Université
de Paris

Université de Paris

ED386 Sciences Mathématiques de Paris Centre — Laboratoire IRIF
Thèse de doctorat, spécialité informatique, sous la direction de Paul-André Melliès

Asynchronous and Relational Soundness Theorems for Concurrent Separation Logic

Léo Stefanesco

Jury

Adrien Guatto	MdC, Université de Paris	Examinateur
Chung-Kil Hur	Assoc Prof, Seoul National University	Rapporteur
Damiano Mazza	DR, CNRS	Rapporteur
Daniela Petrisan	MdC, Université de Paris	Examinatrice
François Pottier	DR, INRIA	Examinateur
Viktor Vafeiadis	Faculty, MPI-SWS	Examinateur
Nobuko Yoshida	Prof, Imperial College	Examinatrice
Paul-André Melliès	DR, CNRS	Directeur
Lars Birkedal	Prof, Aarhus University	Invité

Présentée et soutenue publiquement le 12 novembre 2021.

Abstract — Asynchronous and Relational Soundness Theorems for Concurrent Separation Logic

The subject of this thesis is concurrent separation logic, a program logic for concurrent shared-memory languages. The relation between the proof of a program in a separation logic and the semantics of this program is expressed by the soundness theorem of this logic. This thesis introduces two soundness theorems. The first, the asynchronous soundness theorem, expresses the absence of data race in well specified programs in the language of template games in asynchronous graphs. The second part of this thesis extends the Iris concurrent separation logic with a relational soundness theorem which allows to establish simulations between a concrete program and an abstract model expressed as a state transition system. An application of this theorem is the proof of termination of concurrent programs under the assumption of a fair scheduler.

Keywords: concurrent separation logic, program logics, semantics, Iris, Coq

Résumé — Théorèmes de correction asynchrone et relationnelle de la logique de séparation concurrente

L'objet de cette thèse est la logique de séparation asynchrone, une logique de programme pour les langages de programmation concurrents et à mémoire partagée. Le lien entre une preuve d'un programme dans une logique de séparation concurrente et la sémantique de ce programme est exprimée par le théorème de correction de cette logique. Cette thèse introduit deux théorèmes de corrections. Le premier, le théorème de correction asynchrone, exprime dans le langage des jeux de gabarits sur des graphes asynchrones l'absence de courses des programmes bien spécifiés. L'autre étend la logique de séparation concurrente Iris avec un théorème de correction relationnel qui permet d'établir des simulations entre un programme concurrent concret et un modèle abstrait, formalisé comme un système de transition. Une application de ce théorème est la preuve de terminaison de programmes concurrent sous l'hypothèse d'un ordonnanceur équitable.

Mot-clés: logique de séparation concurrente, logique de prorgramme, sémantique, Iris, Coq

Contents

Résumé en francais

Les logiques de programme, dont la logique de séparation concurrente qui est l'objet de cette thèse, sont des *logiques axiomatiques*, en ce que les théorèmes qu'on prouve dans ces logiques ne sont que des moyens de prouver des propriétés *dans la logique ambiante* sur des programmes. Les propriétés des programmes sont déduites de la prouvabilité de certains théorèmes par le truchement des *théorèmes de correction* —aussi appelés théorèmes d'adéquation— de la logique. Cette thèse s'intéresse principalement à ces théorèmes de correction, pour deux variantes de la logique de séparation concurrente.

Après avoir introduit la logique de séparation concurrente, ce résumé décrit les deux parties largement indépendantes de ce manuscrit qui présentent chacun un théorème de correction : un théorème asynchrone pour la première partie, et un théorème relationnel pour la seconde.

La logique de séparation concurrente

La première logique de programme moderne, dans le sens où les logiques de programme contemporaines utilisent le même formalisme, est la logique de Hoare (1969), qui fut introduite par ce dernier dans son article fondateur *An axiomatic basis for computer programming*. Les prédicats principaux de la logique sont les *triplet de Hoare*[1]

$$\{P\}\,C\,\{Q\}$$

où C est un programme écrit dans un langage de programmation impératif séquentiel et P et Q sont des prédicats sur l'état de la mémoire du programme, qu'on appelle respectivement la *pré-condition* et la *post-condition* du triplet de Hoare. Intuitivement, la signification du triplet de Hoare ci-dessus est que, si l'état initial de la mémoire avant que le programme C ne s'exécute satisfait la pré-condition P, alors d'une part le programme de recontrera pas d'erreur qui stoppe son exécution, et d'autre part, si le programme termine, alors l'état final de la mémoire satisfait nécessairement la post-condition Q.

[1] L'article original utilisait la notation duale $P\ \{C\}\ Q$ pour ce triplet, mais nous utilisons la notation devenue standard par la suite.

Les triplet de Hoare sont prouvés *dans la logique de Hoare* à l'aide de règles d'inférence, avec lesquelles sont construits des *arbres de dérivations*, aussi appelés *preuves*. Par exemple, la règle d'inférence qui correspond à la composition séquentielle de deux programmes est la suivante :

$$\frac{\{P\}\, C_1\, \{Q\} \quad \{Q\}\, C_2\, \{R\}}{\{P\}\, C_1; C_2\, \{Q\}}$$

Comme nous l'avons expliqué précédemment, le lien entre l'existence d'un tel arbre de dérivation dont la conclusion est un triplet de Hoare $\{P\}\, C\, \{Q\}$ et la sémantique du programme C est exprimé par le théorème de correction de la logique de programme. Celui de la logique de Hoare (faible, car elle ne garantie pas la terminaison des programmes) est le suivant :

Theorème 1. *Soit un triplet de Hoare $\{P\}\, C\, \{Q\}$ qui soit prouvable à l'aide des règles d'inférence de la logique de hoare, et soit $\mathfrak{s}$ un état de la mémoire qui satisfait la précondition P. Alors C exécuté à partir de l'état $\mathfrak{s}$ ne rencontre pas d'erreur, et pour tout état mémoire $\mathfrak{s}'$ tel que C exécuté à partir de $\mathfrak{s}$ termine en $\mathfrak{s}'$, cet état $\mathfrak{s}'$ satisfait la postcondition Q.*

Bien sûr, pour que le théorème ci-dessus soit complètement formel, il faut donner une sémantique formelle au langage de programmation, ce qui est aisé dans le cas d'un langage aussi simple que celui considéré par Hoare (1969).

La logique de Hoare a connu de nombreux développements ultérieurs qui l'ont étendu dans différentes directions telles que l'automatisation des preuves, l'augmentation de l'expressivité de la logique ou la gestion de langages de programmation avec des fonctionnalités plus avancées telles que la récursion.

L'une des limites de la logique de Hoare est qu'elle n'est pas adaptée à la gestion des structures de données contenant des pointeurs. En effet, il est difficile d'écrire une formule qui décrit, par exemple, un arbre binaire de recherche *sans partage*.

C'est pour résoudre ces problèmes que la *logique de séparation* fut inventée au tournant des années 2000. La différence fondamentale entre la logique de séparation et la logique de Hoare est que ses prédicates P, Q ne sont plus des formules de la logique du premier ordre avec des formules atomiques décrivant la mémoire, mais une logique sous-structurelle qui étent la logique du premier ordre avec une nouvelle forme de conjonction, la *conjonction séparante* $P * Q$. Un état mémoire $\mathfrak{s}$ satisfait ce prédicat s'il est possible de découper cet état en deux parties disjointe, l'une satisfaisant P et l'autre satisfaisant Q.

Quelques années plus tard, O'Hearn découvrit que la logique de séparation était aussi adaptée à résoudre une autre limitation de la logique de Hoare, les programmes concurrents, et introduit la *logique de séparation concurrente*. En effet, les extensions de la logique de Hoare aux programmes concurrents, telles que la logique d'Owicki-Gries ou les logiques de Rely-Guarantee, doivent explicitement poser les hypothèses que l'environnement du programme qu'on prouve doit respecter.

Au contraire, la localité des prédicats de la logique de séparation permettent de mettre en place une discipline de *propriété* (en anglais : *ownership*) au niveau logique : si un prédicat mentionne une certaine ressource —par exemple de la mémoire— alors le programme peut utiliser cette ressource sans risque d'interférence de la part de l'environnement. En d'autres termes, la seule hypothèse sur l'environnement qui est faite est qu'il a aussi été prouvé avec la logique de séparation concurrente.

Un théorème de correction asynchrone pour la logique de séparation concurrente

La première partie de ce manuscrit étudie une version de la logique de séparation concurrente proche de la version originale proposée par O'Hearn. Le langage considéré étend celui considéré par Hoare avec une opération de composition parallèle $C_1 \parallel C_2$, ainsi que des verrous qui sont alloués statiquement avec la construction $\texttt{resource}\,r\,\texttt{do}\,C$ et qui sont utilisés avec la construction $\texttt{with}\ r\ \texttt{do}\ C$. Ces constructions sont reflétées dans la forme des triplets de la logique de séparation concurrente : ils sont de la forme

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

où Γ est un contexte associant à chaque verrou disponible un invariant, qui est simplement un prédicat de la logique, du même type que les pré-conditions et que les post-conditions.

Dans un cadre concurrent, le théorème de correction de la logique doit aussi garantir qu'un programme qui a été prouvé ne fera pas d'*accès concurrent* (*data race* en anglais). Un accès concurrent est la situation où un deux threads d'un programme accèdent en même temps à la même case mémoire, et qu'au moins l'un de ces accès est une écriture.

Les deux parties du théorème de correction —le programme ne crash pas, le programme de fait pas d'accès concurrent— sont de nature largement différentes. En effet, la première propriété parle de chaque instruction exécutée par le programme, alors que la seconde parle des *paires* d'instructions exécutées par le programme. Dans notre formalisme, cette première condition est exprimée comme une propriété 1-dimensionnelle de la sémantique, alors que la seconde condition correspond à une propriété 2-dimensionnelle.

La grande majorité des preuves de correction de la logique de séparation concurrente, au contraire de notre approche, ramènent la seconde condition à la première en instrumentant la sémantique du langage de programmation pour provoquer un crash en cas d'accès concurrent.

Notre approche générale est la suivante. Étant donné un arbre de dérivation π d'un triplet de la logique de séparation concurrente

$$\begin{array}{c} \vdots\, \pi \\ \Gamma \vdash \{P\}\, C\, \{Q\} \end{array}$$

on interprète la preuve ainsi que le programme (de deux manières différentes) dans le même domaine sémantique de telle manière à ce ces trois intepretations soient liées de manière canoniques par des morphismes correspondant à des simulations :

$$\llbracket \pi \rrbracket_{\mathrm{Sep}} \xrightarrow{\mathcal{S}} \llbracket C \rrbracket_{\mathrm{S}} \xrightarrow{\mathcal{L}} \llbracket C \rrbracket_{\mathrm{L}}.$$

Le domaine sémantique est basé sur la notion de *graphe asynchrone*, une notion de graphe dotés d'informations 2-dimensionnelles, les *tuiles* qui témoignent du fait que deux opérations sont "indépendantes".

Le théorème de correction peut alors s'exprimer à l'aide de notions de 1-fibration et de 2-fibration qui expriment que les arcs et les tuiles de la sémantique du programme C peuvent être relevées en des arcs et des tuiles de la sémantique du la preuve π, qui vivent dans un univers où, par construction, les erreurs et les accès concurrents sont impossibles.

Theorème 2 (Théorème de correction)**.** *Étant donné une preuve π d'un triplet $\Gamma \vdash \{P\}\, C\, \{Q\}$, et avec les notations ci-dessus :*

1. *$\mathcal{S}$ est une 1-fibration du Code ;*
2. *$\mathcal{L} \circ \mathcal{S}$ est une 2-fibration.*

Un point important de cette partie est la méthodologie suivie pour définir les sémantiques des programmes et des preuves de la logique. Nous avons utilisés une version dualisée des *jeux de gabarits* de Melliès, basée sur des cospans au lieu de spans comme la version usuelle. L'idée la plus importante est que les interpretations vivents dans des doubles catégories induites par des *gabarits* (dans notre cas, des opcatégories internes à la catégorie des graphes asynchrones), et toutes les *opérations* sur ces interprétations sont induites par des *structures* sur ces gabarits.

Un théorème de correction relationnelle pour Iris

La seconde partie de cette thèse propose une nouvelle variante de la logique de séparation concurrente qui permet de construire des simulations entre un programme et un modèle abstrait qui peut soit être vu comme une spécification, soit comme une abstraction qui permet de prouver certaines propriétés du programme.

Cette logique est basée sur Iris, une logique de séparation concurrente moderne qui a plusieurs propriétés importantes pour ce travail. D'abord, contrairement aux logique dont nous avons parlé jusqu'à présent, il n'y a pas de stratification entre les prédicats P, Q qui apparaissent en pré-condition par exemple, et les triplets de Hoare. Au contraire, les prédicats logiques de Iris sont suffisamment expressifs pour pouvoir définir les triplets de Hoare. Cela signifie que, pour définir une nouvelle logique de programme, il suffit de définir une nouvelle notion de triplet dans la "logique de base" de Iris, et c'est ce que nous avons fait ici.

Une autre différence importante est qu'Iris est une logique modale avec un opérateur de *point fixe gardé*. En particulier, Iris est doté d'une modalité $\triangleright$ ("later") et son modèle sémantique est basé sur le comptage de pas (*step-indexing*). Cela signifie en particulier que la logique n'est capable que d'exprimer des propriétés de sureté, par opposition à des propriétés de vivacité telles que la terminaison.

Concrètement, nous proposons une logique de programme paramétrisée par un un certain modèle et qui admet une unique nouvelle règle de raisonnement par rapport à la logique de programme standard d'Iris qui permet de "faire un pas" dans le modèle. Le théorème de correction de la logique permet de prouver qu'une certaine relation ξ est contient une simulation entre le programme et le modèle abstrait si l'on a prouvé un certain triplet pour le programme en question.

Une application directe de ce théorème est de montrer qu'un programme distribué implémente l'algorithme de consensus distribué Paxos de Lamport, au sens où le programme et l'algorithme envoient les même messages, modulo encodage. Un autre exemple que nous avons mis en œuvre est la preuve qu'un algorithme distribué converge nécessairement (*eventually consistent*), sous certaines hypothèses de vivacité du réseau et du programme.

Enfin, la dernière application de ce théorème de correction relationnelle est une logique permettant de prouver qu'un programme concurrent termine si l'ordonnanceur est équitable (*fair scheduler*). La limitation due au comptage de pas ne pose pas de problème car la technique utilisée est de construire une relation entre le programme et un modèle abstrait qui admet un ordre bien fondé satisfaisant quelques propriétés de décroissance. La relation préserve l'équité du programme vers le modèle, et la terminaison du modèle vers le programme, ce qui permet de déduire la terminaison des traces équitables du programme de la terminaison des traces équitables du modèle abstrait, qui découle elle-même de l'existence d'un ordre bien fondé sur ses états.

Introduction

Proving the correctness of programs has been the object of scientific interest well before the use of computer software became widespread and the social need for proving their correctness arose, especially in critical contexts, such as cyber-physical systems where computer programs control physical actuators. Formally, the objective is to prove properties about the semantics of some program: most typically that it does not crash and that if it terminates, its final state satisfies a certain property which expresses that it did what was expected.

0.1 Program logics

The first paper on this topic is the now famous paper *Checking a large routine* by Turing (1949). Most current program logics are instances of what is now called *Hoare logic*, which was first developed by Hoare (1969), building on previous works by Floyd (1967). In every case, the method follows the structure of the program, be it in the form of a flowchart for Turing and Floyd, or of its syntax tree for Hoare.

Logics should provide reasoning rules which are as close as possible to the methods programmers use to reason about the correctness of their programs, such as using loop invariants to reason about the correctness of loops, and variants to prove their termination.

Related to the idea that proofs of programs carried out in a program logic should follow the structure of the program, these proofs should be *modular*: proofs of correctness of several part of a program should be carried out in isolation, and then combined into a proof of the whole program, mirroring the modularity mechanisms of the programming language.

0.1.1 Why use program logics?

Some researchers, such as L. Lamport (1994) have argued that instead of using specialized logics, one should prove programs using "usual" mathematics, or at least in a setting as closed to it as possible. For example, the logic of his TLA+ tool is a set theory augmented with temporal modalities from LTL to express properties of execution steps of programs.

However, it seems to us that TLA+'s purpose is not to prove programs, instead it is to prove algorithms, or something intermediate between an algorithm and a program implementing it.

Proving programs and proving algorithms are widely different tasks: programming languages have intricate semantics, and proving that a piece of code is safe can be a subtle task, even if it corresponds to a simple action in the algorithm the program implements. For example, to prove that dereferencing a raw pointer in Rust is safe, which is an unsafe operation in general, the programmer needs to consider which other part of the program is using this pointer, whether the target of the pointer is valid, and what are the other pointers or references which may alias with it. Logics such as Rustbelt (Jung, Jourdan, et al., 2017) provide facilities to carry such arguments in a specialized logic.

Nowadays, realistic languages are formalized using an operational semantics (for example CompCert for C (Leroy, 2009), or JSCert for Javascript (Bodin et al., 2014)), because state transition systems are simple first order objects which are easy to define and manipulate, especially in a proof assistant. Operational semantics, however, are not well suited in practice to reason directly about programs, even programs of moderate size, because the number of cases to consider quickly explodes. This is specially true when, as it is common, the semantics leaves evaluation orders unspecified, leading to non-determinism in the semantics. This is even more true for concurrent programs, where one needs to consider all interleavings. In a way, the logic is required to recover the compositionality which operational semantics lack compared to denotational semantics.

Finally, most programs are composed of many algorithms: for instance, a simple concurrent program would contain a lock implementing the ticket-lock algorithm (Mellor-Crummey and Scott, 1991), which would prevent concurrent access to a skip-list for example. In the same way that the lock module can be used to protect any data structure, it should be given a specification and a proof of correctness that can be used in many contexts. Conversely, the skip list should be proved and specified without mention of locking or concurrency, as it is a sequential data structure.

One avenue we will explore in Part II is to cleanly separate the proof of the algorithm from the proof of the program module implementing it. For example, we consider an abstract model of the Paxos consensus algorithm taken from a TLA+ specification, and prove in the Iris logic that it simulates a concrete implementation. Then, the consensus property of the program is deduced from the consensus property of the abstract algorithm.

0.1.2 Hoare logic

We begin by describing Hoare logic (Hoare, 1969), which is the core of most program logics for imperative programming languages. Its purpose is to prove the safety and the functional correctness of programs written in a simple While language with the following grammar: It has Boolean expressions:

$$B ::= \mathtt{true} \mid \mathtt{false} \mid B \wedge B' \mid B \vee B' \mid \neg B \mid E = E'$$

arithmetic expressions:

$$E ::= 0 \mid 1 \mid \dots \mid x \mid E + E' \mid E * E'$$

and commands:

$$C ::= x \coloneqq E \mid C; C' \mid \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \mid \mathtt{while}\ B\ \mathtt{do}\ C \mid \mathtt{skip}$$

This language is extremely simple: it does not have functions nor local variables, but it captures the essence of sequential imperative programming. For example, the following program ediv computes the Euclidian division of two integers. More precisely, it computes the division of the number stored in the global variable x by the one in global variable y, and stores the quotient in q and the remainder in r.

```
r := x;
q := 0;
while r ≥ y do
  (r := r - y;
   q := q + 1)
```

Of course, this piece of code only satisfies the informal specification above when x and y are positive.

Hoare logic specifies programs C using *Hoare triples*, which are predicates of the form:

$$\{P\}\, C\, \{Q\}$$

where P and Q are formulas which represent logical predicates over the state of the memory, written in a language which is part of Hoare logic. Intuitively, this states that any execution of the program C which begins with an initial memory state $\mathfrak{s}_0$ which satisfies the *precondition* P, does not crash; moreover, if this execution of the program C terminates, then its final state $\mathfrak{s}_f$ satisfies the *postcondition* Q.

In the version of Hoare logic we consider, P and Q are first-order formulas with atomic propositions $x = v$ asserting that, in the current memory state, the program variable x is well defined and contains the value v. The meaning of such a predicate is given by the satisfiability relation $\mathfrak{s} \vDash P$, where the memory state $\mathfrak{s}$ is a finite partial map from variables to values; it is defined as follows

$$\begin{aligned} \mathfrak{s} \vDash x = v \quad &\Longleftrightarrow \quad \mathfrak{s}(x) = v, \\ \mathfrak{s} \vDash P \wedge Q \quad &\Longleftrightarrow \quad \mathfrak{s} \vDash P \ \wedge \ \mathfrak{s} \vDash Q, \end{aligned}$$

and similarly for every first order connectives.

In this thesis, we consider Hoare logics for *weak correctness*, which does not guarantee that programs terminate for initial states satisfying the precondition P. Such logics —called *strong*— do exist and, for sequential programs, are fairly similar to weak logics.

For instance, the `ediv` program above, which implements Euclidian division, would be specified using the following Hoare triple:

$$\{\mathtt{x} \ \geq \ 0 \wedge \mathtt{y} > 0\} \ \mathtt{ediv} \ \{\mathtt{x} = \mathtt{y} \cdot \mathtt{q} + \mathtt{r} \ \ \wedge \ \ 0 \leq \mathtt{r} < \mathtt{y}\}$$

To be accepted, a Hoare triple such as the one above needs to be the conclusion of a proof tree build using the inference rules of the logic. The usual inference rules for Hoare logic are presented in Figure 0.1. Each syntactic construction of the programming language has an associated rule. In this version of Hoare logic, the only rule which does not correspond to the syntax of the program is the *rule of consequence* Conseq. This rule, which allows strengthening the precondition and weakening the postcondition, is essential: it allows the prover to “massage” the pre and postconditions so that the syntax directed rules can be applied; for instance, the premise (and the conlusion) of the While rule must be of a particular shape to be used. The premises of Conseq refer to the entailment relation on predicates, which is defined as follow:

$$P \Rightarrow Q \quad \Longleftrightarrow \quad \forall \mathfrak{s}, \ \mathfrak{s} \vDash P \text{ implies } \mathfrak{s} \vDash Q,$$

or, equivalently, it holds iff the Hoare logic predicate $P \Rightarrow Q$ is a tautology. Importantly, entailment should be seen as a *semantic* construct, unlike the rest of Hoare logic which is a syntactic system.

Let us focus on the rule for `while`. To use it, the prover needs to find a Hoare logic predicate I which is called the *loop invariant*. The premise states that if, before the execution of an iteration, the invariant I and the guard B of the loop both hold, then the invariant is still true after that iteration.

$$\frac{}{\{P\}\,\mathsf{skip}\,\{P\}}\ \textsc{Skip} \qquad \frac{}{\{Q[x \coloneqq a]\}\ x \coloneqq E\ \{Q\}}\ \textsc{Aff}$$

$$\frac{\{P\}\,C_1\,\{Q\} \quad \{Q\}\,C_2\,\{R\}}{\{P\}\,C_1;C_2\,\{R\}}\ \textsc{Seq} \qquad \frac{\{P \land B\}\,C_1\,\{Q\} \quad \{P \land \neg B\}\,C_2\,\{Q\}}{\{P\}\ \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \{Q\}}\ \textsc{If}$$

$$\frac{\{I \land B\}\,C\,\{I\}}{\{I\}\ \mathsf{while}\ B\ \mathsf{do}\ C\ \{I \land \neg B\}}\ \textsc{While} \qquad \frac{P \Rightarrow P' \quad \{P'\}\,C\,\{Q'\} \quad Q' \Rightarrow Q}{\{P\}\,C\,\{Q\}}\ \textsc{Conseq}$$

Figure 0.1: Inference rules for Hoare logic

To see how this works out in practice, we prove that the program `ediv` does satisfy the specification we have given above. Instead of displaying the proof as a tree, because of the amount of redundancy which is contained in Hoare logic proof trees, we annotate the program with predicates enclosed in curly braces, and we denote the use of the Conseq rule with ⇒.

```
{ x ≥ 0 ∧ y > 0 } ⟹
{ x = y·0 + x ∧ x ≥ 0 ∧ y > 0 }
r := x;
{ x = y·0 + r ∧ r ≥ 0 ∧ y > 0 }
q := 0;
{ x = y·q + r ∧ r ≥ 0 ∧ y > 0 }
while r ≥ y do
  (
   { x = y·q + r ∧ r ≥ 0 ∧ y > 0 ∧ r ≥ y } ⟹
   { x = y·(q+1) + (r-y) ∧ r-y ≥ 0 ∧ y > 0 ∧ r-y ≥ 0 }
   r := r - y;
   { x = y·(q+1) + r ∧ r ≥ 0 ∧ y > 0 ∧ r ≥ 0 }
   q := q + 1
   { x = y·q + r ∧ r ≥ 0 ∧ y > 0 ∧ r ≥ 0 }
  )
{ x = y·q + r ∧ r ≥ 0 ∧ y > 0 ∧ r < y } ⟹
{ x = y·q + r ∧ r < y }
```

This example demonstrates that proving the entailment relation used in the consequence rule requires general mathematical reasoning, in this case arithmetic reasoning. This is why considering it as a semantic notion instead of a syntactic one simplifies a lot the definition of the logic.

