Concurrent Separation Logic Meets Template Games

PAUL-ANDRÉ MELLIÈS and LÉO STEFANESCO, IRIF, CNRS & Université de Paris, France

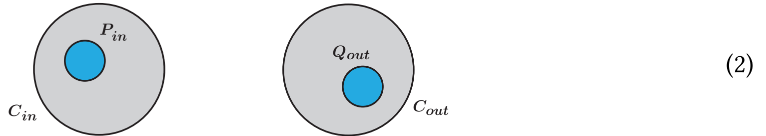
An old dream of concurrency theory and programming language semantics has been to uncover the fundamental synchronization mechanisms which regulate situations as different as game semantics for higher-order programs, and Hoare logic for concurrent programs with shared memory and locks. In this paper, we establish a deep and unexpected connection between two recent lines of work on concurrent separation logic (CSL) and on template game semantics for differential linear logic (DiLL). Thanks to this connection, we reformulate in the purely conceptual style of template games for DiLL the asynchronous and interactive interpretation of CSL designed by Melliès and Stefanesco. We believe that the analysis reveals something important about the secret anatomy of CSL, and more specifically about the subtle interplay, of a categorical nature, between sequential composition, parallel product, errors and locks.

1 INTRODUCTION

There is a fascinating analogy between the notion of Hoare triple $\{P\}C\{Q\}$ in programming language semantics, and a stream of elegant ideas coming from differential geometry and mathematical physics. Consider the typical situation of a code C written in an imperative and sequential programming language, and executed on a set *State* of machine states. By construction, the code C comes equipped with a set C_{in} of input states and a set C_{out} of output states, together with a pair of labeling functions

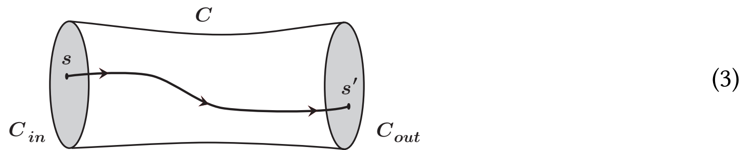
$$\lambda_{in} : C_{in} \longrightarrow State \qquad \lambda_{out} : C_{out} \longrightarrow State \tag{1}$$

which assign to every input state $s \in C_{in}$ and output state $s' \in C_{out}$ of the code C its underlying machine state $\lambda_{in}(s) \in State$ and $\lambda_{out}(s') \in State$. Following the philosophy of Hoare logic, the predicates P and Q describe specific subsets of the set *State* of states of the machine, which induce (by inverse image) predicates P_{in} on C_{in} and Q_{out} on C_{out}. The Hoare triple $\{P\}C\{Q\}$ then expresses the fact that the code C transports every input state $s \in C_{in}$ which satisfies the predicate P_{in} into an output state $s' \in C_{out}$ which satisfies the predicate Q_{out}. Looking at the situation with the eyes of the physicist, the set C_{in} of input states can be depicted as a two-dimensional disk (in gray) with the predicate P_{in} represented as a subset or subdisk (in blue) living inside C_{in} ; and similarly for the predicate Q_{out} on the set C_{out} of output states:



(2)

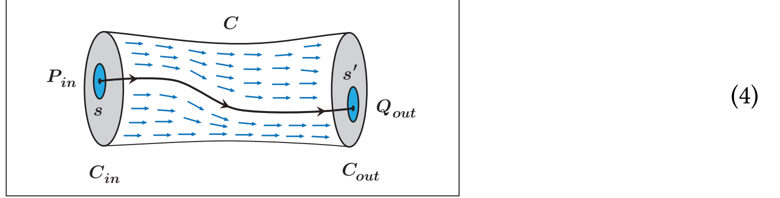
Since the purpose of the code C is to transport the input states $s \in C_{in}$ to output states $s' \in C_{out}$, it makes sense to depict C (in white) as a three-dimensional tube or cylinder connecting the input disk C_{in} to the output disk C_{out}. Here, the three-dimensional cylinder C should be understood as a geometric object living in space and time, and describing the evolution in time of the internal states of the code C in the course of execution:



(3)

Authors' address: Paul-André Melliès, mellies@irif.fr; Léo Stefanesco, leo.stefanesco@irif.fr, IRIF, CNRS & Université de Paris, 8 place Aurélie Nemours, Paris, 75013, France.

The physical intuition that the code C transports states $s \in C_{in}$ to states $s' \in C_{out}$ may be expressed by equipping the cylinder with a (three-dimensional) vector field describing how a state evolves in time through the code C. The description is reminiscent of fluid mechanics, and the way one describes the trajectory of a particle along a flow in the cylinder C. In that graphical representation, the Hoare triple $P\{C\}Q$ indicates that the flow of execution described by the vector field on the cylinder C transports every state $s \in C_{in}$ starting from P_{in} to a state $s' \in C_{out}$ exiting in Q_{out}:

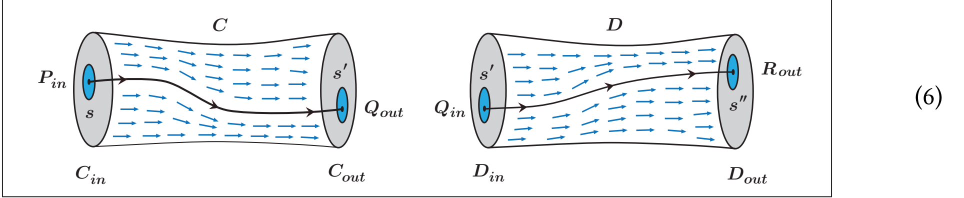


(4)

This intuition underlies the sequential rule of Hoare logic

$$\frac{\{P\}C\{Q\} \qquad \{Q\}D\{R\}}{\{P\}C;D\{R\}} \quad \text{sequential composition} \tag{5}$$

which states that Hoare triples "compose well" in the sense that the Hoare triple $\{P\}C;D\{R\}$ holds whenever the Hoare triples $\{P\}C\{Q\}$ and $\{Q\}D\{R\}$ are assumed to hold. This basic principle of Hoare logic reflects the fact that when the two codes C and D are executed sequentially as below



(6)

every state $s \in C_{in}$ satisfying the predicate P_{in} is transported by the code C to a transitory state $s' \in C_{out}$ satisfying the predicate Q_{out}; and that the same transitory state $s' \in D_{in}$ taken now as input is transported by the code D to a state $s'' \in D_{out}$ satisfying the predicate R_{out}. The careful reader will notice that we make here the simplifying and somewhat unrealistic assumption that the set C_{out} of output states of the code C coincides with the set D_{in} of input states of the code D; as we will see, this point is interesting and far from anecdotal, and we will thus come back to it with great attention at a later stage of the paper, see §5.

Cospans of transition systems. By taking seriously these geometric intuitions and formalizing them in the language of category theory, we establish in this paper a strong and unexpected connection between two recent and largely independent lines of work on concurrent separation logic [18, 19] and on template games for differential linear logic [16, 17]. The connection enables us to disclose for the first time a number of basic and fundamental categorical structures underlying concurrent separation logic, and more specifically, the proof of the asynchronous soundness theorem established in [19]. Our starting point is to define a **transition system** (A, λ_A) on a given labeling graph $\pm_{label}$ as a morphism

$$\lambda_A \quad : \quad A \longrightarrow \pm_{label} \tag{7}$$

in the category **Gph** of directed graphs. The graph A is called the *support* of the transition system (A, λ_A), and λ_A its *labeling map*. It is important here that the transition system (A, λ_A) is labeled with a graph instead of a set, and that it is multi-sorted with sorts provided by the vertices of the

labeling graph $\pm_{label}$. A morphism between two such transition systems

$$(f,\varphi) \quad : \quad (A,\lambda_A) \longrightarrow (B,\lambda_B) \tag{8}$$

is defined as a pair (f,φ) of graph morphisms making the diagram below commute

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ {\scriptstyle \lambda_A}\downarrow & & \downarrow{\scriptstyle \lambda_B} \\ \pm_{label,1} & \xrightarrow{\quad \varphi \quad} & \pm_{label,2} \end{array} \tag{9}$$

in the category **Gph**. The graph morphism $\varphi : \pm_{label,1} \to \pm_{label,2}$ between labeling graphs works in the same way as a "change of base" and is called the *relabeling map* of the morphism (8). At this stage, we are ready to formulate the guiding idea of the paper, which is that the situation of the three-dimensional cylinder C connecting the sets C_{in} and C_{out} of input and output states depicted in (3) can be conveniently formulated as a *cospan of transition systems*

$$(C_{in},\lambda_{in}) \xrightarrow{\quad (in,\eta) \quad} (C,\lambda_{code}) \xleftarrow{\quad (out,\eta) \quad} (C_{out},\lambda_{out}) \tag{10}$$

defining a commutative diagram in the category **Gph**:

$$\begin{array}{ccccc} C_{in} & \xrightarrow{\quad in \quad} & C & \xleftarrow{\quad out \quad} & C_{out} \\ {\scriptstyle \lambda_{in}}\downarrow & & \downarrow{\scriptstyle \lambda_{code}} & & \downarrow{\scriptstyle \lambda_{out}} \\ \pm_S[0] & \xrightarrow{\quad \eta \quad} & \pm_S[1] & \xleftarrow{\quad \eta \quad} & \pm_S[0] \end{array} \tag{11}$$

One important feature of the cospan formulation of (3) is the simple and intuitive definition of the labeling graphs $\pm_S[0]$ and $\pm_S[1]$ and of the relabeling map $\eta : \pm_S[0] \to \pm_S[1]$. Both labeling graphs $\pm_S[0]$ and $\pm_S[1]$ have the set *State* of machine states as set of vertices. The difference between them is that the graph $\pm_S[0]$ is discrete (that is, it has no edge) and can be thus identified with the set *State* itself, while the graph $\pm_S[1]$ has as edges the transitions $inst : s \to s'$ performed by the instructions $inst \in Inst$ of the machine. The relabeling map

$$\eta \; : \; \pm_S[0] \longrightarrow \pm_S[1] \tag{12}$$

is the graph morphism which maps every machine state $s \in State$ to itself. The graph C together with the label map $\lambda_{code} : C \to \pm_S[1]$ describe the transition system (C,λ_{code}) defined by the operational semantics of the code. Note in particular that λ_{code} labels every vertex (or internal state) of the graph C with a machine state $\lambda_{code}(s) \in State$ and every edge (or execution step) $m : s \to s'$ of the graph C with a machine instruction $\lambda_{code}(m) : \lambda_{code}(s) \to \lambda_{code}(s')$. The sets C_{in} and C_{out} of input and output states of the code C are understood in (10) as discrete graphs, and similarly for the functions $\lambda_{in} : C_{in} \to State$ and $\lambda_{out} : C_{out} \to State$ in (1) which are understood as graph homomorphisms. Similarly, the source and target maps $in : C_{in} \to C$ and $out : C_{out} \to C$ are graph morphisms whose purpose is to map every input $s \in C_{in}$ and output state $s' \in C_{out}$ to the underlying internal states $in(s)$ and $out(s')$ of the code C.

The bicategory of cospans. The discussion leads us to the definition of the bicategory $\mathbf{Cospan}(\mathbb{S})$ of cospans associated with a category $\mathbb{S}$ with pushouts. The bicategory $\mathbf{Cospan}(\mathbb{S})$ has the same objects A, B, C as the original category $\mathbb{S}$, and its morphisms are the triples

$$(S, in, out) \quad : \quad A \longrightarrow\!\!\!\!\!\!\!\!+\;\;\; B \tag{13}$$

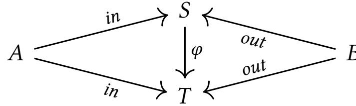
consisting of an object S of the category $\mathbb{S}$ together with a pair

$$A \xrightarrow{\ in\ } S \xleftarrow{\ out\ } B$$

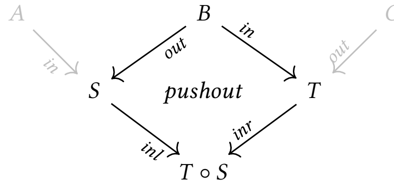
of morphisms of the category $\mathbb{S}$. Such a triple (13) is called a **cospan** between A and B, with **support** the object S of the category $\mathbb{S}$. In many situations, the two morphisms $in : A \to S$ and $out : B \to S$ can be deduced from the context, and we thus simply write $S : A \nrightarrow B$ for the cospan in that case. A 2-dimensional cell in the bicategory **Cospan**($\mathbb{S}$) between two such cospans

$$\varphi \quad : \quad (S, in, out) \Longrightarrow (T, in, out) \quad : \quad A \nrightarrow B$$

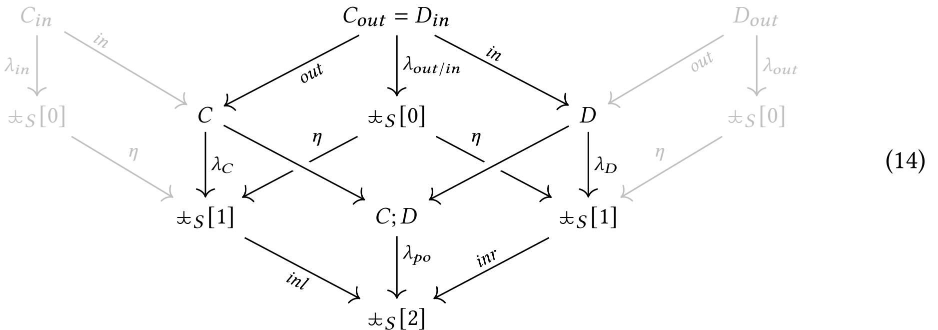
is defined as a morphism $\varphi : S \to T$ of the original category $\mathbb{S}$, making the diagram commute:



Two cospans $S : A \nrightarrow B$ and $T : B \nrightarrow C$ can be put side by side and composed into the cospan $T \circ S : A \nrightarrow C$ by "gluing" their common border B, using a *pushout diagram* performed in the category $\mathbb{S}$:



Sequential composition as pushout + relabeling. One good reason for describing codes C and D as cospans (10) in the category **Trans** of transition systems, is that the very same "gluing" recipe based on pushouts can be applied to compute their *sequential composition* $C; D$, at least in the specific case where the set C_{out} of output states of C coincides with the set D_{in} of input states of D. The pushout construction performed in **Trans** gives rise to the commutative diagram in **Gph** below:



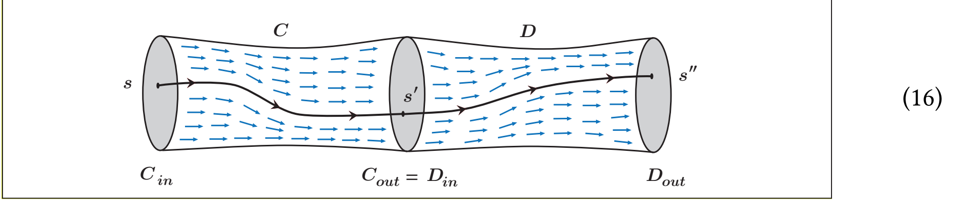
(14)

It should be noted that the pushout diagram in **Trans** is obtained by computing *independently* in **Gph** the pushout diagram $[a]$ defining the support graph $C; D$ as well as the pushout diagram $[b]$

defining the labeling graph $\pm_S[2]$, as shown below:

$$\begin{array}{ccccc} & & C_{in} = D_{out} & & \\ & \swarrow^{out} & & \searrow^{in} & \\ C & & pushout[a] & & D \\ & \searrow_{inl} & & \swarrow_{inr} & \\ & & C;D & & \end{array} \qquad \begin{array}{ccccc} & & \pm_S[0] & & \\ & \swarrow^{\eta} & & \searrow^{\eta} & \\ \pm_S[1] & & pushout[b] & & \pm_S[1] \\ & \searrow_{inl} & & \swarrow_{inr} & \\ & & \pm_S[2] & & \end{array} \tag{15}$$

Pictorially, the purpose of the pushout $[a]$ is to "glue" together the two cylinders C and D represented in (6) along their common border $C_{out} = D_{in}$, so as to obtain the cylinder $C;D$ describing the result of composing the two codes C and D sequentially:



(16)

One main contribution and novelty of the paper is the idea that the pushout diagram $[a]$ performed on the graphs C and D should come together with a pushout diagram $[b]$ performed this time at the level of their labeling graphs. The labeling graph $\pm_S[2]$ produced by the pushout $[b]$ is easy to compute: its vertices are the machine states $s \in State$ and there is a pair of edges noted $inst_l, inst_r : s \longrightarrow s'$ for every edge $inst : s \longrightarrow s'$ of the graph $\pm_S[1]$. The intuition is that the code C performs the instructions of the form $inst_l$ in $\pm_S[2]$ while the code D performs the instructions of the form $inst_r$. The labeling graph $\pm_S[2]$ comes together with a relabeling map

$$\mu \;:\; \pm_S[2] \longrightarrow \pm_S[1] \tag{17}$$

which maps every edge of the form $inst_l, inst_r : s \longrightarrow s'$ of $\pm_S[2]$ to the underlying edge $inst : s \longrightarrow s'$ of $\pm_S[1]$. The transition system $(C;D, \lambda_{C;D})$ defining the sequential composition is simply obtained by relabeling along μ the transition system $(C;D, \lambda_{po})$ computed by the pushout of (C, λ_C) and (D, λ_D) in **Trans**, in order to obtain the transition system with labeling map:

$$C;D \xrightarrow{\lambda_{C;D}} \pm_S[1] \quad = \quad C;D \xrightarrow{\lambda_{po}} \pm_S[2] \xrightarrow{\mu} \pm_S[1]$$

The construction establishes that:

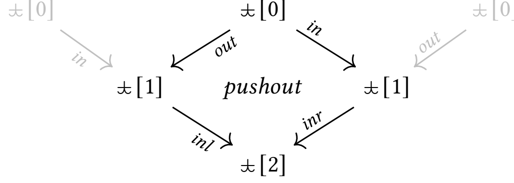
$$\boxed{\text{Sequential composition} \;=\; \text{gluing by pushout} \;+\; \text{relabeling}} \tag{18}$$

Template games and internal opcategories. The fact that the relabeling along $\mu : \pm_S[2] \to \pm_S[1]$ plays such a critical role in the definition of sequential composition $C, D \mapsto C;D$ is reminiscent of a similar observation in the work by Melliès on template games for differential linear logic (DiLL). In that case, the construction of the bicategory **Games**$(\pm)$ of template games, strategies and simulations relies on the assumption that the synchronization template $\pm$ defines an *internal category* in the underlying category $\mathbb{S}$. An important fact observed for the first time by Bénabou [1] is that an internal category $\pm$ in a category $\mathbb{S}$ with pullbacks is the same thing as a monad in the associated bicategory **Span**$(\mathbb{S})$ of spans, see [7] for a discussion. Since we are working in a category $\mathbb{S}$ with pushouts, it makes sense to dualize the situation, and to call **internal opcategory**

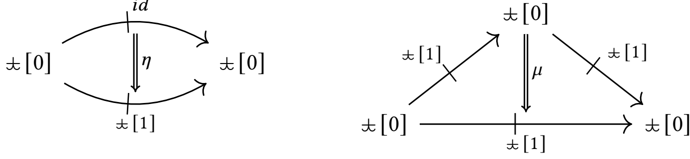
$\pm$ a monad in the bicategory $\mathbf{Cospan}(\mathbb{S})$. In other words, an internal opcategory $\pm$ is defined as a pair of objects $\pm[0]$ and $\pm[1]$ equipped with a cospan and a pair of morphisms:

$$\pm[0] \xrightarrow{in} \pm[1] \xleftarrow{out} \pm[0] \qquad \pm[0] \xrightarrow{\eta} \pm[1] \qquad \pm[2] \xrightarrow{\mu} \pm[1] \tag{19}$$

where the object $\pm[2]$ is defined as the result of the pushout diagram in $\mathbb{S}$:



In addition, a number of diagrams are required to commute, in order to ensure that the two morphisms η and μ define the following 2-cells in the bicategory $\mathbf{Cospan}(\mathbb{S})$



and that these 2-cells satisfy the unitality and associativity axioms required of a monad in the bicategory $\mathbf{Cospan}(\mathbb{S})$. Note in particular that the requirement that η defines the left hand-side 2-cell is very strong, since it implies that the three morphisms $\eta, in, out : \pm[0] \to \pm[1]$ are equal.

A bicategory of cobordisms. We have seen earlier that the labeling graphs $\pm_S[0] = State$ and $\pm_S[1]$ of machine states are equipped with morphisms η and μ explicated in (12) and (17). Somewhat surprisingly, this additional structure happens to be tightly connected to the work by Melliès on template games, as established by the following theorem:

Theorem 1.1. *The cospan*

$$\pm_S[0] \xrightarrow{\pm_S[1]} \pm_S[0] \qquad = \qquad \pm_S[0] \xrightarrow{\eta} \pm_S[1] \xleftarrow{\eta} \pm_S[0]$$

together with the graph morphisms η and μ defines an internal opcategory $\pm$ in the category **Gph**.

