . The purpose of these resources is to provide the fragments of memory shared between the various threads during the execution. The inference rules are given in Figure 2. The inference rule RES associated to **resource** r **do** C moves a piece of memory which is owned by the Code into the shared context Γ , which means it can be accessed concurrently inside C . However, the access to said piece of memory is mediated by the **with** construct, which grants temporary access under the condition that one must give it back (rule WITH). Notice that in the rule CONJ, the context $\Gamma = r_1 : J_1, \dots, r_k : J_k$ is required to be *precise*, in the sense that each of the predicates J_i is precise.

Definition 7.1 (Precise predicate) *A predicate P is precise when, for any $\sigma \in \mathbf{LState}$, there exists at most one $\sigma' \in \mathbf{LState}$ such that $\exists \sigma'', \sigma = \sigma' * \sigma''$ and $\sigma' \models P$.*

8 Separated states

We now introduce our third notion of state, which display which region of (logical) memory belongs to the Code, which region belongs to the Frame, and which region is shared. We suppose given a finite set **LockName** of resource variables.

$$\begin{array}{c}
 \frac{}{\Gamma \vdash \{\text{Own}_\top(x) * X = E\} x := E \{\text{Own}_\top(x) * x = X\}} \text{AFF} \quad \frac{}{\Gamma \vdash \{E \mapsto -\} [E] := E' \{E \mapsto E'\}} \text{STORE} \\
 \\
 \frac{x \notin \text{fv}(E)}{\Gamma \vdash \{E \mapsto^p v * \text{Own}_\top(x)\} x := [E] \{E \mapsto^p v * \text{Own}_\top(x) * x = v\}} \text{LOAD} \\
 \\
 \frac{\Gamma \vdash \{P\} C \{Q\} \quad \Gamma \vdash \{Q\} C' \{R\}}{\Gamma \vdash \{P\} C; C' \{R\}} \text{SEQ} \quad \frac{P \Rightarrow \text{def}(B) \quad \Gamma \vdash \{P \wedge B\} C_1 \{Q\} \quad \Gamma \vdash \{P \wedge \neg B\} C_2 \{Q\}}{\Gamma \vdash \{P\} \text{if } B \text{ then } C_1 \text{ else } C_2 \{Q\}} \text{IF} \\
 \\
 \frac{\Gamma \text{ is precise} \quad \Gamma \vdash \{P_1\} C \{Q_1\} \quad \Gamma \vdash \{P_2\} C \{Q_2\}}{\Gamma \vdash \{P_1 \wedge P_2\} C \{Q_1 \wedge Q_2\}} \text{CONJ} \quad \frac{\Gamma, r : J \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * J\} \text{resource } r \text{ do } C \{Q * J\}} \text{RES} \\
 \\
 \frac{P \Rightarrow \text{def}(B) \quad \Gamma \vdash \{(P * J) \wedge B\} C \{Q * J\}}{\Gamma, r : J \vdash \{P\} \text{with } r \text{ when } B \text{ do } C \{Q\}} \text{WITH} \quad \frac{\Gamma \vdash \{P_1\} C_1 \{Q_1\} \quad \Gamma \vdash \{P_2\} C_2 \{Q_2\}}{\Gamma \vdash \{P_1 * P_2\} C_1 \parallel C_2 \{Q_1 * Q_2\}} \text{PAR} \\
 \\
 \frac{\Gamma \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * R\} C \{Q * R\}} \text{FRAME}
 \end{array}$$

Figure 2. Inference rules of Concurrent Separation Logic

Definition 8.1 *The separated states are the triples*

$$(\sigma_C, \sigma, \sigma_F) \in \mathbf{LState} \times (\mathbf{LockName} \rightarrow \mathbf{LState} + \{C, F\}) \times \mathbf{LState}$$

such that the state below is defined:

$$\sigma_C * \left\{ \bigotimes_{r \in \text{dom}(\sigma)} \sigma(r) \right\} * \sigma_F$$

where

$$\begin{aligned}
 \text{dom}(\sigma) &= \{r \in \mathbf{LockName} \mid \sigma(r) \in \mathbf{LState}\}, \\
 \text{dom}_C(\sigma) &= \{r \in \mathbf{LockName} \mid \sigma(r) = C\}, \\
 \text{dom}_F(\sigma) &= \{r \in \mathbf{LockName} \mid \sigma(r) = F\}.
 \end{aligned}$$

We say that a separated state $(\sigma_C, \sigma, \sigma_F)$ combines into a machine state $\mathfrak{s} = (\mu, L)$ precisely when both $L = \text{dom}_C(\sigma) \uplus \text{dom}_F(\sigma)$ and the memory state $\mu \in \mathbf{State}$ is equal to the image of

$$\sigma_C * \left\{ \bigotimes_{r \in \text{dom}(\sigma)} \sigma(r) \right\} * \sigma_F \in \mathbf{LState} \quad (3)$$

under the function $U : \mathbf{LState} \rightarrow \mathbf{State}$ which forgets the permissions. Note that by definition, every separated state $(\sigma_C, \sigma, \sigma_F)$ combines into a unique machine state, which we write for concision $(\mu, L) = \bigotimes(\sigma_C, \sigma, \sigma_F)$.

9 The graphs of machine and separated states

In this section, we introduce the two labeled graphs $\mathbf{G}(\mathbf{MState})$ and $\mathbf{G}(\mathbf{SState})$ of machine states and of separated states, and construct a graph homomorphism

$$\bigotimes : \mathbf{G}(\mathbf{SState}) \longrightarrow \mathbf{G}(\mathbf{MState}) \quad (4)$$

which maps every separated state $(\sigma_C, \sigma, \sigma_F)$ to its combined machine state (σ, L) , in the way described in the introduction.

Definition 9.1 *The graph of machine states $\mathbf{G}(\mathbf{MState})$ is the graph whose vertices are the machine states $\mathfrak{s} \in \mathbf{MState}$ and whose edges are either Code or Environment transitions of the following kind:*

- a Code transition $\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$ for every machine step $\mathfrak{s} \rightsquigarrow^m \mathfrak{s}'$,
- an Environment transition $\mathfrak{s} \xrightarrow{env} \mathfrak{s}'$ for every pair $\mathfrak{s}, \mathfrak{s}' \in \mathbf{MState}$ of machine states, and where *env* is just a tag indicating that the transition has been fired by the Environment.

Note that a trace $t \in \mathbf{Traces}$ (see Def. 4.1) is the same thing as an alternating path starting and ending with an Environment edge in the graph $\mathbf{G}(\mathbf{MState})$.

Definition 9.2 The graph of separated states $\mathbf{G}(\mathbf{SState})$ is the graph whose vertices are the separated states and whose edges are either Eve moves or Adam moves of the following kind:

- Eve moves of the form

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

labeled by an instruction $m \in \mathbf{Instr}$ such that

$$(\otimes)(\sigma_C, \sigma, \sigma_F) \rightsquigarrow^m (\otimes)(\sigma'_C, \sigma', \sigma_F)$$

between machine states, and such that the following conditions on locked resources are moreover satisfied:

$$\begin{aligned} \forall r \notin \text{lock}(m), \sigma(r) &= \sigma'(r), \\ \forall r \in \text{lock}^+(m), r \in \text{dom}(\sigma) \wedge r &\in \text{dom}_C(\sigma'), \\ \forall r \in \text{lock}^-(m), r \in \text{dom}_C(\sigma) \wedge r &\in \text{dom}(\sigma'); \end{aligned}$$

- Adam moves of the form

$$(\sigma_C, \sigma, \sigma_F) \xrightarrow{env} (\sigma_C, \sigma', \sigma'_F)$$

where *env* is just a tag, and moreover

$$\text{dom}_C(\sigma') = \text{dom}_C(\sigma).$$

The definition of the vertices and of the edges of the graph of separated states $\mathbf{G}(\mathbf{SState})$ is designed to ensure that there exists a graph homomorphism (4) which maps every Eve move to a Code transition, and every Adam move to an Environment transition. The graph homomorphism (4) enables us to study how an execution trace $t \in \mathbf{Traces}$ defined as a path in $\mathbf{G}(\mathbf{MState})$ may be “refined” into a *separated execution trace* \mathbf{p} living in the graph of $\mathbf{G}(\mathbf{SState})$ of separated states, and such that $t = (\otimes)\mathbf{p}$. In that situation, we use the following terminology:

Definition 9.3 We say that a path \mathbf{p} in the labeled graph $\mathbf{G}(\mathbf{SState})$ *combines* into a trace $t \in \mathbf{Traces}$ in the labeled graph $\mathbf{G}(\mathbf{MState})$ when $t = (\otimes)\mathbf{p}$.