Weakest precondition

The rule While has a special status in Hoare logic because it is the only rule for which the pre and postconditions in the premise cannot be computed from those in the conclusion. To see this, it is useful to consider another presentation of Hoare logic using *weakest preconditions*: the weakest precondition $\mathsf{wp}\ C\ \{Q\}$ of a program C and a postcondition Q is the weakest —according to the entailment order— precondition such that the Hoare triple $\{\mathsf{wp}\ C\ \{Q\}\}\ C\ \{Q\}$ holds. We can recover Hoare triples from weakest preconditions with:

$$\{P\}\ C\ \{Q\} \quad \coloneqq \quad P \Rightarrow \mathsf{wp}\ C\ \{Q\}$$

There are logics, such as the Iris logic, which we will present in Part II, which takes weakest preconditions as their primitive notion, from which Hoare triples are deduced.

One advantage of weakest preconditions is that, except for `while` loops as alluded to above, they can be computed by induction on the syntax of the program:

$$\begin{aligned}
\mathsf{wp}\ \mathtt{skip}\ \{Q\} &\quad\coloneqq\quad Q\\
\mathsf{wp}\ x \coloneqq E\ \{Q\} &\quad\coloneqq\quad Q[x \coloneqq E]\\
\mathsf{wp}\ C_1; C_2\ \{Q\} &\quad\coloneqq\quad \mathsf{wp}\ C_1\ \{\mathsf{wp}\ C_2\ \{Q\}\}\\
\mathsf{wp}\ \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \{Q\} &\quad\coloneqq\quad (B \land \mathsf{wp}\ C_1\ \{Q\}) \lor (\neg B \land \mathsf{wp}\ C_2\ \{Q\})
\end{aligned}$$

Therefore, to check that a program (without loop) satisfies a Hoare triple $\{P\}\ C\ \{Q\}$, it suffices to prove the entailment

$$P \Rightarrow \mathsf{wp}\ C\ \{Q\}$$

which, in practice, can be fairly well automated using automatic theorem provers.

A common strategy for loops is to ask the programmer to give a loop invariant, which can then be used to define the weakest precondition:

$$\mathsf{wp}\ \mathtt{while}\ B\{I\}\ \mathtt{do}\ C\ \{Q\} \quad\coloneqq\quad I$$

assuming the side conditions $B \land I \Rightarrow \mathsf{wp}\ C\ \{I\}$ and $\neg B \land I \Rightarrow Q$ hold. Examples (among many) of tools which use refinements of the method just outlined are Why3 (Filliâtre and Paskevich, 2013) for ML or Dafny (Leino, 2010) for C.

Soundness theorem

We have explained informally the meaning of a (weak) Hoare triple, in terms of executions of the programs. The corresponding formal statement is called the *soundness* *theorem* of the logic. To be able to state it, we need to give the language a formal semantics. For example, we can consider its big step semantics $C, \mathfrak{s} \Downarrow \mathfrak{s}'$ which states that the program C started in the initial state $\mathfrak{s}$ terminates, and its final state is $\mathfrak{s}'$, and $C, \mathfrak{s} \Downarrow \lightning$ if the program crashes.

We can now state the soundness theorem:

Theorem 0.1.1. *Given a derivation tree π of a Hoare triple $\{P\}C\{Q\}$, and given a machine state $\mathfrak{s}$ such that $\mathfrak{s} \vDash P$, the program is safe: $C, \mathfrak{s} \not\Downarrow \lightning$ and if $C, \mathfrak{s} \Downarrow \mathfrak{s}'$ then $\mathfrak{s}' \vDash Q$*

Proof. Straightforward induction on the structure of the derivation tree π. □

The reason why the soundness theorem is important is that Hoare logic is an "axiomatic logic", in the terminology of Girard, in that it is only useful as a tool to prove properties of the program expressed in the ambient logic; and this connection is given by the soundness theorem. This point will more salient in Part II because the distance between the Iris logic and the ambient logic is greater, as Iris is a step-indexed modal logic.

Semantic triples and completeness

We can rephrase the soundness theorem above by defining the *semantic validity* of a Hoare triple as:

$$\vDash \{P\}\,C\,\{Q\} \quad := \quad \forall \mathfrak{s},\ \mathfrak{s} \vDash P \implies C, \mathfrak{s} \not\Downarrow \lightning \ \wedge\ \forall \mathfrak{s}',\ C, \mathfrak{s} \Downarrow \mathfrak{s}' \Rightarrow \mathfrak{s}' \vDash Q$$

Then, writing $\vdash \{P\}\,C\,\{Q\}$ for the existence of a derivation tree for that Hoare triple, the soundness theorem can be expressed as

Theorem 0.1.2. $\vdash \{P\}\,C\,\{Q\}$ *implies* $\vDash \{P\}\,C\,\{Q\}$.

The usual way to prove this theorem amounts to proving that each inference rule of the logic has a corresponding lemma where we replace syntactic triples with semantic triples. Some logics such as Iris completely forgo syntactic triples and only have semantics triples with lemmas about them which correspond to the inference lemmas.

Despite its simplicity, Hoare logic is *complete*, in the sense that the converse of the second formulation of the soundness theorem holds:

Theorem 0.1.3. $\vDash \{P\}\,C\,\{Q\}$ *implies* $\vdash \{P\}\,C\,\{Q\}$.

In the sequel we will only consider the soundness of program logics.

0.1.3 Program logics for concurrent programs

As we have discussed, Hoare logic is a program logic well suited for a simple sequential programming language such as WHILE. Intuitively, the logic ensures that at each step of an execution of the program, it is in a state which guarantees that the execution is safe.

In a concurrent setting this is not sufficient anymore, because a non-trivial thread cannot be safe regardless of how the other threads executing concurrently are behaving. Consider indeed a thread which modifies randomly the values of every variables, which are shared between the threads. In these conditions, during the proof of any other thread, it does not make sense to have, say, a loop invariant which depends on the value of a variable since that other thread can invalidate it at any moment.

The fundamental difference between logics for sequential and concurrent languages is that, if we are to prove each thread separately, the proof needs to pose constraints on the behavior of its environment, which is the abstraction of all threads possibly executing concurrently.

A simple imperative shared memory language

To make things more concrete, let us consider a simple concurrent language with atomic sections, which adds the following commands to the WHILE language we considered in the previous section:

$$C ::= \ldots \mid C_1 \parallel C_2 \mid \texttt{atomic}\ C$$

The command $C_1 \parallel C_2$ executes the two commands C_1 and C_2 in parallel and waits for both of them to finish, and the atomic section $\texttt{atomic}\ C$ creates a critical section, in that at most one such critical section is executing in the complete program at any one time. Here is a very simple program taking advantage of these two new constructs:

```
atomic (y:=x; x:=y+1) || atomic (z:=x; x:=z+1)
```

If the initial values of x is 0, then the the final value will be 2. Without the critical sections, both threads could have read the value 0 from x, and then they both would have written 1 into x. (There would have been a data race as well, but we will talk about this later.)

Rely-guarantee

There have been several generalizations of Hoare logic to concurrent settings. An early milestone was proposed by Owicki and Gries (1978), who introduced an extension of Hoare logic to shared-memory concurrent programs, which now bears their name, which is compositional, in that each thread is proved separately. The central rule is the following

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\} \quad \text{the two proofs are not interfering}}{\{P_1 \land P_2\}\,C_1 \parallel C_2\,\{Q_1 \land Q_2\}}$$

The issue however is that the non-interference side-condition is quite complicated and unpractical.

The logic which we consider more carefully is the *Rely-Guarantee* logic (RG), which was introduced by Jones (1983). The two key ideas are, first, to make explicit the assumption about the interferences of the environment on which the proof of the program *relies*, and, second, that conversely, the program should provide *guarantees* to its environment. Formally, a specification in RG is a 5-tuple which we write

$$R; G \vdash \{P\}\,C\,\{Q\}$$

where, as before, P and Q are predicates, and C is the program; the two new items R and G are *relations* on memory states (or a syntactic representation thereof). Intuitively, G contains the set of atomic modifications of the memory which the program is allowed to perform; dually, R represents the set of atomic modifications which the environment is allowed to perform while C is executing.

Since the program needs to be able to rely on its precondition P to hold at the beginning of its execution, the precondition P needs to be stable under the rely relations, in that $\mathfrak{s} \vDash P \land (\mathfrak{s}, \mathfrak{s}') \in R \Rightarrow \mathfrak{s}' \vDash P$. Otherwise, the proof of the program would have to take into account that the environment can invalidate P at any time.

The inference rule for the parallel product ensures that the two specifications are compatible, in that the *rely* relation of each thread follows from the *guarantee* of the other:

$$\frac{R_1; G_1 \vdash \{P_1\}\,C_1\,\{Q_1\} \quad R_2; G_2 \vdash \{P_2\}\,C_2\,\{Q_2\} \quad G_1 \Rightarrow R_2 \quad G_2 \Rightarrow R_1}{R_1 \cap R_2; G_1 \cup G_2 \vdash \{P_1 \land P_2\}\,C_1 \parallel C_2\,\{Q\}}$$

for some Q computed from the Q_i's and the R_i's. The improvement over the Owicki-Gries logic is that the side conditions no longer refer to the proofs of C_1 and of C_2, but only the specifications of C_1 and of C_2.

The other rules ensure that each atomic step taken by a program are part of its guarantee relation, and that all preconditions are stable under the rely relation.

In conclusion,the fundamental difference between sequential Hoare logics and concurrent Hoare logics is that, for the logic to be compositional, the predicates it manipulates must be secure against interferences from any *well-specified* environment.

0.1.4 Limitations of Hoare logics

Despite the power of Hoare logic and its variant —they are complete— they have difficulties with compositionality. One instance of this is related to the *frame rule*, an important rule for compositionality of proofs:

$$\frac{\{P\}\,C\,\{Q\} \quad R \text{ does not contain variables written by } C}{\{P \land R\}\,C\,\{Q \land R\}}$$

It allows for proving a piece of code C using only the needed pre/post-condition P and Q, and the using it in any context where the precondition implies $P \land R$. A typical example where this is useful is in a program with many variables which contains a loop writing to a single variable x. Without such a rule, the loop invariant needs to state that every property about the other variables is preserved, which can be tiresome. More fundamentally, this allow a single piece of code to be proved once and used in several different contexts.

This rule does hold in “classical” Hoare logic, but tends to fail in extensions. In the context of concurrent programming languages, the Hoare triples of RG do not admit a suitable frame rule, making the rely and guarantee relations global.

The second programming language feature which invalidates Hoare logic’s frame rule, and which motivated separation logic originally, is the addition of pointers. The core of the problem is that, unlike for program variables, interference at the level of pointers (addresses in the memory heap) is a semantic property, which cannot be captured using a syntactic side condition like the frame rule presented above.

We add pointers to our simple (sequential) While language. The memory states are now pairs $\mathfrak{s} = (s, h)$ of a stack s as before, and a heap h a finite partial mapping from addresses to values. We identify addresses with natural numbers, and we assume that addresses are values, allowing programs to store pointers. We write $[E]$ to dereference the address represented by the expression E, giving the following commands:

$$C ::= \cdots \mid [E] \coloneqq E' \mid \mathsf{x} \coloneqq [E]$$

It is natural to add an atomic predicate to be able to specify the value contained at some memory location, which mirrors the predicates for the stack:

$$\ell \mapsto v$$

specifies that the address ℓ is allocated and that it contains the value v. However, it is easy to see that the frame rule given above does not hold, as the following could be derived:

$$\frac{\{\ell \mapsto 1\}\ [\ell] \coloneqq 3\ \{\ell \mapsto 3\}}{\{\ell \mapsto 1 \land \ell \mapsto 1\}\ [\ell] \coloneqq 3\ \{\ell \mapsto 3 \land \ell \mapsto 1\}}$$

Of course this example can be ruled out using a version of the side condition mentioned above, but the address ℓ could be stored in a variable, or in the heap.

The fact that a program C admits a Hoare triple $\{P\}\ C\ \{Q\}$ implies, informally, that the memory it manipulates is described by the predicate P. Therefore, for a frame rule to exist, it would be sufficient to be able to state that the frame predicate R describes a part of the memory which is distinct from the memory described by P. The new tools provided by separation logic will allow us to do just that.

Another way this disjointedness property is useful is in specifying programs which manipulate data structures which contain pointers. For example, the obvious program which mutably concatenates two linked lists produces a linked list only if the two lists passed as arguments do not *alias*, that is, only if they span two disjoint areas of the memory.

0.2 Separation logic

Separation logic was introduced around 2000 by Reynolds, O'Hearn, Ishtiaq and Yang to improve reasoning about pointers in Hoare logics. It acquired its name in a survey paper by Reynolds (2002), following works by Ishtiaq and P. W. O'Hearn (2001), Reynolds (2000), P. W. O'Hearn, Reynolds, and Yang (2001).

As we explained above, the main improvement over Hoare logic lies in how predicates can control aliasing. This is achieved using a new logical connector, *the separating conjunction*, written $P * Q$. It states that the predicates P and Q hold in two disjoint regions of the memory:

$$\mathfrak{s} \vDash P * Q \qquad \Longleftrightarrow \qquad \exists \mathfrak{s}_1, \mathfrak{s}_2,\ \mathfrak{s} = \mathfrak{s}_1 \uplus \mathfrak{s}_2 \ \land\ \mathfrak{s}_1 \vDash P \ \land\ \mathfrak{s}_2 \vDash Q$$

This allows to state *at a semantic level* that two predicates are "independent". More fundamentally, predicates now behave like resources: they are not duplicable in general, in that $P \Rightarrow P * P$ does not hold.

There are two flavors of separation logics, depending on whether the logic allows weakening: If it does, the logic is called *affine* (or intuitionistic), otherwise it is *linear* (or classical). Semantically, in logics of the first kind, the set of states which satisfy any predicate P is upward closed:

$$\forall \mathfrak{s}, \mathfrak{s}', \quad \mathfrak{s} \vDash P \ \wedge\ \mathfrak{s} \subseteq \mathfrak{s}' \quad \Longrightarrow \quad \mathfrak{s}' \vDash P$$

For example, the interpretation of the "points-to" predicates which specify the value at some address ℓ of the heap in an affine logic is defined in the same way as for the Hoare logic above:

$$\mathfrak{s} \vDash \ell \mapsto v \qquad \Longleftrightarrow \qquad (\ell, v) \in \mathfrak{s}$$

In such a logic, the unit element of the separating conjunction $*$ is the predicate $\top$ which holds for any memory state. In a separation logic which is linear, however, $\ell \mapsto v$ only holds for the corresponding singleton heap:

$$(s, h) \vDash \ell \mapsto v \qquad \Longleftrightarrow \qquad h = [\ell \mapsto v]$$

In such a logic, the predicate $\top$ is not the unit element of the separating conjunction. Actually, the mapping $P \mapsto P * \top$ defines a translation from a linear to an affine separation logic. The unit element is the atomic predicate **emp**, which holds exactly when the heap is empty. In the remainder of this introduction and in Part I we consider a linear separation logic. Part II uses the Iris separation logic, which is affine.

0.2.1 Ownership

A predicate P can be seen as representing the *ownership* of an area of the memory which satisfies it. As we will see later, this (purely logical) ownership can be transferred among threads when acquiring a lock. Another example of ownership transfer would be a program which sends a pointer in a channel, giving away its ownership of the memory it points to; the program which receives from the other end of the channel would be the new owner of this memory.

The separating conjunction $P * Q$ then represents the ownership of P and the ownership of Q. The usual conjunction $P \wedge Q$, on the other hand, states the ownership of memory which satisfies both P and Q. This is intuitively similar to intersection types, where a term has several types, and can be used as either.

The ownership interpretation of a separation logic Hoare triple $\{P\}\ C\ \{Q\}$ is that the program C needs to be given ownership of the resource P to run, and after it has finished, it gives back to its caller the ownership of the resource Q.

0.2.2 Inference rules

Separation logic allows us to state a frame rule which holds in the presence of pointers:

$$\frac{\{P\}\,C\,\{Q\} \quad R \text{ does not contain variables written by } C}{\{P * R\}\,C\,\{Q * R\}}$$

The reason why there still is a side condition about written program variables is that in the presentation we have given, predicates do not have a notion of ownership of variables. This means that non-interference at the level of program variables needs to be handled in side conditions.

This is why most modern separation logic handle variables like ML: a variable is immutable, but can contain a location whose content can change, as we will see in Part II. In other words, instead of memory states being a stack and a heap, they are a heap and an environment. In the remainder of this introductory chapter, we omit those side conditions, since they can be dealt with.

The remainder of the rules are similar to the ones for Hoare logic. For example, the rule for reading from the heap is the following:

$$\frac{}{\{\mathsf{x} = v' * \ell \mapsto v\}\,\mathsf{x} \coloneqq [\ell]\,\{\mathsf{x} = v * \ell \mapsto v\}} \tag{0.1}$$

The resource $\ell \mapsto v$ is given back in the postcondition because the command does not consume this resource. In contrast, the `dispose` operation which frees the memory has the following specification:

$$\frac{}{\{\ell \mapsto v\}\,\mathtt{dispose}(\ell)\,\{\mathbf{emp}\}}$$

The linearity of the logic allows the user of the logic to ensure that all the memory has been freed using the `dispose` operation at the end of the execution of the program by choosing **emp** as the postcondition of the program.

0.2.3 Bunched implications

The predicates of separation logic are an instance of *the logic of bunched implications* (BI), a substructural logic introduced by P. W. O'Hearn and D. J. Pym (1999). It features two types of conjunctions, additive conjunction $\wedge$ and multiplicative conjunction $*$, each with a right adjoint $\to$ and $\mathrel{-\!\!*}$ (read "magic wand"), respectively.

A sequent in BI is of the form $\Gamma \vdash P$, where the context Γ is a tree of the form:

$$\Gamma \Coloneqq P \mid \Gamma_1, \Gamma_2 \mid \Gamma_1; \Gamma_2$$

The arborescent nature of the contexts gives its name to the logic. The internal nodes of the tree denoted with semicolons correspond to ∧, and the nodes denoted with commas correspond to ∗, as demonstrated by the introduction rules of the implications:

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \mathrel{-\!\!*} Q} \qquad \frac{\Gamma; P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$$

The main difference is that only ; admits structural rules. To express those rules, we denote by $\Gamma(\Delta)$ a context where Δ appears as a subtree, and where Γ is the "rest" of the tree (more formally, Γ is a context in the other meaning of the word).

$$\frac{\Gamma(\Delta) \vdash P}{\Gamma(\Delta; \Delta) \vdash P} \qquad \frac{\Gamma(\Delta; \Delta) \vdash P}{\Gamma(\Delta) \vdash P}$$

As expected, there is a multiplicative unit I and an additive unit ⊤, which correspond to **emp** and ⊤ in the separation logic we have presented.

Models of BI

A natural question to ask is how BI relates to linear logic, another substructural logic. One answer is to look at models of BI. A model of BI is a *doubly closed category*: a Cartesian closed category $(\mathcal{C}, \times, \Rightarrow, \top)$ together with a monoidal closed structure on the *same* category $(\mathcal{C}, *, \mathrel{-\!\!*}, I)$. The semantics of a context Γ is given by interpreting , using the monoidal product ∗, and interpreting ; using the Cartesian product ×. In particular, we have:

$$\mathcal{C}(I, P \mathrel{-\!\!*} Q) \cong \mathcal{C}(P, Q) \cong \mathcal{C}(\top, P \Rightarrow Q)$$

Many separation logics have a magic wand; it is the basic notion of implication which is used in Iris for example.

One class of examples of models of BI are functor categories over monoidal categories, where the monoidal product is the Day tensor product. There is also a game model by McCusker and D. Pym (2007).

0.2.4 Towards concurrent separation logic

The great aptitude of separation logic for controlling interferences make it well-suited for the concurrent setting. Intuitively, the Frame rule implies the following rule for the parallel product:

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \quad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 * P_2\}\, C_1 \parallel C_2\, \{Q_1 * Q_2\}}$$

Indeed, according to the frame rule, each program C_i leaves untouched the memory described by P_{3-i}, so it is reasonable to expect that C_1 and C_2 do not to interfere at all, which means that this rule is semantically valid.

Separation logic can be seen as an implicit instance of the Rely-Guarantee method: a program which can be specified with a separation logic Hoare triple $\{P\}C\{Q\}$ guarantees to only access memory it owns, which, at the start of its execution, is described by its precondition P, and it relies on the environment not to access the memory owned by C.

The purpose of concurrent separation logic is to provide means to prove concurrent programs which interfere in a controlled way.

0.3 Concurrent separation logic

Concurrent separation logic (CSL) was introduced by P. W. O'Hearn (2004) and Brookes (2004) as an extension of separation logic to a concurrent shared memory language with locks.

The language which was considered in the early CSL papers was slightly different from the ones we used in Section 0.1.3: Instead of atomic sections, programs can declare locks, and use critical sections associated to each of these locks. Its grammar replaces the command `atomic` C with:

$$C ::= \cdots | \texttt{resource}\ r\ \texttt{do}\ C \mid \texttt{with}\ r\ \texttt{do}\ C$$

The command `resource` r `do` C declares a new lock r, which is available inside C; `with` r `do` C acquires the lock r, runs C, and then unlocks r. Dynamically, two critical sections for the same lock are guaranteed not to run at the same time.

0.3.1 Resource invariants

The idea of CSL is to associate each lock with a predicate of CSL, which describes the resource protected by the lock while it is available (unlocked). In CSL, Hoare triple are parameterized with a context Γ, making them formally 4-tuples:

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

where P and Q are predicate of the same type as (sequential) separation logic, and Γ is a context

$$\Gamma \ ::= \ \emptyset \mid \Gamma, r : I$$

associating predicates I with locks r. The predicate I is called the *lock invariant* of the lock r. We consider contexts up to permutations and we assume the lock names are distinct.

