Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 336 (2018) 241–256

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Game Semantics of Concurrent Separation Logic

Paul-André Melliès[a] Léo Stefanesco[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.

Keywords: Concurrent separation logic, game semantics specification logic

1 Introduction

Concurrent separation logic (CSL) is an extension of Reynold's separation logic [12] formulated by O'Hearn [10] 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

https://doi.org/10.1016/j.entcs.2018.03.026

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 separation logic [5,6]. His soundness proof was the object of great attention in the community, and it was revisited in a number of different ways, either semantic [13], syntactic [2] or axiomatic [7] and formalised in proof assistants. One main technical challenge in all these proofs of soundness is to establish the validity of the concurrent rule:

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

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

This leads us to a "span"

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

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

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

refines the logical state $\circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ defined by the separation tensor product

$$\circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \quad \stackrel{\text{def}}{=} \quad \sigma_C * \left\{ \bigcircledast_{r \in \mathrm{dom}(\boldsymbol{\sigma})} \boldsymbol{\sigma}(r) \right\} * \sigma_F \tag{2}$$

where $\mathrm{dom}(\boldsymbol{\sigma})$ denotes the set of resources available in $\boldsymbol{\sigma}$, in the sense that $\boldsymbol{\sigma}(r) \neq C, F$. Similarly, every separated state $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ 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 $L = \mathrm{dom_C}(\boldsymbol{\sigma}) \uplus \mathrm{dom_F}(\boldsymbol{\sigma})$, 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 $\sigma = \circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \in \mathbf{LState}$ 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 [5,6] 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:

$$\texttt{read}\ 71\ \texttt{from}\ x,\ \texttt{read}\ 36\ \texttt{from}\ y,\ \texttt{acquire lock r}, \ldots.$$

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

$$\texttt{write}\ 89\ \texttt{in}\ x, \texttt{read}\ 14\ \texttt{from}\ x$$

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 [13] 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 $\boldsymbol{\sigma}$ which tracks the states of each of the available locks.

In contrast to the semantic proofs mentioned above, Balabonski, Pottier and Protzenko [2] 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 [8,9].

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 $s : \mathbf{Var} \rightharpoonup_{fin} \mathbf{Val}$ 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* **State**. *The domains of the partial function s and of h are noted* $\mathrm{vdom}(\mu)$ *and* $\mathrm{hdom}(\mu)$ *respectively, and we write* $\mathrm{dom}(\mu)$ *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 $L \subseteq$ **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:

$$\mathfrak{s} \overset{m}{\rightsquigarrow} \mathfrak{s}' \qquad\qquad \mathfrak{s} \overset{m}{\Rightsquigarrow} \mathfrak{s}'$$

depending on whether the instruction $m \in$ **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:

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

where $x \in$ **Var** is a variable, $r \in$ **LockName** is a resource variable, and E, E' are arithmetic expressions with variables. Typically, the instruction $x \coloneqq E$ 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:

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

Thanks to the inclusion **Loc** $\subseteq$ **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 `nop` (for no-operation) does not alter the logical state, while $x \coloneqq \mathtt{alloc}(E)$ 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 $\mathtt{dispose}(E)$ deallocates the location with address E.

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

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

whose even transitions

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

are labelled by an instruction $m_k \in \mathbf{Instr}$ *such that* $\mathfrak{s}_{2k} \overset{m_k}{\rightsquigarrow} \mathfrak{s}_{2k+1}$ *and whose last transition is played by the environment. The set of traces is denoted by* **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 $\mathsf{len}(t) = p$ is defined as the number of Code transitions in the trace, and

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

denotes the k-th even transition of the trace t, for $1 \leq k \leq \mathsf{len}(t)$. Observe that a trace t always starts and stops by an Environment transition, and that its number of transitions is equal to $2 \times \mathsf{len}(t) + 1$. 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* $1 \leq k \leq \mathsf{len}(t)$.