Note that a path \mathbf{p} which combines into a trace $t \in \mathbf{Traces}$ is alternated between Eve and Adam moves, and that it starts and stops with an Adam move.

10 Separation games

In this section, we explain how to associate to every trace $t \in \mathbf{Traces}$ a *separation game* $\mathbf{SGame}(t)$ on which Eve and Adam interact and try to “justify” every transition played in the execution trace t by the Code or by the Environment, by lifting it to a

separated execution trace p which combines into t .

Definition 10.1 (Game) A game A is a triple $A = (\text{Board}_A, \text{Pol}_A, \text{Plays}_A)$ consisting of a graph $\text{Board}_A = (V, E, \partial_0, \partial_1)$ with source and target functions $\partial_0, \partial_1 : E \rightarrow V$, and whose edges are called moves: of a function $\text{Pol}_A : E \rightarrow \{-1, +1\}$ which assigns a polarity $+1$ to every move played by Eve (Player) and -1 to every move played by Adam (Opponent); of a prefix-closed set Plays_A of finite paths, called the plays of the game A . One requires moreover that every play of the game

$$x_1 \xrightarrow{e_1} x_2 \xrightarrow{e_2} \cdots \longrightarrow x_n \xrightarrow{e_n} x_{n+1}$$

is alternating in the sense that $\text{Pol}_A(e_i) = (-1)^i$ for $1 \leq i \leq n$, and that it starts and stops with an Adam move.

A vertex in a game A is called *initial* when there exists a play $s \in \text{Plays}_A$ with $x = \partial_0(s)$ as source. The set of initial vertices of a game A is noted Init_A . We take below the most general and liberal definition of a strategy. In particular, a strategy in that sense does not need to be deterministic.

Definition 10.2 (Strategy) A strategy of a game is a prefix-closed set of plays.

Every execution trace $t \in \mathbf{Traces}$ induces a game defined below, called the *separation game* associated to t and noted $\mathbf{SGame}(t)$.

Definition 10.3 (Separation Game) The game $\mathbf{SGame}(t) = (\text{Board}, \text{Pol}, \text{Plays})$ is defined as the graph $\text{Board} = \mathbf{G}(\mathbf{SState})$ with plays in Plays defined as the paths

$$p : (\sigma_C, \sigma, \sigma_F) \xrightarrow{*} (\sigma'_C, \sigma', \sigma'_F)$$

in $\mathbf{G}(\mathbf{SState})$ which combine into a path in $\mathbf{G}(\mathbf{MState})$

$$(\otimes)p : (\otimes)(\sigma_C, \sigma, \sigma_F) \xrightarrow{*} (\otimes)(\sigma'_C, \sigma', \sigma'_F)$$

prefix of the trace $t \in \mathbf{Traces}$. The polarity Pol of the moves is derived from the polarity Eve $(+1)$ and Adam (-1) of the edges of the graph $\text{Board} = \mathbf{G}(\mathbf{SState})$ of separated states.

A play of the separation game $\mathbf{SGame}(t)$ may be thus seen as a “psychoanalysis” or rather a “couple therapy” where Eve and Adam try and justify *a posteriori* what has just happened in the execution trace $t \in \mathbf{Traces}$ played by the Code (on the side of Eve) and the Environment (on the side of Adam). At each transition $m : (\sigma, L) \rightarrow (\sigma', L')$ performed by the Code in the execution trace $t \in \mathbf{Traces}$ starting from a machine state $(\sigma, L) = (\otimes)(\sigma_C, \sigma, \sigma_F)$, Eve has to play a move $m : (\sigma_C, \sigma, \sigma_F) \rightarrow (\sigma'_C, \sigma', \sigma'_F)$ which “justifies” the transition by decomposing the machine state (σ', L') into a separated state $(\sigma'_C, \sigma', \sigma'_F)$. And symmetrically for Adam and the Environment.

11 Soundness theorem

At this stage, we establish our soundness theorem for concurrent separation logic, by interpreting every derivation tree as a winning strategy in a specific separation game. We suppose given a Hoare triple $\Gamma \vdash \{P\} C \{Q\}$. We start by describing the winning condition on the separation game $\mathbf{SGame}(t)$ associated to an execution trace $t \in \llbracket C \rrbracket$ in the operational semantics of C .

Definition 11.1 A separated predicate is a triple $\mathbf{P} = (P, \Gamma, Q)$ consisting of two predicates P and Q and of a context $\Gamma = r_1 : J_1, \dots, r_k : J_k$ of variable resources.

Definition 11.2 We write

$$(\sigma_C, \sigma, \sigma_F) \models (P, \Gamma, Q)$$

and say that the separated state $(\sigma_C, \sigma, \sigma_F)$ satisfies the separated predicate $\mathbf{P} = (P, \Gamma, Q)$ precisely when $\sigma_C \models P$ and $\sigma_F \models Q$ and $\forall r \in \text{dom}(\sigma), \sigma(r) \models \Delta(r)$.

We suppose from now on that the execution trace $t \in \llbracket C \rrbracket$ is of length p , and introduce the sequence $\mathbf{P}_1, \dots, \mathbf{P}_{2p+2}$ of separated predicates, defined as:

$$\mathbf{P}_1 = (P, \Gamma, \mathbf{true}) \quad \mathbf{P}_i = (\mathbf{true}, \Gamma, \mathbf{true}) \quad \mathbf{P}_{2p+2} = (Q, \Gamma, \mathbf{true})$$

for $1 < i < 2p+1$ when the execution trace $t \in \llbracket C \rrbracket$ is returning; and defined as

$$\mathbf{P}_1 = (P, \Gamma, \mathbf{true}) \quad \mathbf{P}_i = (\mathbf{true}, \Gamma, \mathbf{true}) \quad \mathbf{P}_{2p+2} = (\mathbf{true}, \Gamma, \mathbf{true})$$

for $1 < i < 2p+2$ when the execution trace $t \notin \llbracket C \rrbracket$ is not returning. Here, we write \mathbf{true} for the constant predicate which is true for every logical state.