The general idea is that the only way for threads to communicate is though locks: for example, suppose the lock r is associated with the resource $\exists v, \ell \mapsto v$, which expresses the ownership of the memory location ℓ. Suppose ℓ contains initially 0, and that one thread wants to signal another thread. It can write the value 1 to the memory location ℓ in a critical section:

$$\texttt{with}\ r\ \texttt{do}\ [\ell] \coloneqq 1$$

The other thread could be reading ℓ in a loop, waiting to read 1 before exiting the loop:

$$\begin{array}{l} \texttt{x} \coloneqq 0; \\ \texttt{while}\ x = 0\ \texttt{do}\ \texttt{with}\ r\ \texttt{do}\ x \coloneqq [\ell] \end{array}$$

At the logical level, the program acquires the resources protected by the lock when it enters the critical section, and it must release the resource at the end of the critical section.

$$\frac{\Gamma \vdash \{P * I\}\, C\, \{Q * I\}}{\Gamma, r : I \vdash \{P\}\, \texttt{with}\ r\ \texttt{do}\ C\, \{Q\}} \qquad \frac{\Gamma, r : I \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P * I\}\, \texttt{resource}\ r\ \texttt{do}\ C\, \{Q * I\}}$$

To declare a new lock r with invariant I, the program needs to let go of the resource I and put it “in the lock”; after the end of the command C the program gets back the resource I. Finally, the rule for the parallel product is the same as the one sketched at the end of the previous section, with the addition of the context which is shared between the two threads.

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

0.3.2 Soundness theorem

The soundness theorem of the CSL states, as for Hoare logic, that every execution of a well-specified program is safe, and that if it terminates, the final state satisfies the postcondition. If the logic uses linear predicates, one can allow the initial state of said execution ton contain a sub-memory state which satisfies the precondition.

Theorem 0.3.1. *Suppose $\emptyset \vdash \{P\}\, C\, \{Q\}$, and suppose that $\mathfrak{s} \vDash P * \top$, then any execution of C which starts from $\mathfrak{s}$ is safe, and its final state, if it exists, satisfies $Q * \top$.*

There is another property which is important to require for a concurrent shared-memory program to be considered safe: it must be data race free. Informally, an execution of a program contains a *data race* if there are two instructions which are executed at the same time and such that one writes to the location the other one is accessing. Therefore, the soundness theorem of CSL contains a second half:

Theorem 0.3.2. *Suppose* $\emptyset \vdash \{P\}\, C\, \{Q\}$, *and suppose that* $\mathfrak{s} \vDash P * \top$, *then any execution of* C *which starts from* $\mathfrak{s}$ *is data-race free.*

This second half of the theorem is different in nature to the first because it requires reasoning about *pairs* of instructions of the same execution.

The proof of soundness of CSL is significantly more difficult than its sequential counterpart, as the meaning of a CSL Hoare triple needs to state formally what is the "rely" of the environment.

Most proofs of soundness of CSL add a rule in the semantics of the programming language which makes the program crash when a data race occurs, this allows one to deduce the second part of the theorem from the first.

Part I of this thesis, on the contrary, develops a proof of soundness of CSL which handles directly this aspect by giving an asynchronous semantics to both the program and the derivation trees of CSL, allowing us to talk about concurrent executions of instructions.

0.3.3 Developments to concurrent separation logic

There have been many advances in the field of concurrent separation logics since the seminal papers of O'Hearn and Brookes. The language we considered above is indeed very simple, and CSL was extended to support richer languages with higher-order functions, higher order store (mutable references which can contain closures), dynamic creation of threads, etc.

Another area of improvement is the concurrency primitives that the programming languages support. The concurrency primitive of the While language above is statically declared locks. Logics for dynamically created locks were developed by Gotsman et al. (2007b), and later for atomic operations such as Compare-and-swap allowing lockless data structures to be proved (Parkinson, Bornat, and P. O'Hearn, 2007).

All the examples above consider programs with sequentially consistent (SC) semantics (Lamport, 1979): The execution of concurrent programs are interleaved together. Parallel computers exhibit non-SC behaviors, making it particularly difficult to reason about concurrent programs. Logics based on CSL were developed to reason about such programs, such as GPS (Turon, Vafeiadis, and Dreyer, 2014).

Other directions include distributed systems —eg. Disel (Sergey, Wilcox, and Tatlock, 2017), or Aneris (Krogh-Jespersen et al., 2020; Gondelman et al., 2021), and language based security (Georges et al., 2021), …

The main technical means which contributed to this increase in expressive power of the logic is the decoupling of *logical state* from *machine state*. Here, we call machine state the notion of state which is used to define the semantics of the programming language, and we call logical state the type of object the predicates of the logic are predicates on. In the instance of CSL above, the meaning of a predicate P is defined by the satisfiability relation $\mathfrak{s} \vDash P$, where $\mathfrak{s}$ is a machine state. In other words, the two notions are identified in this early example.

Permissions

An early instance of this decoupling is the introduction of *permissions* which were introduced by Bornat, Calcagno, P. O'Hearn, et al. (2005) in separation logic, following their invention by Boyland (2003), who used them in a type system.

To avoid data races, the situation to forbid is one thread writing to a location while another is reading or writing to that location. The variant of CSL we have presented above is too strict in that it also disallows concurrent reads. Indeed, to be able to read a location ℓ, using the rule (0.1), a command needs ownership of a corresponding resource $\ell \mapsto v$. Since this resource is not duplicable, either it is protected by a lock or only one thread can own it.

The solution is to consider a more refined notion of ownership: one can own a fraction p $(0 < p \leq 1)$ of a location. Fractional ownership can be combined using the following law:

$$\ell \overset{p_1}{\mapsto} v * \ell \overset{p_2}{\mapsto} v \iff \ell \overset{p_1+p_2}{\mapsto} v$$

if $p_1 + p_2 \leq 1$. Any fractional ownership is sufficient to read from it, but full ownership ($p = 1$) is required to write to it. The invariant enforced by the logic is that if one thread owns a full permission over some location ℓ, then no other thread is able to own any fractional permission to that location, which means that no other thread can read from this location, avoiding data races on location ℓ.

Because predicates now contain permissions, they are predicates over a refined version of the memory which contains permissions. This logical state is related to the machine state by erasing permissions.

Ghost variables and logical state

In Hoare logics, it is very useful to have *ghost code*, code which is added to the program to help with reasoning but does not affect the behavior of the program. A classical example is proving that two threads which each add k to a shared reference end up adding $2k$ to it. Here is the program:

```
resource r do
  with r do (x ≔ [ℓ]; [ℓ] ≔ x + k) ∥ with r do (y ≔ [ℓ]; [ℓ] ≔ y + k)
```

and we want to prove that, if ℓ contains initially 0, in the final state it contains $2k$. Since both thread write to ℓ, we need to put total ownership of it in the lock r, but the invariant cannot express that, depending on which instruction has been executed, the value at location ℓ is either 0, k or $2k$: it has to be of the form $\exists v, \ell \overset{1}{\mapsto} v$ or similar.

The solution is to use ghost code, and add two ghost variables λ and ρ such that λ contains the contribution of the left thread, and ρ that of the right. Because each is written to by a single thread and both are mentioned in the invariant, we split their ownership in two part: one half is given to the relevant thread, and the other is in the invariant:

$$\exists v_1, v_2,\ \ \lambda \overset{1/2}{\mapsto} v_1 * \rho \overset{1/2}{\mapsto} v_2 * \ell \overset{1}{\mapsto} v_1 + v_2$$

We modify the code so that each critical section also increments the corresponding ghost variable by k. At the end of the execution, we can easily prove that λ and ρ both contain k and therefore ℓ contains $2k$.

Usually there are syntactic rules to ensure that non-ghost variables do not depend on ghost variables, and one can prove an erasure theorem stating that the program with the ghost code erased and the original program are equivalent in some sense. This approach has drawback however, as the source code of the program has to be modified, which is detrimental to modularity of proofs.

In recent concurrent separation logics such as Iris (Jung, Swasey, et al., 2015), ghost variables are subsumed by logical variables, which live in the logical state instead of the machine state. This increases further the distance between the machine and the logical states compared to permissions, as they no longer have the same "shape".

Iris

In Part II of this thesis, we will use the Iris (Jung, Swasey, et al., 2015; Jung, Robbert Krebbers, Lars Birkedal, et al., 2016; Robbert Krebbers, Jung, et al., 2017) concurrent separation logic, which unifies under an extensible framework many of the previous concurrent separation logics which had been developed in the 2010s. One powerful aspect of Iris is that Hoare triples are defined in its language of predicate. This has two consequences: First, a Hoare triple can appear in the precondition of some Hoare triple, allowing for specifying higher-order programs. Second, this lowers the effort required to develop new program logics, since one only needs to define a new version of Hoare triples in the base logic of Iris. An instance of this appears in Part II.

The semantic of Iris is defined a step-indexed Kripke semantics: an Iris predicate is a step-indexed predicate over a notion of world, which are defined as the solution to a guarded domain equation. Iris's facilities to solve domain equation have also been used to define step-indexed logical relations (Robbert Krebbers, Timany, and Lars Birkedal, 2017): for example to prove properties about the ST monad (Timany, Léo Stefanesco, et al., 2017), or the soundness of the DOT calculus (Giarrusso et al., 2020), but also to develop a model of the Rust language (Jung, Jourdan, et al., 2017).

The soundness theorem for the usual notion of Hoare triple of Iris is proved using the adequacy theorem of the base logic. The soundness theorem states, as usual, that a well-specified program does not get stuck (the way runtime errors are represented in Iris). Though it does not provide data race freedom, it is possible to define other program logics which do guarantee data race freedom, such as the RustBelt semantic model of the Rust language (Jung, Jourdan, et al., 2017).

In Part II of this thesis, we develop a program logic in the Iris framework with a more intensional soundness theorem. The logic is parameterized by a user-chosen state transition system (STS) which models the behavior they want their program to satisfy. For example, they can chose the TLA+ (L. Lamport, 1994) model of the single-decree Paxos algorithm. It they prove their implementation of Paxos in this logic, the soundness theorem will certify that their Paxos implementation is simulated by the abstract model.

Another use of this logic is to prove that a concurrent program terminates assuming the thread-scheduler is *fair*. This is done by relating the program with an STS whose states are ordered by a well-founded order, and which satisfy a property expressing that enough transitions of the STS make the state decrease.

0.4 Organization of the thesis

This thesis is made of two independent parts. The first part, which reports on joint works with Paul-André Melliès (Melliès and Léo Stefanesco, 2017; Melliès and Léo Stefanesco, 2018; Melliès and Léo Stefanesco, 2020), takes a new approach to the soundness of concurrent separation logic. Both the programs and the derivation trees of CSL are given a truly concurrent interpretation using asynchronous graphs. These semantics are related by a forgetful map between them, and our version of the soundness theorem is expressed as topological properties of this map.

- Chapter 1 is a brief survey of previous proofs of soundness for CSL,
- Chapter 2 introduces our approach to the problem and defines the basic notions which we will use,
- Chapter 3 presents our asynchronous model of CSL and our proof of soundness.

Part II presents a joint work with Lars Birkedal, Léon Gondelman, Abel Nieto, Simon Oddershede Gregersen and Amin Timany which we described briefly just above (Timany, Gregersen, et al., 2021).

- Chapter 4 introduces Iris,
- Chapter 5 defines the new Hoare triples, its soundness theorem and examples relating programs with STSs describing their correctness,
- Chapter 6 develops a logic to prove the fair termination of shared memory concurrent programs based on the logic described in the previous chapter. This is the personal work of the author.

Links to definitions

This thesis is using the excellent *knowledge* package developed by ***Thomas Colcombet***. Definitions are written in bold, as just above, and uses of these notions are shown like this: tiny. Clicking on it in a PDF viewer goes to the definition, and hovering it may display the definition.

Part I

Asynchronous models of CSL

1 Proofs of soundness of CSL

We can distinguish two kinds of proofs of soundness of concurrent separation logic, depending on how the semantics of the programming language is formalized: semantically using traces or operationally using a small-step semantics.

1.1 Trace semantics of the language

When O'Hearn developped the rules of CSL, he asked Brookes, a specialist in the semantics of concurrent shared-memory programming languages, to help him prove the soundness of the logic. We describe here Brookes seminal proof (Brookes, 2004; Brookes, 2011).

He considers a simple While language with parallel composition, locks and critical sections. Its semantics is defined using the notion of *action*, which are of the form

$$\lambda \coloneqq \mathsf{nop} \mid i = v \mid i \coloneqq v \mid [\ell] = v \mid [\ell] \coloneqq v \mid \mathit{acq}(r) \mid \mathit{rel}(r) \mid \mathit{try}(r) \mid \mathit{abort} \ldots$$

An action describes an event in the history of a program, for example, $i = v$ means that the program read variable i and its value was v, and $i \coloneqq v$ means that the program wrote v into the variable i. The notation $[\ell]$ denotes the dereferencing of a memory address ℓ. The last three operations on a lock r respectively mean that the program successfully acquired the lock r, released it, or unsuccessfully tried to acquire it.

Interpretation of the language Programs are interpreted as sets of *traces*, finite or infinite sequences of such actions. But first, we can give a semantics to actions λ by giving them *effects* $\overset{\lambda}{\Rightarrow} \subseteq \mathbf{State} \times \mathbf{State}$. The predicate $\mathfrak{s} \overset{\lambda}{\Rightarrow} \mathfrak{s}'$ between states $\mathfrak{s}$ and $\mathfrak{s}'$ holds when the action λ is compatible with the initial state $\mathfrak{s}$ and final state $\mathfrak{s}'$. For example, the action $i = v$ of reading the value v from the variable i has the following effect:

$$\begin{array}{ll} \mathfrak{s} \overset{i=v}{\Longrightarrow} \mathfrak{s} & \text{if the state } \mathfrak{s} \text{ maps } i \text{ to } v \\ \mathfrak{s} \overset{i=v}{\Longrightarrow} \lightning & \text{if the variable } i \text{ is not in the state } \mathfrak{s} \end{array}$$

where $\lightning \in$ **State** is a special state denoting an error; here reading from an undefined variable. Note that, if $\mathfrak{s}$ maps i to another value that v, then there is no $\mathfrak{s}'$ such that $\mathfrak{s} \xRightarrow{i=v} \mathfrak{s}'$. We say that the action λ is *enabled* when there exists a state $\mathfrak{s}'$ such that $\mathfrak{s} \overset{\lambda}{\Rightarrow} \mathfrak{s}'$.

A trace $\vec{\lambda}$ is called *sequential* (or sequentially consistent) when it is enabled, in that there exist $\mathfrak{s}_0, \dots, \mathfrak{s}_{|\vec{\lambda}|}$ such that

$$\mathfrak{s}_0 \xRightarrow{\lambda_1} \mathfrak{s}_1 \xRightarrow{\lambda_2} \mathfrak{s}_2 \xRightarrow{\lambda_3} \cdots$$

Such a trace describes an execution which makes sense without interference from the environment.

As expected for a denotational semantics, the operators used to compose programs are reflected as operations in the semantic domain. The interpretation of the sequential fragment of the language is standard. For the parallel product, given two sets T_1 and T_2 of traces, $T_1 \parallel T_2$ is defined as the union of the interleavings of $t_1 \in T_1$ and of $t_2 \in T_2$. The set of interleavings of two traces is defined coinductively on their structure, taking into account the set of locks which is held by the program and its environment. The definition is setup in such a way that $t_1 \parallel t_2$ contains the singleton trace *abort* if it contains a data race.

With these operations, we interpret the commands as sets of traces and the expressions as sets of pairs (t, v) of a trace t and a resulting value v in the standard way. Of course, the traces in the semantics of a program are not necessarily sequentially consistent; for example the expression $i + i$ which reads the variable i twice and adds the two values is interpreted as the following set of pairs of a trace and a resulting value:

$$\left\{ ((i = v)\,(i = w)\,, v + w) \;\middle|\; v \text{ and } w \text{ values} \right\}$$

Interpretation of the logic The Hoare triples $\Gamma \vdash \{P\}\,C\,\{Q\}$ which are considered by Brookes (2004) have contexts of the form

$$r_1(X_1) : J_1, \dots, r_n(X_n) : J_n$$

where the X_i are sets of program variables which are said to be owned by the invariant.

The soundness theorem is formalized by defining the notion of valid Hoare triple. Informally, if a Hoare triple $\Gamma \vdash \{P\}\,C\,\{Q\}$ is valid, then all the traces of the program C which start from a state which satisfy the precondition P and the invariants in Γ are safe and their final states satisfy the postcondition Q and the invariants still hold.

This definition is too weak to be an inductive invariant because it does not take into account the interference from the environment, which is necessary to prove that the parallel product preserves validity.

Brookes' solution is to define a local notion of reduction, which he calls the *local enabling relation*

$$(s, h, A) \xrightarrow[\Gamma]{\lambda} (s', h', A')$$

where s is the stack, h is the heap, and A is the set of locks held by the "current" thread. For actions which do not involve locks, local enabling essentially coincides with the global enabling we have talked about above, except that the use of variables which appear in invariants is forbidden: For instance, the read of a variable i

$$(s, h, A) \xrightarrow[\Gamma]{i:=v} \lightning$$

is invalid if $i \notin \mathrm{dom}(s)$, as before, or if i is free in an invariant J_k which is not owned, in that $r_k \notin A$.

The logical nature of the local enabling relation is clearer when looking, for example, at the rules for the action $acq(r)$:

$$(s, h, A) \xrightarrow[\Gamma, r(X):J]{acq(r)} (s \uplus s_r, h \uplus h_r, A \uplus \{r\})$$

for any stack s_r and heap h_r such that $\mathrm{dom}(s_r) = X$ and $(s \uplus s_r, h_r) \vDash J$. When the program acquires a lock, its memory state can be adjoined with any memory state satisfying the invariant associated with the lock r.

Remark 1.1.1. This proof technique puts a requirement of the semantics of the programming language we consider: they must be insensitive to any extension of the memory:

$$\mathfrak{s} \uplus \mathfrak{s}_F \xrightarrow{\lambda} \mathfrak{s}' \qquad \Longrightarrow \qquad \exists \mathfrak{s}'', \mathfrak{s}' = \mathfrak{s}'' \uplus \mathfrak{s}_F \ \ \wedge \ \ \mathfrak{s} \xrightarrow{\lambda} \mathfrak{s}''$$

A common counter-example is semantics with a deterministic allocation strategy, such as `malloc` allocating the least free address. A naive notion of validity would make the following Hoare triple valid

$$\Gamma \vdash \{\mathbf{emp}\}\, \mathtt{malloc}(E)\, \{0 \mapsto E\} \tag{1.1}$$

because we assert that in the precondition that the initial state is empty, and therefore that address 0 is the least free address. This would, however, contradict the Frame-rule of separation logic, since it would let us prove the following rule:

$$\Gamma \vdash \{R\}\, \mathtt{malloc}(E)\, \{R * 0 \mapsto E\}$$

which is is clearly unsound (take $R \coloneqq 0 \mapsto 88$ for example). The technique used by Brookes (2004) ensures the validity of the Frame rule by putting conditions on the semantics of the programming language. Another more flexible proof, for example the proof by Vafeiadis (2011) which we will describe in Section 1.2, bakes in the frame into the definition of validity, making rule (1.1) invalid.

The core of the proof are lemmas relating local traces of a command C with local traces of its subcommands C_i. For example, for the parallel product:

Theorem 1.1.2 (Thm 21 in (Brookes, 2004))**.** *Let C_1 and C_2 be two commands and Γ a CSL context, and suppose that every variable which is free in one and written to in the other is owned by some invariant in Γ. Consider a trace $\alpha_1 \in [\![C_1]\!]$ and a trace $\alpha_2 \in [\![C_2]\!]$. Given a trace $\alpha \in \alpha_1 \parallel \alpha_2$ in their parallel product and $h = h_1 \uplus h_2$:*

1. *If $(s, h, \emptyset) \xrightarrow[\Gamma]{\alpha} \lightning$ then $(s \setminus \mathsf{wr}(\alpha_2), h_1, \emptyset) \xrightarrow[\Gamma]{\alpha_1} \lightning$ or $(s \setminus \mathsf{wr}(\alpha_1), h_2, \emptyset) \xrightarrow[\Gamma]{\alpha_2} \lightning$.*
2. *If $(s, h, \emptyset) \xrightarrow[\Gamma]{\alpha} (s', h', \emptyset)$ then either:*
    - *$(s \setminus \mathsf{wr}(\alpha_2), h_1, \emptyset) \xrightarrow[\Gamma]{\alpha_1} \lightning$, or*
    - *$(s \setminus \mathsf{wr}(\alpha_1), h_2, \emptyset) \xrightarrow[\Gamma]{\alpha_2} \lightning$, or*
    - *there exists $h_1' \uplus h_2' = h'$ such that*
        - *$(s \setminus \mathsf{wr}(\alpha_2), h_1, \emptyset) \xrightarrow[\Gamma]{\alpha_1} (s' \setminus \mathsf{wr}(\alpha_2), h_1', \emptyset)$, and*
        - *$(s \setminus \mathsf{wr}(\alpha_1), h_2, \emptyset) \xrightarrow[\Gamma]{\alpha_2} (s' \setminus \mathsf{wr}(\alpha_1), h_2', \emptyset)$.*

If we assume that α_1 and α_2 do not crash, then this imply that neither do the interleavings. This result is proved using a difficult induction on the length of the traces α_1 and α_2, where the statement is generalized to an arbitrary set A of locked resources instead of $\emptyset$.

Notice also that, because the separation of which program variables are used in which part of the program is handled using side rules, the statements of the lemmas as the one above need to pose subtle restrictions. This was solved in subsequent works with the variables as resources discipline of Bornat, Calcagno, and Yang (2006) (see the next chapter) and later by considering languages with immutable variables which can contain mutable memory-cells, in the tradition of ML.

We can now state the definition of validity of a Hoare triple:

Definition 1.1.3. A Hoare triple $\Gamma \vdash \{P\}\ C\ \{Q\}$ is *valid* if, for any trace $\alpha \in [\![C]\!]$, for any stacks s whose domain contains all program variables which appear in C and in Γ but are not owned by an invariant in Γ, and for any heap h, if $(s, h, \emptyset) \vDash P$ and $(s, h, \emptyset) \xrightarrow{\alpha} \mathfrak{s}'$, then $\mathfrak{s}' \vDash Q$. In particular, $\mathfrak{s}'$ is not the error state $\lightning$ since it satisfies some formula.

As expected, the soundness theorem states:

Theorem 1.1.4 (Soundness)**.** *Every provable Hoare triple* $\Gamma \vdash \{P\}\ C\ \{Q\}$ *is valid.*

It is proved by induction on the derivation tree of the Hoare triple $\Gamma \vdash \{P\}\ C\ \{Q\}$, using decomposition results such as Theorem 1.1.2.

Finally, all that remains to be done is to link validity of a Hoare triple, which is a statement about the *local* enabling relation of the program, to its global semantics, as mentioned in Remark 1.1.1.

1.1.1 Other proofs based on trace semantics

In their work on Subjective Concurrent Separation Logic, Ley-Wild and Nanevski (2013) interpret programs as sets of *action trees*, which one can see as more interactive variants of traces. An instead of having one trace for, say, each value return by a load to memory, the tree contains a branching with one subtree for each value. Unlike in Brookes's proof, action trees are finite approximations of the program, and programs are interpreted as sets of such trees, which form a complete lattice to interpret recursion. A more recent research program which interprets program as (possibly infinite) interactive trees in the Coq proof assistant is the work on *Interaction trees* by Xia et al. (2020) and Zakowski et al. (2020).