This theorem means that the machine model $\pm_S$ is regulated by almost the same algebraic principles as the synchronization templates exhibited by Melliès in order to generate his game semantics of differential linear logic (DiLL). The only difference (it is a key difference though) is that the original internal category $\pm$ in DiLL is "dualized" and replaced in the present situation by an internal opcategory $\pm = \pm_S$ of machine states. Guided by this auspicious connection, we associate a bicategory $\mathbf{Cob}(\pm)$ of **games**, **cobordisms** and **simulations** with every internal opcategory $\pm$ living in a category $\mathbb{S}$ with pushouts. The terminology of *cobordism* pays tribute to our main source of inspiration for the idea of cospan, which is the construction of the *category of cobordisms* in algebraic topology and in topological quantum field theory, see [20]. A **game** (A, λ_A) is a pair

$$\lambda_A \quad : \quad A \longrightarrow \pm[0] \tag{20}$$

A **cobordism** $\sigma : A \;⇸\; B$ between two such games is a quadruple $(S, in, out, \lambda_\sigma)$ consisting of an object S and three morphisms making the diagram below commute:

$$\begin{array}{ccccc} A & \xrightarrow{in} & S & \xleftarrow{out} & B \\ {\scriptstyle\lambda_A}\downarrow & & \downarrow{\scriptstyle\lambda_\sigma} & & \downarrow{\scriptstyle\lambda_B} \\ \pm[0] & \xrightarrow{\eta} & \pm[1] & \xleftarrow{\eta} & \pm[0] \end{array} \tag{21}$$

Here, the object S is called the **support** of the cobordism σ. A **simulation** between two cobordisms

$$\varphi \quad : \quad \sigma \Longrightarrow \tau \quad : \quad A \;⇸\; B$$

is defined as a morphism $\varphi : S \to T$ of the original category $\mathbb{S}$, making the three diagrams commute:

$$\begin{array}{ccc} & A & \\ {\scriptstyle in}\swarrow & & \searrow{\scriptstyle in} \\ S & \xrightarrow[\varphi]{} & T \end{array} \qquad \begin{array}{ccc} & B & \\ {\scriptstyle out}\swarrow & & \searrow{\scriptstyle out} \\ S & \xrightarrow[\varphi]{} & T \end{array} \qquad \begin{array}{ccc} S & \xrightarrow{\varphi} & T \\ {\scriptstyle\lambda_\sigma}\searrow & & \swarrow{\scriptstyle\lambda_\tau} \\ & \pm[1] & \end{array}$$

The composition of cobordisms $\sigma : A \;⇸\; B$ and $\tau : B \;⇸\; C$ is defined by pushout and relabeling:

$$\begin{array}{ccccccccc} A & & & & B & & & & C \\ {\scriptstyle\lambda_A}\downarrow & \searrow{\scriptstyle in} & & {\scriptstyle out}\swarrow & \downarrow{\scriptstyle\lambda_B} & \searrow{\scriptstyle in} & & {\scriptstyle out}\swarrow & \downarrow{\scriptstyle\lambda_C} \\ \pm[0] & & S & & \pm[0] & & T & & \pm[0] \\ & \searrow{\scriptstyle\eta} & \downarrow{\scriptstyle\lambda_\sigma} & \searrow \;\; {\scriptstyle\eta}\swarrow & & \searrow{\scriptstyle\eta} \;\; \swarrow & \downarrow{\scriptstyle\lambda_\tau} & \swarrow{\scriptstyle\eta} & \\ & & \pm[1] & & S +_B T & & \pm[1] & & \\ & & & \searrow{\scriptstyle inl} & \downarrow{\scriptstyle\lambda_{po}} & \swarrow{\scriptstyle inr} & & & \\ & & & & \pm[2] & & & & \\ & & & & \downarrow{\scriptstyle\mu} & & & & \\ & & & & \pm[1] & & & & \end{array} \tag{22}$$

while the identity cobordism $id_A : A \;⇸\; A$ is defined as follows:

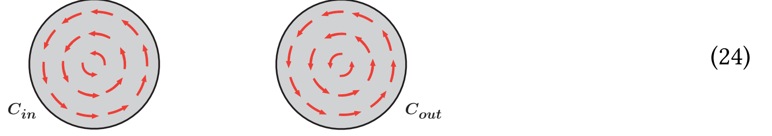
$$\begin{array}{ccccc} A & \xrightarrow{id_A} & A & \xleftarrow{id_A} & A \\ & & \downarrow{\scriptstyle\lambda_A} & & \\ {\scriptstyle\lambda_A}\downarrow & & \pm[0] & & \downarrow{\scriptstyle\lambda_A} \\ & & \downarrow{\scriptstyle\eta} & & \\ \pm[0] & \xrightarrow{in} & \pm[1] & \xleftarrow{out} & \pm[0] \end{array} \tag{23}$$

Given a category $\mathbb{S}$ with pushouts, we establish that

Theorem 1.2. *Every internal opcategory $\pm$ in the category $\mathbb{S}$ defines a bicategory* **Cob**$(\pm)$.

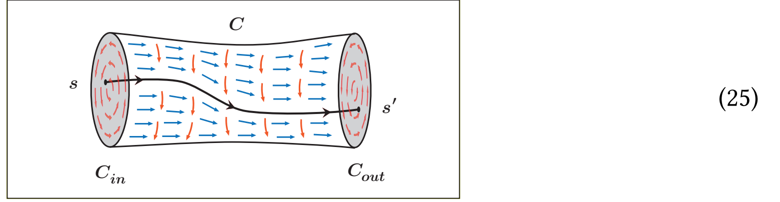
The construction of the bicategory **Cob**$(\pm)$ of cobordisms is surprisingly similar to the construction of the bicategory **Games**$(\pm)$ performed by Melliès [16]. In particular, the composition of strategies in **Games**$(\pm)$ and of cobordisms in **Cob**$(\pm)$ relies in both cases on a relabeling along the "multiplication" morphism $\mu : \pm[2] \to \pm[1]$, while the definition of identities relies on the "unit" morphism $\eta : \pm[0] \to \pm[1]$. There is a simple conceptual explanation for the similitude however, which is that we construct in both cases the bicategory **Games**$(\pm)$ and **Cob**$(\pm)$ as a bicategory *sliced above* a formal monad $\pm$ living either in the bicategory **Span**$(\pm)$ (in the case of games) or in the bicategory **Cospan**$(\pm)$ (in the case of cobordisms), see [7] for details.

Parallel product as pullback + relabeling. We have just described how two cobordisms $\sigma : A \nrightarrow B$ and $\tau : B \nrightarrow C$ can be "glued" together and composed sequentially using a pushout diagram on their common border (B, λ_B). Similarly, given two cobordisms $\sigma : A \nrightarrow B$ and $\tau : C \nrightarrow D$ interpreting concurrent imperative programs, we would like to give a nice and conceptual description of their parallel product $\sigma \| \tau : A \| C \nrightarrow B \| D$. To that purpose, we shift from a *sequential* to a *concurrent* setting by extending our original labeling graphs $\pm_S[0]$ and $\pm_S[1]$ with transitions performed not just by the Code, but also by the Frame (or the Environment). When compared to the sequential situation of (2), (3) and (4), this extension of the original labeling graphs $\pm_S[0]$ and $\pm_S[1]$ conveys the intuition that a concurrent imperative code C connects an input graph C_{in} to an output graph C_{out} whose only transitions (depicted below in red) are performed by the Frame:



(24)

and whose cylinder or cobordism $C : C_{in} \nrightarrow C_{out}$ admits transitions performed either by the Code (in blue) or by the Frame (in red), as shown below:



(25)

The flexibility of our approach based on cobordism is illustrated by the great ease in which the shift of paradigm from sequential to concurrent is performed by moving from the original *one-player* machine model to a *two-player* machine model where transitions may be performed by **C** (for the Code) or **F** (for the Frame). In particular, as it stands, the interpretation of the code C as a cobordism is described by the very same diagram (11) as in the sequential case, except that transitions in $\pm[0]$ are now performed by Frame and those in $\pm[1]$ by Frame and by Code. Finally, in order to reflect the asynchronous structure of interactions, we take this opportunity to upgrade our model, and shift from the ambient category $\mathbb{S} = \mathbf{Gph}$ to the category $\mathbb{S} = \mathbf{AsynGph}$ of asynchronous graphs, described in §3.

At this stage, another striking connection emerges between our bicategory $\mathbf{Cob}(\pm)$ of template games and cobordisms and the original work by Melliès on template games for differential linear logic (DiLL). Indeed, it appears that the parallel product $A \| B$ and $\sigma \| \tau$ of template games and cobordisms in $\mathbf{Cob}(\pm)$ can be defined using the same principles as the tensor product $A \otimes B$ and $\sigma \otimes \tau$ of template games and strategies in the bicategory $\mathbf{Games}(\pm)$ for DiLL. The general recipe is to equip the internal opcategory $\pm$ with a pair of internal functors

$$\pm \times \pm \xleftarrow{\text{pick}} \pm^{\|} \xrightarrow{\text{pince}} \pm \tag{26}$$

from which one derives (see §4) a lax functor $\mathsf{pull}[\mathsf{pick}]$ and a pseudo functor $\mathsf{push}[\mathsf{pince}]$ between bicategories, see [10] for definitions:

$$\mathbf{Cob}(\pm \times \pm) \xrightarrow{\mathsf{pull}[\mathsf{pick}]} \mathbf{Cob}(\pm^{\|}) \xrightarrow{\mathsf{push}[\mathsf{pince}]} \mathbf{Cob}(\pm) \tag{27}$$

obtained by *pulling along pick* and by *pushing along pince*, respectively. The parallel product

$$\| \quad : \quad \mathbf{Cob}(\pitchfork) \times \mathbf{Cob}(\pitchfork) \xrightarrow{\ \mathrm{m}_{\pitchfork,\pitchfork}\ } \mathbf{Cob}(\pitchfork \times \pitchfork) \longrightarrow \mathbf{Cob}(\pitchfork)$$

is then defined as the lax double functor obtained by precomposing (27) with the canonical pseudo functor

$$\mathrm{m}_{\pitchfork_1,\pitchfork_2} \quad : \quad \mathbf{Cob}(\pitchfork_1) \times \mathbf{Cob}(\pitchfork_2) \longrightarrow \mathbf{Cob}(\pitchfork_1 \times \pitchfork_2)$$

The definition is elegantly conceptual, but not entirely transparent, and thus probably worth explicating. The parallel product of two template games A and B is defined by computing the pullback of $\lambda_A \times \lambda_B$ along pick[0], and then relabeling the resulting labeling map λ_{pb} along pince[0]:

$$\begin{array}{ccccc} A \times B & \longleftarrow & A \| B & \xrightarrow{\ \lambda_{A\|B}\ } & \\ {\scriptstyle \lambda_A \times \lambda_B}\downarrow & \textit{pullback} & \downarrow{\scriptstyle \lambda_{pb}} & & \\ \pitchfork[0] \times \pitchfork[0] & \xleftarrow[\ \mathrm{pick}[0]\ ]{} & \pitchfork^{\|}[0] & \xrightarrow[\ \mathrm{pince}[0]\ ]{} & \pitchfork[0] \end{array} \tag{28}$$

In exactly the same way, the parallel product of $\sigma \| \tau : A \| C \nrightarrow B \| D$ of two cobordisms $\sigma : A \nrightarrow B$ and $\tau : C \nrightarrow D$ with respective supports S and T has its support $S \| T$ computed by the following pullback along pick[1], with resulting labeling map λ_{pb} postcomposed with pince[1]:

$$\begin{array}{ccccc} S \times T & \longleftarrow & S \| T & \xrightarrow{\ \lambda_{\sigma\|\tau}\ } & \\ {\scriptstyle \lambda_\sigma \times \lambda_\tau}\downarrow & \textit{pullback} & \downarrow{\scriptstyle \lambda_{pb}} & & \\ \pitchfork[1] \times \pitchfork[1] & \xleftarrow[\ \mathrm{pick}[1]\ ]{} & \pitchfork^{\|}[1] & \xrightarrow[\ \mathrm{pince}[1]\ ]{} & \pitchfork[1] \end{array} \tag{29}$$

As explained in §4, the opcategory $\pitchfork_S^{\|}$ describes a *three-player* machine model where every machine transition may be performed by one of the three players $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{F}$ (for Frame) involved in the parallel product $C_1 \| C_2$. The functor pick describes how the machine model $\pitchfork_S^{\|}$ can be mapped to a pair $\pitchfork_S \times \pitchfork_S$ of two-player machine models, where $\mathbf{C}_1$ identifies the transitions of $\mathbf{C}_2$ as part of its frame $\mathbf{F}$ in the left-hand component $\pitchfork_S$, and $\mathbf{C}_2$ identifies the transitions of $\mathbf{C}_1$ as part of its frame $\mathbf{F}$ in the right-hand component $\pitchfork_S$. The functor pince then identifies the transitions of the players $\mathbf{C}_1$ and of $\mathbf{C}_2$ in the three-player machine model $\pitchfork_S^{\|}$ as the transitions of the code C in the original two-player machine model $\pitchfork_S$.

To summarize, we see in (28) and (29) that the definition of the parallel product is performed by a pullback along the functor pick, whose purpose is to synchronize the transitions performed by $\mathbf{C}_1$, $\mathbf{C}_2$ and $\mathbf{F}$, followed by a relabeling along the functor pince. The construction thus establishes that:

$$\boxed{\text{Parallel product} \quad = \quad \text{synchronizing by pullback} \ + \ \text{relabeling}} \tag{30}$$

The fact that the sequential composition is computed by a pushout and thus a colimit (18) while the parallel product is computed by a pullback and thus a limit (30) has the immediate and remarkable consequence that there exists a natural and coherent family of morphisms

$$Hoare_{C_1,C_2,D_1,D_2} \quad : \quad (C_1 \| C_2);(D_1 \| D_2) \Rightarrow (C_1;D_1) \| (C_2;D_2)$$

which turns $\mathbf{Cob}(\pitchfork)$ into a lax monoidal bicategory. This lax monoidal structure of $\mathbf{Cob}(\pitchfork)$ provides a nice algebraic explanation for the Hoare inequality principle [12], since we derive it from the fact that a colimit (sequential composition) always commutes with a limit (parallel product) up to a (in that case non reversible) coercion morphism.

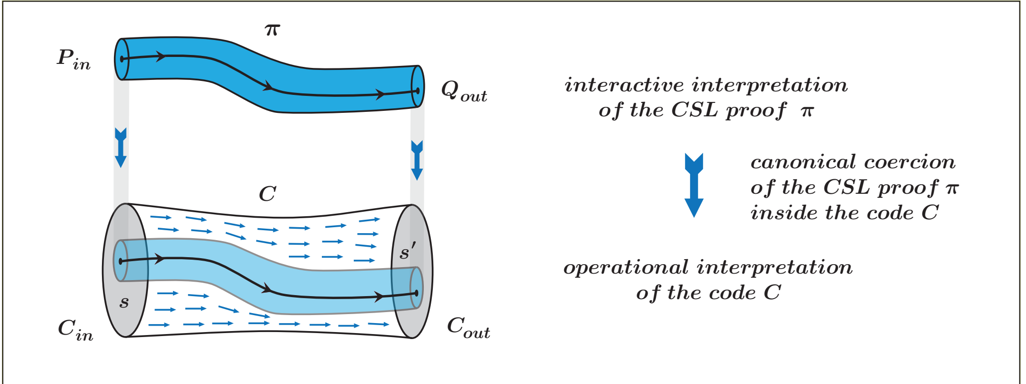
A uniform interpretation of the CSL codes and proofs. One of the main purposes of the present paper is to reunderstand in a more foundational and axiomatic way the *asynchronous soundness theorem* of concurrent separation logic (CSL) recently established by Melliès and Stefanesco [19]. Their proof of soundness is based on the construction of a game-theoretic and asynchronous interpretation of the codes and of the proofs of CSL in its original form, see Brookes [4]. One main advantage of our approach based on template games is to provide a general, uniform and flexible framework to construct models of CSL, both at the level of codes and of proofs. We will explain in §7 how to construct a machine model $\pm_{\mathrm{Sep}}$ based on the notion of *separated states* introduced by Melliès and Stefanesco [18, 19] in order to interpret the proofs of CSL, and not just the codes as before. We show moreover that there exists a chain of functors between the three machine models or internal opcategories introduced in the paper:

$$\pm_{\mathrm{Sep}} \xrightarrow{u_S} \pm_S \xrightarrow{u_{\mathcal{L}}} \pm_L.$$

As it turns out, given a CSL proof π of a Hoare triple $\{P\}C\{Q\}$, this chain of functors induces a chain of translations

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

between the interpretations of the CSL proof π and of the specified code C in the three template game models associated with $\pm_{\mathrm{Sep}}$, $\pm_S$ and $\pm_L$. The translation from the cobordism $[\![\pi]\!]_{Sep}$ interpreting the CSL proof π above the machine model $\pm_{\mathrm{Sep}}$ of separated states to the cobordism $[\![C]\!]_S$ providing the operational semantics of the code C can be depicted as follows:



In the picture, the blue cylinder above describes the asynchronous graph $[\![\pi]\!]_{Sep}$ of separated states where the code C starting from a state $s \in C_{in}$ of the Code satisfying the predicate P will remain and produce an output $s' \in C_{out}$ satisfying the predicate Q, as long as the Frame (or Environment) does not alter the part of the separated state owned by the Code, and respects the invariants required by the CSL judgment. The interpretation of CSL proofs in the machine model $\pm_{\mathrm{Sep}}$ of separated states requires to equip the internal opcategory $\pm_{\mathrm{Sep}}$ with one asynchronous graph $\pm_{\mathrm{Sep}}[0, P]$ for each predicate P of the logic. For that reason, $\pm_{\mathrm{Sep}}$ defines what we call a *colored* J-opcategory where J denotes here the set of predicates of the logic. Technically speaking, $\pm_{\mathrm{Sep}}$ defines a *polyad* in the sense of Bénabou [1] instead of just a monad in the bicategory **Cospan**$(\mathbb{S})$ for $\mathbb{S} =$ **AsynGph**. We explain in §2 how to adapt the construction of the bicategory **Cob**$(\pm)$ discussed in the introduction, to the case of a polyad $\pm$ instead of a monad. We explain at the end of the paper, see §10, how this conceptual toolbox sheds light on the constructions underlying the recent proof of asynchronous soundness established by Melliès and Stefanesco [19].

Synopsis. We start by introducing in §2 the notion of internal *colored* opcategory $\pm$, and explain how to associate a double category $\mathbf{Cob}(\pm)$ of games, cobordisms and simulations with every such synchronization template. Then, we exhibit in §3 two specific internal opcategories $\pm_S$ and $\pm_L$ living in the category **AsynGph** of asynchronous graphs, and providing machine models based, respectively, on a *stateful* and *stateless* account of the machine instructions. We then explain in §4 and §5 how to perform the parallel product and the sequential composition in our template game model. This includes the description in §6 of a convenient monadic treatment of errors in our asynchronous and multi-player transition systems. Once the synchronization template $\pm_{Sep}$ of separated state has been recalled in §7, we develop in §8 an axiomatic and fibrational account of lock acquisition and release. This categorical framework enables us to interpret in §9 codes and CSL proofs in a nice and uniform way. We explain in §10 how to derive from our framework the asynchronous soundness theorem recently established in [19]. We discuss the related works in §11 and conclude in §12.

2 THE DOUBLE CATEGORY $\mathrm{Cob}(\pm)$ OF GAMES AND COBORDISMS

We have explained in the introduction how to associate a bicategory of template games and cobordisms with an internal opcategory $\pm$ in a category $\mathbb{S}$ with pushouts. We show here that we can in fact associate with $\pm$ a double category $\mathbf{Cob}(\pm)$ instead of just a bicategory, and extend the construction to the case of *polyads* in $\mathbf{Cospan}(\mathbb{S})$ instead of just monads. We start by recalling the definition of double category, and then of polyad, introduced by Bénabou [1].

2.1 Double categories

We recall the notion of **(pseudo) double category**, which was introduced by Ehresmann [8]. Formally, a double category $\mathcal{D}$ is a weakly internal category in **Cat**, the 2-category of small categories. More concretely, it is the data of: a collection of objects, a collection of vertical morphisms between objects denoted with a simple arrow $A \to B$, a collection of horizontal morphisms between pairs of objects denoted with a crossed arrow $A \nrightarrow B$, and of 2-cells filling squares of the form:

$$\begin{array}{ccc} A_1 & \overset{F}{\nrightarrow} & B_1 \\ {\scriptstyle f}\downarrow & \Downarrow{\scriptstyle \alpha} & \downarrow{\scriptstyle g} \\ A_2 & \underset{G}{\nrightarrow} & B_2 \end{array}$$

A 2-cell is called **special** if it is globular, in that f and g are identities. Composition of vertical arrows is associative, whereas composition of horizontal arrows are only associative up to special invertible 2-cells.

The intuition is that, as for most examples, vertical morphisms are the usual notion of morphisms associated with the objects, for example ring morphisms for the double category of rings, and the vertical morphisms are relational structures, such are bimodules in the case of rings. The other canonical example, which is relevant for this paper is the double category of cospans: Given a category $\mathbb{S}$ with pushouts, we consider the double category whose objects and vertical morphisms are respectively the objects and the morphisms of $\mathbb{S}$, and whose horizontal morphisms between A and B are the cospans in $\mathbb{S}$ of the form depicted below left. A two cell is given by the morphism $S \to S'$ in the right diagram.

$$A \longrightarrow S \longleftarrow B \qquad\qquad \begin{array}{ccccc} A & \longrightarrow & S & \longleftarrow & B. \\ \downarrow & & \downarrow & & \downarrow \\ A' & \longrightarrow & S' & \longleftarrow & B'. \end{array}$$

Composition of horizontal morphisms is given by pushout. Because pushouts are only unique up to iso, given a choice of pushouts, it is easy to check that the universality of pushouts implies that horizontal composition is indeed associative up to an invertible special two cell.