Definition 11.3 (Winning condition) A play

$$(\sigma_C^1, \sigma^1, \sigma_F^1) \xrightarrow{\text{env}} (\sigma_C^2, \sigma^2, \sigma_F^2) \xrightarrow{m_1} (\sigma_C^3, \sigma^3, \sigma_F^3) \cdots \rightarrow (\sigma_C^q, \sigma^q, \sigma_F^q)$$

in the separation game $\mathbf{SGame}(t)$ is declared winning when

$$\forall i \in \{1, \dots, 2q+2\}, \quad (\sigma_C^i, \sigma^i, \sigma_F^i) \models \mathbf{P}_i.$$

In other words, given a context $\Gamma = r_1 : J_1, \dots, r_k : J_k$, a play is winning when the logical states σ_C^1 and σ_C^2 (which are equal) satisfy the input predicate P ; when the vectors σ 's of logical states satisfy

$$\forall i \in \{1, \dots, q\}, \forall r \in \text{dom}(\Gamma), \quad r \in \text{dom}(\sigma^i) \Rightarrow \sigma^i(r) \models J_i$$

and finally, in the case when $q = 2p+2$ and $t \in \llbracket C \rrbracket$ returns, when the logical states σ_C^{2p+1} and σ_C^{2p+2} (which are equal) satisfy the output predicate Q . Note that the notion of winning play is closed under prefix.

Definition 11.4 A strategy \mathbf{strat} of the separation game $\mathbf{SGame}(t)$ is winning when it contains only winning plays, and moreover:

- the strategy \mathbf{strat} contains every empty and winning play of the separation game,
- for every play \mathbf{p} in the strategy \mathbf{strat} , which can be extended by a move a played by Adam into a winning play $\mathbf{p} \cdot a$ of the separation game $\mathbf{SGame}(t)$, there exists a move e played by Eve such that $\mathbf{p} \cdot a \cdot e$ defines a play in the strategy \mathbf{strat} .

Note that an empty and winning play of the separation game consists of a separated state $(\sigma_C, \sigma, \sigma_F)$ satisfying the predicate $(P, \Gamma, \mathbf{true})$, and in the very special case when the trace $t \in \llbracket C \rrbracket$ is empty and returns, the predicate $(Q, \Gamma, \mathbf{true})$.

We are now able to state the soundness theorem of concurrent separation logic, which is established by structural induction on the derivation tree π of the Hoare triple $\Gamma \vdash \{P\} C \{Q\}$.

Theorem 11.5 (Soundness) Every derivation tree π of $\Gamma \vdash \{P\} C \{Q\}$ defines for every execution trace $t \in \llbracket C \rrbracket$ a winning strategy $\mathbf{strat}(\pi, t)$ in the separation game $\mathbf{SGame}(t)$ determined by the Hoare triple $\Gamma \vdash \{P\} C \{Q\}$ and t .

The reader will find the proof of the theorem in the Appendix. This statement is inspired by game semantics, and the idea of a Curry-Howard correspondence between proofs (here the derivation trees) and winning strategies. This game-theoretic interpretation of derivation trees implies the soundness of concurrent separation logic in the more traditional sense [4,12,1], by considering the case when the context Γ is empty, and the environment is passive, in the following sense.

Definition 11.6 *The environment is passive in a trace*

$$\mathfrak{s}_1 \xrightarrow{env} \mathfrak{s}_2 \xrightarrow{m_1} \mathfrak{s}_3 \xrightarrow{env} \dots \xrightarrow{env} \mathfrak{s}_{2p} \xrightarrow{m_p} \mathfrak{s}_{2p+1} \xrightarrow{env} \mathfrak{s}_{2p+2}$$

when every transition $\mathfrak{s}_{2i+1} \xrightarrow{env} \mathfrak{s}_{2i+2}$ by the environment does not alter the logical state, and is thus the identity $\mathfrak{s}_{2i+1} = \mathfrak{s}_{2i+2}$, for $0 \leq i \leq p$.

Corollary 11.7 *Suppose that the triple $\emptyset \vdash \{P\} C \{Q\}$ has been proved by a derivation tree π of concurrent separation logic, and that $t \in \llbracket C \rrbracket$ is an execution trace*

$$\mathfrak{s}_1 \xrightarrow{id} \mathfrak{s}_1 \xrightarrow{m_1} \mathfrak{s}_3 \xrightarrow{id} \dots \xrightarrow{id} \mathfrak{s}_{2p} \xrightarrow{m_p} \mathfrak{s}_{2p+1} \xrightarrow{id} \mathfrak{s}_{2p+1}$$

in which the Environment is passive. Then, the execution trace t produces no error, in the technical sense that every machine step $m_i : \mathfrak{s}_{2i+1} \rightsquigarrow \mathfrak{s}_{2i+3}$ executed by the Code, for $0 \leq i \leq p-1$ is of the form $\mathfrak{s}_{2i+1} \rightsquigarrow^{m_i} \mathfrak{s}_{2i+3}$ and thus does not produce any error at run-time. Moreover, in the case when the execution trace $t \in \llbracket C \rrbracket$ returns, one has that:

$$\partial_0 t \models P * \mathbf{true} \implies \partial_1 t \models Q * \mathbf{true}.$$

Note that the predicate $P * \mathbf{true}$ means that the logical state σ taken as input by the Code C contains a fragment σ_C which satisfies the predicate P . The winning strategy associated to π ensures that when the trace t returns, the Code C ends with a fragment σ'_C of the logical state σ' returned as output.

References

- [1] Balabonski, T., F. Pottier and J. Protzenko, *Type soundness and race freedom for mezzo*, in: *FLOPS*, 2014.
- [2] Bornat, R., C. Calcagno, P. O'Hearn and M. Parkinson, *Permission accounting in separation logic*, in: *POPL*, 2005.
- [3] Bornat, R., C. Calcagno and H. Yang, *Variables as resource in separation logic*, *ENTCS* **155** (2006), pp. 247–276.
- [4] Brookes, S., *A semantics for concurrent separation logic*, in: *CONCUR*, 2004.
- [5] Brookes, S., *A revisionist history of concurrent separation logic*, *ENTCS* **276** (2011), pp. 5 – 28.
- [6] Dinsdale-Young, T., L. Birkedal, P. Gardner, M. J. Parkinson and H. Yang, *Views: compositional reasoning for concurrent programs*, in: *POPL*, 2013.
- [7] Jung, R., D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal and D. Dreyer, *Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning*, in: *POPL*, 2015.
- [8] Krebbers, R., R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer and L. Birkedal, *The essence of higher-order concurrent separation logic*, in: *ESOP*, 2017.
- [9] O'Hearn, P. W., *Resources, concurrency, and local reasoning*, *TCS* **375** (2007), pp. 271–307.
- [10] Parkinson, M. J., R. Bornat and C. Calcagno, *Variables as resource in hoare logics*, in: *LICS*, 2006, pp. 137–146.
- [11] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *LICS*, 2002.
- [12] Vafeiadis, V., *Concurrent separation logic and operational semantics*, *ENTCS* **276** (2011), pp. 335–351.

A Proof of Soundness

A.1 Preliminaries

A.1.1 Order on games

We can order separation games.

Definition A.1 *Let G and G' be two separation games whose underlying traces are the same. Then we say that G is harder than G' if, whenever a path p realizes G , it also realizes G' .*

Equivalently, for each corresponding predicate P of G and P' of G' ,

$$\forall \sigma \in \mathbf{State}, \quad \sigma \models P \Rightarrow \sigma \models P'.$$

A.1.2 A sufficient condition to be a winning play

In the case of the games induced by Hoare triples, a strategy is a set of paths that realize even-length prefixes of G , and such that, in addition, each play of Eve satisfies the separated predicate that follows it.