1.2 Operational semantics of the language

In this section, we explain how to prove the soundness of concurrent separation logic which is based on a small step operational semantics of the language, instead of a denotational semantics as in Brookes original proof which we described in the previous section. This section is based on the proof of soundness by Vafeiadis (2011). There are other proofs based on operational semantics, such as the Views framework by Dinsdale-Young et al. (2013), the TaDa logic by da Rocha Pinto, Dinsdale-Young, and Gardner (2014) or, more recently, the Iris framework, which we will discuss in the second part of this thesis.

$$\frac{\mathfrak{s} \vDash Q}{\mathsf{safe}(\mathtt{skip}, \mathfrak{s}, Q)} \qquad \frac{(C, \mathfrak{s}) \not\to \lightning \quad \forall C', \mathfrak{s}', (C, \mathfrak{s}) \to (C', \mathfrak{s}') \Rightarrow \mathsf{safe}(C', \mathfrak{s}', Q)}{\mathsf{safe}(C, \mathfrak{s}, Q)}$$

Figure 1.1: Semantic validity for Hoare logic

Vafeiadis expresses the semantics of the programming language under consideration, a simple While language with a parallel product and locks as small-step semantics: a state transition system whose states are pairs $(C, \mathfrak{s})$ of a program C and a machine state $\mathfrak{s}$ which contains the state of the memory and of the locks. A transition

$$(C, \mathfrak{s}) \longrightarrow (C', \mathfrak{s}')$$

expresses that, in machine state $\mathfrak{s}$, the program C reduces to C' and updates the machine state to $\mathfrak{s}'$. The definition is the expected one for a concurrent shared memory programming language, with the following two rules for the parallel product:

$$\frac{(C_1, \mathfrak{s}) \longrightarrow (C_1', \mathfrak{s}')}{(C_1 \parallel C_2, \mathfrak{s}) \longrightarrow (C_1' \parallel C_2, \mathfrak{s}')} \qquad \frac{(C_2, \mathfrak{s}) \longrightarrow (C_2', \mathfrak{s}')}{(C_1 \parallel C_2, \mathfrak{s}) \longrightarrow (C_1 \parallel C_2', \mathfrak{s}')}$$

Note that this is a slight simplification of the setting in the proof of Vafeiadis (2011) in the way we deal with locks.

Safety of a Hoare triple Before we explain the validity of a concurrent separation logic Hoare triple in the setting of Vafeiadis (2011), we go back to the simpler case of Hoare logic for (weak correctness of) sequential programs. In that setting, one can express the semantic validity of a Hoare triple $\vdash \{P\}\, C\, \{Q\}$ at machine state $\mathfrak{s}$ as $\mathsf{safe}(C, \mathfrak{s}, Q)$, where the predicate safe is defined in Figure 1.1. The predicate $\mathsf{safe}(C, \mathfrak{s}, Q)$ explores the tree of all computation which start in state $(C, \mathfrak{s})$ and check that no path leads to the error state $\lightning$, and that all leaves satisfy the postcondition Q. We can then define the validity of a Hoare triple as follows:

$$\vdash \{P\}\, C\, \{Q\} \quad \coloneqq \quad \forall \mathfrak{s},\ \mathfrak{s} \vDash P \implies \mathsf{safe}(C, \mathfrak{s}, Q).$$

The predicate of Figure 1.1 needs to be adapted to the setting of concurrent separation logic: First, separation logic predicates P (in linear variants such as the ones we consider here) hold for "small" memory states; therefore we would have, for example, to replace $\mathfrak{s} \vDash P$ by $\exists \mathfrak{s}', \mathfrak{s}' \subseteq \mathfrak{s} \wedge \mathfrak{s}' \vDash P$. More fundamentally, the predicate safe cannot handle an environment modifying the state, and therefore does not validate the inference rule for the parallel product.

$$\begin{array}{c} \mathfrak{s} \vDash Q \\ \hline\hline \mathsf{safe}(\texttt{skip}, \mathfrak{s}, \Gamma, Q) \end{array}$$

$$\begin{array}{l} \forall \mathfrak{s} \uplus \mathfrak{s}_F,\ (C, \mathfrak{s} \uplus \mathfrak{s}') \not\to \lightning \\ \\ \forall \mathfrak{s} \uplus \mathfrak{s}_\Gamma \uplus \mathfrak{s}_F, \mathfrak{s}_\Gamma \vDash \displaystyle\mathop{\circledast}_{r \in \mathsf{unlocked}(\mathfrak{s})} \Gamma(r) \ \Rightarrow \\ \quad \forall C', \mathfrak{s}', (C, \mathfrak{s} \uplus \mathfrak{s}_\Gamma \uplus \mathfrak{s}_F) \to (C', \mathfrak{s}') \Rightarrow \\ \qquad \exists \mathfrak{s}'' \uplus \mathfrak{s}'_\Gamma \uplus \mathfrak{s}_F = \mathfrak{s}' \ \land\ \mathfrak{s}'_\Gamma \vDash \displaystyle\mathop{\circledast}_{r \in \mathsf{unlocked}(\mathfrak{s}')} \Gamma(r) \ \land\ \mathsf{safe}(C', \mathfrak{s}'', Q) \\ \hline\hline \multicolumn{1}{c}{\mathsf{safe}(C, \mathfrak{s}, \Gamma, Q)} \end{array}$$

Figure 1.2: Semantic validity for concurrent separation logic

Vafeiadis' solution is to use the predicate defined coinductively[1] in Figure 1.2. The predicate $\mathsf{safe}(C, \mathfrak{s}, \Gamma, Q)$ is defined as a predicate over the the executions of C. Unlike its counterpart for Hoare logic, at each step, we consider what happens if C is executed in a state which *contains* $\mathfrak{s}$ and which contains memory which satisfies the invariants of the unlocked resources. One important thing to note and to contrast with the notion of local enabling in the proof of Brookes is that we always consider the reduction of the program in the global state of the memory, which is decomposed as $\mathfrak{s} \uplus \mathfrak{s}_\Gamma \uplus \mathfrak{s}_F$: the memory $\mathfrak{s}$ which is owned by the program, $\mathfrak{s}_\Gamma$ which is being protected by the locks, and $\mathfrak{s}_F$ which is owned by the environment, which we also call the frame. The reason this definition is adequate for the concurrent setting is that, at each step, the pieces of state $\mathfrak{s}_\Gamma$ and $\mathfrak{s}_F$ are universally quantified. As long as the environment does not modify $\mathfrak{s}$ and preserves the invariants, the program C will be safe, and will preserve the memory of the environment and the invariants. Therefore, one can show that, assuming both $\mathsf{safe}(C_1, \mathfrak{s}_1, \Gamma, Q_1)$ and $\mathsf{safe}(C_2, \mathfrak{s}_2, \Gamma, Q_2)$ hold, with $\mathfrak{s}_1 \uplus \mathfrak{s}_2$, then so does $\mathsf{safe}(C_1 \parallel C_2, \mathfrak{s}_1 \uplus \mathfrak{s}_2, \Gamma, Q_1 * Q_2)$.

Finally, Vafeiadis proves the soundness theorem:

Theorem 1.2.1. *If* $\Gamma \vdash \{P\}\ C\ \{Q\}$ *is provable, and if* $\mathfrak{s} \vDash P$*, then* $\mathsf{safe}(C, \mathfrak{s}, \Gamma, Q)$.

To prove that the absence of data-races in well-specified programs, similarly to Brookes'

[1] The paper and its associated mechanizations use the corresponding step-indexed predicate safe_n; and we can define $\mathsf{safe} \coloneqq \bigwedge_n\ \mathsf{safe}_n$ to recover the coinductive predicate.

proof before, the following rule RaceDetect which crashes the program when there is a data-race:

$$\frac{(\mathbf{accesses}(C_1, \mathfrak{s}) \cap \mathbf{writes}(C_2, \mathfrak{s})) \cup (\mathbf{accesses}(C_2, \mathfrak{s}) \cap \mathbf{writes}(C_1, \mathfrak{s})) = \emptyset}{(C_1 \parallel C_2, \mathfrak{s}) \longrightarrow \text{↯}}$$

which states that if C_1 is about to access a variable or a location which C_2 is about to write, or vice-versa, then the program crashes. Since the safe predicate implies safety of executions of closed programs without interference from the environment, this implies the absence of data-race.

1.2.1 Step-indexed models of CSL

There is a line of research (Svendsen and Lars Birkedal, 2014; Dodds et al., 2016) which uses a step-indexed type theory to define the validity of a Hoare triple. This line of research developed into the Iris framework (Jung, Swasey, et al., 2015; Jung, Robbert Krebbers, Lars Birkedal, et al., 2016; Robbert Krebbers, Jung, et al., 2017; Jung, Robbert Krebbers, Jourdan, et al., 2018).

The proof of soundness of the Iris separation logic proceeds by defining a weakest precondition similar to safe expressed in the Iris base logic, which is a modal logic with guarded recursion and a separating conjunction. Instead of syntactic inference rules, the user of the logic is directly manipulating the semantic predicate, with lemmas which roughly correspond to the usual inference rules of the logic. We give a more complete description of Iris in Chapter 4.

One thing to note here is that the "generic" notion of Hoare triple of Iris does not guarantee data-race freedom, or rather, every memory access is considered atomic. One can recover data-race freedom by instrumenting the semantics: For example, in the semantics of λ_{rust}, in the RustBelt development by Jung, Jourdan, et al. (2017), each memory cell is associated with what amounts to a reader-writer lock which crashes the program if two threads access and write the same memory location concurrently.

1.3 Other proofs soundness of CSL

1.3.1 Syntactic proofs

There are many proofs of soundness based on an operational semantics of the programming language and which are modeled after the familiar technique of progress and preservation proofs of safety of type systems. The idea is to see a (derivation tree of) a Hoare triple as a typing derivation and to prove two lemmas: First, a version of *progress* stating that a well-specified program does not immediately crash in a state which satisfies the precondition of the Hoare triple.

Lemma 1.3.1. *Given a Hoare triple* $\Gamma \vdash \{P\}\, C\, \{Q\}$, *and a machine state* $\mathfrak{s} \vDash P$, *the next step of reduction of* C *are safe:*

$$(C, \mathfrak{s}) \not\longrightarrow \lightning.$$

Second, one also proves a version of *preservation*, which states that a well-specified program necessarily reduces to a well-specified program.

Lemma 1.3.2. *Given a Hoare triple* $\Gamma \vdash \{P\}\, C\, \{Q\}$, *and a machine state* $\mathfrak{s} \vDash P$, *for any possible reduction*

$$(C, \mathfrak{s}) \longrightarrow (C', \mathfrak{s}')$$

there exists a predicate P' *of CSL such that* $\mathfrak{s}' \vDash P'$ *and a derivable Hoare triple* $\Gamma \vdash \{P'\}\, C'\, \{Q\}$.

The safety of well specified programs and the validity of the postcondition at final states are easy consequences of the two preceding lemmas.

This proof techniques have been used by many authors, for example François Pottier (2013) and Balabonski, François Pottier, and Protzenko (2014) in works which prove the type soundness of type systems which use features borrowed from separation logic. Gotsman et al. (2007a) use a predicate transformer semantics to prove the soundness of their variant of concurrent separation logic.

1.3.2 Other

There are many proofs of soundness of concurrent separation logic. We cite a few notable ones here, but we do not claim any exhaustiveness. Hayman and Winskel (2008) have given a proof of soundness of CSL using a truly concurrent semantics of the underlying programming language using Petri nets. Unlike the other proofs we have cited here, data-races are not detecting by crashing or clocking the program, but instead, data races are detected using the notion of independence of Petri nets, which state that two transitions have disjoint neighborhoods.

1.4 Conclusion

To our knowledge, most proofs deduce data race-freedom from the safety of programs where the semantics crashes in case of data race. Moreover, in most proofs of correctness of concurrent separation logics, the meaning of a Hoare triple $\{P\}\ C\ \{Q\}$ is a property of the program.

Instead, our proof of soundness interprets the derivation tree π of $\{P\}\ C\ \{Q\}$ itself as an interactive semantics of a program operating with a refined notion of state which tracks the owner of each memory cell.

The asynchronous soundness theorem which we will describe in the next chapter explains the absence of data race in valid executions by the fact that any concurrent access in the semantics of the program C corresponds to an independent access in the semantics of the derivation tree π.

2 Asynchronous soundness for CSL

2.1 Hoare logic as refinement systems

The starting point of our semantics of concurrent separation logic (CSL) is to see such a program logic as a refinement type system: the program C, which we see a being of type **State** → **State**, is *refined* by the Hoare triples $\Gamma \vdash \{P\}\, C\, \{Q\}$ it satisfies, which we could write $\Gamma \vdash C : \{P\} \to \{Q\}$. Indeed, if we consider the input and the output of the program to be the initial and the final states of its executions, the predicates P and Q describe subsets of the input and output respectively. This is characteristic of type refinement systems.

Melliès and Zeilberger (2015) have argued that the semantics of a type refinement system should be given as a functor

$$\mathcal{L} : \mathcal{D} \longrightarrow \mathcal{C}$$

from a category D where the refined programs are interpreted to a category $\mathcal{C}$ where the programs are interpreted. In the case of a Hoare logic for example, the functor $\mathcal{L}$ corresponds to the syntactic “forgetful” map

$$\begin{array}{c} \vdots\, \pi \\ \Gamma \vdash \{P\}\, C\, \{Q\} \end{array} \quad \longmapsto \quad C$$

projecting the program C out of the proof tree π of the Hoare triple $\Gamma \vdash \{P\}\, C\, \{Q\}$.

Our model will follow this guiding idea. In particular, the proofs π of the Hoare triples $\Gamma \vdash \{P\}\, C\, \{Q\}$ will be given their own semantic interpretations $[\![\pi]\!]$. The programs C will also be given a semantics $[\![C]\!]$, and the two will be related through a semantic map

$$\mathcal{L} : [\![\pi]\!] \longrightarrow [\![C]\!].$$

The existence of the map $\mathcal{L}$ suggests that the interpretations of the proofs and of the programs should be similar in nature so that they can easily be compared.

This chapter explains the rationale behind the choices we have made in the construction of our model of concurrent separation logic, and in particular how we deal with data races.

2.2 State transition systems

Perhaps the simplest way to give a semantics to a sequential imperative program is as a *state transformer*: a program C is seen as a function

$$[\![C]\!] \;:\; \textbf{State} \;\longrightarrow\; \textbf{State} + \bot$$

which takes as input an initial memory state and outputs either the final state of the program, when the program terminates on that input, or ⊥ if the program does not halt.

In the semantics of state transformers, the following two programs have the exact same semantics:

```
X := 0;
Y := 0;
```

```
X := 4;
Y := 0;
X := Y;
```

In some context, too much information is lost. When he gave his object based semantics of Idealized Algol, a higher-order sequential language, Reddy (1996) argued the information missing in this semantics is the *changing* of the state, seen as a first-class phenomenon and not simply a piece of information which can (partially) be recovered as the difference between the initial and the final state the program.

Another, more obvious, obstacle appears when adding concurrency to the language: the environment is able to observe the transient state of the program on the right above where the variable `X` is equal to the value 4. For example, the context `(-|| Z := X)` is able to distinguish between the two programs by reading `X` during the time in which it holds the value 4.

To capture the changes induced by a program C, we interpret it as a state transition systems $[\![C]\!]$: a graph where each intermediate state the program goes though is represented by a node, and each action of the program is represented as an edge.

It is important, if not so common, to distinguish between the *internal* and the *external state* of the program. The external state is characterized by the fact that it is observable from the context, and therefore it can be used for synchronization. In the case of the simple imperative and concurrent language we consider, it consists essentially on the state of the memory, since it is what is shared between the program and its environment. In a typed language, the type of the program would also be part of its external state; to a certain extent, this can be seen in how we interpret proofs of CSL, which we can see as programs typed using a refinement type system. The internal state, on the other hand, corresponds in our case to the position of the program the control flow, which would correspond to the program counter in a concrete machine.

To enforce this discipline, the state transition system $[\![C]\!]$ is labeled explicitly using a map $\lambda_{[\![C]\!]} : [\![C]\!] \to$ ま which associate an external state in ま to each node of the state transition system $[\![C]\!]$. We explain the nature of ま in the sequel. The nodes of $[\![C]\!]$ are left “opaque”: all we know about them is their image under $\lambda_{[\![C]\!]}$.

2.3 An imperative shared-memory concurrent language

The language we consider is the same as the one considered in the early concurrent separation logic papers, for example in the proofs of soundness of Brookes (2004) in Vafeiadis (2011).

The grammar of this language is the following. It has Boolean expressions

$$B ::= \mathtt{true} \mid \mathtt{false} \mid B \wedge B' \mid B \vee B' \mid \neg B \mid E = E'$$

arithmetic expressions

$$E ::= 0 \mid 1 \mid \ldots \mid x \mid E + E' \mid E * E'$$

and commands

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

where $x \in \mathbf{Var}$ are program (stack) variables, and the notation $[E]$ dereferences the address denoted by the arithmetic expression E. The action of the construction $\mathtt{with} r \mathtt{do} C$ is to acquire the lock r, execute the command C, and then release the lock r. Locks are introduced with the construction $\mathtt{resource}\, r\, \mathtt{do}\, C$: the lock r is only available inside of the command C.

Let us define the external states of our state transition systems. We decompose the memory in two parts: the *stack* $s : \mathbf{Var} \rightharpoonup_{fin} \mathbf{Val}$ which is a partial finite map from the set **Var** of variables to the set **Val** of values, and the *heap* $h : \mathbf{Loc} \rightharpoonup_{fin} \mathbf{Val}$, which associates a value to each allocated location $\ell \in \mathbf{Loc}$. We identify the set **Loc** of locations with the natural numbers $\mathbb{N}$, and we assume that $\mathbf{Loc} \subseteq \mathbf{Val}$ so that values can serve as addresses in the heap. A ***memory state*** μ is a pair (s, h) of a stack s and of a heap h.

There is one more piece of state which is needed: the locks. Given a set $\mathfrak{L} \subseteq \mathbf{Locks}$ of locks valid in the current context, the ***lock state*** keeps track of the set $L \subseteq \mathfrak{L}$ of locks

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

$$\frac{E(\mu) = \ell \quad \ell \in \mathrm{dom}(\mu) \quad E'(\mu) = v}{(\mu, L) \overset{[E]:=E'}{\rightsquigarrow} (\mu[\ell \coloneqq v], L)} \qquad \frac{r \notin L}{(\mu, L) \overset{P(r)}{\rightsquigarrow} (\mu, L \uplus \{r\})}$$

$$\frac{}{(\mu, L \uplus \{r\}) \overset{V(r)}{\rightsquigarrow} (\mu, L)} \qquad \frac{E(\mu) = v \quad \ell \notin \mathrm{dom}(\mu)}{(\mu, L \uplus \{r\}) \overset{\mathtt{alloc}(E)}{\rightsquigarrow} (\mu[x \coloneqq \ell, \ell \coloneqq v], L)}$$

Figure 2.1: Semantics of machine instructions

which are being held (locked). Finally, a ***machine state*** $\mathfrak{s} \in \mathbf{State}(\mathfrak{L})$ is a pair (μ, L) of a memory state μ and of a set of a lock state L.

We label each transition in the state transition system $[\![C]\!]$ with a label describing the external action the transition corresponds to. There is a strong correspondance between the external actions of the code and the “leaf” commands. The transitions of $[\![C]\!]$ are labeled with the following set $\mathbf{Instr}(\mathfrak{L})$ of actions, which we call ***machine instructions*** m:

$$x \coloneqq E \mid x \coloneqq [E] \mid [E] \coloneqq E' \mid \mathsf{nop}$$
$$x \coloneqq \mathtt{alloc}(E, \ell) \mid \mathtt{dispose}(E) \mid P(r) \mid V(r)$$

The first three correspond to reading and writing the memory, while nop does nothing, $x \coloneqq \mathtt{alloc}(E, \ell)$ corresponds to allocating a new memory cell at address ℓ initializing it with E, $\mathtt{dispose}(E)$ frees the address E. The last two machine instructions $P(r)$ and $V(r)$ respectively acquire and release the lock $r \in \mathfrak{L}$. The semantics of these instructions are expressed by the transition system $⚓_S$ between machine states described in Figure 2.1. The set of nodes of $⚓_S$ is the set $\mathbf{State} + ↯$ of mahchine states adjoined with an error state ↯. Its edges $\cdot \overset{m}{\rightsquigarrow} \cdot$ are labeled with machine instructions m.

Remark 2.3.1. The symbol ⚓ was chosen by Melliès (2019) for its similarity with boat anchors. It was introduced by Jeremy Gibbons as a symbol for tree constructors, as it is inspired by 木, the Chinese ideogram for tree. It can be pronounced “anchor”, “template” or “moo”.

A state transition system for interpreting programs can be defined as a pair $(G, \lambda : G \to ⚓)$ of a graph G and a graph homomorphism λ. Explicitly, each node x of the graph G

is labeled with a machine state $\lambda(x) \in \mathbf{State}$ or the error state $\lightning$. Each edge in G, seen as a transition,

$$x \longrightarrow y$$

is labeled with a transition of the state transition system $\pitchfork_{\mathrm{S}}$ defined in Figure 2.1 of one of the two forms

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

depending on whether executing the machine instruction m in machine state $\mathfrak{s}$ is safe (left) or produces an error (right). Since λ is a graph homomorphism, $\lambda(x) = \mathfrak{s}$ and, respectively, $\lambda(y) = \mathfrak{s}'$ or $\lambda(y) = \lightning$. A quick inspection of the definition of $\pitchfork_{\mathrm{S}}$ shows that the error state is terminal, in that there are no edges outgoing from $\lightning$; hence the description of the image under λ of the edge $x \longrightarrow y$ above is exhaustive. We sometimes write the situations above as

$$\mathfrak{s} \xrightarrow{\quad m \quad} \mathfrak{s}' \qquad\qquad \mathfrak{s} \xrightarrow{\quad m \quad} \lightning$$

always keeping in mind that $\mathfrak{s}$, $\mathfrak{s}'$ and $\lightning$ are merely labels attached to the (unnamed) nodes of the graph G.

2.4 Concurrent transition systems

The transition systems we have described above are inadequate to interpret shared memory programs for two reasons: First, the treatment of data-races is impossible because the semantics of a program which we have sketched so far does not track which pairs of instructions are executed in parallel. Second, the semantics of programs are not stable under an environment which is executing concurrently with the program, this precludes a compositional definition of the parallel product $C_1 \parallel C_2$ of two programs. To handle the first issue, we switch from using graphs to using asynchronous graphs for our transition systems and our machine models. For the second, we consider polarized transition systems which contain both transitions of the program and transitions of the environments.

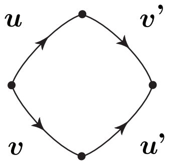
2.4.1 Asynchronous graphs

To solve these problems without abandoning the idea of using state transition systems based on graphs, we use asynchronous graphs, which are a notion of graphs with 2-dimensional information.

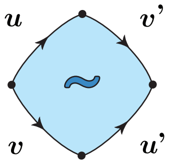
Definition 2.4.1. A graph $G = (V, E, \partial_-, \partial_+)$ consists in a set V of vertices, or nodes, a set E of edges, and two maps $\partial_-, \partial_+ : E \to V$ assigning a source and a target vertex to each edge, respectively.

Given a graph G, a ***square*** in G is a pair of paths of length 2 in G with a common source vertex and a common target vertex.

An ***asynchronous graph*** $\mathcal{G} = (G, \diamond)$ is a graph G equipped with a tuple $\diamond = (T, \partial_\diamond, \sigma)$ of a set T, the elements of which we call ***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 $\partial_\diamond(t) = (f, g)$, then $\partial_\diamond(\sigma(t)) = (g, f)$. A permutation tile mapped to a square $(u \cdot v', v \cdot u')$, where u, u', v, v' are edges of the graph G which form a square



is depicted as a 2-dimensional tile sitting on that square:



A square on which there are no tiles is called a ***hole***. We write $T : p \sim q$ to express that T is a tile which sits between the two paths p and q of length 2.

The intuition is that a tile such as the one depicted above witnesses that the transitions u and v are *independent*. The transitions u' and v' should be seen as being the same, respectively, as u and v, but executed from another state. In particular, the fact that they form a square means that the action of u and of v commute.