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* $\mathsf{len}(t_1) + \mathsf{len}(t_2)$ *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* $f : \{1, ..., p\} \to \{1, ..., q\}$ *induces a restriction function*

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

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

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

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* $\omega : \{1, \ldots, p\} \uplus \{1, \ldots, q\} \to \{1, \ldots, p+q\}$*. 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

$$\omega_1 : \{1, ..., p\} \to \{1, \ldots, p+q\} \quad \text{and} \quad \omega_2 : \{1, ..., q\} \to \{1, \ldots, p+q\}$$

defined by restricting ω to $\{1, ..., p\}$ and to $\{1, ..., q\}$, 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* $\omega \in \mathbf{Shuffles}(\mathsf{len}(t_1), \mathsf{len}(t_2))$.

Note that every trace t in $t_1 \parallel t_2$ satisfies $\mathsf{len}(t) = \mathsf{len}(t_1) + \mathsf{len}(t_2)$ 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 coinitial and cofinal.

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

Definition 4.8 *The function* $\mathsf{hide}[r] : \mathbf{Traces} \to \mathbf{Traces}$ *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

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

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 $\mathbf{T} = (T, \lfloor T \rfloor)$ *is a set of traces* $T \subseteq \mathbf{Traces}$ *closed under prefix, together with a subset* $\lfloor T \rfloor \subseteq T$, *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:*

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

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:*

$$T_1 \parallel T_2 = \bigcup_{t_i \in T_i} t_1 \parallel t_2 \qquad \lfloor T_1 \parallel T_2 \rfloor = \bigcup_{t_i \in \lfloor T_i \rfloor} t_1 \parallel t_2$$

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

$$\mathsf{hide}[r](T) = \{\mathsf{hide}[r](t) \mid t \in T\} \qquad \lfloor \mathsf{hide}[r](T) \rfloor = \{\mathsf{hide}[r](t) \mid t \in \lfloor T \rfloor\}.$$

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

$$\begin{aligned} [\![m]\!] &= \{\mathfrak{s}_1 \xrightarrow{env} \mathfrak{s}_2 \xrightarrow{m} \mathfrak{s}_3 \xrightarrow{env} \mathfrak{s}_4 \mid \mathfrak{s}_2 \rightsquigarrow \mathfrak{s}_3 \} \\ \big|[\![m]\!]\big| &= \{\mathfrak{s}_1 \xrightarrow{env} \mathfrak{s}_2 \xrightarrow{m} \mathfrak{s}_3 \xrightarrow{env} \mathfrak{s}_4 \mid \mathfrak{s}_2 \rightsquigarrow \mathfrak{s}_3 \} \end{aligned}$$

The intuition is that the program interpreted by $[\![m]\!]$ executes the instruction m after the environment has made the transition $\mathfrak{s}_1 \xrightarrow{env} \mathfrak{s}_2$ and returns when the machine step $\mathfrak{s}_2 \xrightarrow{m} \mathfrak{s}_3$ 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* $\mathsf{inside}[r](\mathbf{T})$ *associated to a transition system* $\mathbf{T}$ *and to a lock* $r \in \mathbf{LockName}$ *is defined as follows:*

$$\mathsf{inside}[r](\mathbf{T}) = [\![P(r)]\!]; \mathbf{T}; [\![V(r)]\!].$$

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

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

$$\begin{aligned} \mathsf{whentrue}[P](\mathbf{T}) &= \{t \in T \mid P(\overline{\partial}_0 t) = \mathbf{true}\} \\ \big|\mathsf{whentrue}[P](\mathbf{T})\big| &= \{t \in |T| \mid P(\overline{\partial}_0 t) = \mathbf{true}\} \end{aligned}$$

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

The transition system $\mathsf{whenfalse}[P](\mathbf{T})$ 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 $\mathsf{whenabort}[P, C]$, whose construction is detailed in the Appendix[1].