Proposition A.2 *Let $\Gamma \vdash \{P\} C \{Q\}$ be a triple, let $t \in \llbracket C \rrbracket$ be an execution trace of length $2p+1$ and let $(\mathbf{P}_i)_{1 \leq i \leq 2p+2}$ be the separated predicates they induce. Suppose the play $(\sigma_C^i, \sigma^i, \sigma_F^i)_{1 \leq i \leq 2p+2}$ is such that, for all $i \in \{1, \dots, 2p+1\}$, $(\sigma_C^i, \sigma^i, \sigma_F^i) \models \mathbf{P}_{i+1}$, then it is a winning play (in the sense of Definition 11.3).*

A.1.3 Stability of well-defined boolean formulas

Lemma A.3 *Let B be a boolean formula and t be a trace that is realized by a play p . Call n_1 the first separated state of p , and assume that $n_1 \models \{P, \Gamma, \mathbf{true}\}$, for some P and Γ such that $P \Rightarrow B$.*

Then, if B is true in the first state of t , it is also true in the second state of t .

Proof If B only depends on variables owned by P , Adam cannot touch them. \square

In the remainder of this section, we define the strategy $\mathbf{strat}(\pi, t)$, for all proofs π of a triple $\Gamma \vdash \{P\} C \{Q\}$ and all $t \in \llbracket C \rrbracket$, by structural induction on π .

A.2 Parallel composition

First, we prove the case of PAR, which is the most interesting. Let $C = C_1 \parallel C_2$. The strategy for the parallel composition will first be defined in a new graph, that will keep the part of the memory state that belongs to C_1 separated from the part that belongs to C_2 , and then will be projected into $\mathbf{G}(\mathbf{SState})$. By appealing to Proposition A.2, we only consider plays of even length.

Definition A.4 *The states of the graph of parallel composition are the 4-tuples $(\sigma_1, \sigma_2, \sigma, \sigma_F)$ such that their product is defined, as in Definition 8.1. It has 3 kinds of moves:*

- *Eve₁ moves $(\sigma_1, \sigma_2, \sigma, \sigma_F) \xrightarrow{1} (\sigma'_1, \sigma_2, \sigma', \sigma_F)$,*
- *Eve₂ moves $(\sigma_1, \sigma_2, \sigma, \sigma_F) \xrightarrow{2} (\sigma_1, \sigma'_2, \sigma', \sigma_F)$,*
- *Adam moves $(\sigma_1, \sigma_2, \sigma, \sigma_F) \longrightarrow (\sigma_1, \sigma_2, \sigma', \sigma'_F)$,*

with $\sigma : \mathbf{LockName} \rightarrow \mathbf{State} + \{C_1, C_2, F\}$, and otherwise the same existence conditions on the moves as in Definition 9.2.

We call a *valid path of the graph of parallel composition* a path of even length, that starts with an Adam move, and that is alternated (disregarding which Eve is playing). We can project any valid path p on an even length path $\mathbf{proj}(p)$ of $\mathbf{G}(\mathbf{SState})$ where Adam starts in the graph of separated states by multiplying the first two components and forgetting which Eve is playing.

$$(\sigma_1, \sigma_2, \sigma, \sigma_F) \mapsto (\sigma_1 * \sigma_2, \sigma, \sigma_F)$$

Through this projection, we can define realization and strategies on these paths.

We now define the left and right projections of a valid path p of the graph of parallel composition, which give the subjective views of each execution thread, as paths of the graph of separated states.

Definition A.5 To define \mathbf{proj}_1 , first consider the function defined on the states in the following way:

$$(\sigma_1, \sigma_2, \sigma, \sigma_F) \mapsto (\sigma_1, \underline{\sigma}, \sigma_F * \sigma_2)$$

where

$$\underline{\sigma}(r) = \begin{cases} C & \text{if } \sigma(r) = C_1 \\ F & \text{if } \sigma(r) = C_2 \\ \sigma(r) & \text{otherwise} \end{cases}$$

and on the labels of the moves by mapping Eve_1 moves to Eve moves and Eve_2 moves to env moves. Then \mathbf{proj}_1 is defined as the function above, where consecutive *Environment* moves are merged into one.

By merging consecutive *Environment* moves together, the image of a valid path is an alternated even-length path of $\mathbf{G}(\mathbf{MState})$.

We are now ready to define $\mathbf{strat}(\pi, t)$, where π is the proof:

$$\frac{\begin{array}{c} \vdots \pi_1 \\ \vdots \\ \Gamma \vdash \{P_1\} C_1 \{Q_1\} \end{array} \quad \begin{array}{c} \vdots \pi_2 \\ \vdots \\ \Gamma \vdash \{P_2\} C_2 \{Q_2\} \end{array}}{\Gamma \vdash \{P_1 * P_2\} C \{Q_1 * Q_2\}} \text{PAR}$$

Let $t \in \llbracket C \rrbracket$. By definition of $\llbracket \cdot \rrbracket$, there exist $t_1 \in \llbracket C_1 \rrbracket$ and $t_2 \in \llbracket C_2 \rrbracket$ such that $t \in (t_1 \parallel t_2)$.

We define the strategy $\mathbf{strat}(\pi, t)$ (in the graph of parallel composition) such that for all $p \in \mathbf{strat}(\pi, t)$,

$$\mathbf{proj}_1(p) \in \mathbf{strat}(\pi_1, t_1) \wedge \mathbf{proj}_2(p) \in \mathbf{strat}(\pi_2, t_2). \quad (\text{A.1})$$

Then, the strategy $\mathbf{strat}(\pi, t)$ in $\mathbf{G}(\mathbf{SState})$ will be obtained by projecting the paths of the strategy using \mathbf{proj} .

Note that an Adam move in a path p of the graph of parallel composition corresponds one-to-one with an Adam move in $\mathbf{proj}(p)$ of the graph of separated states.

First, if $n = (\sigma_C, \sigma, \sigma_F)$ is a state that satisfies $\{P_1 * P_2, \Gamma, \mathbf{true}\}$, then by definition, there exists σ_1 and σ_2 such that $\sigma_i \models P_i$ and $\mathbf{proj}(\sigma_1, \sigma_2, \sigma, \sigma_F) = n$.

Suppose now that we have such a path p where Adam played last, and note

$(\sigma_1, \sigma_2, \sigma, \sigma_F)$ its last state. By induction, we suppose that (A.1) holds.

Suppose, for instance, that the next instruction to be executed comes from t_1 . Then, since $\mathbf{strat}(\pi_1, t_1)$ is a winning strategy, there exists a winning move by Eve in response to the play $\mathbf{proj}_1(p)$; it is of the form:

$$(\sigma_1, \underline{\sigma}, \sigma_F * \sigma_2) \xrightarrow{m} (\sigma'_1, \underline{\sigma}', \sigma_F * \sigma_2)$$

therefore, the following move is winning in p :

$$(\sigma_1, \sigma_2, \sigma, \sigma_F) \xrightarrow[1]{m} (\sigma'_1, \sigma_2, \sigma', \sigma_F).$$

As noted before, Adam's next move can be lifted from $\mathbf{proj}(p)$ to p .

By construction, the equations (A.1) still hold with the two new moves appended.

Proposition A.6 *Assuming $\mathbf{strat}(\pi_1, t_1)$ and $\mathbf{strat}(\pi_2, t_2)$ are winning strategies, $\mathbf{strat}(\pi, t)$ is a winning strategy.*

A.3 The Sequential Composition

Let π be a proof of $\Gamma \vdash \{P\} C \{Q\}$ and π' be a proof of $\Gamma \vdash \{Q\} C' \{R\}$. Let us denote by $\tilde{\pi}$ the proof of $\Gamma \vdash \{P\} C; C' \{R\}$ built from π and π' .

Let $\tilde{t} \in \llbracket C; C' \rrbracket$, then there exists $t \in \llbracket C \rrbracket$ and $t' \in \llbracket C' \rrbracket$ such that $\tilde{t} = t \cdot t'$. Define $\mathbf{strat}(\tilde{\pi}, \tilde{t})$ to be

$$\mathbf{strat}(\pi, t) \cup \{p \cdot p' \mid p \in \mathbf{strat}(\pi, t), p' \in \mathbf{strat}(\pi', t'), |p| = |t|\} \quad (\text{A.2})$$

where the concatenation $p \cdot p'$ on path of the graph of separated states is defined similarly to the case of traces, only when the endpoints match.