A morphism $\mathcal{F}$ between asynchronous graphs $\mathcal{G} = (G, \diamond_G)$ and $\mathcal{H} = (H, \diamond_H)$ is called an ***asynchronous morphism***. It is a pair $\mathcal{F} = (\mathcal{F}_G, \mathcal{F}_\diamond)$ of a graph homomorphism $\mathcal{F}_G : G \to G'$ and of a map $\mathcal{F}_\diamond : T_G \to T_H$ between their tiles such that it preserves borders:

$$\mathcal{F}_G \circ \partial_\diamond \ = \ \partial_\diamond \circ \mathcal{F}_\diamond, \qquad \mathcal{F}_G \circ \sigma \ = \ \sigma \circ \mathcal{F}_\diamond.$$

This definition of asynchronous graph, without uniqueness properties on tiles is motivated by the simplicity of the limits and colimits in the category **AsyncGraph** of asynchronous graphs and asynchronous morphisms. This is because the category **AsyncGraph** is the category of presheaves over the category presented by the following graph:

$$\begin{tikzcd}
{[0]} \arrow[r, bend left=10, "s"] \arrow[r, bend right=10, "t"'] & {[1]} \arrow[r, bend left=40, "dl"] \arrow[r, bend left=12, "dr"] \arrow[r, bend right=12, "ur"] \arrow[r, bend right=40, "ul"'] & {[2]} \arrow[loop right, "\sigma"]
\end{tikzcd}$$

and with the equations:

$$\begin{array}{llll}
dl \circ s = ul \circ s & dl \circ t = dr \circ s & dr \circ t = ur \circ t & ul \circ t = ur \circ s \\
\sigma \circ dl = ul & \sigma \circ ul = dl & \sigma \circ dr = ur & \sigma \circ ur = dr.
\end{array}$$

As such, both limits and colimits are computed object-wise. We detail below the case of pullbacks, and consequently of Cartesian products. The pullback of a cospan

$$\begin{tikzcd}
A_1 \arrow[r, "f"] & B & A_2 \arrow[l, "g"']
\end{tikzcd}$$

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 $(u_1, u_2) : (x_1, x_2) \to (y_1, y_2)$ are the pairs consisting of an edge $u_1 : x_1 \to y_1$ in A_1 and of an edge $u_2 : x_2 \to y_2$ 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.

2.4.2 Asynchronous machine models

We now add permutation tiles to the machine model $\pm_{\mathrm{S}}$ which we have defined above to express the behavior of instructions which are executed in parallel with respect to data-races. We will then define another machine model $\pm_{\mathrm{L}}$ which will contain the synchronization behavior of the instructions. First, we define the notion of machine state footprint which we will use to define data races and the permutation tiles in our machine models.

m	rd	wr	lk	mem
nop	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$x \coloneqq E$	$\mathsf{fv}(E)$	$\{x\}$	$\emptyset$	$\emptyset$
$x \coloneqq [E]$	$\mathsf{fv}(E) \cup \{E\}$	$\{x\}$	$\emptyset$	$\emptyset$
$[E] \coloneqq E'$	$\mathsf{fv}(E')$	$\{E\}$	$\emptyset$	$\emptyset$
$x \coloneqq \mathtt{alloc}(E, \ell)$	$\mathsf{fv}(E)$	$\{x\}$	$\emptyset$	$\{\ell\}$
$\mathtt{dispose}(E)$	$\mathsf{fv}(E)$	$\{E\}$	$\emptyset$	$\{E\}$
$P(r)$	$\emptyset$	$\emptyset$	$\{r\}$	$\emptyset$
$V(r)$	$\emptyset$	$\emptyset$	$\{r\}$	$\emptyset$

Table 2.1: Footprint of machine instructions

Data races

We define data races using the notion of footprint of a machine instruction which summarizes both the part of the memory and the set of locks which is accessed by the instruction.

Definition 2.4.2. A ***machine state footprint*** (with valid locks $\mathfrak{L}$)

$$\mathfrak{p} \;\in\; \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(\mathbf{Var} + \mathbf{Loc}) \times \wp(\mathfrak{L}) \times \wp(\mathbf{Loc})$$

contains: 1. $\mathsf{rd}(\mathfrak{p})$ the set of locations that are read, 2. $\mathsf{wr}(\mathfrak{p})$ the set of locations that are written, 3. $\mathsf{lk}(\mathfrak{p})$ the set of locks that are touched (either acquired or released), and 4. $\mathsf{mem}(\mathfrak{p})$ the set of locations which are allocated or deallocated.

Given a machine instruction m and a machine state $\mathfrak{s}$, we define the footprint $\mathfrak{p}_{\mathfrak{s}}(m)$ of m executed at $\mathfrak{s}$ in Table 2.1. The set of free program variables of an expression E is denoted by $\mathsf{fv}(E)$, and we write E for its value seen as a location in **Loc** when evaluated in the state $\mathfrak{s}$.

Informally, two instructions executing at the same time define a data race when their footprints are *not* independent:

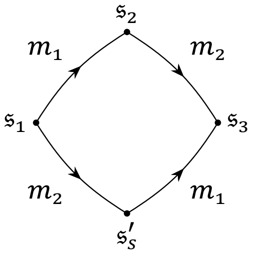
Definition 2.4.3. Two machine state footprints $\mathfrak{p}_1$ and $\mathfrak{p}_2$ are defined to be ***independent*** when:

$$\begin{aligned} (\mathsf{rd}(\mathfrak{p}_2) \cup \mathsf{wr}(\mathfrak{p}_2)) \cap \mathsf{wr}(\mathfrak{p}_1) &= \emptyset & \qquad \mathsf{lk}(\mathfrak{p}_1) \cap \mathsf{lk}(\mathfrak{p}_2) &= \emptyset \\ (\mathsf{rd}(\mathfrak{p}_1) \cup \mathsf{wr}(\mathfrak{p}_1)) \cap \mathsf{wr}(\mathfrak{p}_2) &= \emptyset & \qquad \mathsf{mem}(\mathfrak{p}_1) \cap \mathsf{mem}(\mathfrak{p}_2) &= \emptyset \end{aligned}$$

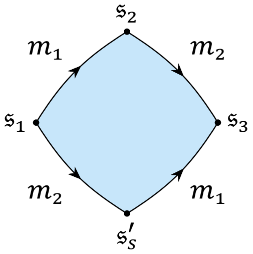
It may not be obvious why the last component $\mathsf{mem}(\mathfrak{p})$ of the footprint is needed. If two instructions `alloc` and `dispose` over the same memory location ℓ are executed, then there must have been some synchronization to propagate the information that the location ℓ is free or no longer free. Hence, the two instructions do not run in parallel. A dedicated component is needed so that the map from the stateful to the stateless semantics, which is defined bellow, is well defined.

Stateful machine model

We can now complete the definition of the stateful machine model $\pitchfork_S \in \mathbf{AsyncGraph}$ by defining its set of tiles. All the tiles will sit on squares of the form



which represent the two schedules of m_1 and m_2 being executed on two different threads. For every such square in $\pitchfork_S$, there is a unique tile



when $\mathfrak{p}_{\mathfrak{s}_1}(m_1)$ and $\mathfrak{p}_{\mathfrak{s}_1}(m_2)$ are independent.

Remark 2.4.4. This condition above does not depend on the state at which are computed the footprints: When the footprints are independent for one of the four states in the square, they are independent at the other three.

The stateful machine model contains the information of when two instructions executed in two different threads create a data race. We use a second machine model to encode the information of when two instructions are unsynchronized when run in two different threads.

Stateless machine model

We define the ***stateless machine model*** $ѫ_{\mathrm{L}}$, which is also parameterized by the set $\mathfrak{L}$ of available locks. Because its purpose is to embody the synchronization behavior of instructions in the language of asynchronous graphs, it needs to contains “more squares” than the stateful machine model $ѫ_{\mathrm{S}}$. Indeed, the fact that the actions of two instructions on the state of the machine commute with each other does not depend on whether these two instructions synchronize or not.

The solution to this problem is to use a coarser notion of state for the machine model $ѫ_{\mathrm{L}}$: a ***lock state***

$$L \in \wp(\mathfrak{L}) + ↯$$

only keeps track of the set of locks which are being held in the current state.

The edges of the asynchronous graph $ѫ_{\mathrm{L}}$ are of the form

$$L \xrightarrow{m} L'$$

where m is a ***lock instruction***

$$m ::= P(r) \mid V(r) \mid \mathtt{alloc}(\ell) \mid \mathtt{dispose}(\ell) \mid \mathsf{nop}.$$

The effect of a lock instruction on a lock state is simple: $\mathtt{alloc}(\ell)$, $\mathtt{dispose}(\ell)$ and nop act as the identity, and there are transitions

$$L \xrightarrow{P(r)} L \uplus \{r\} \qquad\qquad L \uplus \{r\} \xrightarrow{V(r)} L$$

Moreover, since lock states do not contain enough information about whether fallible instructions crash, we add pessimistically the following transitions:

$$\forall L, \forall m, \quad L \xrightarrow{m} ↯$$

In order to define the tiles of the asynchronous graph $ѫ_{\mathrm{L}}$, we define the notion of stateless footprint.

Definition 2.4.5. A ***stateless footprint***

$$\mathfrak{p} \in \wp(\mathfrak{L}) \times \wp(\mathbf{Loc})$$

is the data of a set $\mathsf{lk}(\mathfrak{p})$ of locks, and of a set $\mathsf{mem}(\mathfrak{p})$ of memory locations. Two such footprints $\mathfrak{p}_1$ and $\mathfrak{p}_2$ are ***independent*** when

$$\mathsf{lk}(\mathfrak{p}_1) \cap \mathsf{lk}(\mathfrak{p}_2) = \emptyset \qquad\qquad \mathsf{mem}(\mathfrak{p}_1) \cap \mathsf{mem}(\mathfrak{p}_2) = \emptyset.$$

A square

$$\begin{tikzcd}
& L_2 \arrow[dr, "m_2"] & \\
L_1 \arrow[ur, "m_1"] \arrow[dr, "m_2"'] & & L_3 \\
& L'_s \arrow[ur, "m_1"'] &
\end{tikzcd}$$

in the asynchronous graph $ⵜ_{\mathrm{L}}$ is then associated with a unique tile when the footprints of m_1 and of m_2 are independent.

Relationship between the models

The two models are constructed in such a way that there exists an asynchronous morphism

$$\mathcal{L}_{ⵜ} \;:\; ⵜ_{\mathrm{S}} \;\longrightarrow\; ⵜ_{\mathrm{L}}$$

defined on nodes by projecting out the lock state out of the machine state

$$\mathcal{L}(\mu, L) \coloneqq L$$

and on edges as follows.

$$\begin{array}{ccc}
(\mu, L) \xrightarrow{P(r)} (\mu, L') & \mapsto & L \xrightarrow{P(r)} L' \\
(\mu, L) \xrightarrow{V(r)} (\mu, L') & \mapsto & L \xrightarrow{V(r)} L' \\
(\mu, L) \xrightarrow{\mathtt{alloc}(E,\ell)} (\mu, L') & \mapsto & L \xrightarrow{\mathtt{alloc}(\ell)} L' \\
(\mu, L) \xrightarrow{\mathtt{dispose}(E)} (\mu, L') & \mapsto & L \xrightarrow{\mathtt{dispose}([\![E]\!])(\mu)} L' \\
(\mu, L) \xrightarrow{m} (\mu', L') & \mapsto & L \xrightarrow{\mathsf{nop}} L'
\end{array}$$

where, in the last case, m is any instruction which is not mentioned above. A rapid inspection of the definitions of the stateful machine model $ⵜ_{\mathrm{S}}$ and the stateless machine model $ⵜ_{\mathrm{L}}$ shows that the map $\mathcal{L}$ maps tiles to tiles, and hence defines an asynchronous morphism.

2.4.3 Transition systems

We can now refine the way we interpret programs which we sketched in Section 2.3. A state transition system is a pair $(G, \lambda : G \to \pm)$ of an asynchronous graph and of an asynchronous morphism $\lambda : G \to \pm$ from G to a machine model $\pm$. The machine model $\pm$ contains all the information which is intrinsic to the behavior of instructions: their effect on state, and how pairs of instructions behave when they run in parallel. The asynchronous graph G on the other hand, expresses how and when instructions are dispatched, adding the constraints of the program to their inherent constraints in the machine model.

Another way to look at it is that the program adds *program-order* dependencies to instructions to the dependencies such as *synchronizes-with* which are part of the specification of instructions. The property that the program can only *add* dependencies corresponds to the requirement that an asynchronous morphism sends tiles in the program (intuitively, non-dependent transitions) to tiles in the machine model.

To interpret programs with fine-grained information about their concurrent behavior using asynchronous graphs, we interpret each program with two different machine models. As we will see in the next section, the stateful interpretation $[\![C]\!]_{\mathrm{S}}$ and the stateless interpretation $[\![C]\!]_{\mathrm{L}}$ will be related by a map $\mathcal{L}$, resulting in a commutative diagram in the category **AsyncGraph** of the form:

$$\begin{tikzcd}
{[\![C]\!]_{\mathrm{S}}} \arrow[d, "\mathcal{L}"'] & & G_S \arrow[r, "\lambda_S"] \arrow[d, "\mathcal{L}"'] & \pm_{\mathrm{S}} \arrow[d, "\mathcal{L}_{\pm}"] \\
{[\![C]\!]_{\mathrm{L}}} & = & G_L \arrow[r, "\lambda_L"] & \pm_{\mathrm{L}}
\end{tikzcd} \tag{2.1}$$

To demonstrate why this double interpretation is useful, we describe data-races in this semantics.

2.4.4 Data-races

Recall that a data-race happens when two instructions are executed in parallel and touch the same area of memory, with at least an instruction writing. In particular, we need to know when two instructions are executed in parallel without synchronization. This corresponds in our semantics to a tile in the asynchronous graph G_L, with the notations of diagram (2.1) above.

Now, we need to characterize the second half of the definition of a data race, which says that the two instructions are in conflict. Since this is defined with respect to the

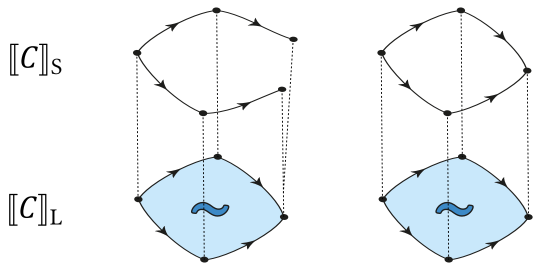
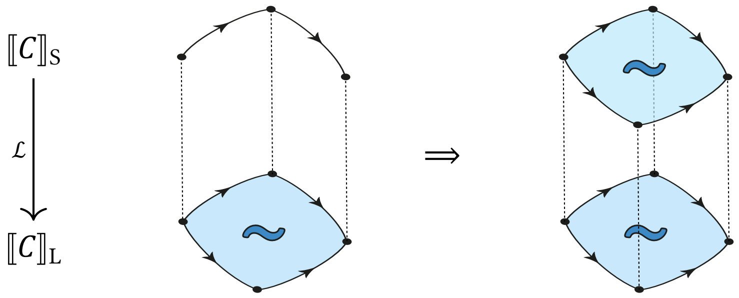


Figure 2.2: A data race in the interpretation of a program

memory, this is a property of the stateful interpretation $\lambda_S : G_S \to \pitchfork_S$. There are two cases. If the instructions involved in the data-race do not commute, for example $x := 4$ and $x := 5$, then there will be a "fork" mapped under $\mathcal{L}$ to the tile in G_L as depicted on the left of Figure 2.2. The second case occurs when the two instructions happen to commute, but still produce a data-race, for example $x := 1$ in parallel with $x := 1$. In that case there will be a hole, since there a tile in $\pitchfork_S$ only when there is no data-race, which is mapped to the tile in G_L; this situation is depicted on the right of Figure 2.2.

Because our goal is to prove the *absence* of data races, we need to prove the negation of the above: for every tile in G_L which is part of an execution, there exists a tile in G_S which is mapped to that tile under the map $\mathcal{L} : G_S \to G_L$. The condition that the tile is part of an execution is necessary because the stateless semantics $[\![C]\!]_L$ contains nonsensical executions; the stateless semantics is a tool to for adding *local* information to transitions in the stateful semantics $[\![C]\!]_S$.

More precisely, a set of executions of $[\![C]\!]_S$ is data race-free if, whenever two consecutive transitions $m_1 \cdot m_2$ of some execution are mapped under $\mathcal{L}$ to the edge of some tile (depicted below left), the tile can be lifted along $\mathcal{L}$ to a tile whose edge is $m_1 \cdot m_2$ (below right):



2.4.5 Polarized asynchronous transition systems

We now explain how to solve the second problem outlined at the beginning of Section 2.4: the interpretation of programs should be closed under the action of the environment. For this, we explicitly add Environment transitions in the machine models and in the program interpretations so that any possible interference of the environment is accounted for.

First, we adapt our stateful and stateless machine models to have two copies of each edge, which we write

$$\mathfrak{s} \xrightarrow{m:\mathbf{C}} \mathfrak{s}' \qquad\qquad \mathfrak{s} \xrightarrow{m:\mathbf{F}} \mathfrak{s}'$$

for, respectively, a transition of the Code, and a transition of the Frame (or Environment). We write $\pm_{\mathrm{S}}^{\circ\bullet}$ and $\pm_{\mathrm{L}}^{\circ\bullet}$ for the version with two polarities. To make the notation more explicit about the number of polarities, we write the unpolarized machine models as $\pm_{\mathrm{S}}^{\bullet}$ and $\pm_{\mathrm{L}}^{\bullet}$ in the sequel instead of notation $\pm_{\mathrm{S}}$ and $\pm_{\mathrm{L}}$ which we have been using until now.

A transition system is therefore a pair (G, λ) of an asynchronous graph G and of an asynchronous morphism

$$\lambda \ : \ G \ \longrightarrow \ \pm^{\circ\bullet}$$

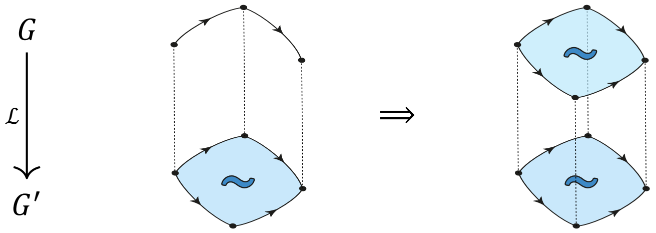
Because the transitions with the Frame polarity F need to account for every possible transition by the Environment, there must be “enough” Frame transitions in G. From the point of view of the Frame, in any state x in G, the set of transitions available to the Frame must be the same as the set of transitions available from $\lambda(x) \in \pm^{\circ\bullet}$. This is made formal by stating that λ is an Environment 1-fibration.

Definition 2.4.6. Let $f : G \to G'$ be an asynchronous morphism between two asynchronous graphs G and G'. Let $\mathcal{M}$ be a set of edges in G'.

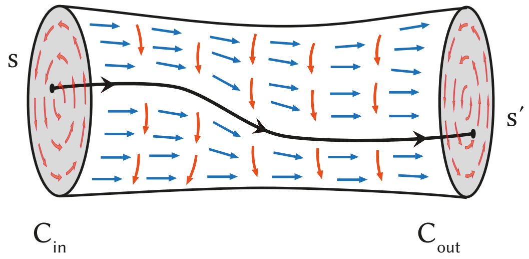
The asynchronous morphism f is called an ***$\mathcal{M}$-1-fibration*** if, for all $x \in G$, and for all edges $e' : f(x) \to y'$ in $\mathcal{M}$, there exists an edge $e : x \to y$ in G such that $f(y) = y'$ and $f(e) = e'$, as depicted below. When there is a notion of Code or Environment edges, this defines the notions of ***Code 1-fibration*** and ***Environment 1-fibration***.

$$\begin{tikzcd}
x \arrow[d, dotted, dash, "f"'] & & & & x \arrow[r] \arrow[d, dotted, dash, "f"'] & y \arrow[d, dotted, dash, "f"] \\
f(x) \arrow[r] & y' & \Longrightarrow & & f(x) \arrow[r] & y'
\end{tikzcd}$$

The asynchronous morphism f is called a ***2-fibration*** if, for all paths $p : x \twoheadrightarrow z$ of length 2 in G, and for all tiles $T' : f(p) \sim q'$ in G', there exists a tile $T : p \sim q$ in G such that $f(q) = q'$ and $f(T) = f(T')$.



Such a transition system, which interpret a program C can be depicted as follows:



At each state, there are many Environment transitions, depicted in red. The Code transitions, the Code transitions, make the execution of the program progress. A path from an initial state on the left to a final state on the right will begin with any number of Frame transitions, followed by one Code transitions, followed by any number of Frame transitions, and so on.

We finally have a suitable definition of a transition system.

Definition 2.4.7. A transition system $(G, \lambda, G_{\mathrm{in}}, G_{\mathrm{out}})$ over the machine model $\pm \in \{\pm_{\mathrm{S}}, \pm_{\mathrm{L}}\}$ is the data of:

1. an asynchronous morphism $\lambda : G \to \pm^{\circ\bullet}$, which is an Environment 1-fibration;
2. two sub-asynchronous graphs $G_{\mathrm{in}}, G_{\mathrm{out}} \subseteq G$ containing only Frame transitions denoting the initial and final states of the program.

This definition will be reformulated in a more categorical and convenient language in the next chapter. The basic idea is that a transition system as above can be seen as a cospan

$$\begin{tikzcd}
G_{\mathrm{in}} \arrow[dr, "in"] & & G_{\mathrm{out}} \arrow[dl, "out"'] \\
& G &
\end{tikzcd}$$

in the category of asynchronous graphs. These asynchronous graphs will be organized to be horizontal morphisms in a certain double category.

Moreover, following Melliès methodology of *template games* (Melliès, 2019), but in a dual setting, with cospans instead of spans, the spans we consider are "typed" using *templates* ᛦ:

$$\begin{tikzcd}
G_{\text{in}} \arrow[rd, "in"] \arrow[d, "\lambda_I"'] & & G_{\text{out}} \arrow[ld, "out"'] \arrow[d, "\lambda_O"] \\
ᛦ[0] \arrow[rd] & G & ᛦ[0] \arrow[ld] \\
& ᛦ[1] &
\end{tikzcd}$$

The purpose of the templates is that every operation which we will define on transition systems will be induced by structures at the level of templates. This has the practical advantage that the templates are simple objects, and so are easier to manipulate.

2.5 Concurrent separation logic

In the previous section, we have explained how to interpret programs as transition systems, and we have sketched the formal statement of the data-race freedom property we wish to prove about programs which have been proved using CSL.

In this section, we describe the version of CSL which we consider, and we explain how to interpret proofs, that is, proof trees, of CSL as transition systems over an appropriate machine model ᛦ$_{\text{Sep}}$. We conclude the section and the chapter by stating the asynchronous soundness theorem of the logic.

2.5.1 The logic

The version of CSL which we consider here is the same as the original papers by P. O'Hearn (2007) and Brookes (2004) as well as the first operational soundness proof, by Vafeiadis (2011). The main differences is that the version of CSL we consider has permissions which were introduced for separation logic by Bornat, Calcagno, P. O'Hearn, et al. (2005), following their introduction by Boyland (2003). The purpose of permissions allow the logic to prove programs where several threads read memory without synchronization. We also use a variant of the "variables as resources" descipline introduced by Bornat, Calcagno, and Yang (2006) to remove side conditions to the rules of the logic such as the inference rule for the parallel product and the frame rule.

The judgments of CSL are Hoare triples of the form

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

where P and Q are formulas of CSL, and $\Gamma = r_1 : J_1, \ldots, r_n : J_n$ is a context associating a formula J_i to each lock r_i.