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:

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

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 $\texttt{resource}\ r$ and acquired using $\texttt{with}\, r\, \texttt{when}\, B\, \texttt{do}\, C$, 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 $[\![C]\!]$ which describes the possible interactive executions of C, and whether they return.

$$\text{Code } C \quad \xrightarrow{\text{translation}} \quad \text{Transition system } [\![C]\!]$$

The interpretation $[\![C]\!]$ is defined by structural induction on the syntax of the command C. To each leaf node C, one associates an instruction $m \in \mathbf{Instr}$

$$x \coloneqq E \mid x \coloneqq [E] \mid [E] \coloneqq [E'] \mid \texttt{nop} \mid x \coloneqq \texttt{alloc}(E) \mid \texttt{dispose}(E)$$

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

$$[\![C \parallel C']\!] \stackrel{\text{def}}{=} [\![C]\!] \parallel [\![C']\!], \qquad [\![C; C']\!] \stackrel{\text{def}}{=} [\![C]\!]; [\![C']\!],$$

$$[\![\texttt{resource}\, r\, \texttt{do}\, C]\!] \stackrel{\text{def}}{=} \mathsf{hide}[r]\,\big([\![C]\!]\big),$$

$$[\![\texttt{with}\, r\, \texttt{when}\, B\, \texttt{do}\, C]\!] \stackrel{\text{def}}{=} \mathsf{whentrue}[B]\Big(\mathsf{inside}[r]\big([\![C]\!]\big)\Big) \cup \mathsf{whenabort}[B, C']$$

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

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

and the while loop

$$[\![\texttt{while}\, B\, \texttt{do}\, C]\!] \stackrel{\text{def}}{=} \bigcup_{n \geq 0} F^n(\emptyset)$$

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

$$\begin{aligned} F(\mathbf{T}) \quad = \quad & \mathsf{whentrue}[B]\big([\![\texttt{nop}]\!]\big); [\![C]\!]; \mathbf{T} \ \cup\ \mathsf{whenfalse}[B]\big([\![\texttt{nop}]\!]\big) \ \cup \\ & \mathsf{whenabort}[B, \texttt{while}\, B\, \texttt{do}\, C]. \end{aligned}$$

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

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

Figure 1. Semantics of the predicates of concurrent separation logic

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 [10,5]. One difference is that we benefit from the work in [3,4,11] and use the permissions and the $\mathrm{Own}_p(x)$ predicate in order to handle the heap as well as variables in the stack. So, we suppose given an arbitrary partial cancellative commutative monoid **Perm** that we call the *permission monoid*, following [3]. 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 **LState** of logical states is defined in a similar way as the set **State** of memory states, with the addition of permissions:

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

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

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

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 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:

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

The semantics of the formulas is expressed as the satisfaction predicate $\sigma \vDash P$ 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 $\Gamma = r_1 : J_1, \ldots, r_k : J_k$ 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 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 $\Gamma = r_1 : J_1, \ldots, r_k : J_k$ 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 $\sigma \in$ **LState**, there exists at most one $\sigma' \in$ **LState** such that $\exists \sigma'',\ \sigma = \sigma' * \sigma''$ and $\sigma' \vDash P$.*

$$\frac{}{\Gamma \vdash \{\mathrm{Own}_\top(x) * X = E\}\, x := E\, \{\mathrm{Own}_\top(x) * x = X\}}\ \textsc{Aff} \qquad \frac{}{\Gamma \vdash \{E \mapsto -\}\, [E] := E'\, \{E \mapsto E'\}}\ \textsc{Store}$$

$$\frac{x \notin \mathrm{fv}(E)}{\Gamma \vdash \{E \mapsto^p v * \mathrm{Own}_\top(x)\}\, x := [E]\, \{E \mapsto^p v * \mathrm{Own}_\top(x) * x = v\}}\ \textsc{Load}$$

$$\frac{\Gamma \vdash \{P\}\, C\, \{Q\} \quad \Gamma \vdash \{Q\}\, C'\, \{R\}}{\Gamma \vdash \{P\}\, C; C'\, \{R\}}\ \textsc{Seq}$$