Lemma A.7 *Suppose $\mathbf{strat}(\pi, t)$ and $\mathbf{strat}(\pi', t')$ are winning strategies, then $\mathbf{strat}(\tilde{\pi}, \tilde{t})$ is winning as well.*

Proof Denote by G , G' and \tilde{G} the games associated with t , t' , and \tilde{t} (and their Hoare triples). Notice that $\partial_1 G = \partial_0 G$ and that \tilde{G} is equal to $G \cdot G'$ where the junction point is $\{\mathbf{true}, \Gamma, \mathbf{true}\}$ instead of $\{Q, \Gamma, \mathbf{true}\}$. Moreover, the even length prefixes of $G \cdot G'$ are:

$$G \cup \{G \cdot g' \mid g' \text{ is an even length prefix of } G'\}$$

therefore $\mathbf{strat}(\tilde{\pi}, \tilde{t})$ is a winning strategy for that game, which is harder than \tilde{G} . Hence, $\mathbf{strat}(\tilde{\pi}, \tilde{t})$ is a winning strategy. \square

A.4 Resource Introduction

We assume we have a winning strategy $\mathbf{strat}(\pi, t)$, for the triple $\Gamma, r : J \vdash \{P\} C \{Q\}$ and we want to construct one for $\Gamma \vdash \{P * J\} \mathbf{resource } r \text{ do } C \{Q * J\}$ and $t' = \mathbf{hide}[r](t)$.

We can extend the $\mathbf{hide}[r]$ function to paths of the graph of separated states like so:

$$\begin{aligned} (\sigma_C, \sigma \uplus [r \mapsto \sigma], \sigma_F) &\longmapsto (\sigma_C * \sigma, \sigma \upharpoonright_{\text{dom}(\Gamma)}, \sigma_F) \\ (\sigma_C, \sigma, \sigma_F) &\longmapsto (\sigma_C, \sigma \upharpoonright_{\text{dom}(\Gamma)}, \sigma_F) \end{aligned}$$

and acts on the transitions labels like its counterpart on traces.

It is easy to check that if p is a path in the graph of separated states, then so is $\text{hide}[r](p)$.

Lemma A.8 *If p realizes the trace t , then $\text{hide}[r](p)$ realizes $\text{hide}[r](t)$.*

A.5 Locking

Let π be a proof of $\Gamma \vdash \{(P * J) \wedge B\} C \{Q * J\}$, and π' be that of the proof built from π using the rule **WITH**. Let $t' \in \llbracket \text{with } r \text{ when } B \text{ do } C \rrbracket$.

Let $t \in \llbracket C \rrbracket$ such that $t' \in \text{whentrue}[B](\text{inside}[r](\{t\}))$. Then, since the first instruction of t' is $P(r)$, Adam gives Eve $\sigma \in \mathbf{State}$ such that: $\sigma \models J$. That is, Adam's move is of the form:

$$(\sigma_C, \sigma, \sigma_F) \xrightarrow{env} (\sigma_C * \sigma, \sigma', \sigma'_F)$$

with $r \in \text{dom}(\sigma')$. And, by hypothesis, $(\sigma_C * \sigma, \sigma', \sigma_F) \models B$ (and B is well defined, hence there is no error).

Therefore, that state satisfies the triple $\{(P * J) \wedge B, \Gamma, \mathbf{true}\}$, and thus is a valid starting point for t . Eve can therefore play like $\mathbf{strat}(\pi, t)$ until the end of t . Let p be that play. If $t \in \llbracket C \rrbracket$, then the last node n of p satisfies $\{Q * J, \Gamma, \mathbf{true}\}$, and thus Eve can give back the resource r .

A.6 Framing

Let π be a proof of $\Gamma \vdash \{P\} C \{Q\}$, $t \in \llbracket C \rrbracket$ and R be a predicate. Let π' be the following proof:

$$\frac{\begin{array}{c} \vdots \\ \pi \\ \vdots \end{array} \quad \Gamma \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * R\} C \{Q * R\}} \text{FRAME}$$

By definition, plays of G' , the game associated to π and t , start with a separated state $n = (\sigma_C, \sigma, \sigma_F)$ satisfying $\{P * R, \Gamma, \mathbf{true}\}$, thus $\sigma_C = \bar{\sigma}_C * \sigma_R$, with $\sigma_R \models R$ and $\bar{\sigma}_C \models P$. That is, in each play of $\mathbf{strat}(\pi', t)$ that start with n , σ_R will be a factor of the first component. It is possible because Adam cannot change the first component of states, and because Eve can respond to a play where σ_R is in the third component, by playing like $\mathbf{strat}(\pi, t)$. More formally, if $p \in \mathbf{strat}(\pi', t)$ starts with n , the image of p by the function on states

$$(\sigma_C * \sigma_R, \sigma, \sigma_F) \mapsto (\sigma_C, \sigma, \sigma_F * \sigma_F)$$

belongs to $\mathbf{strat}(\pi, t)$.

Lemma A.9 *$\mathbf{strat}(\pi', t)$ is a winning strategy.*

A.7 Conditional statements

Let C_1, C_2 be commands, E be an arithmetic expression, and C' be the command **if** E **then** C_1 **else** C_2 . Let π' be the proof:

$$\frac{\begin{array}{c} \vdots \\ \pi_1 \\ \vdots \end{array} \quad \begin{array}{c} \vdots \\ \pi_2 \\ \vdots \end{array} \quad \frac{P \Rightarrow \text{def}(B) \quad \Gamma \vdash \{P \wedge B\} C_1 \{Q\} \quad \Gamma \vdash \{P \wedge \neg B\} C_2 \{Q\}}{\Gamma \vdash \{P\} C' \{Q\}} \text{IF}$$

Let $t \in \llbracket C' \rrbracket$. We build the strategy $\mathbf{strat}(\pi', t)$. A play starts at a state n that satisfies $\{P, \Gamma, \mathbf{true}\}$, and since $P \Rightarrow \text{def}(B)$, the first Code transition in t does not fail.

Let us suppose, for example, that B is true in $(\otimes) n$; then $t = \varepsilon \cdot t_1$, with $\varepsilon \in \llbracket \mathbf{nop} \rrbracket$ and $t_1 \in \llbracket C_1 \rrbracket$. Moreover, the next move of Adam is necessarily a separated state that satisfies $\{P \wedge B, \Gamma, \mathbf{true}\}$, so $\mathbf{strat}(\pi, t)$ can then behave exactly like $\mathbf{strat}(\pi_1, t_1)$.

It is easy to see that:

Lemma A.10 $\mathbf{strat}(\pi', t)$ is a winning strategy.

A.8 Loops

Let C be a command, I be a predicate and B be a boolean formula. Assume π' is the proof:

$$\frac{\begin{array}{c} \vdots \\ \pi \\ \vdots \end{array} \quad \frac{I \Rightarrow \text{def}(B) \quad \Gamma \vdash \{I \wedge B\} C \{I\}}{\Gamma \vdash \{I\} C' \{I\}} \text{WHILE}}{\Gamma \vdash \{I\} C' \{I\}} \text{WHILE}$$

where C' is **while** B **do** C . Let $t \in \llbracket C' \rrbracket$, by definition of the semantics for **while**, t can be decomposed as

$$t = \left(\prod_{i=0}^k t_i^\varepsilon \cdot t_i \right) \cdot t_{k+1}$$

with $t_i^\varepsilon \in \llbracket \text{whentrue}[B](\llbracket \mathbf{nop} \rrbracket) \rrbracket$, $t_i \in \llbracket C \rrbracket$ for $0 \leq i \leq k$ and $t_{k+1} \in F(\llbracket C \rrbracket)$ (F is defined in §6).