Formulas of CSL

Formulas denote predicates over a logical version of the memory augmented with permissions. Their syntax is the following:

$$\begin{aligned} P, Q, R, J ::= {} & \mathbf{emp} \mid P * Q \mid \top \mid \bot \mid P \vee Q \mid P \wedge Q \mid \neg P \\ & \mid \forall a.P \mid \exists a.P \mid\ v \overset{p}{\mapsto} w \mid \mathrm{Own}_p(x) \mid E'_1 = E'_2 \end{aligned}$$

where $x \in \mathbf{Var}$, $p \in \mathbf{Perm}$ is a permission, $v, w \in \mathbf{Val}$, and the E'_i are arithmetic expressions which can contain logic variables a, b, c. The idea behind permissions is that a permission $p \in \mathbf{Perm}$ represents a degree of ownership of a memory cell either in the heap through the atomic formula $x \overset{p}{\mapsto} v$ or in the stack with $\mathrm{Own}_p(x)$. Any permission is enough to *read* the memory, but only the total permission $\top$ allows for writing to the memory. This leads to the following definition.

Definition 2.5.1. A ***permission monoid*** **Perm** is a partial cancellative commutative semi-group with a distinguished element $\top \in \mathbf{Perm}$ called the ***total permission*** of **Perm** which allows no multiple:

$$\forall x \in \mathbf{Perm},\ x + \top \text{ is not defined.}$$

This implies in particular that **Perm** has no neutral element. Cancellativity means that for all elements $x, y, z \in \mathbf{Perm}$, if $x + y = x + z$ (and both sides of the equality are well defined), then $y = z$.

Of course, such permission monoids are partial semi-groups, but this name is traditional.

The canonical example of permission monoid are the fractional permissions $(0, 1]$ of the additive monoid $\mathbb{Q}$ restricted to values greater that 0 and less than or equal to 1. It is easy to check that 1 is a total permission since, for any $x > 0$, $x + 1$ is greater that 1 and thus not defined in the partial monoid of fractional permissions.

Since the formulas can specify the level of permission over pieces of memory, they cannot be interpreted as predicates over memory states. Instead, they are predicates over logical states.

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

Figure 2.3: Satisfaction of CSL formulas

Definition 2.5.2. A ***logical state*** $\sigma \in \mathbf{LogState}$ is a pair (s,h) of a logical stack $s \in \mathbf{Var} \rightharpoonup_{fin} (\mathbf{Val} \times \mathbf{Perm})$ and a logical heap $h \in \mathbf{Loc} \rightharpoonup_{fin} (\mathbf{Val} \times \mathbf{Perm})$ which associate to each variable and each heap location a value and a permission.

Permissions for a resource can be split and combined:

$$x \overset{p+q}{\mapsto} v \ \dashv\vdash \ x \overset{p}{\mapsto} v * x \overset{q}{\mapsto} v$$

therefore, to interpret the separating conjunction, we need an operator to play the role of the disjoint union of heaps in a way that takes permissions into account. Reading the equivalence above tells us that we can combine logical states which contain the same values, and in that case the permissions are added (if the sum is well-defined of course).

Definition 2.5.3. Given two logical states σ_1 and σ_2, their product $\sigma_1 * \sigma_2$, if it is defined, is a logical state whose domain is the union of the domains of σ_1 and of σ_2. For $a \in \mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2) \subseteq \mathbf{Var} \uplus \mathbf{Loc}$,

$$
(\sigma_1 * \sigma_2)(a) \coloneqq \begin{cases} \sigma_1(a) & \text{if } a \in \mathrm{dom}(\sigma_1) \setminus \mathrm{dom}(\sigma_2) \\ \sigma_2(a) & \text{if } a \in \mathrm{dom}(\sigma_2) \setminus \mathrm{dom}(\sigma_1) \\ (v, p + p') & \text{if } \sigma_1(a) = (v,p) \text{ and } \sigma_2(a) = (v,p') \end{cases}
$$

If any the right hand-side is undefined for any a, then the product $\sigma_1 * \sigma_2$ as a whole is undefined.

We define the satisfaction relation $\sigma \vDash P$ in Figure 2.3, using the operation above for the case of the separating conjunction. We write $\mathrm{dom}(\sigma) \subseteq \mathbf{Var} \uplus \mathbf{Loc}$ for the complete domain of the state σ, and $\mathrm{vdom}(\sigma)$ for the domain of its stack component s. The formula $v \overset{p}{\mapsto} w$ is only satisfied for logical states of the form $\sigma = (\emptyset, [v \mapsto (w, p)])$, similarly, the formula $\mathrm{Own}_p(x)$ is only satisfied by singleton stacks whose domains are $\{x\}$. The value of variables is specified using the predicates $E_1 = E_2$, which can only be true if the heap contains all the variables mentionned in E_1 and E_2. Therefore, one can use the formula $\mathrm{Own}_1(x) \wedge x = 5$ to state that x must contain the value 5. As expected, $P * Q$ holds for σ if the logical state σ can be split using the operation of Definition 2.5.3 in two logical states σ_1 and σ_2 which satisfy P and Q respectively.

Given the satisfaction relation, we can define the entailment of predicates as

$$P \Rightarrow Q \qquad \Leftrightarrow \qquad \forall \sigma,\ \sigma \vDash P \ \Rightarrow\ \sigma \vDash Q.$$

Inference rules of CSL

Hoare triples $\Gamma \vdash \{P\}\, C\, \{Q\}$ of CSL are constructed by building derivation trees, or proof trees, using the inference rules presented in Figure 2.4. The first premise of the rule If, $P \Rightarrow \mathrm{def}(B)$ states that if P holds for some logical state σ, then all the program variables appearing in B are owned with some permission.

The rules Aff for writing into variables and St for writing into the heap require full permission over the respective memory resources; while we only require partial permission for reading, implicitly for variables —otherwise equalities involving expressions would be false— and explicitly in rule Ld for heap locations. The other rules are standard and were explained in Section 0.1.2.

2.5.2 Semantic interpretation of CSL

Our method for proving the soundness of the logic we describe above is to interpret proof trees of CSL as transition systems over an adequate machine model. The first task is to define the notion of state which will be used by the machine model.

The predicates P which appear in the Hoare triples of the logic describe the logical memory which is owned by the Code. Since we have chosen to describe explicitly the actions of the Environment, or Frame, in our transition systems, our states need to contain the (logical) memory which is owned by the Frame.

This dichotomy does not account for all the memory: shared resources are owned by the locks, and therefore are owned by neither the Code nor the Frame. These states will keep track, for each lock, of the logical state it owns or the identity of its holder (Code or Frame). Putting all of this together, we define the notion of state which we will use to interpret proofs of CSL.

$$\dfrac{}{\Gamma \vdash \{(\mathrm{Own}_{\top}(x) * P) \land E = v\}\; x \coloneqq E\; \{(\mathrm{Own}_{\top}(x) * P) \land x = v\}}\;\textsc{Aff}$$

$$\dfrac{}{\Gamma \vdash \{(P \land E = w \land E' = v) * w \overset{\top}{\mapsto} -\}\; [E] \coloneqq E'\; \{(P \land E = w \land E' = v) * E \overset{\top}{\mapsto} w\}}\;\textsc{St}$$

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

$$\dfrac{P \Rightarrow \mathrm{def}(B) \quad \Gamma \vdash \{I \land B\}\; C\; \{I\}}{\Gamma \vdash \{I\}\; \texttt{while}\; B\; \texttt{do}\; C\; \{I \land \neg B\}}\;\textsc{While}$$

$$\dfrac{}{\Gamma \vdash \{((\mathrm{Own}_{\top}(x) * P) \land E = w) * w \overset{p}{\mapsto} v\}\; x \coloneqq [E]\; \{((\mathrm{Own}_{\top}(x) * P) \land x = v) * w \overset{p}{\mapsto} v\}}\;\textsc{Ld}$$

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

$$\dfrac{\Gamma, r : J \vdash \{P\}\; C\; \{Q\}}{\Gamma \vdash \{P * J\}\; \texttt{resource}\; r\; \texttt{do}\; C\; \{Q * J\}}\;\textsc{Res} \qquad \dfrac{\Gamma \vdash \{(P * J) \land B\}\; C\; \{Q * J\}}{\Gamma, r : J \vdash \{P\}\; \texttt{with}\; r\; \texttt{do}\; C\; \{Q\}}\;\textsc{With}$$

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

$$\dfrac{P \Rightarrow P' \quad \Gamma \vdash \{P'\}\; C\; \{Q'\} \quad Q' \Rightarrow Q}{\Gamma \vdash \{P\}\; C\; \{Q\}}\;\textsc{Conseq}$$

Figure 2.4: Inference rules of CSL

Definition 2.5.4. Given a CSL context Γ, a ***separated state*** is a triple

$$(\sigma_C, \vec{\sigma}, \sigma_F) \in \mathbf{SepState}(\Gamma)$$

with $\sigma_C, \sigma_F \in \mathbf{LogState}$, and $\vec{\sigma} : \mathsf{dom}(\Gamma) \to \mathbf{LogState} + \{\mathbf{C}, \mathbf{F}\}$, and which satisfies two properties:

1. Locked resources satisfy their invariants: $\forall r \in \mathsf{unlocked}(\vec{\sigma}),\ \vec{\sigma}(r) \vDash \Gamma(r)$;
2. The components are coherent: the logical state below is defined:

$$\sigma_C * \left\{ \bigcircledast_{r \in \mathsf{dom}(\vec{\sigma})} \vec{\sigma}(r) \right\} * \sigma_F \quad \in \quad \mathbf{LogState} \tag{2.2}$$

A separated state $(\sigma_C, \vec{\sigma}, \sigma_F)$ defines as machine state $\circledast(\sigma_C, \vec{\sigma}, \sigma_F) = (\mu, L)$ whose memory state μ is obtained by forgetting permissions in the logical state (2.2) above, and whose set L of available locks is $\mathsf{unlocked}(\vec{\sigma})$.

Unlike the two machine models $\pitchfork_{\mathrm{S}}$ and $\pitchfork_{\mathrm{L}}$ which we use to interpret programs, the machine model for the proofs distinguishes between transitions of the Code and of the Frame. It is defined as follows.

Definition 2.5.5. The ***machine model of separated states*** $\pitchfork^{\circ\bullet}_{\mathrm{Sep}}(\Gamma)$ parameterized by a CSL context Γ is an asynchronous graph whose nodes are the separated states over the context Γ and with Code transitions

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

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

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

is a transition in $\pitchfork^{\bullet}_{\mathrm{S}}$, and such that the following conditions are satisfied:

$$\begin{array}{ll} \forall \ell \notin \mathsf{wr}(m),\ \sigma_C(\ell) = \sigma'_C(\ell) & \mathsf{wr}(m) \cup \mathsf{rd}(m) \subseteq \mathsf{dom}(\sigma_C) \\ \mathsf{lk}(m) \subseteq \mathsf{dom}(\vec{\sigma}) \cup \vec{\sigma}^{-1}(\mathbf{C}) & \forall r \notin \mathsf{lk}(m), \vec{\sigma}(r) = \vec{\sigma}'(r). \end{array}$$

Frame transitions are of the form $(\sigma_C, \vec{\sigma}, \sigma_F) \xrightarrow{m:\mathbf{F}} (\sigma_C, \vec{\sigma}', \sigma'_F)$ with symmetric conditions:

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

is a transition in $\pitchfork^{\bullet}_{\mathrm{S}}$ and

$$\begin{array}{ll} \forall \ell \notin \mathsf{wr}(m),\ \sigma_F(\ell) = \sigma'_F(\ell) & \mathsf{wr}(m) \cup \mathsf{rd}(m) \subseteq \mathsf{dom}(\sigma_F) \\ \mathsf{lk}(m) \subseteq \mathsf{dom}(\vec{\sigma}) \cup \vec{\sigma}^{-1}(\mathbf{F}) & \forall r \notin \mathsf{lk}(m), \vec{\sigma}(r) = \vec{\sigma}'(r). \end{array}$$

The tiles are defined in the same way as in the stateful machine model, using the same notion of footprint: a square in $\mp_{\mathrm{Sep}}$ of the form:

$$\begin{tikzcd} & b \arrow[dr, "m_2"] & \\ a \arrow[ur, "m_1"] \arrow[dr, "m_2"'] & & c \\ & b' \arrow[ur, "m_1"'] & \end{tikzcd}$$

has a unique tile if the footprint of m_1 and of m_2 computed at the machine state $\circledast a$ are independent.

This new machine model of separated states induces a notion of asynchronous transition system which can be used to interpret proofs trees of CSL by induction of the inference rules. A proof $\pi : \Gamma \vdash \{P\}\ C\ \{Q\}$ is interpreted as a pair $(G_\pi, \lambda_\pi, G_{\mathrm{in}}, G_{\mathrm{out}})$ where G_π is an asynchronous graph, and λ_π is an asynchronous morphism

$$\lambda_\pi \ : \ G_\pi \ \longrightarrow \ \mp^{\circ\bullet}_{\mathrm{Sep}}$$

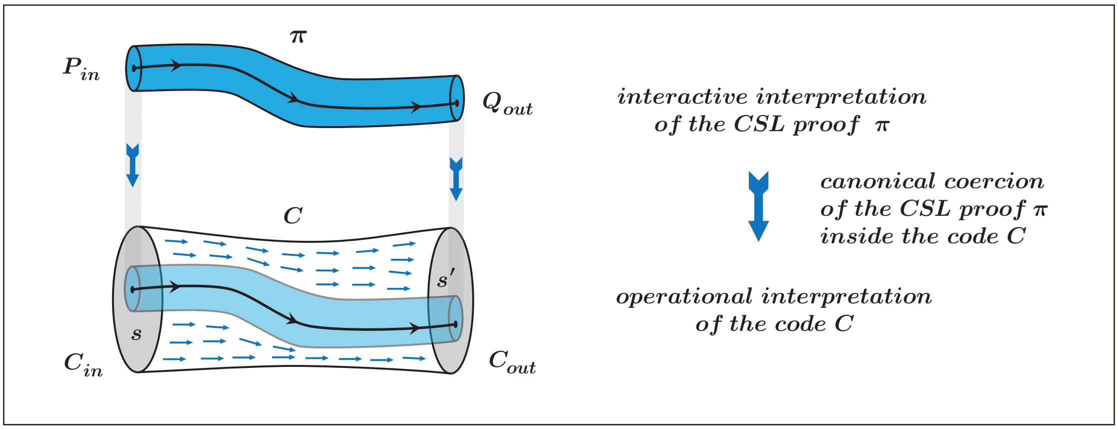
which maps all initial states in $G_{\mathrm{in}} \subseteq G_\pi$ to separated states $(\sigma_C, \vec{\sigma}, \sigma_F)$ which satisfy the precondition P (in that $\sigma_C \vDash P$) and, symmetrically, all final states in $G_{\mathrm{out}} \subseteq G_\pi$ are mapped to separated states which satisfy the postcondition Q.

In the same way that there is a map from the stateful machine model to the stateless machine model $\mathcal{L}_{\mp} : \mp^{\circ\bullet}_{\mathrm{S}} \to \mp^{\circ\bullet}_{\mathrm{L}}$, the operation $(\sigma_C, \vec{\sigma}, \sigma_F) \mapsto \circledast(\sigma_C, \vec{\sigma}, \sigma_F)$ defines an asynchronous morphism $\mathcal{S}_{\mp} : \mp^{\circ\bullet}_{\mathrm{Sep}} \to \mp^{\circ\bullet}_{\mathrm{S}}$.

As it turns out, we will see in the next chapter that, if π is a proof of the Hoare triple $\Gamma \vdash \{P\}C\{Q\}$, the interpretation of π and the two interpretations of the program C are related as follows, completing the diagram (2.1) page 50:

$$\begin{tikzcd} \llbracket \pi \rrbracket_{\mathrm{Sep}} \arrow[d, "\mathcal{S}"'] & G_\pi \arrow[r, "\lambda_\pi"] \arrow[d, "\mathcal{S}"'] & \mp_{\mathrm{Sep}} \arrow[d, "\mathcal{S}_{\mp}"] \\ \llbracket C \rrbracket_{\mathrm{S}} \arrow[d, "\mathcal{L}"'] & G_S \arrow[r, "\lambda_S"] \arrow[d, "\mathcal{L}"'] & \mp_{\mathrm{S}} \arrow[d, "\mathcal{L}_{\mp}"] \\ \llbracket C \rrbracket_{\mathrm{L}} & G_L \arrow[r, "\lambda_L"] & \mp_{\mathrm{L}} \end{tikzcd}$$

The map $\mathcal{S}$ induces a sub-asynchronous graph of $\llbracket C \rrbracket_{\mathrm{S}}$ of the executions which are supported by the CSL proof:


π
P_{in}
Q_{out}
C
s
s'
C_{in}
C_{out}
interactive interpretation of the CSL proof π
canonical coercion of the CSL proof π inside the code C
operational interpretation of the code C

This sub-asynchronous graph cannot be characterized at the level of the stateful semantics, it is only makes sense seen as the paths of $[\![C]\!]_{\mathrm{S}}$ which are refined by the proof π. This is the basis of the asynchronous soundness theorem of CSL.

2.5.3 The asynchronous soundness theorem

The soundness theorem of CSL states that programs which have been proved with the logic are safe, satisfy their postconditions if they terminate, and do not produce any data-race. In our setting, these properties are expressed as properties of the maps

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

which relate the interpretation $[\![\pi]\!]_{\mathrm{Sep}}$ of a proof π of the Hoare triple $\Gamma \vdash \{P\}\ C\ \{Q\}$ and the stateful $[\![C]\!]_{\mathrm{S}}$ and stateless $[\![C]\!]_{\mathrm{L}}$ interpretations of the program C. This property is expressed using the language of fibrations, which we defined in Definition 2.4.6. First, the safety of the program is expressed as a property about $\mathcal{S}$:

Theorem 2.5.6. *The asynchronous morphism $\mathcal{S}$ is a Code 1-fibration.*

Unfolding the definitions, this means that for all Code transitions

$$\mathfrak{s}_1 \xrightarrow{m} \mathfrak{s}_2$$

in $[\![C]\!]_{\mathrm{S}}$, if there is a separated state $(\sigma_C, \vec{\sigma}, \sigma_F)$ which corresponds to the machine state $\mathfrak{s}_1$ thought $\mathcal{S}$, then there exists a transition

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

with $\circledast(\sigma'_C, \vec{\sigma}', \S_F) = \mathfrak{s}_2$. In particular, since the image of the $\circledast$ operation is always a well-defined machine state and not the error state $\lightning$, this means that the operation m did not fail.

There are two ways to understand why this implies the safety of the whole program. The first is that all Code transitions preserve the safety of the program, and that it is up to the Environment to make sure that its transitions follow the discipline of CSL and preserve the correctness of the program. The other point of view is that, given a closed and complete program, one can forbid actions from the Environment. Then the asynchronous morphism $\mathcal{S}$ guarantees that any path which starts in a machine state which corresponds to a separated state which satisfies the precondition P does not contain the error state $\lightning$.

Now, let us state the second part of the soundness theorem, which states that there are no data-races in well-specified programs.

Theorem 2.5.7. *The asynchronous morphism $\mathcal{L} \circ \mathcal{S}$ is a 2-fibration.*

This means that whenever there is a tile in the stateless semantics $[\![C]\!]_{\mathrm{L}}$ of the program C, meaning that two instructions are executed in parallel and do not synchronize between each other, if one of the edges of the tile corresponds to a path in the semantics of the proof $[\![\pi]\!]_{\mathrm{Sep}}$, which means that this path corresponds to a piece of execution which follows the memory management rules of CSL, then there is a tile completing the path in $[\![\pi]\!]_{\mathrm{Sep}}$.

The asynchronous morphism $\mathcal{S}$ maps this tile in $[\![\pi]\!]_{\mathrm{Sep}}$ into a tile in $[\![C]\!]_{\mathrm{S}}$ which, by definition of tiles in $\pitchfork_{\mathrm{S}}$, means that there is no data-race between the two instructions.

3 An asynchronous template game model of CSL

Our semantic interpretation of CSL is based on the methodology of ***templates*** (Melliès, 2019). The general idea is that the interpretations live in a semantic domain which is parameterized by a template ⫛. In the work of Melliès (2019) for example, a template ⫛ is defined to be an internal category, which induces a bicategory of games **Games**(⫛) based on spans.

The slogan of this approach is that every operation at the level of the interpretation, say tensor products, should be induced by a *structure* at the level of templates, for tensors: span-monoidal internal categories in (Melliès, 2019).

In this work, our notion of template is almost dual, as they are a generalization of internal opcategories, and programs and proofs of CSL are interpreted as cobordisms, which are based on cospans.

3.1 The double category Cob(⫛) of games and cobordisms

3.1.1 Internal categories

Internal categories are the result of spelling out the definition of a category in the language of categories, in the same way that one can define a monoid object or a group object in any category with enough structures. We will use this notion in two ways: First, double categories are by definition weakly internal categories in the 2-category **Cat** of categories; and, second, they will be generalized and dualized in the sequel to define internal J-opcategories, the notion of template which our model will be based on.

To a first approximation, a category is a reflexive graph (or, more precisely, quiver): a set $\mathcal{C}[0]$ of objects, and a set $\mathcal{C}[1]$ of morphisms. Each morphism has a source and a target: this defines two maps $s, t : \mathcal{C}[1] \to \mathcal{C}[0]$. Similarly, each object $A \in \mathcal{C}[0]$ is associated with an identity morphism $\mathrm{Id}_A : A \to A$, this defines a map $e : \mathcal{C}[0] \to \mathcal{C}[1]$. One then needs to express that, for instance, the source and target of the identity on A are equal to A, and to formalize the composition operator. This leads to the following definition, discovered by Ehresmann (1963).

Definition 3.1.1. We consider a category $\mathbb{S}$ with finite limits. An ***internal category*** $\mathcal{C}$ in $\mathbb{S}$ contains the following data. Two objects $\mathcal{C}[0]$ and $\mathcal{C}[1]$ of $\mathbb{S}$ which we call the space of objects and the space of morphisms of $\mathcal{C}$ respectively. Morphisms $s, t : \mathcal{C}[1] \to \mathcal{C}[0]$ and $e : \mathcal{C}[0] \to \mathcal{C}[1]$ as explained above. A composition morphism $m : \mathcal{C}[2] \to \mathcal{C}[1]$ which associates morphisms to composable pairs of morphisms defined as the pullback

$$\begin{tikzcd}
 & \mathcal{C}[2] \arrow[dl, "l"'] \arrow[dr, "r"] & \\
\mathcal{C}[1] \arrow[dr, "t"'] & pb & \mathcal{C}[1] \arrow[dl, "s"] \\
 & \mathcal{C}[0] &
\end{tikzcd}$$

In the category of sets, this corresponds to pairs of arrows such that the target of the first is equal to the source of the second.