2.2 Polyads

Definition 2.1. The forgetful functor $|-|:\mathbf{Cat}\to\mathbf{Set}$ which transports every small category to its set of objects has a right adjoint $\mathbf{chaos}:\mathbf{Set}\to\mathbf{Cat}$ which transports every set J to its **chaotic category** defined as the category $\mathbf{chaos}(J)$ whose objects are the elements i,j,k of the set J, with a unique morphism between each pair of objects. A **polyad** in a bicategory $\mathcal{D}$ is a lax double functor of double category

$$\mathbf{chaos}(J) \longrightarrow \mathcal{D}$$

from the chaotic category over some set J, seen as a double category, to $\mathcal{D}$. A polyad is called a **monad** when J is a singleton.

It is worth mentioning that a polyad $\pm$ in a bicategory $\mathcal{W}$ is the same thing as an enriched category $\pm$ over the bicategory $\mathcal{W}$ in the terminology of the Australian school [24]. Given a category $\mathbb{S}$ with pushouts, we call **internal J-opcategory** a polyad $\pm$ in the double category $\mathbf{Cospan}(\mathbb{S})$. The definition can be expounded as follows. An internal J-opcategory $\pm$ consists of an object $\pm[0,i]$ of $\mathbb{S}$ for each element $i\in J$, and of a cospan

$$\pm[1,ij] \quad : \quad \pm[0,i] \;\not\to\; \pm[0,j] \quad = \quad \pm[0,i] \xrightarrow{\mathrm{in}_{ij}} \pm[1,ij] \xleftarrow{\mathrm{out}_{ij}} \pm[0,j]$$

for each pair $i,j\in J$, together with two coherent families η and μ of 2-cells between cospans:

$$\eta_i : id \Rightarrow \pm[1,ii] : \pm[0,i] \not\to \pm[0,i] \qquad\qquad \mu_{ijk} : \pm[1,ij] \,;\, \pm[1,jk] \Rightarrow \pm[1,ik] : \pm[0,i] \not\to \pm[0,k]$$

More explicitly, μ_{ijk} is given by the map g_{ijk} in the following commutative diagram:

$$\begin{array}{ccccccc}
\pm[0,i] & \xrightarrow{\mathrm{in}_{ij}} & \pm[1,ij] & \xleftarrow{\mathrm{out}_{ij}} \pm[0,j] \xrightarrow{\mathrm{in}_{jk}} & \pm[1,jk] & \xleftarrow{\mathrm{out}_{jk}} & \pm[0,k] \\
 & & \searrow & \pm[2,ijk] & \swarrow & & \\
 & & & \downarrow g_{ijk} & & & \\
 & \xrightarrow{\mathrm{in}_{ik}} & & \pm[1,ik] & & \xleftarrow{\mathrm{out}_{ik}} &
\end{array}$$

Finally, there are some conditions about the associativity of the composition, and about the identity; they can be found in [1, def. 5.5.1]. An **internal opcategory** $\pm$ is defined as an internal J-opcategory where the set J is a singleton $J=\{j\}$. We write in that case $\pm[0]$ and $\pm[1]$ for the objects $\pm[0,j]$ and $\pm[1,jj]$ of the ambient category $\mathbb{S}$, respectively.

2.3 The double category $\mathrm{Cob}(\pm)$ of games and cobordisms

Suppose given an internal J-opcategory $\pm$ in a category $\mathbb{S}$ with pushouts. A **j-colored game** (A,λ_A) is defined as an object A of $\mathbb{S}$ equipped with a morphism $\lambda_A : A\longrightarrow \pm[0,j]$. An **$ij$-colored cobordism** from a i-colored game (A,λ_A) to a j-colored game (B,λ_B) is defined as a cospan in $\mathbb{S}$

together with colors $i, j \in J$ and a map λ_σ such that the following diagram commutes:

$$\begin{array}{ccccc} A & \xrightarrow{in} & S & \xleftarrow{out} & B \\ \downarrow{\scriptstyle \lambda_A} & & \downarrow{\scriptstyle \lambda_\sigma} & & \downarrow{\scriptstyle \lambda_B} \\ \pm[0,i] & \xrightarrow{\mathrm{in}_{ij}} & \pm[1,ij] & \xleftarrow{\mathrm{out}_{ij}} & \pm[0,j] \end{array}$$

As in the case (22) of an internal opcategory, two such ij-colored cobordism $\sigma : A \nrightarrow B$ and jk-colored cobordism $\tau : B \nrightarrow C$ can be composed into an ik-colored cobordism $\sigma;\tau : A \nrightarrow C$ using a pushout and a relabeling along the 2-cell μ_{ijk} provided by the internal J-opcategory, as shown below:

$$\begin{array}{ccccc} A & \overset{\mathrm{C}}{\nrightarrow} & B & \overset{\mathrm{C}'}{\nrightarrow} & C \\ \downarrow & \Downarrow & \downarrow & \Downarrow & \downarrow \\ \pm[0,i] & \underset{\pm[1,ij]}{\nrightarrow} & \pm[0,j] & \underset{\pm[1,jk]}{\nrightarrow} & \pm[0,i] \\ & & \Downarrow{\scriptstyle \mu_{ijk}} & & \\ & & \pm[0,i] \underset{\pm[1,ik]}{\nrightarrow} \pm[0,i] & & \end{array}$$

In order to give cobordisms over an internal J-opcategory $\pm$ a structure of double category, we need identities, which are given by:

$$\begin{array}{ccc} A & \overset{\mathbb{1}_A}{\nrightarrow} & A \\ \downarrow{\scriptstyle \lambda_A} & \Downarrow{\scriptstyle \lambda_A} & \downarrow{\scriptstyle \lambda_A} \\ \pm[0,i] & \underset{\mathbb{1}_{\pm[0,i]}}{\nrightarrow} & \pm[0,i] \\ & \Downarrow{\scriptstyle \eta_i} & \\ \pm[0,i] & \underset{\pm[1,ii]}{\nrightarrow} & \pm[0,i] \end{array}$$

The vertical morphisms are the maps $f : A \longrightarrow A'$ such that $\lambda_A = \lambda_B \circ f$, and the two-cells are triples of maps $f_{\mathrm{in}} : A \longrightarrow A'$, $f_{\mathrm{out}} : B \longrightarrow B'$, $f : S \longrightarrow S'$ such that the following diagram commutes

$$\begin{array}{ccccc} A & \xrightarrow{\mathrm{in}_{\mathrm{C}}} & S & \xleftarrow{\mathrm{out}_{\mathrm{C}}} & B \\ \downarrow{\scriptstyle f_{\mathrm{in}}} & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle f_{\mathrm{out}}} \\ A' & \xrightarrow{\mathrm{in}'_{\mathrm{C}}} & S' & \xleftarrow{\mathrm{out}'_{\mathrm{C}}} & B' \end{array}$$

such that moreover, f_{in} and f_{out} are vertical morphisms, and similarly $\lambda_\sigma = \lambda_{\sigma'} \circ f$. One obtains:

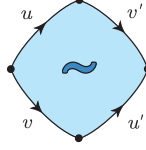
Theorem 2.2. *Every internal J-opcategory $\pm$ induces a double category* $\mathbf{Cob}(\pm)$ *whose objects are the j-colored games and whose horizontal maps are the cobordisms with composition defined as above.*

3 THE STATEFUL AND STATELESS SYNCHRONIZATION TEMPLATES

We have seen in §2 how to associate a double category $\mathbf{Cob}(\pm)$ with every internal J-opcategory $\pm$ living in an ambient category $\mathbb{S}$ with pushouts. As a matter of fact, all the interpretations performed in the present paper will live in the same ambient category $\mathbb{S} = \mathbf{AsynGph}$ of **asynchronous graphs**, which are graphs equipped with two-dimensional tiles. After describing this category in §3.1, we recall in §3.2 the stateful and stateless machine models used in the paper, and simply formulated as asynchronous graphs $\pm^\bullet_S$ and $\pm^\bullet_L$, along a recipe initiated in [19]. We explain in §3.3 how to derive from $\pm^\bullet_S$ and $\pm^\bullet_L$ the internal opcategories $\pm_S$ and $\pm_L$ living in the ambient category $\mathbb{S} = \mathbf{AsynGph}$ of asynchronous graphs, and associated with the stateful and stateless machine models, respectively.

3.1 The category of asynchronous graphs

A **graph** $G = (V, E, \partial^-, \partial^+)$ consists of a set V of vertices or nodes, a set of E of edges or transitions, and a source and target functions $\partial^-, \partial^+ : E \longrightarrow V$. We call a **square** a pair (f, g) of paths $P \twoheadrightarrow Q$ of length 2, with the same source and target vertices. An **asynchronous graph** $(G, \diamond)$ is a graph G equipped with a tuple $\diamond = (T, \partial_\diamond, \sigma)$ of a set T of **permutation tiles**, a function $\partial_\diamond$ from the set T to the set of squares of the graph G, and an endofunction σ on T such that if a tile $t \in T$ is mapped by $\partial_\diamond$ to a square (f, g), then $\partial_\diamond(\sigma(t)) = (g, f)$. We call the function σ the symmetry on $\diamond$. A tile which is mapped to a square $(u \cdot v', v \cdot u')$ is depicted as a 2-dimensional tile between the paths $f = u \cdot v'$ and $g = v \cdot u'$ as follows:



(31)

The intuition conveyed by such a permutation tile $t : u \cdot v' \diamond v \cdot u'$ is that the two transitions u and v are independent. For that reason, the two paths $u \cdot v'$ and $v \cdot u'$ may be seen as equivalent up to scheduling. Two paths $f, g : M \twoheadrightarrow N$ of an asynchronous graph are **equivalent modulo one permutation tile** $h_1 \diamond h_2$ when there exists a tile in T between h_1 and h_2 and when f and g factor as $f = d \cdot h_1 \cdot e$ and $g = d \cdot h_2 \cdot e$ for two paths $d : M \twoheadrightarrow P$ and $e : Q \twoheadrightarrow N$. We write $f \sim g$ when the path $f : M \twoheadrightarrow N$ is "equivalent" to the path $g : M \twoheadrightarrow N$ modulo a number of such permutation tiles. Note that the relation $\sim$ is symmetric, reflexive and transitive, and thus defines an equivalence relation, closed under composition.

Definition 3.1. An **asynchronous graph homomorphism**, or **asynchronous morphism**,

$$\mathcal{F} \quad : \quad (G, \diamond_G) \longrightarrow (H, \diamond_H) \tag{32}$$

is a graph homomorphism $\mathcal{F} : G \rightarrow H$ between the underlying graphs, together with a function $\mathcal{F}_\diamond$ from T_G to T_H such that $\mathcal{F} \circ \partial = \partial \circ \mathcal{F}_\diamond$, where we extend $\mathcal{F}$ in the usual way to paths and squares.

This defines the category **AsynGph** of asynchronous graphs and asynchronous morphisms. As it is needed for the cobordism construction, this category has all small limits and colimits, since it is a presheaf category, as is it shown below. The limits and colimits are computed pointwise for nodes, edges and tiles. We detail below the case of pullbacks, and consequently of Cartesian products. The pullback of a cospan

$$A_1 \xrightarrow{\ f\ } B \xleftarrow{\ g\ } A_2$$

of asynchronous graphs is defined as the asynchronous graph $A_1 \times_B A_2$ below. Its nodes are the pairs (x_1, x_2) consisting of a node x_1 in A_1 and of a node x_2 in A_2, such that $f(x_1) = g(x_2)$. Its edges $(u_1, u_2) : (x_1, x_2) \longrightarrow (y_1, y_2)$ are the pairs consisting of an edge $u_1 : x_1 \longrightarrow y_1$ in A_1 and of an edge $u_2 : x_2 \longrightarrow y_2$ in A_2, such that $f(u_1) = g(u_2)$. In the same way, a tile $(\alpha_1, \alpha_2) : (u_1, u_2) \cdot (v_1', v_2') \diamond (v_1, v_2) \cdot (u_1', u_2')$ is a pair consisting of a tile $\alpha_1 : u_1 \cdot v_1' \diamond v_1 \cdot u_1'$ of the asynchronous graph A_1 and of a tile $\alpha_2 : u_2 \cdot v_2' \diamond v_2 \cdot u_2'$ of the asynchronous graph A_2, such that the two tiles $f(\alpha_1)$ and $g(\alpha_2)$ are equal in the asynchronous graph B. The Cartesian product $A_1 \times A_2$ of two asynchronous graphs is obtained by considering the special case when B is the terminal asynchronous graph, with one node, one edge, and one tile. The definition of $A_1 \times A_2$ thus amounts to forgetting the equality conditions in the definition of the pullback $A_1 \times_B A_2$ above.

Remark 3.2. *The category* **AsynGph** *of asynchronous graphs can be seen as the category of presheaves over the category presented by the graph:*

$$[0] \underset{t}{\overset{s}{\rightrightarrows}} [1] \;\overset{dl,\,dr,\,ur,\,ul}{\rightrightarrows}\; [2] \circlearrowleft \sigma$$

and with the equations:

$$dl \circ s = ul \circ s \qquad dl \circ t = dr \circ s \qquad dr \circ t = ur \circ t \qquad ul \circ t = ur \circ s$$
$$\sigma \circ dl = ul \qquad \sigma \circ ul = dl \qquad \sigma \circ dr = ur \qquad \sigma \circ ur = dr$$

3.2 The stateful and stateless machine models

Following [19], we define the stateful and the stateless *machine models* as asynchronous graphs $\pm^{\bullet}_{S}$ and $\pm^{\bullet}_{L}$, describing the primitive semantics of the machine: the states, the transitions between them, and the tiles which explain how to permute the order of execution of two transitions performed in parallel. Note that each model depends on an implicit set $\mathbb{L}$ of free locks ; we will come back to this point and make it precise in §8.

3.2.1 The stateful model $\pm^{\bullet}_{S}$. A **memory state** is an element $\mu = (s, h)$ of the set $(\mathbf{Var} \rightharpoonup_{fin} \mathbf{Val}) \times (\mathbf{Loc} \rightharpoonup_{fin} \mathbf{Val})$ with $\mathbb{N} \subseteq \mathbf{Loc} \subseteq \mathbf{Val}$, where the finite partial map s stands for the stack and h for the heap. A **machine state** is a pair $\mathfrak{s} = (\mu, L)$ of a memory state and of a subset of $\mathbb{L}$, which represents the available locks. A **machine state footprint**

$$\rho \in \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(\mathbb{L}) \times \wp(\mathbf{Loc})$$

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

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

The **stateful model** $\pm^{\bullet}_{S}$ is defined as the following asynchronous graph: its nodes are the machine states and $\lightning$, its transitions

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

are given by the semantics of the instructions, which are of the form:

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

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

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

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

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

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

where their footprints are independent in the sense above.

3.2.2 The stateless model $\pitchfork_L^\bullet$. A **lock footprint**

$$\rho \in \wp(\mathbb{L}) \times \wp(\mathbf{Loc})$$

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

$$L \xrightarrow{P(r)} L \uplus \{r\} \qquad L \xrightarrow{\texttt{alloc}(\ell)} L \qquad L \xrightarrow{\tau} L \qquad L \uplus \{r\} \xrightarrow{V(r)} L \qquad L \xrightarrow{\texttt{dispose}(\ell)} L \qquad L \xrightarrow{m} \lightning$$

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

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

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

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

when the lock footprints of x and y are independent. It is worth noting that L' may be equal to $\lightning$ in such an asynchronous tile. Note that the asynchronous graph $\pitchfork_L^\bullet$ is more liberal than $\pitchfork_S^\bullet$ about which footprints commute, because it only takes into account the locks as well as the allocated and deallocated locations. As explained in the introduction, this mismatch enables us to detect *data races* in the machine as well as in the code, see also [19].

3.3 The stateful and the stateless internal opcategories $\pitchfork_S$ and $\pitchfork_L$

We have just described in §3.2 the asynchronous graphs $\pitchfork_S^\bullet$ and $\pitchfork_L^\bullet$ which we take as stateful and stateless machine models in the paper. We explain now how we turn these machine models $\pitchfork_S^\bullet$ and $\pitchfork_L^\bullet$ into the internal opcategories $\pitchfork_S$ and $\pitchfork_L$ which we will use, in the ambient category $\mathbb{S} = \mathbf{AsynGph}$ of asynchronous graphs. The constructions of the internal opcategories $\pitchfork = \pitchfork_S, \pitchfork_L$ from the asynchronous graphs $\pitchfork^\bullet = \pitchfork_S^\bullet, \pitchfork_L^\bullet$ follows exactly the same recipe: both of them are defined as cospans of monomorphisms in the ambient category $\mathbb{S} = \mathbf{AsynGph}$ of asynchronous graphs

$$\pitchfork[0] \xhookrightarrow{\ \mathrm{in}\ } \pitchfork[1] \xhookleftarrow{\ \mathrm{out}\ } \pitchfork[0]$$

describing $\pitchfork^\bullet = \pitchfork_S^\bullet, \pitchfork_L^\bullet$ as a monad (or polyad with a single color $j \in J$) in the double category $\mathbf{Cospan}(\mathbb{S})$, see §2 for details. We like to think of the asynchronous graphs $\pitchfork_S^\bullet$ and $\pitchfork_L^\bullet$ defined in §3.2 as "solipsistic" games with only one player: the underlying machine. Building on this intuition,

we follow the conceptual track discussed in the introduction, and construct an asynchronous graph $\text{ホ}[1] = \text{ホ}_S[1], \text{ホ}_L[1]$ with two players (for Code and Environment) instead of one, both in the stateful case of $\text{ホ}[1] = \text{ホ}_S[1]$ and in the stateless case $\text{ホ}[1] = \text{ホ}_L[1]$. The shift from one player to two players is performed in a very simple way. We consider the functor $\Omega : \mathbf{Set} \rightarrow \mathbf{AsynGph}$ which transports a given set $\mathcal{L}$ of labels to the asynchronous graph $\Omega(\mathcal{L})$ with one single node, the elements of $\mathcal{L}$ as edges, and one tile for each square. We are particularly interested in the case when the set of labels $\mathcal{L} = \{\mathbf{C}, \mathbf{F}\}$ contains the two polarities $\mathbf{C}$ and $\mathbf{F}$ associated with the Code and to the Frame (or the Environment), respectively. The asynchronous graph $\Omega(\{\mathbf{C}, \mathbf{F}\})$ enables us to define the *two-player* stateful and stateless machine models $\text{ホ}_S^{\circ\bullet}$ and $\text{ホ}_L^{\circ\bullet}$ as the Cartesian product of asynchronous graphs

$$\text{ホ}_S^{\circ\bullet} = \text{ホ}_S^{\bullet} \times \Omega(\{\mathbf{C}, \mathbf{F}\}) \qquad \text{ホ}_L^{\circ\bullet} = \text{ホ}_L^{\bullet} \times \Omega(\{\mathbf{C}, \mathbf{F}\})$$

Note that the resulting asynchronous graph $\text{ホ}^{\circ\bullet} = \text{ホ}_S^{\circ\bullet}, \text{ホ}_L^{\circ\bullet}$ has the same nodes as $\text{ホ}^{\bullet} = \text{ホ}_S^{\bullet}, \text{ホ}_L^{\bullet}$ and two edges, the first one labeled with a polarity $\mathbf{C}$ for Code, the second one labeled with a polarity $\mathbf{F}$ for Frame, for each edge in the original asynchronous graph $\text{ホ}^{\bullet} = \text{ホ}_S^{\bullet}, \text{ホ}_L^{\bullet}$. The two circles $\circ$ and $\bullet$ in the notation $\text{ホ}^{\circ\bullet}$ are mnemonics designed to remind us that there are two players in the game: the Code playing the white side ($\circ$) and the Environment playing the black side ($\bullet$). We have accumulated enough material at this stage to define the internal opcategories $\text{ホ} = \text{ホ}_S, \text{ホ}_L$ in the ambient category $\mathbb{S} = \mathbf{AsynGph}$. The asynchronous graph $\text{ホ}[1]$ is defined as the two-player machine model $\text{ホ}^{\circ\bullet}$ while the asynchronous graph $\text{ホ}[0]$ is defined as the one-player machine model. In summary:

$$\text{ホ}[1] = \text{ホ}^{\circ\bullet} \qquad \text{ホ}[0] = \text{ホ}^{\bullet}.$$

The asynchronous graph $\text{ホ}[0]$ with one player is then embedded in the asynchronous graph $\text{ホ}[1]$ with two players by the homomorphism $\mathrm{in} = \mathrm{out} : \text{ホ}[0] \rightarrow \text{ホ}[1]$ obtained by transporting every node of $\text{ホ}[0]$ to the corresponding node in $\text{ホ}[1]$, and every edge of $\text{ホ}[0]$ to the corresponding edge in $\text{ホ}[1]$ with the polarity $\mathbf{F}$ of the Frame.

4 THE PARALLEL PRODUCT

We describe below in full detail how to construct the parallel product $C_1 \| C_2$ of two codes C_1 and C_2 by applying a *push* and *pull* functors along a pair (26) of internal functors. We start by formulating in §4.1 the notion of **internal functor** from an internal J-opcategory ホ to an internal J'-opcategory $\text{ホ}'$ living in the same ambient category $\mathbb{S}$ with pushouts. We then introduce in §4.3 the notion of *acute span*, which leads us to the notion of *span-monoidal* internal J-opcategory formulated in §4.4. The parallel product of $\|$ of two codes is then described in §4.5.

4.1 Internal functors between internal J-opcategories

The notion of internal functor between internal J-opcategories is defined as expected.