$$\frac{P \Rightarrow \mathrm{def}(B) \quad \Gamma \vdash \{P \wedge B\}\, C_1\, \{Q\} \quad \Gamma \vdash \{P \wedge \neg B\}\, C_2\, \{Q\}}{\Gamma \vdash \{P\}\, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\, \{Q\}}\ \textsc{If}$$

$$\frac{\Gamma \text{ is precise} \quad \Gamma \vdash \{P_1\}\, C\, \{Q_1\} \quad \Gamma \vdash \{P_2\}\, C\, \{Q_2\}}{\Gamma \vdash \{P_1 \wedge P_2\}\, C\, \{Q_1 \wedge Q_2\}}\ \textsc{Conj} \qquad \frac{\Gamma, r : J \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P * J\}\, \texttt{resource}\ r\ \texttt{do}\ C\, \{Q * J\}}\ \textsc{Res}$$

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

$$\frac{\Gamma \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P * R\}\, C\, \{Q * R\}}\ \textsc{Frame}$$

Figure 2. Inference rules of Concurrent Separation Logic

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.

Definition 8.1 *The* separated states *are the triples*

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

such that the state below is defined:

$$\sigma_C * \left\{ \bigoplus_{r \in \mathrm{dom}(\boldsymbol{\sigma})} \boldsymbol{\sigma}(r) \right\} * \sigma_F$$

where

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

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

$$\sigma_C * \left\{ \bigoplus_{r \in \mathrm{dom}(\boldsymbol{\sigma})} \boldsymbol{\sigma}(r) \right\} * \sigma_F \quad \in \quad \mathbf{LState} \tag{3}$$

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

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

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

which maps every separated state $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ 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*

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

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

$$\circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \rightarrowtail^{m} \circledast(\sigma'_C, \boldsymbol{\sigma}', \sigma_F)$$

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

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

- *Adam moves of the form*

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

where env is just a tag, and moreover

$$\mathrm{dom}_{\mathrm{C}}(\boldsymbol{\sigma}') = \mathrm{dom}_{\mathrm{C}}(\boldsymbol{\sigma}).$$

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* p living in the graph of $\mathbf{G}(\mathbf{SState})$ of separated states, and such that $t = \circledast\, p$. In that situation, we use the following terminology:

Definition 9.3 We say that a path 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 $t = \circledast\, p$.

Note that a path 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* $A = (\mathsf{Board}_A, \mathsf{Pol}_A, \mathsf{Plays}_A)$ *consisting of a graph* $\mathsf{Board}_A = (V, E, \partial_0, \partial_1)$ *with source and target functions* $\partial_0, \partial_1 : E \to V$*, and whose edges are called moves: of a function* $\mathsf{Pol}_A : E \to \{-1, +1\}$ *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 $\mathsf{Pol}_A(e_i) = (-1)^i$ *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 $s \in \mathsf{Plays}_A$ 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* $\mathbf{SGame}(t) = (\mathsf{Board}, \mathsf{Pol}, \mathsf{Plays})$ *is defined as the graph* $\mathsf{Board} = \mathbf{G}(\mathbf{SState})$ *with plays in* Plays *defined as the paths*

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

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

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

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* $\mathsf{Board} = \mathbf{G}(\mathbf{SState})$ *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 $m : (\sigma, L) \to (\sigma', L')$ performed by the Code in the execution trace $t \in \mathbf{Traces}$ starting from a machine state $(\sigma, L) = \circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$, Eve has to play a move $m : (\sigma_C, \boldsymbol{\sigma}, \sigma_F) \to (\sigma'_C, \boldsymbol{\sigma}', \sigma'_F)$ which "justifies" the transition by decomposing the machine state (σ', L') into a separated state $(\sigma'_C, \boldsymbol{\sigma}', \sigma'_F)$. 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 $t \in [\![C]\!]$ 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* $\Gamma = r_1 : J_1, \ldots, r_k : J_k$ *of variable resources.*

Definition 11.2 *We write*

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \vDash (P, \Gamma, Q)$$

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

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

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

for $1 < i < 2p + 1$ when the execution trace $t \in \big|[\![C]\!]\big|$ is returning; and defined as

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

for $1 < i < 2p+2$ when the execution trace $t \notin \left|[\![C]\!]\right|$ is not returning. Here, we write **true** for the constant predicate which is true for every logical state.