Let $\mathbf{idstrat}$ be the identity strategy

$$\mathbf{idstrat} = \{n_1 \rightarrow n_2 \rightarrow n_2 \rightarrow n_3 \mid \forall n_1, n_2, n_3\}.$$

We can define the strategy associated to **while** as:

$$\mathbf{strat}(\pi', t) = \prod_{i=0}^{k+1} \mathbf{idstrat} \cdot \mathbf{strat}(\pi, t_i).$$

Lemma A.11 $\mathbf{strat}(\pi, t)$ is a winning strategy.

Proof Let G be the game associated to $\Gamma \vdash \{I\} C' \{I \wedge \neg B\}$ and t , it can be decomposed in a similar way as t and the strategy:

$$G = \left(\prod_{i=0}^k G_{\text{id}}(t_i^\varepsilon) \cdot G_i \right) \cdot G_{k+1}.$$

where for $i \leq k$, $G_{\text{id}}(t_i^\varepsilon)$ is the game whose underlying trace is $t_i^\varepsilon \in \llbracket \text{nop} \rrbracket$, whose precondition is I and whose post-condition is $I \wedge B$; and G_i is the game induced by $\Gamma \vdash \{I \wedge B\} C \{I\}$ and t_i .

None of the transitions of t are errors, since, by induction, $\text{strat}(\pi, t_i)$ is a winning strategy for G_i , and idstrat is a winning strategy for G_{id} .

If $t \in \llbracket C' \rrbracket$, then the end of t comes from $\text{whenfalse}[B](\llbracket \text{nop} \rrbracket)$, therefore the last two states satisfy $\neg B$ (by lemma A.3). \square

B Additional Material on the Semantics

B.1 Handling errors in the semantics

We leave the way the errors are handled unspecified: the semantics is parameterized by a function

$$\text{Handler} : \text{Code} \times \text{MState} \rightarrow \mathcal{O}(\text{MState}).$$

We only ask that **Handler** be total, because, in the sequel, it allows for a more regular definition of the interactive trace semantics. An excerpt definition of those transitions is in Figure C.1, the full definition can be found in the Appendix. Note that we use **Handler** as a non-deterministic function with range **MState**.

The full definition of $\text{whenabort}[P, C]$ makes use of **Handler** as follows:

Definition B.1 *The transition system $\text{whenabort}[P, C]$ associated to a predicate*

$$P : \text{MState} \rightarrow \{\text{true}, \text{false}, \text{abort}\}$$

on memory states and to a code $C \in \text{Code}$ is defined as a set of traces of the form

$$(\sigma_1, L_1) \xrightarrow{\text{env}} (\sigma_2, L_2) \xrightarrow{\text{nop}} (\sigma_3, L_3) \xrightarrow{\text{env}} (\sigma_4, L_4)$$

such that $P(\sigma_2, L_2) = \text{abort}$, and $L_2 = L_3$. The set of returning traces is empty: $|\text{whenabort}[P, C]| = \emptyset$. (For more details, see the Appendix).

B.2 The Operational Semantics

We give our language an operational semantics very close to [12], we define it as a labeled transition system in Figures B.1 and B.2. In order to keep track of which thread holds which lock in the nodes of the labeled transition system, we add, following [12], a new command, **within** r **do** C which represents the state of a thread that has acquired r previously and not yet released it.

B.3 Relating the Two Semantics

The goal of the remainder of this section is to relate the operational semantics above to the interactive semantics of the previous section.

We define the set of operational traces in the same way as the execution traces.

Definition B.2 (Operational Traces) *An operational trace t is a sequence of machine configurations*

$$(C_1, \mathfrak{s}_1) \longrightarrow (C_2, \mathfrak{s}_2) \xrightarrow{m_1} (C_3, \mathfrak{s}_3) \longrightarrow \dots$$

$$\begin{array}{c}
 \frac{B(\sigma) = \mathbf{true}}{(\text{if } B \text{ then } C_1 \text{ else } C_2, (\sigma, L)) \rightsquigarrow^{\text{nop}} (C_1, (\sigma, L))} \\
 \\
 \frac{B(\sigma) = \mathbf{false}}{(\text{if } B \text{ then } C_1 \text{ else } C_2, (\sigma, L)) \rightsquigarrow^{\text{nop}} (C_2, (\sigma, L))} \\
 \\
 \frac{B(\sigma) = \mathbf{abort}}{(\text{if } B \text{ then } C_1 \text{ else } C_2, (\sigma, L)) \rightsquigarrow^{\text{nop}} \text{Handler}(\text{if } B \text{ then } C_1 \text{ else } C_2, (\sigma, L))} \\
 \\
 \frac{B(\sigma) = \mathbf{true}}{(\text{while } B \text{ do } C, (\sigma, L)) \rightsquigarrow^{\text{nop}} (C; \text{while } B \text{ do } C, (\sigma, L))} \\
 \\
 \frac{B(\sigma) = \mathbf{false}}{(\text{while } B \text{ do } C, (\sigma, L)) \rightsquigarrow^{\text{nop}} (\text{skip}, (\sigma, L))} \\
 \\
 \frac{B(\sigma) = \mathbf{abort}}{(\text{while } B \text{ do } C, (\sigma, L)) \rightsquigarrow^{\text{nop}} \text{Handler}(\text{while } B \text{ do } C, (\sigma, L))}
 \end{array}$$

Figure B.1. Operational semantics (control flow)

with an odd number of transitions and whose odd transitions satisfy $C_{2k+1} = C_{2k+2}$, and whose even transitions $\mathfrak{s}_{2k} \xrightarrow{m_k} \mathfrak{s}_{2k+1}$ for $1 \leq k \leq p$ are transitions as defined in Figure B.2. We denote by **OperationalTraces** the set of operational traces.

Note that this closely matches how the environment is handled in [6].

Like in the case of traces, we define operational transition systems.

Definition B.3 (Interactive Operational Semantics) Every code $C \in \mathbf{Code}$ induces a code transition system

$$[C]_{\text{oper}} \subseteq \mathbf{OperationalTraces}$$

as follows: if t is an operational trace, $t \in [C]_{\text{oper}}$ if it starts with a configuration of the form (C, \mathfrak{s}) . A trace returns if its last configuration is of the form $(\text{skip}, \mathfrak{s})$.

Definition B.4 Given an operational trace $t \in \mathbf{OperationalTraces}$, we denote $\bar{t} \in \mathbf{Traces}$ the trace where we have projected the configurations on the second component: $(C, \mathfrak{s}) \mapsto \mathfrak{s}$. We extend this definition to a function from operational transition systems to transition systems.

Theorem B.5 For every $C \in \mathbf{Code}$, $\overline{[C]_{\text{oper}}} = \llbracket C \rrbracket$.

Proof We proceed by induction on C .