Definition 4.1 (Internal functor). An internal functor $F = (f, F[0, \cdot], F[1, \cdot])$ from ホ to $\text{ホ}'$ is a triple consisting of a function $f : J \rightarrow J'$ between the sets of colors, together with:

- for each $i \in J$, a map $F[0, i]$ in the ambient category $\mathbb{S}$:

$$F[0, i] : \text{ホ}[0, i] \longrightarrow \text{ホ}'[0, f(i)],$$

- for each $i, j \in J$, a map $F[1, ij]$ in the ambient category $\mathbb{S}$:

$$F[1, ij] : \text{ホ}[1, ij] \longrightarrow \text{ホ}'[1, f(ij)]$$

where we use the lighter notation $f(ij)$ for $f(i)f(j)$. One asks moreover that the diagram below commutes

$$\begin{array}{ccccc}
\pitchfork[0,i] & \xrightarrow{\ \mathrm{in}_{ij}\ } & \pitchfork[1,ij] & \xleftarrow{\ \mathrm{out}_{ij}\ } & \pitchfork[0,j] \\
{\scriptstyle F[0,i]}\downarrow & & \downarrow{\scriptstyle F[1,ij]} & & \downarrow{\scriptstyle F[0,j]} \\
\pitchfork'[0,f(i)] & \xrightarrow{\ \mathrm{in}_{f(i)f(j)}\ } & \pitchfork'[1,f(ij)] & \xleftarrow{\ \mathrm{out}_{f(i)f(j)}\ } & \pitchfork'[0,f(j)]
\end{array}$$

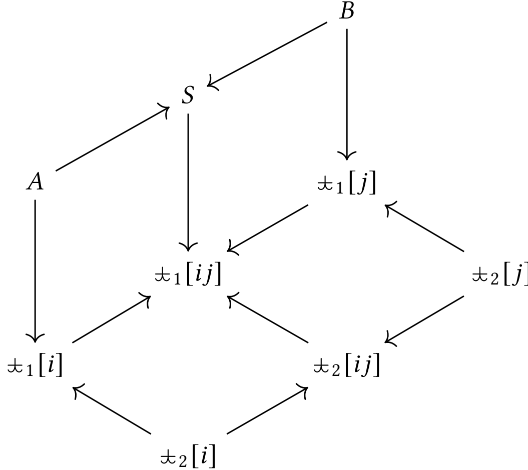
and that the internal functor is compatible with the identities: the diagram on the right below commutes; and with composition: the maps $F[2,ijk] : \pitchfork[2,ijk] \to \pitchfork'[2,f(ijk)]$ induced by universality of the pushout must make the left diagram commute.

$$\begin{array}{ccc}
\pitchfork[2,ijk] & \longrightarrow & \pitchfork[1,ik] \\
{\scriptstyle F[2,ijk]}\downarrow & & \downarrow{\scriptstyle F[1,ik]} \\
\pitchfork'[2,f(ijk)] & \longrightarrow & \pitchfork'[1,f(ik)]
\end{array}
\qquad\qquad
\begin{array}{ccc}
\pitchfork[0,i] & \xrightarrow{\ \eta_i\ } & \pitchfork[1,ik] \\
{\scriptstyle F[0,i]}\downarrow & & \downarrow{\scriptstyle F[1,ii]} \\
\pitchfork[0,f(i)] & \xrightarrow[\ \eta'_{f(i)}\ ]{} & \pitchfork'[1,f(ii)]
\end{array}$$

The internal J-opcategories and internal functors form a category $\mathbf{opCat}(\mathbb{S})$. This category admits products: the index set of the product of two internal categories is the product of their index sets, and $(\pitchfork \times \pitchfork')[0,(i,j)] = \pitchfork[0,i] \times \pitchfork'[0,j]$. This fact will play a central role in the sequel.

4.2 Plain internal functors

An important family of internal functors are **plain internal functors**, which are functors whose action on the colors is the identity. With the notations of Definition 4.1, they are the internal functors such that $f = id$. Plain internal functors are useful because they define a pull operation when $\mathbb{S}$ has pullbacks, given by the obvious three pullbacks in the following diagram:



This construction induces a lax double functor between $\mathbf{Cob}(\pitchfork_2)$ and $\mathbf{Cob}(\pitchfork_1)$. The converse operation of pushing exists for any internal functor.

Lemma 1. *A plain internal functor $F : \pitchfork_1 \to \pitchfork_2$ induces a lax double functor of double category defined by taking the pointwise pullbacks along the components of F*

$$\mathsf{pull}[F] : \mathbf{Cob}(\pitchfork_2) \longrightarrow \mathbf{Cob}(\pitchfork_1).$$

Conversely, any internal functor $F : \pm_1 \to \pm_2$ induces a pseudo functor by post-composition

$$\mathsf{push}[F] : \mathbf{Cob}(\pm_1) \longrightarrow \mathbf{Cob}(\pm_2).$$

4.3 Acute spans of internal functors

We start by introducing the notion of **acute span** of internal functors, and then describe how transport of structure works along an acute span.

Acute spans. An **acute span** between an internal J_1-opcategory $\pm_1$ and an internal J_2-opcategory $\pm_2$ is defined as a span of internal functors

$$\pm_1 \xleftarrow[=]{F} \pm_0 \xrightarrow{G} \pm_2$$

where $\pm_0$ is an internal J_0-opcategory and the = sign indicates that the internal functor F is plain. Here, we suppose that the ambient category $\mathbb{S}$ has pushouts and pullbacks. In that case, the definition gives rise to a bicategory **AcuteSpan**, whose objects are the internal J-opcategories in the ambient category $\mathbb{S}$, whose 1-cells are acute spans, and whose 2-cells are commuting diagrams of the form:

$$\begin{array}{ccccc} & & \pm_0 & & \\ & \overset{=}{\swarrow} & \downarrow{\scriptstyle \phi} & \searrow & \\ \pm_1 & & & & \pm_2 \\ & \underset{=}{\nwarrow} & & \nearrow & \\ & & \pm_0' & & \end{array}$$

where the internal functor $\phi : \pm_0 \to \pm_0'$ is plain. Acute spans are composed using pullbacks of plain internal functors along internal functors, in the category of internal J-opcategories. The pullback is defined as follows. Assume that we are given two internal functors:

$$\pm_1 \xrightarrow{(f,F)} \pm_2 \xleftarrow[=]{(id,G)} \pm_3$$

between an internal J_1-opcategory $\pm_1$ and a J_3-opcategory $\pm_3$. Their pullback is defined as the internal J_1-opcategory $\pm_1 \times_{\pm_2} \pm_3$ described below. At the level of its components and objects of $\mathbb{S}$, for each $i \in J_1$, the two internal functors give the following diagram:

$$\pm_1[0,i] \xrightarrow{F[0,i]} \pm_2[0,f(i)] \xleftarrow{G[0,f(i)]} \pm_3[0,f(i)]$$

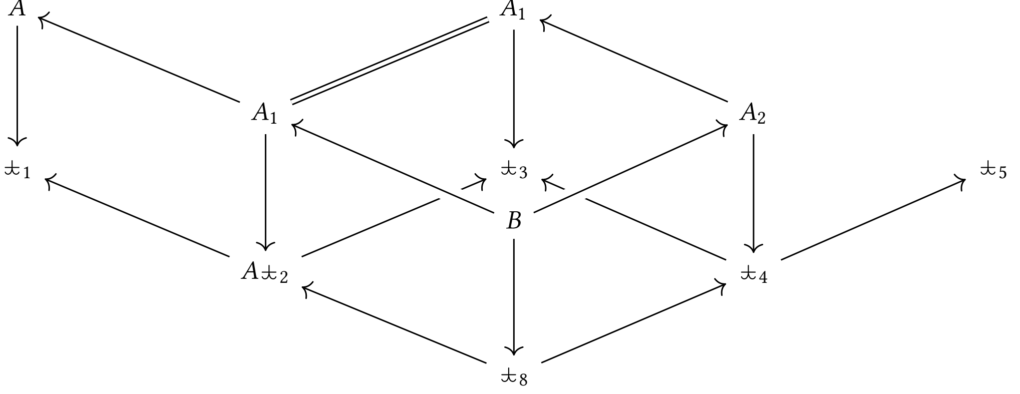
and we simply define $(\pm_1 \times_{\pm_2} \pm_3)[0,i]$ as the pullback of that diagram in the ambient category $\mathbb{S}$. We proceed similarly to define the $(\pm_1 \times_{\pm_2} \pm_3)[1,ij]$; and the universality of the pullbacks in $\mathbb{S}$ gives the structural maps between the two.

Transport along acute spans. The ultimate *raison d'être* of acute spans is to induce an operation of transport by "pull-then-push" along the two legs of a span. This operation plays a fundamental role in the paper, in particular because we derive our definition of parallel product from it. This operation can be formulated as a pseudo functor $\mathbf{Cob} : \mathbf{AcuteSpan} \to \mathbf{DblCat}_{\text{lax}}$ of bicategories, from the bicategory **AcuteSpan** just defined to the bicategory $\mathbf{DblCat}_{\text{lax}}$ of double categories and lax double functors. It is defined on objects and 1-cells by:

$$\begin{array}{cccc} \mathbf{Cob} & : & \mathbf{AcuteSpan} & \longrightarrow & \mathbf{DblCat}_{\text{lax}} \\ & & \pm & \longmapsto & \mathbf{Cob}(\pm) \\ & & (F,G) & \longmapsto & \mathsf{push}[G] \circ \mathsf{pull}[F] \end{array}$$

To prove that this induces a pseudo functor, one needs in particular to check that, given two composable acute spans F and G, $\mathbf{Cob}(F \circ G) \cong \mathbf{Cob}(F) \circ \mathbf{Cob}(G)$. This is indeed the case because,

in the following commutative diagram, the map from B to A_2 is an isomorphism, for the top face of the cube is a pullback according to the pasting lemma for pullbacks.



4.4 Span-monoidal internal J-opcategories

We use the span operation above to define the parallel product of two programs. Categorically, we construct a lax-monoidal structure on $\mathbf{Cob}(\pitchfork)$, that is to say a lax double functor

$$\parallel\, : \mathbf{Cob}(\pitchfork) \times \mathbf{Cob}(\pitchfork) \to \mathbf{Cob}(\pitchfork).$$

By lax double functor, we mean that there exists a natural and coherent family of maps:

$$(C_1 \parallel C_2) \,;(D_1 \parallel D_2) \quad \longrightarrow \quad (C_1; D_1) \parallel (C_2; D_2).$$

Since the composition of cobordisms corresponds to the sequential composition of programs, these coercion maps capture the famous Hoare inequality [12]

$$(C_1 \parallel C_2) \,;(D_1 \parallel D_2) \quad \subseteq \quad (C_1; D_1) \parallel (C_2; D_2).$$

In our setting, this follows from Proposition 1, and therefore from the fact that there is always a map from a colimit of limits to the corresponding limit of colimits.

This tensor product is built the same way as in the case of template games [16], using the notion of internal span-monoidal opcategory. We remarked below Definition 4.1, the category $\mathbf{opCat}(\mathbb{S})$ of internal opcategories in the ambient category $\mathbb{S}$ is a Cartesian category. As a consequence, the lax functor $\mathbf{Cob}$ is lax monoidal, where we equip both $\mathbf{AcuteSpan}$ and $\mathbf{DblCat}_{\text{lax}}$ with the monoidal structure induced by their Cartesian products. In particular, there exist coercions living in the Cartesian category $\mathbf{DblCat}_{\text{lax}}$, and thus provided by pseudo functors of double categories

$$\begin{array}{rcccc} \mathrm{m}_{\pitchfork_1,\pitchfork_2} & : & \mathbf{Cob}(\pitchfork_1) \times \mathbf{Cob}(\pitchfork_2) & \longrightarrow & \mathbf{Cob}(\pitchfork_1 \times \pitchfork_2) \\ \mathrm{m}_{\mathbf{1}} & : & \mathbb{1} & \longrightarrow & \mathbf{Cob}(\mathbf{1}) \end{array}$$

The first coercion is obtained in the natural way by taking the "pointwise" Cartesian product of the two cobordisms, and the second is the trivial cobordism, given by the initial opcategory $\mathbf{1}$.

Definition 4.2. A **span-monoidal internal J-opcategory** $(\pitchfork, \parallel, \eta)$ is a symmetric pseudomonoid object in the symmetric monoidal bicategory **AcuteSpan**. In particular, $\parallel$ and η are two acute spans:

$$\pitchfork \times \pitchfork \xleftarrow{\text{pick}} \pitchfork^{\parallel} \xrightarrow{\text{pince}} \pitchfork \qquad\qquad \mathbf{1} \longleftarrow \pitchfork^{\eta} \longrightarrow \pitchfork.$$

together with invertible 2-cells that witness the associativity of $\parallel$, and the fact that η is is a left and right identity of $\parallel$. See [5, §3] for the complete definition.

Now, combining the external operations $\mathrm{m}_{\pm,\pm}$ and m_{1} on the one hand, and the internal operations $\|, \eta$ on the other, we get a lax-monoidal structure on $\mathbf{Cob}(\pm)$ whose multiplication and unit are respectively given by:

$$\mathbf{Cob}(\pm) \times \mathbf{Cob}(\pm) \xrightarrow{\quad \mathrm{m}_{\pm,\pm} \quad} \mathbf{Cob}(\pm \times \pm) \xrightarrow{\quad \mathbf{Cob}(\|) \quad} \mathbf{Cob}(\pm)$$

$$\mathbb{1} \xrightarrow{\qquad \mathrm{m}_{1} \qquad} \mathbf{Cob}(\mathbf{1}) \xrightarrow{\quad \mathbf{Cob}(\eta) \quad} \mathbf{Cob}(\pm)$$

Theorem 4.3. *Every span-monoidal internal J-opcategory $\pm$ induces a symmetric lax-monoidal double category* $\mathbf{Cob}(\pm)$.

4.5 Illustration: parallel product of codes

Now that we have a general method for equipping the double category $\mathbf{Cob}(\pm)$ with a lax-monoidal structure, we use it to define the parallel product for the stateful and the stateless semantics of the Code. The case of separated states is a bit more involved and will be treated later, see §7.4 for details. The basic idea is to think of the code $C_1 \parallel C_2$ as a situation where (1) there are *three* players involved: the Code of C_1, the Code of C_2 and the overall Frame F, but where (2) we have forgotten the identities of the codes C_1 and C_2 by considering both of them to be the Code C. This idea leads us to define the three-player machine models $\pm_S^{\circ\circ\bullet}$ and $\pm_L^{\circ\circ\bullet}$, with polarities $\mathbf{C}_1, \mathbf{C}_2$ and $\mathbf{F}$. In the same way the two-player versions of the machine models are deduced from their one-player versions in §3.3 using a product, we use here the pullback:

$$\begin{array}{ccc}
\pm^{\circ\circ\bullet} & \xrightarrow{\pi^{(12)(\mathbf{F})}_{\mathrm{pol}}} & \Omega\{\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}\} \\
{\scriptstyle \pi^{(12)(\mathbf{F})}_{\mathrm{state}}} \downarrow & & \downarrow {\scriptstyle \Omega(12)(\mathbf{F})} \\
\pm^{\circ\bullet} & \xrightarrow{\quad \pi_{\mathrm{pol}} \quad} & \Omega\{\mathbf{C}, \mathbf{F}\}
\end{array}$$

where $(12)(\mathbf{F})$ denotes the function $\{\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}\} \to \{\mathbf{C}, \mathbf{F}\}$ which maps the polarities $\mathbf{C}_1$ and $\mathbf{C}_2$ to $\mathbf{C}$, and the polarity $\mathbf{F}$ to itself. More explicitly, $\pm^{\circ\circ\bullet}$ has three copies for each transition of $\pm^{\bullet}$, one copy for each of the three polarities $\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}$. The span-monoidal structures on the internal opcategories $\pm = \pm_S, \pm_L$ are defined in the same way:

$$\pm^{\circ\bullet} \times \pm^{\circ\bullet} \xleftarrow{\left\langle \pi^{(1)(2\mathbf{F})}_{\mathrm{state}}, \pi^{(2)(1\mathbf{F})}_{\mathrm{state}} \right\rangle} \pm^{\circ\circ\bullet} \xrightarrow{\pi^{(12)(\mathbf{F})}_{\mathrm{state}}} \pm^{\circ\bullet} \qquad\qquad \mathbf{1} \longleftarrow \pm^{\bullet} \xrightarrow{\ \iota_{\mathbf{F}}\ } \pm^{\circ\bullet}.$$

where, for instance, $\pi^{(1)(2\mathbf{F})}_{\mathrm{state}}$ maps every transition with polarity $\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}$ to the corresponding transition with polarity given by the map $\mathbf{C}_1 \mapsto \mathbf{C}$ and $\mathbf{C}_2, \mathbf{F} \mapsto \mathbf{F}$; while the homomorphism $\iota_{\mathbf{F}}$ embeds the asynchronous graph $\pm^{\bullet}$ into $\pm^{\circ\bullet}$ with every edge transported to the corresponding edge of polarity $\mathbf{F}$. The resulting lax-monoidal product $\|$ is essentially the same as the parallel product which was defined by hand in Melliès and Stefanesco [19]. The product synchronizes transitions of the Code in C_1 with transitions of the Environment in C_2 which are mapped to the same transition in $\pm$. The intuition here is that a transition of C_1 is seen by C_2 as a transition of its Environment. Note that the parallel product preserves tiles from C_1 and from C_2, and adds Code/Code tiles when one transition comes from C_1 and the other transition from C_2—meaning that the two instructions are executed on two different "threads"—and their image in $\pm^{\circ\bullet} \times \pm^{\circ\bullet}$ forms a tile—meaning that these two instructions are independent.

5 SEQUENTIAL COMPOSITION

So far, to compose cobordisms horizontally, the target of the first cobordism must exactly match the source of the second. This is not a reasonable assumption, for the initial and the final states of a

program are part of its internal state. To summarize, in general, the two cobordisms we wish to compose look like:

$$\begin{array}{ccc} A & \longrightarrow\!\!\!\!\!\!\!\!+\quad\;\; & B \\ \downarrow & & \downarrow \lambda_{\text{out}} \\ \pm[0,i] & \xrightarrow{\pm[1,ij]} & \pm[0,j] \end{array} \qquad\qquad \begin{array}{ccc} B' & \longrightarrow\!\!\!\!\!\!\!\!+\quad\;\; & C' \\ \lambda_{\text{in}} \downarrow & & \downarrow \\ \pm[0,i'] & \xrightarrow{\pm[1,i'j']} & \pm[0,j'] \end{array}$$

In practice, in all the cases we consider, $\pm[0,j]$ and $\pm[0,i']$ will be equal, but B and B' will be different. To bridge the gap between $(B, \lambda_{\text{out}})$ and $(B', \lambda_{\text{in}})$, we will use a **filling system** over the internal J-opcategory $\pm$. With each pair of interfaces $\lambda_{\text{out}} : B \to \pm[0,j]$ and $\lambda_{\text{in}} : B' \to \pm[0,i']$, the filling system associates a cobordism

$$(B, \lambda_{\text{out}}) \longrightarrow \mathsf{fill}((B, \lambda_{\text{out}}), (B', \lambda_{\text{in}})) \longleftarrow (B', \lambda_{\text{in}})$$

from $(B, \lambda_{\text{out}})$ to $(B', \lambda_{\text{in}})$. Thanks to this mediating cobordism, it becomes possible to compose the two cobordisms using the usual composition of cobordism. Given such a filling system, we write $C_1 \mathbin{⨟} C_2$ for this generalized form of composition.

Proposition 1. *Suppose that there exists a map* $\mathsf{fill}(\lambda \parallel \lambda', \mu \parallel \mu') \to \mathsf{fill}(\lambda, \mu) \parallel \mathsf{fill}(\lambda', \mu')$. *In that case, the Hoare inequality holds:* $(C_1 \parallel C_1') \mathbin{⨟} (C_2 \parallel C_2') \to (C_1 \mathbin{⨟} C_2) \parallel (C_1' \mathbin{⨟} C_2')$.

Proof. The Hoare inequality for the usual composition applied twice gives us the map below, from which we can conclude using the hypothesis on the filling system.

$$(C_1 \parallel C_1'); (\mathsf{fill}(\lambda, \mu) \parallel \mathsf{fill}(\lambda', \mu')); (C_2 \parallel C_2') \to (C_1 \mathbin{⨟} C_2) \parallel (C_1' \mathbin{⨟} C_2') \qquad \square$$

When the base category $\mathbb{S}$ has pullbacks as well as pushouts, which is the case in the examples we are considering, a filling system always exists. It is defined by the following diagram:

$$\begin{array}{ccccc} & \swarrow & B_j \times_{\pm[1,jj']} A_{j'} & \searrow & \\ B_j & & & & A_{j'} \\ & \searrow & B_j \cup_{\pm[1,jj']} A_{j'} & \swarrow & \\ \downarrow \lambda & & \vdots & & \downarrow \lambda' \\ \pm[0,j] & \xrightarrow{\text{in}_{jj'}} & \pm[1,jj'] & \xleftarrow{\text{in}_{jj'}} & \pm[0,j'] \end{array}$$

where $B_j \times_{\pm[1,jj']} A_{j'}$ is a pullback, and where $B_j \cup_{\pm[1,jj']} A_{j'}$ is a pushout above that pullback. Intuitively, it identifies all nodes of B_j and of $A_{j'}$ that have the same underlying state in $\pm[1,jj']$. This is the filling system which will be used in our interpretation of sequential composition. We establish in the Appendix §B that the hypothesis of Proposition 1 holds in the case of the stateful and stateless templates $\pm_S$ and $\pm_L$. Interestingly, this is not necessarily the case for the template $\pm_{\text{Sep}}$ of separated states regulating the interpretation of CSL proofs. The reason is that there are several ways to decompose a given separated state between two players **C**, **F** in $\pm_{Sep}^{\circ\bullet}$ into a separated state between three players $\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}$ in $\pm_{Sep}^{\circ\circ\bullet}$.