Definition 11.3 (Winning condition) *A play*

$$(\sigma_C^1, \boldsymbol{\sigma}^1, \sigma_F^1) \xrightarrow{env} (\sigma_C^2, \boldsymbol{\sigma}^2, \sigma_F^2) \xrightarrow{m_1} (\sigma_C^3, \boldsymbol{\sigma}^3, \sigma_F^3) \to \cdots \to (\sigma_C^{2q+2}, \boldsymbol{\sigma}^{2q+2}, \sigma_F^{2q+2})$$

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

$$\forall i \in \{1, \ldots, 2q+2\}, \qquad (\sigma_C^i, \boldsymbol{\sigma}^i, \sigma_F^i) \vDash \mathbf{P}_i.$$

Note that the notion of winning play is closed under prefix.

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

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

Note that an empty and winning play of the separation game consists of a separated state $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ satisfying the predicate $(P, \Gamma, \mathbf{true})$, and in the very special case when the trace $t \in \left|[\![C]\!]\right|$ 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* $t \in [\![C]\!]$ *a winning strategy* $\mathtt{strat}(\pi, t)$ *in the separation game* $\mathbf{SGame}(t)$ *determined by the Hoare triple* $\Gamma \vdash \{P\}\, C\, \{Q\}$ *and* t*.*

The proof of the theorem is in the Appendix[1]. This statement is inspired by game semantics, and the idea of a Curry-Howard correspondence between proofs (derivation trees) and winning strategies. This interpretation of proofs implies the soundness of concurrent separation logic in the traditional sense [5,13,2] 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*

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

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* $t \in [\![C]\!]$ *is an execution trace*

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

in which the Environment is passive, and such that $\mathfrak{s}_1 \vDash P * \mathbf{true}$*. Then, the execution trace* t *produces no error, in the technical sense that every machine step* $m_i : s_{2i+1} \rightsquigarrow s_{2i+3}$ *executed by the Code, for* $0 \leq i \leq p-1$ *is of the form*

$s_{2i+1} \overset{m_i}{\rightsquigarrow} s_{2i+3}$ *and thus does not produce any error at run-time. Moreover, when* $t \in \left|C\right|$ *returns, one has that:*

$$\partial_1 t \vDash Q * \mathbf{true}\,.$$

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] *Appendix to a game semantics of concurrent separation logic.* URL http://stefanesco.com/documents/mfps17.pdf

[2] Balabonski, T., F. Pottier and J. Protzenko, *Type soundness and race freedom for mezzo*, in: *FLOPS*, 2014.

[3] Bornat, R., C. Calcagno, P. O'Hearn and M. Parkinson, *Permission accounting in separation logic*, in: *POPL*, 2005.

[4] Bornat, R., C. Calcagno and H. Yang, *Variables as resource in separation logic*, ENTCS **155** (2006), pp. 247–276.

[5] Brookes, S., *A semantics for concurrent separation logic*, in: *CONCUR*, 2004.

[6] Brookes, S., *A revisionist history of concurrent separation logic*, ENTCS **276** (2011), pp. 5 – 28.

[7] Dinsdale-Young, T., L. Birkedal, P. Gardner, M. J. Parkinson and H. Yang, *Views: compositional reasoning for concurrent programs*, in: *POPL*, 2013.

[8] 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.

[9] 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.

[10] OHearn, P. W., *Resources, concurrency, and local reasoning*, TCS **375** (2007), pp. 271–307.

[11] Parkinson, M. J., R. Bornat and C. Calcagno, *Variables as resource in hoare logics*, in: *LICS*, 2006, pp. 137–146.

[12] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *LICS*, 2002.

[13] Vafeiadis, V., *Concurrent separation logic and operational semantics*, ENTCS **276** (2011), pp. 335–351.