This structure $\mathcal{C} = (\mathcal{C}[0], \mathcal{C}[1], s, t, e, m)$ must satisfy certain properties:

1. The identities have the right endpoints: $s \circ e = t \circ e = \mathrm{Id}_{\mathcal{C}[0]}$;
2. Morphism composition preserves endpoints: $s \circ l = s \circ m$ and $t \circ r = t \circ m$;
3. Identities are identities for composition: the following diagram commutes

$$\begin{tikzcd}
\mathcal{C}[0] \times_{\mathcal{C}[0]} \mathcal{C}[1] \arrow[r, "e \times_{\mathcal{C}[0]} \mathrm{Id}"] \arrow[dr, "\pi_2"'] & \mathcal{C}[2] \arrow[d, "m"] & \mathcal{C}[1] \times_{\mathcal{C}[0]} \mathcal{C}[0] \arrow[l, "\mathrm{Id} \times_{\mathcal{C}[0]} e"'] \arrow[dl, "\pi_1"] \\
 & \mathcal{C}[1] &
\end{tikzcd} \tag{3.1}$$

4. and composition must be associative: writing $\mathcal{C}[3] = \mathcal{C}[1] \times_{\mathcal{C}[0]} \mathcal{C}[1] \times_{\mathcal{C}[0]} \mathcal{C}[1]$ the space of triples of composable arrows, the following diagram must commute:

$$\begin{tikzcd}
\mathcal{C}[3] \arrow[r, "m \times_{\mathcal{C}[0]} \mathrm{Id}"] \arrow[d, "\mathrm{Id} \times_{\mathcal{C}[0]} m"'] & \mathcal{C}[2] \arrow[d, "m"] \\
\mathcal{C}[2] \arrow[r, "m"'] & \mathcal{C}[1]
\end{tikzcd} \tag{3.2}$$

As expected, a (small) category is an internal category in the category **Set** of sets and functions. We ask that the ambient category $\mathbb{S}$ has finite limits in the definition above as this is general enough for our purposes, in general only the pullbacks of s along t needs to exist. The instance we are most interested in are strict double categories, which are categories internal to **Cat**. More precisely, we need the weaker notion of (*pseudo*) double category, where the composition operator $m : \mathcal{C}[2] \to \mathcal{C}[1]$ is only associative up to an isomorphism. Accordingly, we define a ***weakly internal category*** in a 2-category $\mathbb{S}$ with finite limits to be the same as an internal category except that diagrams (3.1) and (3.2) only needs to commute up to isos which satisfy the pentagon and triangle identities of monoidal categories. See Appendix A.2 of the PhD thesis of Courser (2020) for the full definitions of double categories and related objects.

A nice characterization of internal categories, which is going to be useful to generalize internal categories to polyads, is the following:

Lemma 3.1.2. *An internal category in $\mathbb{S}$ is the same as a monad in the bicategory* **Span**($\mathbb{S}$) *of spans in $\mathbb{S}$. Equivalently, this is also the same as a lax pseudo-functor* $\mathbf{1} \to \mathbf{Span}(\mathbb{S})$, *where* **1** *is the terminal bicategory.*

In the same way the definition of a category was expressed in the language of categories, there is an internal notion of functor between internal categories:

Definition 3.1.3. An ***internal functor of categories*** F from $\mathcal{C}$ to $\mathcal{D}$, two internal categories in a category $\mathbb{S}$ with finite limits, is the data of two morphisms in $\mathbb{S}$ between the two spaces of objects and the two spaces to morphisms:

$$F[0] : \mathcal{C}[0] \longrightarrow \mathcal{D}[0] \qquad \text{and} \qquad F[1] : \mathcal{C}[1] \longrightarrow \mathcal{D}[1]$$

satisfying the expected conditions:

1. F acts coherently with respect to the source and target maps, in that the following diagram commutes:

$$\begin{tikzcd}
\mathcal{C}[0] \arrow[d, "{F[0]}"'] & \mathcal{C}[1] \arrow[l, "s"'] \arrow[r, "t"] \arrow[d, "{F[1]}"] & \mathcal{C}[0] \arrow[d, "{F[0]}"] \\
\mathcal{D}[0] & \mathcal{D}[1] \arrow[l, "s"'] \arrow[r, "t"] & \mathcal{D}[0]
\end{tikzcd}$$

2. The commutation of the diagram above induces a map $F[2] : \mathcal{C}[2] \to \mathcal{D}[2]$ by the universality of the pullback defining $\mathcal{D}[2]$. The following diagram must commute,

expressing the fact that the composite of the images is equal to the image of the composite and that F preserves identity maps:

$$\begin{tikzcd}
\mathcal{C}[2] \arrow[r, "m"] \arrow[d, "F[2]"'] & \mathcal{C}[1] \arrow[d, "F[1]"] & \mathcal{C}[0] \arrow[l, "e"'] \arrow[d, "F[0]"] \\
\mathcal{D}[0] \arrow[r, "m"] & \mathcal{D}[1] & \mathcal{D}[0] \arrow[l, "e"']
\end{tikzcd}$$

3.1.2 Double categories

Double categories are a notion of two-dimensional categories introduced by Ehresmann (1963). Unlike 2-categories and bicategories, double categories have two distinct classes of 1-cells, called vertical and horizontal. This notion is central in this work, as the programs and the proof tree of CSL will be interpreted as horizontal morphisms in a double category.

Definition 3.1.4. A ***double category*** $\mathcal{D}$ is a weakly internal category in **Cat**, the 2-category of small categories. Let us spell it out. The double category $\mathcal{D}$ is a tuple $\mathcal{D} = (\mathcal{D}[0], \mathcal{D}[1], s, t, e, m)$ which contains two categories $\mathcal{D}[0]$ and $\mathcal{D}[1]$. The objects of $\mathcal{D}[0]$ are called the objects of the double category $\mathcal{D}$, and its maps are called the ***vertical morphisms*** of $\mathcal{D}$ and are denoted with plain arrows $A \rightarrow B$. Each object of $\mathcal{D}[1]$ is associated with a source and a target object in $\mathcal{D}[0]$ through the two functors s and t, and is called a ***horizontal morphism***. They are denoted with crossed arrows $A \mathrel{\mapstochar\rightarrow} B$. Finally, a morphism α in $\mathcal{D}[1]$ is called a ***2-cell*** of the double category $\mathcal{D}$. The 2-cell α fills a square of two horizontal and two vertical maps of the form:

$$\begin{tikzcd}
A_1 \arrow[r, "F", "\shortmid"{marking}] \arrow[d, "f"'] \arrow[dr, phantom, "\Downarrow\alpha"] & B_1 \arrow[d, "g"] \\
A_2 \arrow[r, "G"', "\shortmid"{marking}] & B_2
\end{tikzcd}$$

A 2-cell as above is called ***special*** if the two vertical morphisms f and g are identities. Composition of horizontal morphisms is associative up to coherent special two-cells.

Most examples of double categories have a similar flavor: vertical morphisms are the usual notion of map for the objects under consideration, and the horizontal morphisms correspond to relation-like mappings between objects. For example, there is a double category where the horizontal morphisms between rings A and B are the AB-bimodules, while vertical morphisms are simply ring homomorphisms. In other words, the underlying vertical category ($\mathcal{D}[0]$ in the notation of the definition) is the usual category of rings. This intuition is captured by the notion of *framed bicategory* introduced by Shulman (2008).

Another important family of example of double categories which follow the same principle are the double categories $\mathbf{Cospan}(\mathbb{S})$, parameterized by a category $\mathbb{S}$ with pushouts. The object and vertical morphisms are the same as $\mathbb{S}$, and horizontal morphisms between objects A and B of $\mathbb{S}$ are cospans:

$$\begin{tikzcd}
A \arrow[dr, "l"'] & & B \arrow[dl, "r"] \\
& S &
\end{tikzcd}$$

The maps l and r are called its legs, and S is its support. We sometimes write $S : A \nrightarrow B$ when the legs are clear from context. The composite of two cospans $S : A \nrightarrow B$ and $T : B \nrightarrow C$ is defined using the pushout of the right leg of S and the left leg of T as depicted in this diagram:

$$\begin{tikzcd}
A \arrow[dr, "l"] & & B \arrow[dl, "r"'] \arrow[dr, "l'"] & & C \arrow[dl, "r'"'] \\
& S \arrow[dr] & po & T \arrow[dl] & \\
& & T \circ S & &
\end{tikzcd}$$

The 2-cells between horizontal maps $S : A \nrightarrow B$ and $T : C \nrightarrow D$ and vertical maps $f : A \to C$ and $g : B \to D$ are the maps $\alpha : S \to T$ which make the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "l"] \arrow[d, "f"'] & S \arrow[d, "\alpha"] & B \arrow[l, "r"'] \arrow[d, "g"] \\
C \arrow[r, "l'"'] & T & D \arrow[l, "r'"]
\end{tikzcd}$$

Of course, there is a similar double category $\mathbf{Span}(\mathbb{S})$ of spans when the category $\mathbb{S}$ has all pullbacks. We can recover the more common *bicategory* of cospans if we only consider special 2-cells; this comes from a general fact:

Lemma 3.1.5. *Any double category $\mathcal{D}$ induces a bicategory $\mathbf{H}\mathcal{D}$ whose objects are the same as $\mathcal{D}$'s, whose 1-cells are the horizontal morphisms of $\mathcal{D}$, and whose 2-cells are the special 2-cells of $\mathcal{D}$.*

There is a natural notion of map between double categories:

Definition 3.1.6. A ***double functor*** $\mathcal{F}$ between two double categories $\mathcal{C}$ and $\mathcal{D}$ is an internal functor between them. Explicitly, $\mathcal{F}$ maps objects, vertical 1-cells and horizontal 1-cells and 2-cells of the double category $\mathcal{C}$ to the same type of cells of the double category $\mathcal{D}$ while preserving source and targets, as expected.

Moreover, for two composable vertical morphisms f, g in $\mathcal{C}$, $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$ in $\mathcal{D}$, and, for two composable horizontal morphisms F, G in $\mathcal{C}$, $\mathcal{F}(G \circ F)$ and $\mathcal{F}(G) \circ \mathcal{F}(F)$ are related by an invertible special 2-cell. Finally, $\mathcal{F}$ preserves vertical identities on the nose and vertical identities up to invertible special 2-cells.

A weaker notion which will be useful is the following

Definition 3.1.7. A ***lax double functor*** $\mathcal{F}$ between two double categories $\mathcal{C}$ and $\mathcal{D}$ is the same as a double functor except that the special 2-cells which relates $\mathcal{F}(G \circ F)$ and $\mathcal{F}(G) \circ \mathcal{F}(F)$

$$\begin{tikzcd}
\mathcal{F}(A) \arrow[r, "\mathcal{F}F"] \arrow[d, equal] & \mathcal{F}(B) \arrow[r, "\mathcal{F}G"] \arrow[d, Rightarrow, "\alpha"] & \mathcal{F}(A) \arrow[d, equal] \\
\mathcal{F}(A) \arrow[rr, "\mathcal{F}(G\circ F)"'] & {} & \mathcal{F}(C)
\end{tikzcd}$$

is not necessarily invertible. Again, we refer the reader to Appendix A.2 of (Courser, 2020) for the full definition.

3.1.3 Polyads and internal J-opcategories

Polyads are a generalization of monads with several objects which were introduced by Bénabou (1967). They are used to define the notion of internal J-opcategory, which will be the basic notion of ***template*** in this work.

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.

Definition 3.1.8. Given a set J, a ***J-polyad*** in a double category $\mathcal{D}$ is a lax double functor

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

from the chaotic category over the set J, seen as a double category whose horizontal morphisms are the morphisms of $\mathbf{chaos}(J)$, to $\mathcal{D}$. The elements of J are called the colors of the polyad. A polyad is a J-polyad for some set J.

Polyads generalize monads as a monad is simply a $\{*\}$-polyad. This leads us, inspired by Lemma 3.1.2, to define an ***internal J-opcategory*** to be a polyad in the double category $\mathbf{Cospan}(\mathbb{S})$. They are, in a sense, dual to internal categories in that they are based on cospans instead of spans; they also are a generalization to multiple objects.

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

$$\begin{tikzcd} \pm[1,ij] \; : \; \pm[0,i] \arrow[r, "|" marking] & \pm[0,j] \arrow[r, phantom, "="] & \pm[0,i] \arrow[r, "\mathrm{in}_{ij}"] & \pm[1,ij] & \pm[0,j] \arrow[l, "\mathrm{out}_{ij}"'] \end{tikzcd}$$

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

$$\begin{tikzcd} \pm[0,i] \arrow[rr, bend left=50, "\mathrm{Id}", "|" marking, ""{name=U, below}] \arrow[rr, bend right=50, "{\pm[1,ii]}"', "|" marking, ""{name=D, above}] & & \pm[0,i] \arrow[from=U, to=D, Rightarrow, "\eta_i"] \end{tikzcd} \qquad \begin{tikzcd} & \pm[0,j] \arrow[dr, "{\pm[1,jk]}", "|" marking] \arrow[d, Rightarrow, "\mu_{ijk}"] & \\ \pm[0,i] \arrow[ur, "{\pm[1,ij]}", "|" marking] \arrow[rr, "{\pm[1,ik]}"', "|" marking] & {} & \pm[0,k] \end{tikzcd}$$

In particular, the left condition implies that the cospans $\pm[1,ii]$ are of the form

$$\begin{tikzcd} \pm[0,i] \arrow[r, "\eta_i", "|" marking] & \pm[1,ii] & \pm[0,i] \arrow[l, "\eta_i"', "|" marking] \end{tikzcd}$$

More explicitly, μ_{ijk} is given by the map g_{ijk} in the following commutative diagram in the category $\mathbb{S}$:

$$\begin{tikzcd} \pm[0,i] \arrow[dr, "\mathrm{in}_{ij}"] \arrow[dddrr, bend right=40, "\mathrm{in}_{ik}"'] & & \pm[0,j] \arrow[dl, "\mathrm{out}_{ij}"'] \arrow[dr, "\mathrm{in}_{jk}"] & & \pm[0,k] \arrow[dl, "\mathrm{out}_{jk}"'] \arrow[dddll, bend left=40, "\mathrm{out}_{ik}"] \\ & \pm[1,ij] \arrow[dr] & po & \pm[1,jk] \arrow[dl] & \\ & & \pm[2,ijk] \arrow[d, "g_{ijk}"] & & \\ & & \pm[1,ik] & & \end{tikzcd}$$

Finally, there are some conditions about the associativity of the composition, and about the identity; they can be found in (Bénabou, 1967, def. 5.5.1). An ***internal opcategory*** $\pm$ is defined as an internal $\{*\}$-opcategory. We write in that case $\pm[0]$ and $\pm[1]$ for the objects $\pm[0,*]$ and $\pm[1,**]$ of the ambient category $\mathbb{S}$, respectively.

The notion of internal J-opcategory comes with a natural notion of morphism between them: internal functors of J-opcategories:

Definition 3.1.9. An ***internal functor*** $F = (f, F[0,\cdot], F[1,\cdot])$ from $\pitchfork$ to $\pitchfork'$ is a triple consisting of a function $f : J \to J'$ between the sets of colors, and:

1. for each $i \in J$, a map $F[0,i]$ in the ambient category $\mathbb{S}$:
$$F[0,i] : \pitchfork[0,i] \longrightarrow \pitchfork'[0,f(i)],$$
2. for each $i,j \in J$, a map $F[1,ij]$ in the ambient category $\mathbb{S}$:
$$F[1,ij] : \pitchfork[1,ij] \longrightarrow \pitchfork'[1,f(ij)]$$

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

$$\begin{tikzcd}
\pitchfork[0,i] \arrow[r, "\mathrm{in}_{ij}"] \arrow[d, "F[0,i]"'] & \pitchfork[1,ij] \arrow[d, "F[1,ij]"] & \pitchfork[0,j] \arrow[l, "\mathrm{out}_{ij}"'] \arrow[d, "F[0,j]"] \\
\pitchfork'[0,f(i)] \arrow[r, "\mathrm{in}_{f(i)f(j)}"] & \pitchfork'[1,f(ij)] & \pitchfork'[0,f(j)] \arrow[l, "\mathrm{out}_{f(i)f(j)}"']
\end{tikzcd}$$

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.

$$\begin{tikzcd}
\pitchfork[2,ijk] \arrow[r] \arrow[d, "F[2,ijk]"'] & \pitchfork[1,ik] \arrow[d, "F[1,ik]"] \\
\pitchfork'[2,f(ijk)] \arrow[r] & \pitchfork'[1,f(ik)]
\end{tikzcd}
\qquad
\begin{tikzcd}
\pitchfork[0,i] \arrow[r, "\eta_i"] \arrow[d, "F[0,i]"'] & \pitchfork[1,ii] \arrow[d, "F[1,ii]"] \\
\pitchfork[0,f(i)] \arrow[r, "\eta'_{f(i)}"'] & \pitchfork'[1,f(ii)]
\end{tikzcd}$$

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 important role in the sequel.

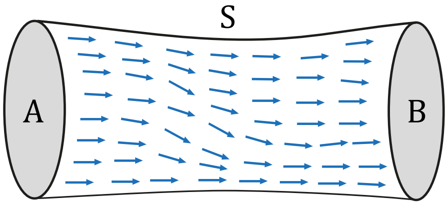
3.1.4 The double category Cob($\pitchfork$) of games and cobordisms

Suppose given an internal J-opcategory $\pitchfork$ 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 $\lambda_A : A \to \pitchfork[0,j]$. An ***$ij$-colored cobordism*** from an i-colored game (A,λ_A) to a j-colored game (B,λ_B)

is defined as a cospan in $\mathcal{C}$ together with colors $i, j \in J$ and a map λ_σ such that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "\mathrm{in}"] \arrow[d, "\lambda_A"'] & S \arrow[d, "\lambda_\sigma"] & B \arrow[l, "\mathrm{out}"'] \arrow[d, "\lambda_B"] \\
\pitchfork[0,i] \arrow[r, "\mathrm{in}_{ij}"] & \pitchfork[1,ij] & \pitchfork[0,j] \arrow[l, "\mathrm{out}_{ij}"']
\end{tikzcd}$$

We think of a cobordism as a space S of all executions of a program, visualized as a topological space, with two “borders” which are interpreted as the initial (or input) states and the final (or output) states of the program. The cospan at the level of templates is a way of typing, or coloring this object. They are depicted like this:



Two such ij-colored cobordism $\sigma : A \rightarrow\!\!\!\!\!+ B$ and jk-colored cobordism $\tau : B \rightarrow\!\!\!\!\!+ C$ can be composed into an ik-colored cobordism $\sigma;\tau : A \rightarrow\!\!\!\!\!+ C$ using a pushout and a relabeling along the 2-cell μ_{ijk} provided by the internal J-opcategory, as shown below:

$$\begin{tikzcd}
A \arrow[rr, "\mathbf{c}", "|" marking] \arrow[d] & {} \arrow[d, Rightarrow] & B \arrow[rr, "\mathbf{c}'", "|" marking] \arrow[d] & {} \arrow[d, Rightarrow] & C \arrow[d] \\
\pitchfork[0,i] \arrow[rr, "{\pitchfork[1,ij]}"', "|" marking] \arrow[rrrr, bend right=40, "{\pitchfork[1,ik]}"', "|" marking] & {} & \pitchfork[0,j] \arrow[rr, "{\pitchfork[1,jk]}"', "|" marking] \arrow[d, Rightarrow, "\mu_{ijk}"] & {} & \pitchfork[0,i] \\
 & & {} & &
\end{tikzcd}$$

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

$$\begin{tikzcd}
A \arrow[rr, "\mathrm{Id}_A", "|" marking] \arrow[d, "\lambda_A"'] & {} \arrow[d, Rightarrow, "\lambda_A"] & A \arrow[d, "\lambda_A"] \\
\pitchfork[0,i] \arrow[rr, "\mathrm{Id}_{\pitchfork[0,i]}"', "|" marking] \arrow[rr, bend right=60, "{\pitchfork[1,ii]}"', "|" marking] & {} \arrow[d, Rightarrow, "\eta_i"] & \pitchfork[0,i] \\
 & {} &
\end{tikzcd}$$

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

$$\begin{tikzcd}
A \arrow[r, "\text{in}_C"] \arrow[d, "f_{\text{in}}"'] & S \arrow[d, "f"] & B \arrow[l, "\text{out}_C"'] \arrow[d, "f_{\text{out}}"] \\
A' \arrow[r, "\text{in}'_C"] & S' & B' \arrow[l, "\text{out}'_C"']
\end{tikzcd}$$

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

Theorem 3.1.10. *Every internal J-opcategory ⫛ induces a double category* **Cob**(⫛) *whose objects are the j-colored games and whose horizontal maps are the cobordisms with composition defined as above.*

3.2 Three internal J-opcategories: ⫛$_{\text{L}}$, ⫛$_{\text{S}}$, ⫛$_{\text{Sep}}$

We instantiate the framework described above to define the three J-opcategories in the category $\mathbb{S}$ = **AsyncGraph** of asynchronous graphs which we will use to give a semantics to the concurrent shared memory language we describe in Section 2.3 and to the proofs of CSL, described in Section 2.5.

They are based on the machine models ⫛$^{\bullet}_{\text{L}}$, ⫛$^{\bullet}_{\text{S}}$, and ⫛$^{\bullet}_{\text{Sep}}$ which we defined in Chapter 2. The machine models describe the basic operations supported by the underlying machine, and how the interact when they are executed in parallel. The cobordisms over these internal J-opcategories will contain the information of how programs cause these instructions to be executed and how they react to changes in the global memory state.

3.2.1 The internal opcategories ⫛$_{\text{L}}$ and ⫛$_{\text{S}}$ for the code

Recall that both the stateless machine model ⫛$^{\bullet}_{\text{L}}$ and the stateful machine model ⫛$^{\bullet}_{\text{S}}$ are parameterized by a set $\mathfrak{L}$ of allocated locks. In this section we assume given such a set and we do not write it explicitly.

The construction of the internal opcategories from the machine models is the same for both the ***stateless internal opcategory*** ⫛$_{\text{L}}$ and the ***stateful internal opcategory*** ⫛$_{\text{S}}$. We write ⫛$^{\bullet}$ for either ⫛$^{\bullet}_{\text{L}}$ or ⫛$^{\bullet}_{\text{S}}$ and we write ⫛ for either ⫛$_{\text{L}}$ or ⫛$_{\text{S}}$, respectively.

Recall that defining an internal opcategory $\pitchfork$ comes down to defining a cospan in the ambient category **AsyncGraph**. In our case, it will be of the form:

$$\begin{tikzcd}
\pitchfork[0] \arrow[dr, hook, "\iota"] & & \pitchfork[0] \arrow[dl, hook', "\iota"'] \\
& \pitchfork[1] &
\end{tikzcd}$$

That is, both its legs are equal and are monomorphisms.

We construct an asynchronous graph $\pitchfork[1]$ with two players (for Code and Environment) from the asynchronous graph $\pitchfork^{\bullet}$ which we see as having a single player. This shift from one player to two players is performed in a very simple way. We consider the functor $\Omega : \mathbf{Set} \to \mathbf{AsyncGraph}$ 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. Note that the functor preserves limits: in particular, the asynchronous graph $\Omega(1)$ is the terminal object of **AsyncGraph**. The asynchronous graph $\Omega(\{\mathbf{C}, \mathbf{F}\})$ enables us to define the *two-player* machine models $\pitchfork^{\circ\bullet}$ as the Cartesian product of asynchronous graphs

$$\pitchfork^{\circ\bullet} \quad = \quad \pitchfork^{\bullet} \times \Omega(\{\mathbf{C}, \mathbf{F}\}).$$

The resulting asynchronous graph $\pitchfork^{\circ\bullet}$ has the same nodes as $\pitchfork^{\bullet}$ and contains two edges

$$x \xrightarrow{m:\mathbf{C}} y \qquad\qquad x \xrightarrow{m:\mathbf{F}} y$$

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

$$x \xrightarrow{m} y$$

in the original one-player, or apolar, asynchronous graph $\pitchfork^{\bullet}$. The two circles $\circ$ and $\bullet$ in the notation $\pitchfork^{\circ\bullet}$ 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 $\pitchfork = \pitchfork_{\mathrm{S}}, \pitchfork_{\mathrm{L}}$ in the ambient category **AsyncGraph**. The asynchronous graph $\pitchfork[1]$ is defined as the two-player machine model $\pitchfork^{\circ\bullet}$ while the asynchronous graph $\pitchfork[0]$ is defined as the one-player machine model, which we see here as played only by the Frame. In summary:

$$\pitchfork[1] = \pitchfork^{\circ\bullet} \qquad\qquad \pitchfork\,[0] = \pitchfork^{\bullet}.$$

The asynchronous graph $\pitchfork[0]$ with one player is then embedded in the two-player asynchronous graph $\pitchfork[1]$ by the asynchronous morphism $\iota = \pitchfork[0] \to \pitchfork[1]$ obtained by transporting every node of $\pm[0]$ to the corresponding node in $\pm[1]$, and every edge of $\pm[0]$ to the corresponding edge in $\pm[1]$ with the polarity **F** of the Frame.