6 THE ERROR MONAD

We want to keep track of the errors in our transition systems, in particular we want to carefully control how errors propagate in the various constructions. Our method is to use the **error monad T** over the category of asynchronous graphs, which adds an isolated node $\bullet$. Then we swap our base category from the category $\mathbb{S}$ to its category $\mathbb{S}^{\mathbf{T}}$ of **T**-algebras. In the case of asynchronous graphs, a **T**-algebra is simply a pointed asynchronous graph (x, G) where x is a node of G, and a morphism

of algebras is an asynchronous morphism which maps the distinguished node of its source to that of its target. In our case, the distinguished node will correspond to the error. As we will see, this category $\mathbb{S}^{\mathbf{T}}$ of $\mathbf{T}$-algebras inherits from $\mathbb{S}$ the properties that are needed to define the operations that we use to interpret programs and proofs. This principled approach to error handling ensures that the error is represented as a single node in the transition systems.

6.1 Liftings

Both the stateful and the stateless machine models have natural $\mathbf{T}$-algebra structures: the distinguished node is the error node $\lightning$. Given an opcategory $\pm$ in the category of algebras $\mathbb{S}^{\mathbf{T}}$, the monad $\mathbf{T}$ can be lifted to a monad $\dot{\mathbf{T}}$ on $\mathbf{Cob}(\pm)$, where we omit the writing of the forgetful functor from the algebra to the underlying category. The idea is to define the image under $\dot{\mathbf{T}}$ of a cobordism to be the cobordism

$$\begin{array}{ccccc}
\mathbf{T}A & \xrightarrow{\mathbf{T}in} & \mathbf{T}S & \xleftarrow{\mathbf{T}out} & \mathbf{T}B \\
\downarrow{\scriptstyle \mathbf{T}\lambda_A} & & \downarrow{\scriptstyle \mathbf{T}\lambda_\sigma} & & \downarrow{\scriptstyle \mathbf{T}\lambda_B} \\
\mathbf{T}\pm[0] & \xrightarrow{\mathbf{T}\mathrm{in}} & \mathbf{T}\pm[1] & \xleftarrow{\mathbf{T}\mathrm{out}} & \mathbf{T}\pm[0] \\
\downarrow{\scriptstyle a_0} & & \downarrow{\scriptstyle a_1} & & \downarrow{\scriptstyle a_0} \\
\pm[0] & \xrightarrow{\mathrm{in}} & \pm[1] & \xleftarrow{\mathrm{out}} & \pm[0]
\end{array}$$

where a_0 and a_1 are the structural morphism of $\pm[0]$ and $\pm[1]$ respectively. The multiplication $\dot{\mu}$ is given for games by the diagram:

$$\begin{array}{ccc}
\mathbf{TT}A & \xrightarrow{\mu} & \mathbf{T}A \\
\downarrow{\scriptstyle \mathbf{TT}\lambda_A} & & \downarrow{\scriptstyle \mathbf{T}\lambda_A} \\
\mathbf{TT}\pm[0] & \xrightarrow{\mu} & \mathbf{T}\pm[0] \\
\downarrow{\scriptstyle \mathbf{T}a_0} & & \downarrow{\scriptstyle a_0} \\
\mathbf{T}\pm[0] & \xrightarrow{a_0} & \pm[0] \\
\downarrow{\scriptstyle a_0} & \nearrow{\scriptstyle id} & \\
\pm[0] & &
\end{array}$$

and similarly for cobordisms. The case of the unit $\dot{\eta}$ follows the same idea. The fact that $\dot{\mathbf{T}}$ is compatible with horizontal composition and that it satisfies the monad laws follows from the fact that the error monad T is a cocartesian monad: as a functor, it preserves pushouts, and all the naturality squares of μ and of η are pushout squares. Further, we have an inclusion from cobordisms in the ambient category of $\mathbf{T}$-algebras into the algebras of the lifted monad $\dot{\mathbf{T}}$ on cobordisms:

$$\mathbf{Cob}_{\mathbb{S}^{\mathbf{T}}}(\pm) \hookrightarrow \mathbf{Cob}_{\mathbb{S}}(\pm)^{\dot{\mathbf{T}}}$$

By monad on a double category $\mathcal{D}$, we mean a monad in the vertical 2-category $\mathbf{DblCat}_v$ of double categories (see [10]).

In the case of the error monad on asynchronous graphs, the category of $\mathbf{T}$-algebras is complete (as the forgetful functor always creates all limits), cocomplete (as $\mathbf{T}$ preserves reflexive coequalizers) and adhesive (as the forgetful functor creates pullbacks and pushouts).

6.2 Tensor product

In order to define the parallel product in this new ambient category, we need a tensor product in the category of $\mathbf{T}$-algebras. We achieve this by lifting the Cartesian product into the **smash product**

of pointed asynchronous graphs $(x_1, G_1) \boxtimes (x_2, G_2)$ which identifies any pair of nodes of the form (x_1, x_2), $(x_1, -)$ or $(-, x_2)$ and make this node the distinguished node of the smash product. In our case, the smash product is a lifting to the Eilenberg-Moore category of the the Cartesian product on asynchronous graphs, in that they commute with both the forgetful functor and the free algebra functors in the expected way. and the forgetful functor in the expected ways. See Lemma 2.20 in [21], and [13] for more details.

7 SEPARATED STATES

We now define the last of the three internal J-opcategories: $\pm_{\mathrm{Sep}}$, that will be used to interpret the proofs of CSL. Its structure is richer than the other two as it is a proper internal J-opcategory, indexed by the set of predicates of CSL. First, we define the notion of separated states, then we define the machine model $\pm^{\bullet}_{Sep}$, and finally we define the J-internal category structure $\pm_{\mathrm{Sep}}$.

7.1 Separated states

We use the same notion of separated states as Melliès and Stefanesco [18, 19]. We suppose given an arbitrary partial cancellative commutative monoid **Perm** which we call the **permission monoid**, with a top element $\top$, following Bornat et al. [3]. We require $\top$ to admit no multiples: $\forall x \in \mathbf{Perm}$, $\top \cdot x$ is not defined. The set **LState** of logical states is defined in much the same way as the set of memory states, with the addition of permissions to each variable and heap location:

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

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

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

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

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

The separation product $\sigma * \sigma'$ of the two logical states σ and σ' is not defined otherwise. In particular, the memory states underlying σ and σ' agree on the values of the shared variables and heap locations when the separation product is well defined.

The predicates of CSL are predicates on logical states. Their grammar is the following, it consists mainly in first order logic enriched with the separating conjunction:

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

where $x \in \mathbf{Var}$, $p \in \mathbf{Perm}$, $v, w \in \mathbf{Val}$, and the E_i' are arithmetic expressions that can contain metavariables $a, b, c, \ldots$ that are used for quantifiers. The semantics of these predicates is given by

the judgment $\sigma \vDash P$; it is defined by induction on the structure of P, some rules are:

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

We recall the notion of *separated state* whose purpose is to separate the logical memory state into one region controlled by the Code, one region controlled by the Frame, and one independent region for each unlocked resource. In order to define the notion, we suppose given a finite set $\mathbb{L} \subseteq \mathbf{Locks}$ of locks, and for each lock r, a predicate I. We write $\Gamma = r_1 : I_1, \ldots, r_n : I_n$.

Definition 7.1. A *separated state* is a triple

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

such that the logical state below is defined:

$$\sigma_C \ * \ \left\{ \bigcircledast_{r \in \mathrm{dom}(\boldsymbol{\sigma})} \boldsymbol{\sigma}(r) \right\} \ * \ \sigma_F \quad \in \quad \mathbf{LState} \tag{33}$$

where we write $\mathrm{dom}(\boldsymbol{\sigma})$ for $\boldsymbol{\sigma}^{-1}(\mathbf{LState})$, and similarly for $\mathrm{dom}_{\mathrm{C}}(\boldsymbol{\sigma})$ and $\mathrm{dom}_{\mathrm{F}}(\boldsymbol{\sigma})$, and such that, for all $r_k \in \mathrm{dom}(\boldsymbol{\sigma})$, $\boldsymbol{\sigma}(r) \vDash I_k$.

The logical state σ_C describes the part of the (logical and refined) memory which is owned by the Code; while σ_F describes the part which is owned by the Frame or the Environment; finally, the function $\boldsymbol{\sigma}$ indicates, for each lock r, who is holding it between Code and Frame, and if nobody is holding it, which part $\boldsymbol{\sigma}(r)$ of the logical memory the lock is currently protecting, or owning. Note that $\boldsymbol{\sigma}(r)$ is the piece of the logical memory which the Code or the Frame will get when it acquires the lock. For example, in the traditional example where the monoid of permissions is defined as $\mathbf{Perm} = (0, 1]$ with addition, the situation where both the Code and the Environment have read access to a location ℓ can be represented by the separated state $([\ell \mapsto (4, 1/2)], \emptyset, [\ell \mapsto (4, 1/2)])$ where the function $\boldsymbol{\sigma}$ is empty.

7.2 The machine model of separated states $\pm^{\bullet}_{Sep}$

In contrast to the stateful and the stateless machine models $\pm^{\bullet}_{S}$ and $\pm^{\bullet}_{L}$, which are one-player games, the machine model of separated states involves two players Code and Frame. Similarly to the graphs of states and locks, we define a notion of footprints, which turn out to be the same as the footprints of associated with the machine states:

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

The **machine model of separated states** $\pm^{\bullet}_{Sep}[\Gamma]$ is the asynchronous graph whose nodes are the separated states and whose edges are either Code or Frame transitions: Code moves are the

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \xrightarrow{m:C} (\sigma'_C, \boldsymbol{\sigma}', \sigma_F) \quad \text{such that} \quad \circledast(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \xrightarrow{m} \circledast(\sigma'_C, \boldsymbol{\sigma}', \sigma_F) \ \text{ in } \pm^{\bullet}_{S}$$

where $m \in \mathbf{Instr}$ is an instruction, and such that the following conditions are satisfied:

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

Frame moves are of the form $(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \xrightarrow{m:F} (\sigma_C, \boldsymbol{\sigma}', \sigma'_F)$ with symmetric conditions. In the same way as the other machine models, the tiles of $\pm^{\bullet}_{Sep}$ are the squares such that the footprints of the opposite instructions are independent.

7.3 The internal category $\pm_{\mathrm{Sep}}$

Recall that the internal category $\pm_{\mathrm{Sep}}$ is indexed by the set of predicates of CSL. In order to define the asynchronous graph $\pm[0, P]$ for each predicate P, we need to lift the satisfaction relation of a predicate from logical states to separated states by defining:

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \vDash P \iff \sigma_C \vDash P.$$

Then, we can define $\pm_{\mathrm{Sep}}[0, P]$ to be the subgraph of $\pm^{\bullet}_{Sep}$ induced by the separated states that satisfy P, keeping only the Frame moves. We are now ready to define the internal J-opcategory, the unique support is the whole two player graph $\pm_{\mathrm{Sep}}[1] = \pm^{\bullet}_{Sep}$, and the interfaces is the collection of all the $\pm_{\mathrm{Sep}}[0, P]$, with the obvious inclusion morphism into $\pm_{\mathrm{Sep}}[1]$. The map $\mu_P : \pm_{\mathrm{Sep}}[2, P] \to \pm_{\mathrm{Sep}}[1]$ is given by the universality of the pushout.

7.4 Parallel product understood as a span-monoidal structure on $\pm_{\mathrm{Sep}}$

In order to define the parallel product at the level of proofs, which we will use to interpret the CSL rule for the parallel product, we endow $\pm_{\mathrm{Sep}}$ with a span-monoidal structure. The main piece of this structure is the asynchronous graph $\pm^{\circ\circ\bullet}_{Sep}$ of *three-player separated states*, which will be the basis for the support of the span

$$\pm_{\mathrm{Sep}} \times \pm_{\mathrm{Sep}} \xleftarrow{\ \mathrm{pick}\ } \pm^{\parallel}_{\mathrm{Sep}} \xrightarrow{\ \mathrm{pince}\ } \pm_{\mathrm{Sep}}$$

of Definition 4.2. It is the straightforward generalization of separated states to the case where there are three players, $\mathbf{C}_1, \mathbf{C}_2$ and $\mathbf{F}$; its nodes are the states $(\sigma_1, \sigma_2, \boldsymbol{\sigma}, \sigma_F)$, such that the product is well defined. The colors are pairs of predicates, which correspond to σ_1 and σ_2. The internal functor pince has the following action on the supports of the internal J-categories: it maps a three-player state to the (two-player) separated state $(\sigma_1 * \sigma_2, \boldsymbol{\sigma}', \sigma_F)$, where $\boldsymbol{\sigma}'$ is the same as $\boldsymbol{\sigma}$ where C_1 and C_2 have been remapped to C. The morphism pick maps that three-player state to the pair $\langle(\sigma_1, \boldsymbol{\sigma}_1, \sigma_2 * \sigma_F), (\sigma_2, \boldsymbol{\sigma}_1, \sigma_1 * \sigma_F)\rangle$. The unit $\pm^{\parallel\eta}_{\mathrm{Sep}}$ of the structure is the subgraph of $\pm_{\mathrm{Sep}}[1]$ with only Frame moves.

8 CHANGE OF LOCKS

The three machine models $\pm = \pm_{\mathrm{Sep}}, \pm_S, \pm_L$ considered in this paper are parameterized by the set of *free locks* which the programs can access. In the case of the two internal opcategories $\pm_S$ and $\pm_L$, the free locks are simply described by a set $\mathbb{L}$ of lock names. In the case of the internal opcategories $\pm_{\mathrm{Sep}}$ of separated states, the free locks are described by a context $\Gamma = r_1 : I_1, \ldots, r_n : I_n$ which associates with each free lock r_k the predicate I_k of a CSL invariant. As we are considering the general case, we write Γ, r to denote a context in either of the two cases. The operations of *introducing a new lock* and of *creating a critical section* transport cobordisms across machine models parameterized with different free locks. The change-of-basis operations are induced by the two

acute spans below:

$$\pm(\Gamma) \underset{\mathsf{hide}[r]}{\overset{\mathsf{when}[r]}{\rightleftarrows}} \pm(\Gamma, r)$$

where we write explicitly the dependence of the models $\pm(\Gamma)$ on the contexts (or lists) Γ of free locks. We formalize the situation by defining the graph **LockGraph** whose vertices are the lock contexts Γ, with edges defined as transitions *adding* or *removing* one specific lock in the context:

$$\Gamma \underset{\pi_r^\Gamma}{\overset{\iota_r^\Gamma}{\rightleftarrows}} \Gamma, r$$

Consider **LockGraph*** the locally posetal bicategory freely generated by this graph, with invertible 2-cells between paths that are equal up to the reorderings: $\pi_r \circ \iota_{r'} \sim \iota_{r'} \circ \pi_r$, $\pi_r \circ \pi_{r'} \sim \pi_{r'} \circ \pi_r$ and $\iota_r \circ \iota_{r'} \sim \iota_{r'} \circ \iota_r$, for $r \neq r'$, and leaving the Γ's implicit. By locally posetal, we mean that there is at most one tile (or invertible 2-cell) between any pair of morphisms, witnessing that they are equal up to the reorderings above. We call a **LockGraph-template** a pseudo functor from this bicategory to **AcuteSpan**$(\mathbb{S})$, the bicategory of internal J-opcategories and acute spans we defined in §4.3. The invertible 2-cells in the domain reflect the fact that the order of the operations does not matter, up to isomorphism. The three families $\pm = \pm_{\mathrm{Sep}}, \pm_S, \pm_L$ of internal J-opcategories defined so far are **LockGraph**-templates; we explain how in the remainder of this section. Practically, this means that, by post-composing with the lax functor **Cob**$(\cdot)$, we are able to transport a cobordism defined on the internal category $\pm(\Gamma)$ to one defined on the internal category $\pm(\Gamma, r : I)$, to interpret critical sections, and back, for resource introduction. We abuse the notation and write these lax double functors as follows:

$$\mathbf{Cob}(\pm(\Gamma)) \underset{\mathsf{hide}[r]}{\overset{\mathsf{when}[r]}{\rightleftarrows}} \mathbf{Cob}(\pm(\Gamma, r))$$

8.1 Hiding

To hide a lock r, we proceed in two steps for all the templates that we consider $\pm = \pm_L, \pm_S, \pm_{\mathrm{Sep}}$. First, we prevent the Environment from touching that lock, and then we remove this lock from the states and we transform all transitions $P(r), V(r)$ into nops. Formally, hiding is defined by a pull and push operation along the acute span $\mathsf{hide}[r]$:

$$\pm(\Gamma, r) \xleftarrow{\mathrm{inj}_C} \pm^{\langle r \rangle}(\Gamma, r) \xrightarrow{\mathsf{hide}_C} \pm(\Gamma).$$

The support $\pm^{\langle r \rangle}(\Gamma, r)$ of the span is defined to be the same as the template $\pm(\Gamma, r)$, except that all Environment transitions $P(r), V(r)$ are deleted, and, only in the case of the template of separated states, we remove the states of the form $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$, where the lock r is held by the Environment. By definition of $\pm^{\langle r \rangle}(\Gamma, r)$ as a restriction of $\pm(\Gamma, r)$, there is a canonical injection inj_C. The map hide_C is defined differently depending on the kind of template we are considering. In the case of the templates used for the code, respectively $\pm_L$ and $\pm_S$, we map the states $L \subseteq \mathbb{L}$ to $L \setminus \{r\}$, and $\mathfrak{s} = (\mu, L)$ to $(\mu, L \setminus \{r\})$ respectively. In the case of the template of separated states $\pm_{\mathrm{Sep}}$ used to interpret proofs, it is defined as follows on separated states:

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \mapsto \begin{cases} (\sigma_C * \boldsymbol{\sigma}(r), \boldsymbol{\sigma} \setminus r, \sigma_F) & \text{if } \boldsymbol{\sigma}(r) \in \mathbf{State} \\ (\sigma_C, \boldsymbol{\sigma} \setminus r, \sigma_F) & \text{if } \boldsymbol{\sigma}(r) = C \end{cases}$$

In all cases, hide_C maps $P(r)$ and $V(r)$ to nop, and otherwise preserves the edges. The intuition behind the definition of hide_C for the separated states is that when we forget about the lock r, even when nobody is holding it, we need to do something with the resources that is associated with the lock: we chose to give this resource to the Code. This means that outside of the binder for r, the resource associated with r is not shared, but belongs to the Code. Remark that the lock r does not appear in the result, which means that the semantics of `resource` r `do` C is invariant under α-conversion.

8.2 Critical sections

The case of critical sections is more delicate, as the definition differs between $\pm_L$, $\pm_S$ on the one hand, and $\pm_{\mathrm{Sep}}$ on the other. In the case of the semantics of the code, we wish to let the Environment be wild and change the states of locks whenever it wants, even if it happens to be held by the Code. On the other hand, when it comes to the semantics of the proofs, we enforce the discipline that a lock can only be unlocked by the player that locked it. This difference of requirements is reflected by differences in the definition of when$[r]$.

Stateful and stateless semantics. We consider the machine model $\pm_{\langle r\rangle}(\mathbb{L} \uplus \{r\})$ which is the restriction of $\pm(\mathbb{L} \uplus \{r\})$ where *only the Environment* is able to use the lock r. This corresponds to the fact that inside a critical section the Code loses access to the corresponding lock until the end of the section. There exists a map ∇ from $\pm_{\langle r\rangle}(\mathbb{L} \uplus \{r\})$ to $\pm(\mathbb{L} \uplus \{r\})$ given by:

$$(\mu, L) \mapsto (\mu, L \cap \mathbb{L})$$
$$P(r), V(r) : F \mapsto \mathsf{nop}$$
$$m \mapsto m$$

Then, the lifting of a cobordism for the critical sections is given by the following acute span:

$$\pm(\mathbb{L}) \xleftarrow{\ \nabla\ } \pm_{\langle r\rangle}(\mathbb{L} \uplus \{r\}) \xrightarrow{\ \mathrm{incl}\ } \pm(\mathbb{L} \uplus \{r\})$$

where the right leg is the obvious inclusion. Intuitively, the operation of pulling along ∇, which defines the when$[r]$ operation, duplicates the whole transition system, with one version for each state of the lock r. It is the Environment which is in control of this, and the Code is oblivious to the state of the lock r.

Separated state semantics. The lifting operation, that we will use to deal with critical sections, is simply defined as the push-forward along the map

$$\pm_{\mathrm{Sep}}(\Gamma) \longrightarrow \pm_{\mathrm{Sep}}(\Gamma, r : I)$$

which sends a separated state $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ over Γ to the separated state over $\Gamma, r : I$ where the lock is held by the code $(\sigma_C, \boldsymbol{\sigma} \uplus [r \mapsto C], \sigma_F)$.

9 A UNIFORM INTERPRETATION OF CODES AND PROOFS

We have carefully studied in previous sections how to express the main operations of concurrent separation logic (sequential composition, parallel product and change of lock) in the conceptual language of template games and cobordisms inspired from [16]. Now that each of these basic operations has been rigorously defined, the interpretation of the code and of the proofs of concurrent separation logic (CSL) is uniform and essentially straightforward.