Instructions

The result is obvious, since C corresponds to a instruction m .

$$\begin{array}{c}
 \frac{(C_1, \mathfrak{s}) \rightsquigarrow^m (C'_1, \mathfrak{s}')}{(C_1 \parallel C_2, \mathfrak{s}) \rightsquigarrow^m (C'_1 \parallel C_2, \mathfrak{s}')} \qquad \frac{(C_2, \mathfrak{s}) \rightsquigarrow^m (C'_2, \mathfrak{s}')}{(C_1 \parallel C_2, \mathfrak{s}) \rightsquigarrow^m (C_1 \parallel C'_2, \mathfrak{s}')} \\
 \\
 \frac{(C_1, \mathfrak{s}) \rightsquigarrow^m (C'_1, \mathfrak{s}')}{(C_1; C_2, \mathfrak{s}) \rightsquigarrow^m (C'_1; C_2, \mathfrak{s}')} \qquad \frac{(C_1, \mathfrak{s}) \rightsquigarrow^m (C'_1, \mathfrak{s}')}{(C_1; C_2, \mathfrak{s}) \rightsquigarrow^m (C'_1, \mathfrak{s}')} \\
 \\
 \frac{(C, \mathfrak{s}) \rightsquigarrow^m (C', \mathfrak{s}') \quad m \in \{P(r), V(r)\}}{(\text{resource } r \text{ do } C, \mathfrak{s}) \rightsquigarrow^{\text{nop}} (\text{resource } r \text{ do } C', \mathfrak{s})} \\
 \\
 \frac{(C, \mathfrak{s}) \rightsquigarrow^m (C', \mathfrak{s}') \quad m \notin \{P(r), V(r)\}}{(\text{resource } r \text{ do } C, \mathfrak{s}) \rightsquigarrow^m (\text{resource } r \text{ do } C', \mathfrak{s})} \\
 \\
 \frac{B(\sigma) = \mathbf{true} \quad r \notin L}{(\text{with } r \text{ when } B \text{ do } C, (\sigma, L)) \rightsquigarrow^{P(r)} (\text{within } r \text{ do } C, (\sigma, L \cup \{r\}))} \\
 \\
 \frac{B(\sigma) = \mathbf{abort}}{(\text{with } r \text{ when } B \text{ do } C, (\sigma, L)) \rightsquigarrow^{\text{nop}} \text{Handler}(\text{with } r \text{ when } B \text{ do } C, (\sigma, L))} \\
 \\
 \frac{(C, \mathfrak{s}) \rightsquigarrow^m (C', \mathfrak{s}')}{(\text{within } r \text{ do } C, \mathfrak{s}) \rightsquigarrow^m (\text{within } r \text{ do } C', \mathfrak{s}')} \\
 \\
 \frac{r \in L}{(\text{within } r \text{ do skip}, (\sigma, L)) \rightsquigarrow^{V(r)} (\text{skip}, (\sigma, L \setminus \{r\}))} \\
 \\
 \frac{m \in \mathbf{Instr} \quad \mathfrak{s} \rightsquigarrow^m \mathfrak{s}'}{(m, \mathfrak{s}) \rightsquigarrow^m (\text{skip}, \mathfrak{s}')}
 \end{array}$$

Figure B.2. Operational semantics (continued)

$$\begin{array}{c}
 C \Longrightarrow C \qquad \text{skip}; C \Longrightarrow C \qquad \text{skip} \parallel \text{skip} \Longrightarrow \text{skip} \qquad \text{resource } r \text{ do skip} \Longrightarrow \text{skip} \\
 \\
 \frac{C_1 \Longrightarrow C_2 \quad (C_2, \mathfrak{s}) \rightsquigarrow^m (C_3, \mathfrak{s}') \quad C_3 \Longrightarrow C_4}{(C_1, \mathfrak{s}) \rightsquigarrow^m (C_4, \mathfrak{s}')}
 \end{array}$$

Figure B.3. Rewriting rules

Sequential composition

Suppose now that $C = C_1; C_2$. Let $t \in \llbracket C \rrbracket$, there are two cases. If $t \in \llbracket C_1 \rrbracket$, then by induction hypothesis,

$$t \in \overline{[C_1]_{\text{oper}}} \subseteq \overline{[C]_{\text{oper}}}.$$

Otherwise, there exist $t_1 \in \llbracket C_1 \rrbracket$ and $t_2 \in \llbracket C_2 \rrbracket$ such that $t = t_1 \cdot t_2$. By the induction hypothesis, $t_1 \in \overline{[C_1]_{\text{oper}}}$; therefore there exists an operational trace $\tau_1 \in [C_1]_{\text{oper}}$ such that $\overline{\tau_1} = t_1$ and such that $\partial_1 \tau_1$ is of the form $(\mathbf{skip}, \mathfrak{s})$. Thus, there is an operational trace τ'_1 from $C_1; C_2$ to $\mathbf{skip}; C_2$. Moreover by the induction hypothesis there exists a trace $\tau_2 \in [C_2]_{\text{oper}}$ such that $\overline{\tau_2} = t_2$. We can glue τ'_1 and τ_2 thanks to the rewrite rule $\mathbf{skip}; C_2 \Longrightarrow C_2$ into an operational trace τ such that $\overline{\tau} = t$.

Hence, we have proved the inclusion

$$\llbracket C_1; C_2 \rrbracket \subseteq \overline{[C_1; C_2]_{\text{oper}}}.$$

Let $\tau \in [C_1; C_2]_{\text{oper}}$. The operational trace τ induces an operational trace $\tau_1 \in [C_1]_{\text{oper}}$ since the execution of $C_1; C_2$ starts by executing C_1 . Depending on whether the execution of C_1 returns, we can have another trace $\tau_2 \in [C_2]_{\text{oper}}$. We can deduce from the induction hypothesis that $\overline{\tau} \in \llbracket C_1; C_2 \rrbracket$.

Parallel composition

Suppose now that $C = C_1 \parallel C_2$. It suffices to show that:

$$\overline{[C_1 \parallel C_2]_{\text{oper}}} = \llbracket C_1 \rrbracket \parallel \llbracket C_2 \rrbracket.$$

It follows from the fact that $[C_1 \parallel C_2]_{\text{oper}} = [C_1]_{\text{oper}} \parallel [C_2]_{\text{oper}}$. This is the case because an operational trace is obtained by applying one of the two rules that execute an instruction from C_1 or C_2 , alternatively with an arbitrary transition from the environment. And each transition from, say, C_1 can be simulated by the a Environment transition in $\llbracket C_2 \rrbracket$.

Resource introduction

Let $C' = \mathbf{resource} \ r \ \mathbf{do} \ C$, with $C \in \mathbf{Code}$. Let $t' \in \llbracket C' \rrbracket$, there exists $t \in \llbracket C \rrbracket$ such that $t' = \mathbf{hide}[r](t)$. By the induction hypothesis, there is an operational trace τ such that $\overline{\tau} = t$. By applying repeatedly the two rules for **resource** of Figure B.2, we obtain a trace τ' such that $\overline{\tau'} = t'$.

Conversely, if $\tau' \in [C']_{\text{oper}}$, all the Code transtitions must come from the aforementioned two rules and from another operational trace $\tau \in [C]_{\text{oper}}$. By the hypothesis induction, $\overline{\tau} \in \llbracket C \rrbracket$. Thus $\overline{\tau'} \in \llbracket C' \rrbracket$.

Locking

Let $C' = \mathbf{with} \ B \ \mathbf{when} \ r \ \mathbf{do} \ C$, and let $t \in \llbracket C' \rrbracket$. By definition of the semantics,

$$\llbracket C' \rrbracket = \mathbf{whentrue}[B] \left(\mathbf{inside}[r](\llbracket C \rrbracket) \right) \cup \mathbf{whenabort}[B, C'].$$

Suppose $t \in \mathbf{whentrue}[B] \left(\mathbf{inside}[r](\llbracket C \rrbracket) \right)$. If B is false or $r \in L$ in the second state of t , then t has length 1, and, clearly, $t \in \overline{[C']_{\text{oper}}}$. If B is true and $r \notin L$, then

$t \in \llbracket P(r) \rrbracket; \mathbf{T}; \llbracket V(r) \rrbracket$, and it corresponds to an operational trace generated the rule for taking the lock, then, by repeated application of the rule for **within** and by induction, an operational trace of $[C]_{\text{oper}}$, and then the rule to release the lock.