9.1 Stateful and stateless interpretations of the code

We begin by describing how the stateful and stateless interpretations $[\![C]\!]_S$ and $[\![C]\!]_L$ of a given code C are computed in our template game model. Since the interpretation is uniform in $\pm_S$ and $\pm_L$ and does only depend on the combinators of C, we find convenient to write $\pm$ alternatively for $\pm_S$ or $\pm_L$. We recall below the grammar of the concurrent programming language with shared memory, dynamic allocation and locks designed by Brookes [4] in his seminal paper on the semantics of CSL:

$$B ::= \textbf{true} \mid \textbf{false} \mid B \wedge B' \mid B \vee B' \mid E = E' \qquad E ::= 0 \mid 1 \mid \ldots \mid x \mid E + E' \mid E * E'$$

$$C ::= x := E \mid x := [E] \mid [E] := E' \mid \texttt{skip} \mid \texttt{dispose}(E) \mid x := \texttt{malloc}(E) \mid C; C' \mid C_1 \parallel C_2$$
$$\mid \texttt{while}\ B\ \texttt{do}\ C \mid \texttt{resource}\ r\ \texttt{do}\ C \mid \texttt{with}\ r\ \texttt{do}\ C \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2$$

The code C is interpreted by structural induction as a (pointed) cobordism of the form

$$[\![C]\!] \quad = \quad \begin{array}{ccccc} [\![C]\!]_{in} & \longrightarrow & [\![C]\!]_{support} & \longleftarrow & [\![C]\!]_{out} \\ \lambda_{\text{in}}\downarrow & & \downarrow & & \downarrow\lambda_{\text{out}} \\ \pm(\mathbb{L})[0] & \longrightarrow & \pm(\mathbb{L})[1] & \longleftarrow & \pm(\mathbb{L})[0]. \end{array} \tag{34}$$

where the set L of available locks is taken as implicit parameter. The interpretation of every non-leaf command of the language corresponds to an operation on cobordisms already defined, and it is thus straightforward. Then, except for `malloc` which is treated in full details in the Appendix, every leaf command $x := E \mid x := [E] \mid [E] := E' \mid \texttt{skip} \mid \texttt{dispose}(E)$ of the language corresponds to a specific machine instruction m which is interpreted as a cobordism $[\![m]\!]$ living in the double category $\mathbf{Cob}(\pm(\mathbb{L}))$, with the set $\mathbb{L}$ of available locks as implicit parameter. The cobordism $[\![m]\!]$ is constructed in the following way: its input and output borders $[\![m]\!]_{in}$ and $[\![m]\!]_{out}$ are defined as the asynchronous graph $\pm(\mathbb{L})[0]$, while its support $[\![m]\!]_{support}$ consists of the disjoint union of $[\![m]\!]_{in}$ and $T[\![m]\!]_{out}$ augmented with an edge $s_1 \to s_2$ from $s_1 \in [\![m]\!]_{in}$ to $s_2 \in T[\![m]\!]_{out}$ for every machine transition $m : s_1 \to s_2$ performed by the instruction m. Note that $[\![m]\!]_{in}$ and $[\![m]\!]_{out}$ contain only Frame transitions, and that all the "transverse" edges $s_1 \to s_2$ from $[\![m]\!]_{in}$ to $T[\![m]\!]_{out}$ are Code transitions, with the state s_2 potentially equal to the error state.

We now detail how to give a semantics to any code C as a cobordism $[\![C]\!]$, by induction on its structure. This lets us build $[\![C]\!]_S$ and $[\![C]\!]_L$ in the same way.

Instructions. We explain first how to define the cobordism that interprets a single instruction m using a well chosen pullback. Consider the following asynchronous graph

$$A = \quad \overset{F_1}{\circlearrowleft}\bullet \xrightarrow{\ C\ } \bullet\overset{F_2}{\circlearrowleft}$$

with a tile $F_1 \cdot C \sim C \cdot F_2$. Then, we can construct the pullback, where **Instr** is the set of instruction:

$$\begin{array}{ccc} G(m) & \hookrightarrow & A \times \underline{\pm}^1 \\ \downarrow & & \downarrow f \\ \Omega(\{F, m\}) & \hookrightarrow & \Omega(\{F\} \cup \mathbf{Instr}) \end{array}$$

where the map f sends edges of the form $(F_1, \cdot)$ and $((F_2, \cdot))$ to F and edges of the form (C, m') to the edge m' in $\Omega(\{F\} \cup \mathbf{Instr})$, for all instructions m'. It is then easy to deduce the maps from the borders using the universal property of the pullback.

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

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

Conditionals. Conditional branching is interpreted as

$$[\![\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]$$

defined as

$$[\![\texttt{test}(B)]\!]; [\![C_1]\!] \oplus [\![\texttt{test}(\neg B)]\!]; [\![C_2]\!] \tag{35}$$

recomposed with

$$\mathsf{fill}(\pm[0] \xrightarrow{id} \pm[0], \pm[0] + \pm[0] \xrightarrow{\nabla} \pm[0])$$

Here, the purpose of precomposing with the filling is to identify the two copies of the input $\pm[0]$ appearing on each sides of the disjoint sum (35).

Sequential and parallel compositions. We use the sequential and the parallel product of cobordisms to interpret their syntactic counterparts:

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

Resource introduction and critical sections. We use the change of locks operations. The interpretation of $\texttt{resource}\ r\ \mathsf{do}\ C$ is defined as

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

The interpretation of $[\![\texttt{with}\ r\ \mathsf{do}\ C]\!]$ is defined as:

$$[\![P(r)]\!] \,⨾\, \text{when}[r]\big([\![C]\!]\big) \,⨾\, [\![V(r)]\!].$$

Loops. For loops, the interpretation of $C' = \texttt{while}\ B\ \mathsf{do}\ C$ is defined as the unfolding of

$$X \mapsto [\![\texttt{test}(B)]\!] \,⨾\, [\![C]\!] \oplus [\![\texttt{test}(\neg B)]\!]$$

see the Appendix for details.

9.2 Interpretation of the proofs

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

9.3 Interactive and separated interpretations of the proofs

We recall in Fig. 1 the main inference rules of concurrent separation logic (CSL) as it appears in its original form, see Brookes [4]. The inference rule Res associated with $\texttt{resource}\ r\ \mathsf{do}\ C$ moves to the shared context Γ a piece of the logical state owned and potentially used by the Code, so that the new resource $r : J$ can be accessed concurrently inside the code C. However, the access to that resource $r : J$ is typically mediated by the `with` constructor, which grants temporary access under the condition that one gives it back. Note that the rule With has the side condition $P \Rightarrow \mathrm{def}(B)$ which means that if P is true in some logical state, then it implies that for every free variable x of B, there exists a permission p such that $\mathrm{own}_p(x)$ holds. At this stage, our purpose is to interpret by

$$\frac{\Gamma \vdash \{P\}C_1\{Q\} \quad \Gamma \vdash \{Q\}C_2\{R\}}{\Gamma \vdash \{P\}C_1;C_2\{R\}}\textsc{Seq} \qquad \frac{\Gamma \vdash \{P_1\}C\{Q_1\} \quad \Gamma \vdash \{P_2\}C\{Q_2\}}{\Gamma \vdash \{P_1 \vee P_2\}C\{Q_1 \vee Q_2\}}\textsc{Disj} \qquad \frac{\Gamma, r : J \vdash \{P\}C\{Q\}}{\Gamma \vdash \{P * J\}\mathtt{resource}\ r\ \mathtt{do}\ C\{Q * J\}}\textsc{Res}$$

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

Fig. 1. Inference rules of Concurrent Separation Logic

structural induction every CSL proof $\pi : \Gamma \vdash \{P\}C\{Q\}$ as a cobordism

$$[\![\pi]\!]_{Sep} \quad = \quad \begin{array}{ccccc} [\![\pi]\!]_{Sep,in} & \longrightarrow & [\![\pi]\!]_{Sep,support} & \longleftarrow & [\![\pi]\!]_{Sep,out} \\ \downarrow & & \downarrow & & \downarrow \\ \pm_{\mathrm{Sep}}(\Gamma)[0,P] & \longrightarrow & \pm_{\mathrm{Sep}}(\Gamma)[1] & \longleftarrow & \pm_{\mathrm{Sep}}(\Gamma)[0,Q]. \end{array} \tag{36}$$

living in the double category $\mathbf{Cob}(\pm_{\mathrm{Sep}}(\Gamma))$ associated with the template $\pm_{\mathrm{Sep}}(\Gamma)$ of separated states, parameterized by the context Γ. As it stands, the interpretation is essentially straightforward, since most of the rules of the logic correspond to an operation on cobordisms already carefully defined. We refer the reader to the Appendix for the comprehensive definitions. There is apparently one exception however: the Frame rule does not seem, at least at first sight, to correspond to a basic operation on cobordisms. Given a cobordism $[\![\pi]\!]_{Sep}$ which interprets a proof π of the Hoare triple $\Gamma \vdash \{P\}C\{Q\}$, we need to define a new cobordism associated with the Hoare triple $\Gamma \vdash \{P*R\}C\{Q*R\}$. The solution is not difficult to find however: we define the new cobordism as the parallel product $[\![\pi]\!]_{Sep} \parallel \pm_{\mathrm{Sep}}[0,R]$, where the asynchronous graph $\pm_{\mathrm{Sep}}[0,R]$ is lifted to the identity cobordism defined in the expected way in the double category $\mathbf{Cob}(\pm_{Sep})$.

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

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

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

$$\left[\!\!\left[\frac{\overset{\vdots\, \pi}{\Gamma \vdash \{P\} C \{Q\}}}{\Gamma \vdash \{P * R\} C \{Q * R\}} \right]\!\!\right]_{Sep} = [\![\pi]\!]_{Sep} \parallel id_{\pm_{\mathrm{Sep}}[0,R]}$$

$$\left[\!\!\left[\frac{\overset{\vdots\, \pi}{\Gamma, r : J \vdash \{P\} C \{Q\}}}{\Gamma \vdash \{P * J\} \mathtt{resource}\ r\ \mathtt{do}\ C \{Q * J\}} \right]\!\!\right]_{Sep} = \mathsf{hide}[r]\big([\![\pi]\!]\big)$$

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

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

10 THE ASYNCHRONOUS SOUNDNESS THEOREM REVISITED

We briefly explain how our principled and axiomatic description of CSL leads us to a radically new way to understand and also to prove the asynchronous soundness theorem recently established by Melliès and Stefanesco [19].

10.1 Comparing the three interpretations

We have just shown how to recover by purely conceptual means the three interpretations $[\![-]\!]_S$, $[\![-]\!]_L$ and $[\![-]\!]_{Sep}$ of the codes and proofs of CSL designed by Melliès and Stefanesco in their proof of the asynchronous soundness theorem. Given a code C and a proof π of $\Gamma \vdash \{P\} C \{Q\}$, their proof of asynchronous soundness relies on the existence of a chain of *translations*

$$[\![\pi]\!]_{Sep} \xrightarrow{\ \mathcal{S}\ } [\![C]\!]_S \xrightarrow{\ \mathcal{L}\ } [\![C]\!]_L \tag{37}$$

between cobordisms living respectively in $\pm_{\mathrm{Sep}}(\Gamma)$, $\pm_S(\mathbb{L})$ and $\pm_L(\mathbb{L})$. Here, we write $\mathbb{L}$ for the domain of Γ. In order to clarify the functorial nature of these translations, one starts by observing the existence of internal functors between the colored internal opcategories:

$$\pm_{\mathrm{Sep}}(\Gamma) \xrightarrow{\ u_{\mathcal{S}}\ } \pm_S(\mathbb{L}) \xrightarrow{\ u_{\mathcal{L}}\ } \pm_L(\mathbb{L}).$$

The first internal functor $u_{\mathcal{S}}$ transports every separated state $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$ into the machine state obtained by multiplying all its components as in (33) and by forgetting all the permissions. The second internal functor $u_{\mathcal{L}}$ forgets the memory from a machine state in order to obtain the corresponding lock state. The three systems of tiles which equip the synchronization templates $\pm_{Sep}$, $\pm_S$ and $\pm_L$ were carefully designed in order to ensure that these internal functors do indeed exist. Since we can compare by a simulation two cobordisms defined over the same internal J-opcategory, we can also compare by "change-of-basis" two cobordisms over different internal

J-opcategories. As a matter of fact, the translations in (37) are simulations:

$$\mathsf{push}[u_{\mathcal{S}}](\llbracket \pi \rrbracket_{Sep}) \xrightarrow{\ \mathcal{S}\ } \llbracket C \rrbracket_S \quad \text{and} \quad \mathsf{push}[u_{\mathcal{L}}](\llbracket C \rrbracket_S) \xrightarrow{\ \mathcal{L}\ } \llbracket C \rrbracket_L.$$

The definition of internal J-functor implies that we have a natural family of isomorphisms

$$\mathsf{push}[u](C\,;\,D) \xrightarrow{\ iso\ } \mathsf{push}[u](C)\,;\,\mathsf{push}[u](D)$$

in the ambient category $\mathbb{S} = \mathbf{AsynGph}$. Intuitively, the reason why this is an isomorphism is that, in both cases, we perform the same pushout for cospan composition $C, D \mapsto C; D$ at the level of the supports of games and cobordisms. The corresponding (non invertible) comparison map associated with the parallel product

$$\mathsf{push}[u](C \parallel D) \longrightarrow \mathsf{push}[u](C) \parallel \mathsf{push}[u](D)$$

is derived from the fact that the internal functor u can be equipped with the structure of a span-monoidal functor, whose main component is a map $u^{\parallel}$ that makes the following diagram commute:

$$\begin{array}{ccccc}
\pitchfork \times \pitchfork & \xleftarrow{\ \text{pick}\ } & \pitchfork^{\parallel} & \xrightarrow{\ \text{pince}\ } & \pitchfork \\
{\scriptstyle u \times u}\downarrow & & \downarrow{\scriptstyle u^{\parallel}} & & \downarrow{\scriptstyle u} \\
\pitchfork' \times \pitchfork' & \xleftarrow{\ \text{pick}\ } & \pitchfork'^{\parallel} & \xrightarrow{\ \text{pince}\ } & \pitchfork'
\end{array}$$

10.2 The asynchronous soundness theorem

Suppose given two cobordisms σ and τ with respective supports S and T on one of the three machine models $\pitchfork = \pitchfork_{\text{Sep}}, \pitchfork_S, \pitchfork_L$, together with an asynchronous morphism $f : S \longrightarrow T$ satisfying the expected equation $\lambda_\sigma = \lambda_\tau \circ f$. Following the terminology introduced by Melliès and Stefanesco [19], we declare that the morphism $f : A \longrightarrow B$ is

- a **Code 1-fibration** when every transition $n : f(s) \longrightarrow s'$ performed by the Code in B can be lifted to a transition $m : s \longrightarrow s''$ performed by the Code in A, such that $f(m) = n$,
- a **2-fibration** when every tile of the form $f(s_1 \longrightarrow s_2 \longrightarrow s_3) \sim f(s_1) \longrightarrow t_2' \longrightarrow f(s_3)$ in B can be lifted to a tile $s_1 \longrightarrow s_2 \longrightarrow s_3 \sim s_1 \longrightarrow s_2' \longrightarrow s_3$ in A, satisfying in particular $f(s_2') = t_2'$.

We can now state the asynchronous soundness theorem for CSL established by Melliès and Stefanesco [19] which relies on these two notions of fibrations in order to express the *safety* and *data-race freedom* of the code in a clean topological way. In order to prove these two properties, the theorem focuses on the nature of the asynchronous comparison maps between cobordisms

$$\llbracket \pi \rrbracket_{Sep} \xrightarrow{\ \ \mathcal{S}\ \ } \llbracket C \rrbracket_S \xrightarrow{\ \ \mathcal{L}\ \ } \llbracket C \rrbracket_L$$

discussed in §10.1. The theorem states that for every CSL proof π of $\Gamma \vdash \{P\}C\{Q\}$,

Theorem 10.1 (Asynchronous soundness theorem). *The comparison map $\mathcal{S} : \llbracket \pi \rrbracket_{Sep} \longrightarrow \llbracket C \rrbracket_S$ is a Code 1-fibration, and the comparison map $\mathcal{L} \circ \mathcal{S} : \llbracket \pi \rrbracket_{Sep} \longrightarrow \llbracket C \rrbracket_L$ is a 2-fibration.*

The first part of the theorem ensures that a program specified in CSL does not crash. Indeed, the cobordism $\llbracket \pi \rrbracket_{Sep}$ lives above $\pitchfork_{\text{Sep}}[1]$ which does not include the error state $\lightning$. Since every step performed by the Code in $\llbracket C \rrbracket_S$ can be lifted to $\llbracket \pi \rrbracket_{Sep}$ by the 1-fibrational property, the specified Code cannot produce any error. The second part of the theorem ensures that a specified program does not produce nor encounter any data race. Indeed, every time two instructions are executed in parallel in the machine, they define a tile in the cobordism $\llbracket C \rrbracket_L$, which can be lifted by the 2-fibrational property to a tile in the cobordism $\llbracket \pi \rrbracket_{Sep}$ of separated states. There, the very existence of the tile implies that these two instructions are independent and do not produce any data race.

The conceptual and axiomatic framework of template games enables us to reunderstand in a radically new way the original proof of the asynchronous soundness theorem [19] by reducing it to the preservation properties of the 1-dimensional and 2-dimensional fibrations with respect to pullbacks, in the hom-categories of cobordisms. The proof relies on the observation that the chain of translations (37) induces a sequence of pullback diagrams between the interpretation of the code in (34) and of the CSL proof in (36):

$$\begin{array}{ccccc}
[\![\pi]\!]_{Sep,in} & \longrightarrow & [\![\pi]\!]_{Sep,support} & \longleftarrow & [\![\pi]\!]_{Sep,out} \\
\downarrow & \textit{pullback} & \downarrow & \textit{pullback} & \downarrow \\
[\![C]\!]_{S,in} & \longrightarrow & [\![C]\!]_{S,support} & \longleftarrow & \mathbf{T}[\![C]\!]_{S,out} \\
\downarrow & \textit{pullback} & \downarrow & \textit{pullback} & \downarrow \\
[\![C]\!]_{L,in} & \longrightarrow & [\![C]\!]_{L,support} & \longleftarrow & \mathbf{T}[\![C]\!]_{L,out}
\end{array}$$

These pullback diagrams indicate that the input and output of the various cobordisms interpreting C and π behave properly from the point of view of the translations. This pullback property is a remarkable consequence of the fact that the ambient category **AsynGph** of asynchronous graphs is adhesive, as a presheaf category, see E.1.1 for a definition of adhesivity. Then, building on the existence of these pullback diagrams, one establishes using careful and diagrammatic arguments around the notion of van Kampen square (see Lemma 6) a key compatibility property between the 2-dimensional fibrations (stable by limits) and the pushout (a colimit) defining the sequential composition.

10.3 Structural properties of cobordisms

The proof of Theorem 10.1 relies on the cobordisms that interpret programs and on proofs to have strong structural properties. Most of these properties are of a fibrational nature, ranging in a wider gamut than what we have introduced so far. In order to handle uniformly this variety of notions of fibrations, it is quite useful to express them as right-lifting properties. Given an asynchronous morphism $f : G \longrightarrow H$ between two asynchronous graphs whose edges have C and F polarities, we define the notions of **Code 1-fibration**, **Environment 1-fibration**, **Environment 1-op-fibration** and of **2-fibration** as right lifting properties, denoted in the four diagrams below, respectively. Each means that whenever there is such a square that commute, there exists a map as the one denoted by a dashed arrow that makes the two triangle commute.

$$\begin{array}{ccc}
\bullet & \longrightarrow & G \\
{\scriptstyle s}\downarrow & \nearrow & \downarrow{\scriptstyle f} \\
\cdot \rightarrow_C \cdot & \longrightarrow & H
\end{array}
\qquad
\begin{array}{ccc}
\bullet & \longrightarrow & G \\
{\scriptstyle s}\downarrow & \nearrow & \downarrow{\scriptstyle f} \\
\cdot \rightarrow_F \cdot & \longrightarrow & H
\end{array}
\qquad
\begin{array}{ccc}
\bullet & \longrightarrow & G \\
{\scriptstyle t}\downarrow & \nearrow & \downarrow{\scriptstyle f} \\
\cdot \rightarrow_F \cdot & \longrightarrow & H
\end{array}
\qquad
\begin{array}{ccc}
\wedge & \longrightarrow & G \\
{\scriptstyle u}\downarrow & \nearrow & \downarrow{\scriptstyle f} \\
\blacklozenge & \longrightarrow & H
\end{array}$$

The asynchronous graph $\bullet$ denotes the graph consisting of a single node; $\cdot \rightarrow_C \cdot$ denotes the graph made up of two nodes and a single Code arrow between them, $\cdot \rightarrow \cdot$ denotes the same, but with a Frame polarity; $\wedge$ denotes the asynchronous graph made up of a path of length 2, and finally $\blacklozenge$ is the graph made up of a single tile. The two maps called s map the node into the source of the edge, t maps into the target, and u maps the path of length 2 into the upper path of the tile. A map that is both a fibration and an opfibration is called a **bifibration**. We say that a **map of cobordism is a fibration** of a certain kind when its underlying map between the supports of the cobordisms is a

fibration of that kind. One advantage of this formulation by right-lifting is that it makes is obvious that all notions of fibration are *stable under pullback*, which will be extremely useful in the sequel.

We now state the structural properties that the cobordisms that correspond to interpretations of proofs and of programs satisfy. With the following notations for the cobordism,

$$\begin{array}{ccccc} A & \xrightarrow{\;in\;} & S & \xleftarrow{\;out\;} & B \\ {\scriptstyle\lambda_A}\downarrow & & \downarrow{\scriptstyle\lambda_\sigma} & & \downarrow{\scriptstyle\lambda_B} \\ \pm[0,i] & \xrightarrow{\;\text{in}_{ij}\;} & \pm[1,ij] & \xleftarrow{\;\text{out}_{ij}\;} & \pm[0,j] \end{array}$$

(1) the map $in : A \longrightarrow \pm[0,i]$ is an epi,
(2) the map $\lambda_\sigma : A \longrightarrow \pm[1,ij]$ is an Environment 1-fibration,
(3) the map $out : B \longrightarrow S$ is a monomorphism,
(4) the pullback of *in* along *out* is the initial object.

Conditions (1) and (2) are related to receptivity in game semantics: the program must let the environment start from any state, and it must accept any legal mode that the environment wishes to play. The last two conditions are more technical and are needed to ensure that sequential composition behaves nicely. Moreover, we ask of each of our templates that the maps in_{ij} and out_{ij} be monos, Environment 1-fibrations and 2-fibrations

We prove that all interpretations of proofs and of programs satisfy these conditions by induction on their structures. The case of the parallel product is quite simple. Conditions (1) and (4) are preserved because epis and monos are stable under pullbacks (all epis are regular in **AsynGph**), and pince : $\pm^{\otimes}_{\text{Sep}}[P] \longrightarrow \pm_{\text{Sep}}[P]$ is an epimorphism. Conditions (2) and (3) are preserved because fibrations are preserved by products and pullbacks.

The case of sequential composition is bit more interesting. Condition (1) is obvious, condition (4) follows from the fact that monos are preserved under pushouts in an adhesive category. Preservation of condition (2) is less obvious, and relies on the fact that the walking node and the walking arrow $\cdot \to \cdot$ is **tiny**, (which means that the functor $\mathbf{Hom}(\cdot \to \cdot, -)$ preserves small colimits) since they are representable. It follows from the following lemma:

Lemma 2. *Given the following diagram:*

$$\begin{array}{ccccccc} & Y & & \xrightarrow{\quad\lambda_2\quad} & & C & \\ {\scriptstyle f}\nearrow & & \searrow{\scriptstyle f'} & & \nearrow{\scriptstyle i} & & \searrow{\scriptstyle i'} \\ & & Z & \overset{\mu}{\dashrightarrow} & & & D \\ W & \text{po} & & \longrightarrow & A & \text{po} & \\ {\scriptstyle g}\searrow & & \nearrow{\scriptstyle g'} & & \searrow{\scriptstyle j} & & \nearrow{\scriptstyle j'} \\ & X & & \xrightarrow{\quad\lambda_1\quad} & & B & \end{array}$$

where lambda$_1$, λ_2, i and j are Environment 1-fibrations, and i, i', j, j' are monos. Then the map μ induced by the universality of the pushout is an Environment 1-fibration as well.

Proof. Suppose we have a node x in Z which is mapped to the source node of an Environment transition t in D. Because the walking arrow is tiny, either B of C contains a transition t' which is mapped to the transition t. Without loss of generality, suppose that it is in B. Similarly, the node x has an antecedent x' in either X or Y. Suppose first that it is in X. Then, because j' is a monomorphism, it must be that this node is mapped by λ_1 to the source node of t', and then we conclude using the fact that λ_1 is an Environment 1-fibration. Suppose now that this node x' is in

Y. Then the node $\lambda_2(x')$ in C is mapped to the same node in D as the source node of t', therefore there is a node y in A which is mapped under j to the source node of t', and under i to $\lambda_2(x')$. We get back to the previous case by using the fact that j is an Environment 1-fibration. □

We now establish that composing with a filling system also preserves these structural properties. This is where condition (4) comes into play. The reason condition (3), which states that the map from the output states to the support is a mono, is preserved is that the inputs states I and the output states O are disjoints, which implies that the possible identifications the input states the that happens during the pushout do not interfere with the output states.

Lemma 3. *Given the following commutative diagram, where the square is a pushout and o' is a mono:*

$$\begin{array}{ccccc} & M & & & O' \\ {\scriptstyle o}\swarrow & & \searrow{\scriptstyle i'} & {\scriptstyle o'}\swarrow & \\ C & & & C' & \\ {\scriptstyle f}\searrow & & \swarrow{\scriptstyle g} & & \\ & D & & & \end{array}$$

if the pullback of i' and o' is the initial object, then the composite $o' \circ g$ is a mono. This clearly implies that composing with a filling system preserves condition (3).

Proof. First, we establish that the C' contains the disjoint union of the image of M under i' and the image of O' under o'. For that purpose, consider an epi-mono factorization $i'_2 \circ i'_1$ of i' and the following diagram, where the two squares are pullbacks:

$$\begin{array}{ccc} M & \longleftarrow & \emptyset \\ {\scriptstyle i'_1}\downarrow & & \downarrow \\ C'_M & \longleftarrow & X \\ {\scriptstyle i'_2}\downarrow & & \downarrow \\ C' & \longleftarrow & O' \end{array}$$

As epimorphisms are preserved under pullbacks in **AsynGph**, the map $\emptyset \to X$ is an epi, which means that X is the initial object $\emptyset$. From that, we deduce that the map

$$I' + O' \longrightarrow C'_M$$

is a monomorphism, using the fact that colimits are universal. This is an instance of the general fact that the union of two subobjects is given by the pushout above the pullback. Now, consider the

following diagram, where all squares are pushouts:

$$\begin{array}{ccccc} C & \xleftarrow{\;o\;} & M & & \\ \downarrow & & \downarrow{\scriptstyle i'_1} & & \\ A & \xleftarrow{\;h\;} & C'_M & & \\ \downarrow & & \downarrow{\scriptstyle \iota_1} & & \\ A+O' & \xleftarrow{\;h+id\;} & C'_M+O' & \xleftarrow{\;\iota_2\;} & O' \\ {\scriptstyle k}\downarrow & & \downarrow{\scriptstyle [i'_2,o']} & & \\ D & \xleftarrow{\;g\;} & C' & & \end{array}$$

Because monos are preserved under pushouts, the composite $k \circ (h + id) \circ \iota_2 = k_2 = o'$ is indeed a mono, where we write $k = [k_1, k_2]$. ◻

10.4 Structural properties of the comparison maps

As we explained above, the comparisons maps

$$[\![\pi]\!]_{Sep} \xrightarrow{\;\;\mathcal{S}\;\;} [\![C]\!]_S \xrightarrow{\;\;\mathcal{L}\;\;} [\![C]\!]_L$$

of the Theorem satisfy a strong structural property which we call **strictness**.

Definition 10.2. Suppose given two cobordisms $C \in \mathbf{Cob}(\pm)$ and $B \in \mathbf{Cob}(\pm')$, we say that a map of cobordisms $(f, f_\pm) : A \to B$ is **strict** when the two squares in the following diagram are pullbacks.

$$\begin{array}{ccccc} I & \xrightarrow{\;in\;} & C & \xleftarrow{\;out\;} & O \\ {\scriptstyle f_I}\downarrow & & \downarrow{\scriptstyle f_C} & & \downarrow{\scriptstyle f_O} \\ I' & \xrightarrow[\;in'\;]{} & C' & \xleftarrow[\;out'\;]{} & O' \end{array}$$

Again, we prove this fact by induction on the structure of the proof and of the programs. The fact that parallel product preserves strictness follows from the fact that it is defined using pullbacks, and from the usual pullback pasting lemma; see Lemma 8 in the Appendix for a detailed proof.

The case of the sequential composition is quite interesting, as it follows from the fact that the ambient category is adhesive. The diagrammatic situation is the following:

$$\begin{array}{ccccccccc} I & & & & M & & & & O \\ \downarrow & \searrow & & \swarrow & \downarrow & \searrow & & \swarrow & \downarrow \\ I' & & C_1 & & M' & & C_2 & & O' \\ & \searrow & \downarrow & \searrow\;\swarrow & & \searrow\;\swarrow & \downarrow & \swarrow & \\ & & C'_1 & & C_1;C_2 & & C'_2 & & \\ & & & \searrow & \downarrow & \swarrow & & & \\ & & & & C'_1;C'_2 & & & & \end{array} \tag{38}$$

Focusing on the commutative cube, what we need to show is that the two front faces are pullbacks. This follows from the fact that the bottom square is a van Kampen square, since it is a pushout along a monomorphism in an adhesive category. Given that the top face is a pushout as well, the definition of van Kampen square gives us directly that the two front faces are pullbacks if and only if the two back faces are pullbacks, which they are by hypothesis.

10.5 Proof of the Theorem

Equipped with the structural properties of the two sections above, we are ready to prove the theorem itself. As expected for a semantic proof of soundness of a program logic, we proceed by induction on the structure of the proof π of some Hoare triple $\Gamma \vdash \{P\}C\{Q\}$. The Theorem is naturally decomposed into the two following independent statements, which we prove in turn.

Theorem 10.3 (1-soundness). *The comparison map $\mathcal{S} : [\![\pi]\!]_{Sep} \to [\![C]\!]_S$ is a Code 1-fibration.*

Theorem 10.4 (2-soundness). *The comparison map $\mathcal{L} \circ \mathcal{S} : [\![\pi]\!]_{Sep} \to [\![C]\!]_L$ is a 2-fibration.*

We detail the cases of the rules for the parallel product and the sequential product, first of 2-soundness, and then for 1-soundness, for the former is simpler.

Proof of the 2-soundness theorem. To prove that the parallel product preserves 2-fibrations, we use the following fact:

Lemma 4. *If $g \circ f$ is a 2-fibration and g is a monomorphism, then f is a 2-fibration.*

in the following diagram.

$$
\begin{array}{ccccccc}
\wedge & & & \longrightarrow & & & \blacklozenge \\
\downarrow & & & & & & \downarrow \\
S_1 \parallel S_2 & & & \overset{f_1 \parallel f_2}{\dashrightarrow} & & & S_1' \parallel S_2' \\
 & \searrow^{(\lambda_1,\lambda_2)} & & & & \swarrow^{(\lambda_1',\lambda_2')} & \\
\downarrow & & S_1 \times S_2 & \xrightarrow{f_1 \times f_2} & S_1' \times S_2' & & \downarrow \\
 & & \downarrow & & \downarrow & & \\
 & & \pitchfork \times \pitchfork & \xrightarrow{u \times u} & \pitchfork' \times \pitchfork' & & \\
 & \nearrow^{(\lambda_1,\lambda_2)} & & & & \nwarrow^{(\lambda_1',\lambda_2')} & \\
\pitchfork^{\parallel} & & & \xrightarrow{u^{\parallel}} & & & \pitchfork'^{\parallel} \\
\downarrow{\scriptstyle \mathbf{proj}} & & & & & & \downarrow{\scriptstyle \mathbf{proj}'} \\
\pitchfork & & & \xrightarrow{u} & & & \pitchfork'
\end{array}
$$

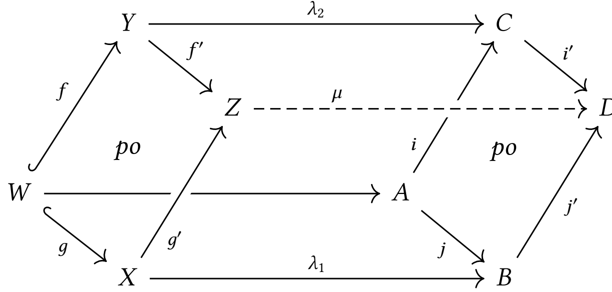
where the map $f_1 \parallel f_2$ is induced by the universal property of the left pullback, and where $(u, u^{\parallel})$ is the span-monoidal functor structure. Indeed, in the case of $\pitchfork = \pitchfork_{\mathrm{Sep}}$, we have the following fact:

Lemma 5. *The map pick : $\pitchfork^{\parallel}_{\mathrm{Sep}} \to \pitchfork_{\mathrm{Sep}} \times \pitchfork_{\mathrm{Sep}}$ is a 2-fibration.*

which means that, by the preservation of fibrations uder pullbacks, the map $(\lambda_1, \lambda_2) : S_1 \parallel S_2 \to S_1 \times S_2$ is one as well. Similarly, the map $(\lambda_1', \lambda_2') : S_1' \parallel S_2' \to S_1' \times S_2'$ is a mono since pick is a mono. We then conclude using Lemma 4.

For the case of sequential composition, we heavily rely on the fact that van Kampen cubes preserve 2-fibrations using the lemma below. The preservation result then follows directly from the fact that the cube in the center of diagram (38) is van Kampen.

Lemma 6. *Consider the following diagram in* **AsynGph***:*



such that:

(1) λ_1 and λ_2 are 2-fibrations,
(2) the cube is van Kampen,
(3) the two back faces are pullbacks.

Then, the asynchronous map μ is a 2-fibration.

Proof. Let us assume there is a path of length 2 in Z which is mapped by μ to the top of a tile in D. We will use the fact that representables are tiny, in particular, tiles are tiny. Assume without loss of generality that the tile in D has a preimage T_C in C. Then the upper path of the tile in D is the target of a path of length 2 in Z and the target of a path of length 2 in C (the upper path of T_C). Since the cube is van Kampen, Y is the pullback of i' along μ, which means that there is a path of length 2 in Y that is mapped to the one in C. We can conclude from there using the fact that λ_2 is a 2-fibration. ◻

Proof of the 1-soundness theorem. Preservation of Code 1-fibrations by sequential composition follows directly from a variant of Lemma 2 where all occurrences of Environment 1-fibrations are replaced with Code 1-fibrations. The proof that the parallel product preserves Code 1-fibrations is a bit more intricate than the case of 2-fibrations. The reason is that the map pick : $\pm_S^{\otimes}[1] \to \pm_S[1] \times \pm[1]$ is not a Code 1-fibration. Instead, we need to rely on the fact that the map $S \longrightarrow \pm_S[1]$ from the support of the cobordism to the template is an Environment 1-fibration and on the fact that the morphism pick above never pairs two Code transitions.

Lemma 7 (Parallel product preserves 1-fibrations). *Given two maps of cobordisms*

$$\alpha_i : C_i \to C_i'$$

for $i = 1, 2$, such that C_1 and C_2, respectively C_1' and C_2', can be sequentially composed, if α_1 and α_2 are 1-fibrations on Code transitions, then the induced morphism

$$\alpha_1 \parallel \alpha_2 : C_1 \parallel C_2 \to C_1' \parallel C_2'$$

is a 1-fibration on Code transitions as well.

Proof. Let us focus on the maps between the supports. We call the supports S_1, S_2, S_1' and S_2', and we call the maps between that are contained in α_1 and α_2:

$$f_1 : S_1 \longrightarrow S_1' \qquad\qquad f_2 : S_2 \longrightarrow S_2'$$

Finally, we write (λ_1, λ_2) both for the map pick in the span-monoidal structure, and for the map it induces by pullback at the level of the supports of the cobordisms.

Recall that the map $f_1 \parallel f_2$ is defined by the following diagram, where $(u, u^{\parallel})$ is the span-monoidal functor structure:

$$
\begin{array}{ccccccc}
\cdot & & & \longrightarrow & & & \cdot \to_C \cdot \\
\downarrow & & & & & & \downarrow \\
S_1 \parallel S_2 & & & \overset{f_1 \parallel f_2}{\dashrightarrow} & & & S_1' \parallel S_2' \\
& \searrow^{(\lambda_1,\lambda_2)} & S_1 \times S_2 & \xrightarrow{f_1 \times f_2} & S_1' \times S_2' & \swarrow^{(\lambda_1',\lambda_2')} & \\
& & \downarrow & & \downarrow & & \\
& & \pm \times \pm & \xrightarrow{u \times u} & \pm' \times \pm' & & \\
\downarrow & \nearrow^{(\lambda_1,\lambda_2)} & & & & \nwarrow^{(\lambda_1',\lambda_2')} & \downarrow \\
\pm^{\parallel} & & & \xrightarrow{u^{\parallel}} & & & \pm'^{\parallel} \\
\downarrow \mathbf{proj} & & & & & & \downarrow \mathbf{proj}' \\
\pm & & & \xrightarrow{u} & & & \pm'
\end{array}
$$

The image of the unique edge of $\cdot \to_C \cdot$ in $\pm^{\parallel}$ is either C_1 or C_2 (recall that the polarities of $\pm_{\parallel}^{\mathrm{pol}}$ are C_1, C_2 and F). Because the situation is symmetric, we can suppose without loss of generality that this polarity is C_1. In that case, the edge $\cdot \to_C \cdot$ is mapped to a Code transition in S_1'. By assumption, f_1 is a 1-fibration on Code transitions, so we can lift this arrow to S. Moreover, because $\lambda_1 : \pm^{\parallel} \to \pm$ is a 1-fibration on Code transitions as well, we can also lift uniquely $\cdot \to_C \cdot$ to $\pm^{\parallel}$. Therefore, we have the following situation:

$$
\begin{array}{ccccccc}
\cdot & & & \longrightarrow & S_1 \times S_2 & & \\
& & S_1 & \overset{\pi_1}{\longleftarrow} & \downarrow \langle f_1, f_2 \rangle & \overset{\pi_2}{\longrightarrow} & S_2 \\
& \cdot \to_C \cdot \overset{(1)}{\dashrightarrow} S_1 & \downarrow \lambda_1 & & \pm[1] \times \pm[1] & & \downarrow \lambda_2 \\
& & \pm[1] & \overset{\pi_1}{\longleftarrow} & & \overset{\pi_2}{\longrightarrow} & \pm[1] \\
\downarrow & & & \overset{\lambda_1}{\nwarrow} & & \overset{\lambda_2}{\nearrow} & \\
\cdot \to_{C_1} \cdot & & & \longrightarrow & \pm^{\parallel}[1] & & \\
& & & \searrow & \cdot \to_F \cdot & \overset{(2)}{\dashrightarrow} S_2 &
\end{array}
$$

The map (1) exists because f_1 is a 1-fibration on Code transitions, as mentioned above, and the map (2) exists because λ_2 is a 1-fibration on Environment moves. Therefore, we can lift the arrow $\cdot \to_{C_1} \cdot$ to $S_1 \times S_2$. We conclude by using the fact that 1-fibrations on Code transitions are stable by pushout, and that $\langle \lambda_1, \lambda_2 \rangle : \pm^{\parallel} \to \pm \times \pm$ is a 1-fibration. $\square$

11 RELATED WORKS

The first proof of soundness for CSL was established by Brookes [4] using a trace-based and stateless semantics. Another proof of soundness was then given a few years later by [25] and Dinsdale-Young et al. [6] using a more direct operational approach. The first proof of soundness based on a truly concurrent semantics of CSL was designed by Hayman and Winskel [11] using an encoding of the Code into Petri nets. The approach Iris [14] takes is to prove the soundness of a modal logic, and then defining Hoare triples in this logic. Our present work follows the true concurrency tradition of interpreting the parallel product as more than a simple interleaving; more specifically, we use a notion of homotopy to talk about independence of instructions, in a way closely related to directed homotopy [9, 26]. There is a well-established line of research on cospans by Bonchi et al. [2], Sassone and Sobocinski [23] for example, who generate LTSs using graph rewriting techniques based on cospans. One main difference with our work is that we use cospans to manipulate and compose our transition systems, instead of deriving them explicitly from rewriting systems.

12 CONCLUSION

One foundational and guiding principle of template game semantics is that one cannot have a direct access to the internal states of a program, because this access is necessarily mediated and regulated by the labels of a specific template $\pitchfork$ of interest. This idea inspired from dependent type theory implies that the basic operations on programs (composition, synchronization, errors, locks, etc.) should be defined by applying cleverly designed *change-of-basis* functors on the labeling templates. We establish in the present paper that, somewhat unexpectedly, the very same categorical yoga based on *pull* and *push* functors, works for concurrent separation logic (CSL) and for differential linear logic (DiLL). This is achieved by designing a notion of *cobordism* $\sigma : A \relbar\joinrel\mapstochar\joinrel\rightarrow B$ based on cospans for CSL which conveniently replaces the notion of *strategy* $\sigma : A \relbar\joinrel\mapstochar\joinrel\rightarrow B$ based on spans for DiLL. One nice outcome is a categorical explanation for the Hoare inequality of concurrency, which is derived here from a lax commutation property between sequential composition (understood as a colimit) and parallel product (understood as a limit). We see this healthy convergence between CSL and DiLL as a strong evidence for the relevance and surprising expressivity of template games. One main challenge for future work is to combine these two lines of research on DiLL and CSL in order to obtain an asynchronous soundness theorem for a higher-order version of CSL, based on a better understanding of the relationship with Iris [14] and FCSL [22].

REFERENCES

[1] Jean Bénabou. 1967. Introduction to bicategories. In *Reports of the Midwest Category Seminar*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–77.

[2] F. Bonchi, B. König, and U. Montanari. 2006. Saturated Semantics for Reactive Systems. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. 69–80.

[3] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *POPL*.

[4] Stephen Brookes. 2004. A semantics for concurrent separation logic. In *CONCUR*.

[5] Brian Day and Ross Street. 1997. Monoidal Bicategories and Hopf Algebroids. *Advances in Mathematics* 129, 1 (1997), 99 – 157.

[6] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *POPL*.

[7] Clovis Eberhart, Tom Hirschowitz, and Alexis Laouar. 2019. Template Games, Simple Games, and Day Convolution. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany.* 16:1–16:19.

[8] Charles Ehresmann. 1963. Catégories structurées. *Annales scientifiques de l'École Normale Supérieure* 80, 4 (1963), 349–426.