Conversly, an operational trace $\tau \in [C']_{\text{oper}}$ is necessarily generated as described above, so its erasure $\bar{\tau}$ is in $\llbracket C' \rrbracket$.

Conditional statement

This case is similar to the previous one and the one for sequential composition, since, by definition, the semantics of $C' = \text{if } B \text{ then } C_1 \text{ else } C_2$ is:

$$\begin{aligned} \llbracket \text{if } B \text{ then } C_1 \text{ else } C_2 \rrbracket &= \text{whentrue}[B](\llbracket \text{nop} \rrbracket); \llbracket C_1 \rrbracket \\ &\cup \text{whenfalse}[B](\llbracket \text{nop} \rrbracket); \llbracket C_2 \rrbracket \\ &\cup \text{whenabort}[B, C'], \end{aligned}$$

Loops

Similar to the case for conditional statements. □

C Full Definitions

$$\begin{array}{c}
 \frac{}{(\sigma, L) \rightsquigarrow^{\text{nop}} (\sigma, L)} \qquad \frac{E(\sigma) = v}{(\sigma, L) \rightsquigarrow^{x:=E} (\sigma[x \mapsto v], L)} \\
 \\
 \frac{E(\sigma) = \mathbf{abort}}{(\sigma, L) \rightsquigarrow^{x:=E} \text{Handler}(x := E, (\sigma, L))} \\
 \\
 \frac{E(\sigma) = l \quad l \in \text{hdom}(\sigma)}{(\sigma, L) \rightsquigarrow^{x:=\llbracket E \rrbracket} (\sigma[x \mapsto \sigma(l)], L)} \\
 \\
 \frac{E(\sigma) = \mathbf{abort} \quad \text{or} \quad E(\sigma) \notin \text{hdom}(\sigma)}{(\sigma, L) \rightsquigarrow^{x:=\llbracket E \rrbracket} \text{Handler}(x := \llbracket E \rrbracket, (\sigma, L))} \\
 \\
 \frac{E_1(\sigma) = l \quad E_2(\sigma) = v \quad l \in \text{hdom}(\sigma)}{(\sigma, L) \rightsquigarrow^{\llbracket E_1 \rrbracket := E_2} (\sigma[l \mapsto v], L)} \\
 \\
 \frac{E_1(\sigma) = \mathbf{abort} \quad \text{or} \quad E_2(\sigma) = \mathbf{abort} \quad \text{or} \quad E_1(\sigma) \notin \text{hdom}(\sigma)}{(\sigma, L) \rightsquigarrow^{\llbracket E_1 \rrbracket := E_2} \text{Handler}(\llbracket E_1 \rrbracket := E_2, (\sigma, L))} \\
 \\
 \frac{E(\sigma) = v \quad l \notin \text{hdom}(\sigma)}{(\sigma, L) \rightsquigarrow^{x:=\text{alloc}(E)} (\sigma[x \mapsto l] \uplus [l \mapsto v], L)} \\
 \\
 \frac{E(\sigma) = \mathbf{abort}}{(\sigma, L) \rightsquigarrow^{x:=\text{alloc}(E)} \text{Handler}(x := \text{alloc}(E), (\sigma, L))} \\
 \\
 \frac{E(\sigma) = l \quad l \in \text{hdom}(\sigma)}{(\sigma, L) \rightsquigarrow^{\text{dispose}(E)} (\sigma \setminus l, L)} \qquad \frac{E(\sigma) = \mathbf{abort} \quad \text{or} \quad l \notin \text{hdom}(\sigma)}{(\sigma, L) \rightsquigarrow^{\text{dispose}(E)} \text{Handler}(\text{dispose}(E), (\sigma, L))} \\
 \\
 \frac{r \notin L}{(\sigma, L) \rightsquigarrow^{P(r)} (\sigma, L \uplus \{r\})} \qquad \frac{r \in L}{(\sigma, L) \rightsquigarrow^{V(r)} (\sigma, L \setminus \{r\})} \\
 \\
 \frac{r \notin L}{(\sigma, L) \rightsquigarrow^{V(r)} \text{Handler}(V(r), (\sigma, L))}
 \end{array}$$

Figure C.1. Semantics of instructions

$$\begin{array}{c}
 \frac{}{\Gamma \vdash \{\text{Own}_\top(x) * X = E\} x := E \{\text{Own}_\top(x) * x = X\}} \text{AFF} \\
 \\
 \frac{}{\Gamma \vdash \{E \mapsto -\} [E] := E' \{E \mapsto E'\}} \text{STORE} \\
 \\
 \frac{x \notin \text{fv}(E)}{\Gamma \vdash \{E \mapsto^p v * \text{Own}_\top(x)\} x := [E] \{E \mapsto^p v * \text{Own}_\top(x) * x = v\}} \text{LOAD} \\
 \\
 \frac{}{\Gamma \vdash \{\text{Own}_\top(x)\} x := \text{alloc}(E) \{\text{Own}_\top(x) * x = l * l \mapsto E\}} \text{ALLOC} \\
 \\
 \frac{}{\Gamma \vdash \{E \mapsto -\} \text{dispose } E \{\mathbf{emp}\}} \text{DISP} \quad \frac{\Gamma \vdash \{P\} C \{Q\} \quad \Gamma \vdash \{Q\} C' \{R\}}{\Gamma \vdash \{P\} C; C' \{R\}} \text{SEQ} \\
 \\
 \frac{P \Rightarrow \text{def}(B) \quad \Gamma \vdash \{P \wedge B\} C_1 \{Q\} \quad \Gamma \vdash \{P \wedge \neg B\} C_2 \{Q\}}{\Gamma \vdash \{P\} \text{if } B \text{ then } C_1 \text{ else } C_2 \{Q\}} \text{IF} \\
 \\
 \frac{I \Rightarrow \text{def}(B) \quad \Gamma \vdash \{I \wedge B\} C \{I\}}{\Gamma \vdash \{I\} \text{while } B \text{ do } C \{I \wedge \neg B\}} \text{WHILE} \\
 \\
 \frac{\Gamma \text{ is precise} \quad \Gamma \vdash \{P_1\} C \{Q_1\} \quad \Gamma \vdash \{P_2\} C \{Q_2\}}{\Gamma \vdash \{P_1 \wedge P_2\} C \{Q_1 \wedge Q_2\}} \text{CONJ} \\
 \\
 \frac{\Gamma, r : J \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * J\} \text{resource } r \text{ do } C \{Q * J\}} \text{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\}} \text{DISJ} \\
 \\
 \frac{\Gamma, r : J \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * J\} \text{resource } r \text{ do } C \{Q * J\}} \text{RES} \\
 \\
 \frac{P \Rightarrow \text{def}(B) \quad \Gamma \vdash \{(P * J) \wedge B\} C \{Q * J\}}{\Gamma, r : J \vdash \{P\} \text{with } r \text{ when } B \text{ do } C \{Q\}} \text{WITH} \\
 \\
 \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\}} \text{PAR} \quad \frac{\Gamma \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * R\} C \{Q * R\}} \text{FRAME} \\
 \\
 \frac{P' \Rightarrow P \quad \Gamma \vdash \{P\} C \{Q\} \quad Q \Rightarrow Q'}{\Gamma \vdash \{P'\} C \{Q'\}} \text{CONSEQ}
 \end{array}$$

Figure C.2. Inference rules of Concurrent Separation Logic