[9] Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. 2016. *Directed Algebraic Topology and Concurrency.* Springer.
[10] Richard Garner. 2006. *Polycategories.* Ph.D. Dissertation. University of Cambridge.
[11] Jonathan Hayman and Glynn Winskel. 2008. Independence and Concurrent Separation Logic. *LMCS* (2008).
[12] Tony Hoare. 2013. Unifying Semantics for Concurrent Programming. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky - Essays Dedicated to Samson Abramsky on the Occasion of His 60th Birthday.* 139–149.
[13] P. T. Johnstone. 1975. Adjoint Lifting Theorems for Categories of Algebras. *Bulletin of the London Mathematical Society* 7, 3 (11 1975), 294–297.
[14] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20.
[15] Stephen Lack and Paweł Sobociński. 2004. Adhesive Categories. In *Foundations of Software Science and Computation Structures*, Igor Walukiewicz (Ed.). Springer Berlin Heidelberg, 273–288.
[16] Paul-André Melliès. 2019. Categorical Combinatorics of Scheduling and Synchronization in Game Semantics. *Proc. ACM Program. Lang.* 3, POPL, Article 23 (Jan. 2019), 30 pages.
[17] Paul-André Melliès. 2019. Template games and differential linear logic. In *LICS 2019*.
[18] Paul-André Melliès and Léo Stefanesco. 2017. A Game Semantics for Concurrent Separation Logic. In *MFPS*.
[19] Paul-André Melliès and Léo Stefanesco. 2018. An Asynchronous Soundness Theorem for Concurrent Separation Logic. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*.
[20] Jeffrey Morton. 2006. Double Bicategories and Double Cospans. *Journal of Homotopy and Related Structures* 4 (11 2006).
[21] P.S. Mulry. 2002. Lifting results for categories of algebras. *Theoretical Computer Science* 278, 1 (2002), 257 – 269. Mathematical Foundations of Programming Semantics 1996.
[22] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*.
[23] V. Sassone and P. Sobocinski. 2005. Reactive systems over cospans. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 311–320.
[24] Ross Street. 1983. Enriched Categories and Cohomology. *Quaestiones Mathematicae* 6, 1-3 (1983), 265–283.
[25] Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *ENTCS* 276 (2011).
[26] Rob J. van Glabbeek. 2006. On the expressiveness of higher dimensional automata. *Theoretical Computer Science* 356, 3 (2006).

CONTENTS

Abstract		1
1	Introduction	1
2	The double category $\mathbf{Cob}(\pm)$ of games and cobordisms	11
2.1	Double categories	11
2.2	Polyads	12
2.3	The double category $\mathbf{Cob}(\pm)$ of games and cobordisms	12
3	The stateful and stateless synchronization templates	13
3.1	The category of asynchronous graphs	14
3.2	The stateful and stateless machine models	15
3.2.1	The stateful model $\pm_S^\bullet$	15
3.2.2	The stateless model $\pm_L^\bullet$	16
3.3	The stateful and the stateless internal opcategories $\pm_S$ and $\pm_L$	16
4	The parallel product	17
4.1	Internal functors between internal J-opcategories	17
4.2	Plain internal functors	18
4.3	Acute spans of internal functors	19
4.4	Span-monoidal internal J-opcategories	20
4.5	Illustration: parallel product of codes	21
5	Sequential composition	21
6	The error monad	22
6.1	Liftings	23
6.2	Tensor product	23
7	Separated states	24
7.1	Separated states	24
7.2	The machine model of separated states $\pm_{Sep}^\bullet$	25
7.3	The internal category $\pm_{\mathrm{Sep}}$	26
7.4	Parallel product understood as a span-monoidal structure on $\pm_{\mathrm{Sep}}$	26
8	Change of locks	26
8.1	Hiding	27
8.2	Critical sections	28
9	A uniform interpretation of codes and proofs	28
9.1	Stateful and stateless interpretations of the code	29
9.2	Interpretation of the proofs	30
9.3	Interactive and separated interpretations of the proofs	30
10	The asynchronous soundness theorem revisited	32
10.1	Comparing the three interpretations	32
10.2	The asynchronous soundness theorem	33
10.3	Structural properties of cobordisms	34
10.4	Structural properties of the comparison maps	37
10.5	Proof of the Theorem	38
11	Related works	41
12	Conclusion	41
References		41
Contents		43
A	Other constructions on cobordisms	44
A.1	Conjunction	44

A.2 Disjunction 44
A.3 Loops 44
B Filling systems and the Hoare inequality 44
C Semantics of the instructions 45
C.1 Stateful instructions 45
C.2 Stateless instructions 46
D The full proof system 46
E Proof of the Soundness Theorem, continued 47
E.1 Properties of the interpretation 47
E.1.1 Adhesiveness and van Kampen cubes 47
E.1.2 Proof of the preservation of strictness 48
E.2 Fibrational properties of change of lock operations 48
E.3 Other constructions 50

A OTHER CONSTRUCTIONS ON COBORDISMS

We present three additional constructions, having to do with union, intersection, and loops. We need the internal J-category to have a $\vee$-structure: a map $\vee : ⅄ + ⅄ \longrightarrow ⅄$. Write $\vee$ for the function on the indices, and $(l[0, ij], r[0, ij]) : ⅄[0, i] + ⅄[0, j] \longrightarrow ⅄[0, i \vee j]$ for the maps of the internal functor.

A.1 Conjunction

Conjunction is defined as the pointwise pullback along the maps into $⅄[0, i \vee j]$ that the $\vee$-structure gives us. The action on the interfaces is as follows; the action on the cobordisms is defined similarly by pullback. Given $\lambda : A \longrightarrow ⅄[0, i]$ and $\lambda' : B \longrightarrow ⅄[0, j]$, the $\vee$-structure gives two squares, one from λ to $id_{⅄[0,i\vee j]}$ and another from λ' to $id_{⅄[0,i\vee j]}$. The conjunction of λ and λ' is then the pullback of these two maps.

A.2 Disjunction

Using the same notations as for the conjunction, we define the disjunction of λ and λ' to be the obvious map $A + B \longrightarrow ⅄[0, i \vee j]$.

A.3 Loops

To interpret a loop `while` B do C, we build its infinite unfolding as the least fixpoint of the map:

$$F(X) = [\![\texttt{test}(B)]\!] \mathbin{⨟} [\![C]\!] \mathbin{⨟} X \;\oplus\; [\![\texttt{test}(\neg B)]\!]$$

seen as an endofunctor on the category of arrows of the double category $\mathbf{Cob}(⅄)$ seen as an internal category in $\mathbf{Cat}$. It exists because that category has all colimits of ω-chains, and F preserves such colimits because it is itself defined using colimits.

B FILLING SYSTEMS AND THE HOARE INEQUALITY

In this section, we suppose that our template and internal opcategory $⅄$ is either the stateful template $⅄_S$ or the stateless template $⅄_L$. Given two games $\lambda : A \longrightarrow ⅄[0]$ and $\mu : B \longrightarrow ⅄[0]$ formulated as asynchronous morphisms, let us describe $\mathsf{fill}(\lambda, \mu)$. Every node x of A comes with a state $\mathrm{in}_{\lambda(x)} \in ⅄[1]$ which we call the underlying state of x. Similarly, every node y of B comes with an underlying state $\mathrm{in}_{\lambda(y)} \in ⅄[1]$. The pullback $A \times_{⅄[1]} B$ contains all the pairs $(a, b) \in A \times B$ which share the same underlying state. Hence, when we perform the pushout, we identify all

such nodes a and b; in particular, in the case where there is another node a' of A with the same underlying states, the pullback will also contain (a', b), and the nodes a and a' will be identified in the pushout. In summary, the support S of the filling $\mathsf{fill}(\lambda, \mu)$ is made of three kinds of nodes:

(1) the nodes x of A such that no node of B has the same underlying state;
(2) the nodes y of B such that no node of A has the same underlying state;
(3) the states $s \in \pm[1]$ such that there exists nodes in A and nodes in B whose underlying states is s ; we use the notation $[s]$ in order to denote these specific nodes.

Let us prove now that the filling system defined at the end of Section 5 satisfies the property that there exists a map:

$$\mathsf{fill}(\lambda \parallel \lambda', \mu \parallel \mu') \longrightarrow \mathsf{fill}(\lambda, \mu) \parallel \mathsf{fill}(\lambda', \mu')$$

The map is constructed in the following way. Consider a node of the support of $\mathsf{fill}(\lambda \parallel \lambda', \mu \parallel \mu')$. As we have just mentioned, there are three possibilities:

(1) In the first case, the element is a node of $A \parallel A'$, and thus a pair $(x, x') \in A \times A'$ consisting of two elements $x \in A$ and $x' \in A'$ with the same underlying state s. Recall indeed that the asynchronous morphism $\mathrm{pince}[1] : \pm^{\parallel}[1] \longrightarrow \pm[1]$ is *injective on states*. Since we are in the first case, there exists no node of $B \parallel B'$ with underlying state s. This means that either:
    (a) neither B nor B' have nodes whose underlying state is s; in that case the node x is in $\mathsf{fill}(\lambda, \mu)$, and the node x' is in $\mathsf{fill}(\lambda', \mu')$, and we map (x, x') to (x, x').
    (b) B has a node whose underlying state is s, but not B'; in the same way essentially as in the preceding case, we can map (x, x') to $([s], x')$,
    (c) B' has a node whose underlying state is s, but not B; this case is symmetric to the previous one.
(2) the second case is symmetric to the previous one.
(3) last case: the node of the support of $\mathsf{fill}(\lambda \parallel \lambda', \mu \parallel \mu')$ is of the form $[s]$. In that case, there are nodes in each of the four graphs A, A', B, B' whose underlying states is s. We are thus allowed to map $[s]$ to the pair $([s], [s])$.

C SEMANTICS OF THE INSTRUCTIONS

C.1 Stateful instructions

The *machine instructions* $m \in \mathbf{Instr}$ which label the machine steps are of the following form:

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

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

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

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

The inclusion $\mathbf{Loc} \subseteq \mathbf{Val}$ means that an expression E may also denote a location. In that case, $[E]$ refers to the value stored at location E in the heap. The instruction $x := \mathtt{alloc}(E, \ell)$ allocates some

memory space on the heap at address $\ell \in \mathbf{Loc}$, initializes it with the value of the expression E, and assigns the address ℓ to the variable $x \in \mathbf{Var}$ if ℓ was free, otherwise there is no transition. `dispose`(E) deallocates the location denoted by E when it is allocated, and returns $\lightning$ otherwise. Finally, the instruction nop (for no-operation) does not alter the state.

C.2 Stateless instructions

The machine instructions of the stateless semantics as given bellow, with their action on the lock state.

$$L \xrightarrow{P(r)} L \uplus \{r\} \qquad L \xrightarrow{\texttt{alloc}(\ell)} L \qquad L \xrightarrow{\tau} L$$
$$L \uplus \{r\} \xrightarrow{V(r)} L \qquad L \xrightarrow{\texttt{dispose}(\ell)} L \qquad L \xrightarrow{m} \lightning$$

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

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

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

D THE FULL PROOF SYSTEM

The syntax and the semantics of the formulas of Concurrent Separation Logic is the same as in Separation Logic. The grammar of formulas is:

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

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

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

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

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

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

$$\frac{}{\Gamma \vdash \{(\mathrm{own}_{\top}(x) * P) \wedge E = v\} x := E \{(\mathrm{own}_{\top}(x) * P) \wedge x = v\}} \textsc{Aff}$$

$$\frac{}{\Gamma \vdash \{E \mapsto -\} [E] := E' \{E \mapsto E'\}} \textsc{Store}$$

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

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

$$\frac{\Gamma \vdash \{P\} C_1 \{Q\} \quad \Gamma \vdash \{Q\} C_2 \{R\}}{\Gamma \vdash \{P\} C_1 ; C_2 \{R\}} \textsc{Seq} \qquad \frac{\Gamma \vdash \{P_1\} C \{Q_1\} \quad \Gamma \vdash \{P_2\} C \{Q_2\}}{\Gamma \vdash \{P_1 \vee P_2\} C \{Q_1 \vee Q_2\}} \textsc{Disj}$$

$$\frac{\Gamma, r : J \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * J\} \texttt{resource } r \texttt{ do } C \{Q * J\}} \textsc{Res} \qquad \frac{P \Rightarrow \mathrm{def}(B) \quad \Gamma \vdash \{(P * J) \wedge B\} C \{Q * J\}}{\Gamma, r : J \vdash \{P\} \texttt{with } r \texttt{ do } C \{Q\}} \textsc{With}$$

$$\frac{\Gamma \vdash \{P_1\} C_1 \{Q_1\} \quad \Gamma \vdash \{P_2\} C_2 \{Q_2\}}{\Gamma \vdash \{P_1 * P_2\} C_1 \parallel C_2 \{Q_1 * Q_2\}} \textsc{Par} \qquad \frac{\Gamma \vdash \{P\} C \{Q\}}{\Gamma \vdash \{P * R\} C \{Q * R\}} \textsc{Frame}$$

E PROOF OF THE SOUNDNESS THEOREM, CONTINUED

E.1 Properties of the interpretation

E.1.1 Adhesiveness and van Kampen cubes. In order to reason about the sequential composition, we use the fact that our ambient category **AsynGph** is adhesive (as is every topos, hence every presheaf category). We recall the notion from [15].

Definition E.1 (Adhesive category). A category $\mathbb{S}$ is adhesive if (1) it has pushouts along monos, (2) it has all pullbacks, and (3) all pushouts along monomorphisms are van Kampen squares.

A **van Kampen square** is a commutative square

$$\begin{array}{ccc} A & \longrightarrow & B \\ \downarrow & & \downarrow \\ C & \longrightarrow & D \end{array}$$

such that for every commutative cube

$$\begin{tikzcd} Y \arrow[rr] \arrow[dd] & & Z \arrow[dd] & \\ & W \arrow[ul] \arrow[rr] \arrow[dd] & & X \arrow[ul] \arrow[dd] \\ C \arrow[rr] & & D & \\ & A \arrow[ul] \arrow[rr] & & B \arrow[ul] \end{tikzcd}$$

whose bottom face is the square above, and whose two back faces are pullbacks, the top face is a pushout if and only if the two front faces are pullbacks. We also call such a commutative cube a **van Kampen cube**.

E.1.2 *Proof of the preservation of strictness.*

Lemma 8. *The parallel product preserves strictness.*

Proof. First, we remark that, in the definition of the parallel product, the square

$$\begin{array}{ccc} I_1 \parallel I_2 & \longrightarrow & C_1 \parallel C_2 \\ \downarrow & & \downarrow \\ I_1 \times I_2 & \longrightarrow & C_1 \times C_2 \end{array}$$

is a pullback square. The reason is that in the following cube, the bottom, right and left faces are pullback squares:

$$\begin{array}{ccccccc} I_1 \times I_2 & & \longrightarrow & & C_1 \times C_2 & & \\ & \nwarrow & & & \downarrow & \nwarrow & \\ \downarrow & & I_1 \parallel I_2 & \longrightarrow & & & C_1 \parallel C_2 \\ & & \downarrow & & & & \downarrow \\ \pitchfork[0] \times \pitchfork[0] & & \longrightarrow & & \pitchfork[1] \times \pitchfork[1] & & \\ & \nwarrow & \downarrow & & & \nwarrow & \downarrow \\ & & \pitchfork^{\parallel}[0] & \longrightarrow & & & \pitchfork^{\parallel}[1] \end{array}$$

The latteral faces are defined to be pullbacks, and the bottom one is a pullback for the three templates that we use. Now, we consider the following cube:

$$\begin{array}{ccccccc} I_1 \times I_2 & & \longrightarrow & & C_1 \times C_2 & & \\ & \nwarrow & & & \downarrow & \nwarrow & \\ \downarrow & & I_1 \parallel I_2 & \longrightarrow & & & C_1 \parallel C_2 \\ & & \downarrow & & & & \downarrow \\ I_1' \times I_2' & & \longrightarrow & & C_1' \times C_2' & & \\ & \nwarrow & \downarrow & & & \nwarrow & \downarrow \\ & & I_1' \parallel I_2' & \longrightarrow & & & C_1' \parallel C_2' \end{array}$$

According to the remark above, the top and bottom faces are pullbacks, and by hypothesis the back face is a pullback. This concludes the proof. □

E.2 Fibrational properties of change of lock operations

Recall that change of locks are a push-then-pull operation, where pulling is achieved by a pullback and pushing by postcomposition. As fibrations, epis and monos are preserved under pullbacks in our ambient category, pulling preserves all the structural properties of our cobordisms. Hence, the preservation of the properties of the cobordisms under change of locks will be determined by the properties of the asynchronous we push along.

Lemma 9. *Given an acute span of J-opcategories*

$$\pitchfork_1 \xleftarrow[F]{} \pitchfork_2 \xrightarrow{G} \pitchfork_3$$

and a cobordism C in $\mathbf{Cob}(\pitchfork_1)$ which satisfies the structural properties of section 10.3:

- *if every map $g[i] : \pm_2[i] \to \pm_3[f(i)]$ is an epi, then condition (1) is preserved,*
- *if every map $g[ij] : \pm_2[ij] \to \pm_3[f(ij)]$ is an Environment 1-fibration, then condition (2) is preserved.*

In the case of critical sections, we only use the third condition above, as the other condition is recovered afterward through the sequential compositions with $[\![P(r)]\!]$ and $[\![V(r)]\!]$ which determine the input and the output states of $[\![\texttt{with}\ r\ \texttt{do}\ C]\!]$. In the case of the semantics of proofs, this corresponds to the fact that initial states of a critical section must be states where the lock is held by the code. The case for hiding is more straightforward as the components of hide_C are epimorphisms and Environment 1-fibrations. Thus, we have established that change of locks preserve the structural properties of cobordisms. We now prove that they also preserve the structural properties of maps between cobordisms, using the following lemma:

Lemma 10. *Given a pair of plain internal functors of J-opcategories*

$$F : \pm_2 \to \pm_1 \qquad \textit{and} \qquad F' : \pm_2' \to \pm_1'$$

and two internal functors of J-opcategories

$$G_1 : \pm_1 \to \pm_1' \qquad \textit{and} \qquad G_2 : \pm_2 \to \pm_2'$$

such that the obvious square that they compose commutes, and given two cobordisms C and C' in $\mathbf{Cob}(\pm_1)$ and $\mathbf{Cob}(\pm_2)$ respectively, together with a comparison map $\mathcal{L}$ between them that sits over G_1, then operation of pulling along F and F' gives rise to two cobordisms in $\mathbf{Cob}(\pm_1')$ and $\mathbf{Cob}(\pm_2')$ respectively, together with a comparison map $\mathcal{L}'$ which sits over the internal functor G_2. Assuming $\mathcal{L}$ is strict, the induced map $\mathcal{L}'$ is strict as well providing that the following squares are pullbacks:

$$\begin{array}{ccc} \pm_2[i] & \longrightarrow & \pm_2[ij] \\ {\scriptstyle F}\downarrow & & \downarrow{\scriptstyle F} \\ \pm_1[i] & \longrightarrow & \pm_1[ij] \end{array} \qquad\qquad \begin{array}{ccc} \pm_2'[i] & \longrightarrow & \pm_2'[ij] \\ {\scriptstyle F'}\downarrow & & \downarrow{\scriptstyle F'} \\ \pm_1'[i] & \longrightarrow & \pm_1'[ij]. \end{array}$$

Proof. We use the pasting lemma for pullbacks in the cube on the left and its counterpart with primes, and then again in the right cube.

$$\begin{array}{ccccccc} I & & \longrightarrow & & C & & \\ & \nwarrow & & & & \nwarrow & \\ \downarrow & & I_1 & \longrightarrow & \downarrow & & C_1 \\ \pm_1[i] & & \longrightarrow & & \pm_1[ij] & & \downarrow \\ & \nwarrow & \downarrow & & & \nwarrow & \\ & & \pm_2[i] & \longrightarrow & & & \pm_2[ij] \end{array} \qquad \begin{array}{ccccccc} I & & \longrightarrow & & C & & \\ & \nwarrow & & & & \nwarrow & \\ \downarrow & & I_1 & \longrightarrow & \downarrow & & C_1 \\ I' & & \longrightarrow & & C' & & \downarrow \\ & \nwarrow & \downarrow & & & \nwarrow & \\ & & I_1' & \longrightarrow & & & C_1' \end{array}$$

□

The last fibrational property we would like the change of lock construction to preserve is the fact that the comparison morphism itself being a fibration of some kind. The following lemma will handle each of our uses of the construction.

Lemma 11. *Using the same assumptions and notations as the the preceding lemma, if the components*

$$F[ij] : \pm_2[ij] \to \pm_1[ij]$$

are fibrations defined by a right lifting property wrt $S_1 \to S_2$ and if, for every pairs of maps l and r, if the following diagram commutes:

$$S_2 \underset{r}{\overset{l}{\rightrightarrows}} \pitchfork'_1[1] \longrightarrow \pitchfork'_2[1]$$

then $l = r$, then the change of locks construction preserves that kind of fibration.

Proof. The crux of the proof is the rely on the fact that given that the following square is a pullback

$$\begin{array}{ccc} C & \longleftarrow & C_1 \\ \downarrow & & \downarrow \\ \pitchfork_1[1] & \longleftarrow & \pitchfork_2[1] \end{array}$$

the map $C_1 \to C$ is a fibration as well using the hypothesis on F. Then, one concludes by a diagram chasing and the second hypothesis. □

E.3 Other constructions

It is easy to check that the *coproduct of cobordisms* preserves all the properties of the cobordisms and of comparison maps between that we consider. From this and from the properties established about sequential composition, we deduce that it is also the case for *conditionals*, as they are defined as a combination of these constructions.

For the case of loops, this follows from the fact that loops are defined as am unfolding of sequential compositions and of sums.