3.2.2 The internal J-opcategory $\pm_{\mathrm{Sep}}$ for the proofs

The case of the separated state internal J-opcategory differs from the two internal opcategories we have just defined in two ways: First, the underlying machine model of separated states already has two players because the notion of state contains distinguishes several the area owned by the Code from the area owned by the Environment.

Second, the set J of colors is not a singleton: it is the set of predicates of the logic. The reason is that a proof π of a Hoare triple $\Gamma \vdash \{P\}\, C\, \{Q\}$ has a meaning only when, in the initial state, the memory owned by the Code satisfies the precondition P, and dually it guarantees that the final state satisfies Q. This typing information is contained in the space of objects of the internal J-opcategory of a cobordism interpreting the proof π:

$$\begin{tikzcd}
I \arrow[rr] \arrow[d] & & S \arrow[d] & & O \arrow[ll] \arrow[d] \\
\pm_{\mathrm{Sep}}(\Gamma)[0,P] \arrow[rr] & & \pm_{\mathrm{Sep}}(\Gamma)[1] & & \pm_{\mathrm{Sep}}(\Gamma)[0,Q] \arrow[ll]
\end{tikzcd}$$

This expresses that the initial (internal) states of the proof, represented by the asynchronous graph I have (external) states which satisfy the predicate P, and similarly for final states and the postcondition Q.

Following this intuition, the nodes of the asynchronous graph $\pm_{\mathrm{Sep}}[0,P]$ are all the separated states whose Code-components satisfy the predicate P:

$$(\sigma_C, \vec{\sigma}, \sigma_F) \in \pm_{\mathrm{Sep}}[0,P] \qquad \Leftrightarrow \qquad \sigma_C \vDash P.$$

As in the previous section for the internal opcategories for the code, we omit the parameter Γ. The transitions in $\pm_{\mathrm{Sep}}[0,P]$ between such states are those of $\pm^{\circ\bullet}_{\mathrm{Sep}}$ played by Frame.

The asynchronous graph $\pm_{\mathrm{Sep}}[1]$ is simply defined to be $\pm^{\circ\bullet}_{\mathrm{Sep}}$, and the maps

$$\pm_{\mathrm{Sep}}[0,P] \xhookrightarrow{\iota} \pm_{\mathrm{Sep}}[1]$$

are the obvious inclusions.

3.3 Parallel product

The double categories $\mathbf{Cob}(\pm)$, where $\pm$ is one of the three internal opcategories $\pm_{\mathrm{L}}$, $\pm_{\mathrm{S}}$ or $\pm_{\mathrm{Sep}}$, need additional structures to interpret the constructions of the programming language we consider and the corresponding rules of CSL. We begin with the parallel product.

It would be natural to interpret it as a monoidal product $\parallel$ in the model. However, given maps $C_i : A_i \twoheadrightarrow B_i$ and $D_i : B_i \twoheadrightarrow E_i$, for $i = 1, 2$, there would have to be an invertible special 2-cell:

$$(C_1 \parallel C_2); (D_1 \parallel D_2) \implies (C_1; D_1) \parallel (C_2; D_2) \tag{3.3}$$

between two horizontal morphisms $A_1 \parallel A_2 \twoheadrightarrow E_1 \parallel E_2$. This contradicts the well-known fact that in general the concatenation of interleavings are *included* in the interleaving of the concatenations of traces, expressed by the so-called Hoare inequality:

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

Hence, the right structure to interpret the parallel product is that of *lax monoidal products*, where the 2-cell (3.3) is not necessarily invertible.

As for the other constructions on cobordisms, this lax monoidal product is going to be induced by a structure at the level of templates: span monoidal internal J-opcategories. In order to define them, we introduce plain internal functors, which are internal functors of J-opcategories which enjoy the property that one can pull along them.

3.3.1 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 3.1.9, they are the internal functors of J-opcategories such that $f = \mathrm{Id}$. Plain internal functors $F : \pm_1 \to \pm_2$ are useful because they define a pull operation on cobordisms $\mathbf{C}$ in $\mathbf{Cob}(\pm_2)$ operation when $\mathbb{S}$ has pullbacks, given by the obvious three pullbacks along the components of F in the following diagram:

$$\begin{tikzcd}
 & & O \arrow[dl, "\mathrm{out}_{\mathrm{C}}"'] \arrow[dd] & \\
 & S \arrow[dd] & & \\
I \arrow[ur, "\mathrm{in}_{\mathrm{C}}"] \arrow[dd] & & \pm_2[0,j] \arrow[dl, "\mathrm{out}_{ij}"'] & \\
 & \pm_2[1,ij] & & \pm_1[0,j] \arrow[ul, "{F[0,j]}"'] \arrow[dl, "\mathrm{out}_{ij}"] \\
\pm_2[0,i] \arrow[ur, "\mathrm{in}_{ij}"] & & \pm_1[1,ij] \arrow[ul, "{F[0,ij]}"'] & \\
 & \pm_1[0,i] \arrow[ul, "{F[0,i]}"] \arrow[ur, "\mathrm{in}_{ij}"'] & &
\end{tikzcd}$$

This construction induces a lax double functor between $\mathbf{Cob}(\pm_2)$ and $\mathbf{Cob}(\pm_1)$. The converse operation of pushing exists for any internal functor.

Lemma 3.3.1. *A plain internal functor $F : \pm_1 \to \pm_2$ induces a lax double functor defined by taking the pointwise pullbacks along the components of F*

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

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

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

Proof. The double functor $\mathsf{pull}[F]$ is lax, in that there is a (not necessarily invertible) map $\mathsf{pull}[F](C); \mathsf{pull}[F](D) \to \mathsf{pull}[F](C; D)$ induced by the universal properties of colimits and limits. □

3.3.2 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 $ナ_1$ and an internal J_2-opcategory $ナ_2$ is defined as a span of internal functors

$$\begin{tikzcd} ナ_1 & ナ_0 \arrow[l, "F"', "="] \arrow[r, "G"] & ナ_2 \end{tikzcd}$$

where $ナ_0$ 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 double category **AcuteSpan**, whose objects are the internal J-opcategories in the ambient category $\mathbb{S}$, whose horizontal 1-cells are acute spans, whose vertical 1-cells are internal functors of J-opcategories and whose 2-cells are ***maps of acute spans***: commuting diagrams of the form:

$$\begin{tikzcd} ナ_1 \arrow[d] & ナ_2 \arrow[l, "="'] \arrow[r] \arrow[d, "\phi", "="'] & ナ_3 \arrow[d] \\ ナ'_1 & ナ'_2 \arrow[l, "="] \arrow[r] & ナ'_3 \end{tikzcd}$$

where the internal functor $\phi : ナ \to ナ'_0$ 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:

$$\begin{tikzcd} ナ_1 \arrow[r, "F"] & ナ_2 & ナ_3 \arrow[l, "G"', "="] \end{tikzcd}$$

between an internal J_1-opcategory $ナ_1$ and a J_3-opcategory $ナ_3$. Their pullback is defined as the internal J_1-opcategory $ナ_1 \times_{ナ_2} ナ_3$ described as follows. 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:

$$\begin{tikzcd} ナ_1[0,i] \arrow[r, "{F[0,i]}"] & ナ_2[0,f(i)] & ナ_3[0,f(i)] \arrow[l, "{G[0,f(i)]}"'] \end{tikzcd}$$

where f is the action of the internal functor F on colors; and we simply define $(ナ_1 \times_{ナ_2} ナ_3)[0,i]$ as the pullback of that diagram in the ambient category $\mathbb{S}$. We proceed similarly to define $(ナ_1 \times_{ナ_2} ナ_3)[1,ij]$, and the universality of the pullbacks in $\mathbb{S}$ gives the structural maps between the two.

Transport along acute spans The *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 Part I of this thesis, in particular because we derive our definition of parallel product, and later of change of lock operations, from it.

To be able to characterize this operation, we need the notion of vertical transformation, a generalization of natural transformations to double categories.

Definition 3.3.2. Given two double categories $\mathcal{C}$ and $\mathcal{D}$, and two lax double functors $\mathcal{F}, \mathcal{G} : \mathcal{C} \to \mathcal{D}$ between them, a ***vertical transformation*** $\alpha : \mathcal{F} \Rightarrow G$ is defined as the following.

- For all object $A \in \mathcal{C}$, a vertical 1-cell

$$\begin{tikzcd}
\mathcal{F}(A) \arrow[d, "\alpha_A"] \\
\mathcal{G}(A)
\end{tikzcd}$$

  in $\mathcal{D}$ which is natural with respect to composition with vertical 1-cells in the double category $\mathcal{C}$.

- For every horizontal 1-cell $F : A \to B$ in $\mathcal{C}$, a 2-cell

$$\begin{tikzcd}
\mathcal{F}(A) \arrow[r, "\mathcal{F}(F)"] \arrow[d, "\alpha_A"] & \mathcal{F}(B) \arrow[d, "\alpha_B"] \\
\mathcal{G}(A) \arrow[r, "\mathcal{G}(F)"'] & \mathcal{G}(B)
\arrow[from=1-1, to=2-2, phantom, "\Downarrow \alpha_F"]
\end{tikzcd}$$

- For composable horizontal 1-cells $F : A \to B$ and $G : B \to C$, the horizontal composite of α_F and α_G is $\alpha_{G \circ F}$.

The operation of transport along an acute span can be formulated as a double functor

$$\mathbf{Cob} : \mathbf{AcuteSpan} \to \mathbf{Dbl}_{vstrict,hlax}$$

between the double category **AcuteSpan** of acute spans defined just above, and the following double category:

Definition 3.3.3. The double category of double categories, double functors and vertical transformations $\mathbf{Dbl}_{vstrict,hlax}$ is defined as follows:

- 0-cells are double categories;
- vertical 1-cells are double functors;
- horizontal 1-cells are lax double functors;
- 2-cells are vertical transformations between the two composite lax double functors.

Lemma 3.3.4. *The operation* **Cob** *defines a double functor*

$$\mathbf{Cob} : \mathbf{AcuteSpan} \to \mathbf{Dbl}_{vstrict,hlax}$$

from the double category **AcuteSpan** *just defined to the double category* $\mathbf{Dbl}_{vstrict,hlax}$ *of double categories, double functors, lax double functors and vertical transformations. It is defined as follows:*

$$\begin{array}{rcrcl} \mathbf{Cob} & : & \mathbf{AcuteSpan} & \longrightarrow & \mathbf{Dbl}_{vstrict,hlax} \\ & & \pitchfork & \longmapsto & \mathbf{Cob}(\pitchfork) \\ & & f & \longmapsto & \mathsf{push}[f] \\ & & (F,G) & \longmapsto & \mathsf{push}[G] \circ \mathsf{pull}[F] \end{array}$$

and the 2-cells are given by the universal properties of the pullback in the definition of $\mathsf{pull}[\cdot]$.

Proof. To prove that this induces a double functor, one needs in particular to check that, given two composable acute spans $\mathbf{F} : \pitchfork_1 \relbar\joinrel\mapstochar\joinrel\rightarrow \pitchfork_3$ with support $\pitchfork_2$ and $\mathbf{G} : \pitchfork_3 \relbar\joinrel\mapstochar\joinrel\rightarrow \pitchfork_5$ with support $\pitchfork_4$, there is an invertible special 2-cell, that is, an invertible vertical transformation $\alpha : \mathbf{Cob}(\mathbf{G} \circ \mathbf{F}) \cong \mathbf{Cob}(\mathbf{G}) \circ \mathbf{Cob}(\mathbf{F})$. In particular, given an object $\lambda : A \to \pitchfork_1[0, i]$ of $\mathbf{Cob}(\pitchfork_1)$, there must be a vertical isomorphism

$$\alpha_A : \mathbf{Cob}(\mathbf{G} \circ \mathbf{F})(A) \xrightarrow{\sim} (\mathbf{Cob}(\mathbf{G}) \circ \mathbf{Cob}(\mathbf{F}))(A)$$

The following diagram shows how to construct this isomorphism.

$$\begin{tikzcd}
A \arrow[dd, "\lambda"'] & & & & A_1 \arrow[dd] & & & & \\
& & A_1 \arrow[ull] \arrow[urr, equals] \arrow[ddd] & & & & A_2 \arrow[ull] \arrow[ddd] & & \\
\pitchfork_1[1] & & & & \pitchfork_3[1] & & & & \pitchfork_5[1] \\
& & & & B \arrow[uull] \arrow[uurr, "\alpha_A"] \arrow[dd] & & & & \\
& & \pitchfork_2[1] \arrow[uull] \arrow[uurr] & & & & \pitchfork_4[1] \arrow[uull] \arrow[uurr] & & \\
& & & & \pitchfork_6[1] \arrow[ull] \arrow[urr] & & & &
\end{tikzcd}$$

Here, $ᛦ_6[1]$ is the pullback defining the composition of F and of G, and A_1 and A_2 are the pullbacks in the action of the acute spans F and G. Finally, B is the pullback of the action of $G \circ F$, along the map from $ᛦ_6[1]$ to $ᛦ_1[1]$.

Using the pasting lemma on pullbacks, the face of the cube with B, A_1 and $ᛦ_6[1]$ is a pullback. Using the same lemma with the other faces of the cube, we conclude that the top face is a pullback as well.

This implies that map from B to A_2 is an isomorphism, since it is the pullback of the identity on A_1. This means that B and A_2 are related by an isomorphism which commutes with the labeling maps:

$$\begin{tikzcd}
B \arrow[rr, "\alpha_A"] \arrow[dr] & & A_2 \arrow[dl] \\
& ᛦ_5[1] &
\end{tikzcd}$$

We can do the same with the $ᛦ[0, i]$'s to construct the desired isomorphism. □

3.3.3 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 $\mathbf{Cob}(ᛦ)$, that is to say a lax double functor

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

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

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

In our setting, this follows from Lemma 3.3.1, and therefore from the fact that there is always a map from a colimit of limits to the corresponding limit of colimits.

The formal statement is that $\mathbf{Cob}(ᛦ)$ is a monoid object in the bicategory $\mathbf{Dbl}_{\text{lax}}$ of double categories and lax double functors. This is why we will use the fact (Lemma 3.1.5) that **Cob** induces a lax pseudo functor of bicategories **H Cob** between the horizontal bicategories **H AcuteSpan** and $\mathbf{H\,Dbl}_{vstrict,hlax} = \mathbf{Dbl}_{\text{lax}}$ of double categories and lax double functors. The objects of bicategory **AcuteSpan** are the same as the corresponding double category, its 1-cells are the acute spans and a two cell $\phi : (F', G') \Rightarrow (F', G')$ is a special 2-cell in the double category, that is, a plain internal functors ϕ which make the following diagram commute:

$$\begin{tikzcd}
 & \pitchfork_0 \arrow[dl, "F"'] \arrow[dr, "G"] \arrow[dd, "\phi", "\shortparallel"'] & \\
\pitchfork_1 & & \pitchfork_2 \\
 & \pitchfork'_0 \arrow[ul, "F'"] \arrow[ur, "G'"'] &
\end{tikzcd}$$

In this section, **Cob** will denote this lax pseudo functor and **AcuteSpan** will denote the bicategory of acute spans.

This tensor product is built the same way as in the case of template games (Melliès, 2019), but using the notion of span-monoidal internal J-opcategory. We remarked below Definition 3.1.9 that the category $\mathbf{opCat}(\mathbb{S})$ of internal opcategories in the ambient category $\mathbb{S}$ is a Cartesian category.

As a consequence, the lax pseudo functor **Cob** is lax monoidal, where we equip both **AcuteSpan** and $\mathbf{Dbl}_{\mathrm{lax}}$ with the monoidal structure induced by their Cartesian products. In particular, there exist coercions living in the Cartesian category $\mathbf{Dbl}_{\mathrm{lax}}$, in the form of lax double functors

$$\begin{array}{rcccc} \mathbf{m}_{\pitchfork_1,\pitchfork_2} & : & \mathbf{Cob}(\pitchfork_1) \times \mathbf{Cob}(\pitchfork_2) & \longrightarrow & \mathbf{Cob}(\pitchfork_1 \times \pitchfork_2) \\ \mathbf{m}_1 & : & \mathbb{1} & \longrightarrow & \mathbf{Cob}(\mathbf{1}) \end{array}$$

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 **1**.

Definition 3.3.5. A ***span-monoidal internal J-opcategory*** $(\pitchfork, \shortparallel, \eta)$ is a symmetric pseudomonoid object in the symmetric monoidal bicategory **AcuteSpan**. In particular, $\shortparallel$ and η are two acute spans:

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

together with invertible 2-cells that witness the associativity of $\shortparallel$, and the fact that η is a left and right identity of $\shortparallel$. See (Day and Street, 1997, §3) for the complete definition.

Now, combining the external operations $\mathbf{m}_{\pitchfork,\pitchfork}$ and $\mathbf{m}_1$ on the one hand, and the internal operations $\shortparallel, \eta$ on the other, we get a lax-monoidal structure on $\mathbf{Cob}(\pitchfork)$ whose multiplication and unit are respectively given by:

$$\mathbf{Cob}(\pitchfork) \times \mathbf{Cob}(\pitchfork) \xrightarrow{\mathbf{m}_{\pitchfork,\pitchfork}} \mathbf{Cob}(\pitchfork \times \pitchfork) \xrightarrow{\mathbf{Cob}(\shortparallel)} \mathbf{Cob}(\pitchfork)$$

$$\mathbb{1} \xrightarrow{\mathbf{m}_1} \mathbf{Cob}(\mathbf{1}) \xrightarrow{\mathbf{Cob}(\eta)} \mathbf{Cob}(\pitchfork)$$

Theorem 3.3.6. *Every span-monoidal internal J-opcategory $\pitchfork$ induces a symmetric lax-monoidal double category* $\mathbf{Cob}(\pitchfork)$.

3.3.4 Parallel products of code and of proofs

Now that we have a general method for equipping the double category $\mathbf{Cob}(\pitchfork)$ with a lax-monoidal structure, we use it to define the parallel product for the stateful and the stateless semantics of the Code.

Parallel product of codes 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 $\pitchfork_{\mathrm{L}}^{\circ\circ\bullet}$ and $\pitchfork_{\mathrm{S}}^{\circ\circ\bullet}$, 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 $\pitchfork_{\mathrm{L}}^{\bullet}$ and $\pitchfork_{\mathrm{S}}^{\bullet}$ in Section3.2.1 using a product, we use here the pullback:

$$\begin{tikzcd}
\pitchfork^{\circ\circ\bullet} \arrow[r, "\pi_{\mathrm{pol}}^{(12)(\mathbf{F})}"] \arrow[d, "\pi_{\mathrm{state}}^{(12)(\mathbf{F})}"'] & \Omega\{\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}\} \arrow[d, "\Omega(12)(\mathbf{F})"] \\
\pitchfork^{\circ\bullet} \arrow[r, "\pi_{\mathrm{pol}}"] & \Omega\{\mathbf{C}, \mathbf{F}\}
\end{tikzcd}$$

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, $\pitchfork^{\circ\circ\bullet}$ has the same states as $\pitchfork^{\bullet}$ and three copies for each transition of $\pitchfork^{\bullet}$, one copy for each of the three polarities $\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}$. The span-monoidal structures on the internal opcategories $\pitchfork = \pitchfork_{\mathrm{S}}, \pitchfork_{\mathrm{L}}$ are defined in a common way:

$$\begin{tikzcd}
\pitchfork^{\circ\bullet} \times \pitchfork^{\circ\bullet} & \pitchfork^{\circ\circ\bullet} \arrow[l, "\left\langle \pi_{\mathrm{state}}^{(1)(2\mathbf{F})}, \pi_{\mathrm{state}}^{(2)(1\mathbf{F})} \right\rangle"'] \arrow[r, "\pi_{\mathrm{state}}^{(12)(\mathbf{F})}"] & \pitchfork^{\circ\bullet} & & \mathbf{1} & \pitchfork^{\bullet} \arrow[l] \arrow[r, "\iota_{\mathbf{F}}"] & \pitchfork^{\circ\bullet}.
\end{tikzcd}$$

where, for instance, $\pi_{\mathrm{state}}^{(1)(2\mathbf{F})}$ 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 $\pitchfork^{\bullet}$ into $\pitchfork^{\circ\bullet}$ with every edge transported to the corresponding edge of polarity $\mathbf{F}$.

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 $\pitchfork$. 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 $\pitchfork^{\circ\bullet} \times \pitchfork^{\circ\bullet}$ forms a tile–meaning that these two instructions are independent.

Parallel product of proofs 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 $\pitchfork_{\mathrm{Sep}}$ with a span-monoidal structure. The main piece of this structure is the asynchronous graph $\pitchfork_{\mathrm{Sep}}^{\circ\circ\bullet}$ of *three-player separated states*, which will be the basis for the support of the span

$$\begin{tikzcd} \pitchfork_{\mathrm{Sep}} \times \pitchfork_{\mathrm{Sep}} & \pitchfork_{\mathrm{Sep}}^{\parallel} \arrow[l, "\mathsf{pick}"'] \arrow[r, "\mathsf{pince}"] & \pitchfork_{\mathrm{Sep}} \end{tikzcd}$$

of Definition 3.3.5. It is the straightforward generalization of separated states to the case where there are three players, $\mathbf{C}_1, \mathbf{C}_2$ and $\mathbf{F}$.

Definition 3.3.7. A ***three-player separated state*** $(\sigma_1, \sigma_2, \vec{\sigma}, \sigma_F)$ is a triple of three logical states σ_1, σ_2 and σ_F and a map $\vec{\sigma} : \mathfrak{L} \to \mathbf{LogState} + \{\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}\}$ which satisfies the following properties. For all $r \in \vec{\sigma}^{-1}(\mathbf{LogState})$, $\vec{\sigma}(r) \vDash \Gamma(r)$, and the following logical state is well defined:

$$\sigma_1 \ * \ \sigma_2 \ * \ \Big\{ \mathop{\circledast}\limits_{r \in \vec{\sigma}^{-1}(\mathbf{LogState})} \vec{\sigma}(r) \Big\} \ * \ \sigma_F \quad \in \quad \mathbf{LogState}$$

As for two-player separated states, using the logical state above, we can define a map $\circledast$ which maps a three-player separated state to its underlying machine state.

The asynchronous graph $\pitchfork_{\mathrm{Sep}}^{\circ\circ\bullet}$, the three-player analogue to $\pitchfork_{\mathrm{Sep}}^{\circ\bullet}$ which is the basis to the $\pitchfork_{\mathrm{Sep}}$ internal J-opcategory is defined as follows. Its nodes are the three-player separated states defined above. It has three kinds of transitions: The $\mathbf{C}_1$ transitions are of the form:

$$(\sigma_1, \sigma_2, \vec{\sigma}, \sigma_F) \xrightarrow{m:\mathbf{C}_1} (\sigma_1', \sigma_2, \vec{\sigma}', \sigma_F)$$

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

$$\circledast(\sigma_1, \sigma_2, \vec{\sigma}, \sigma_F) \xrightarrow{m} \circledast(\sigma_1', \sigma_2, \vec{\sigma}', \sigma_F)$$

is a transition in $\pitchfork_{\mathrm{S}}^{\bullet}$, and such that the following conditions are satisfied:

$$\begin{array}{ll} \forall \ell \notin \mathsf{wr}(m),\ \sigma_1(\ell) = \sigma_1'(\ell) & \mathsf{wr}(m) \cup \mathsf{rd}(m) \subseteq \mathsf{dom}(\sigma_1) \\ \mathsf{lk}(m) \subseteq \mathsf{dom}(\vec{\sigma}) \cup \vec{\sigma}^{-1}(\mathbf{C}_1) & \forall r \notin \mathsf{lk}(m), \vec{\sigma}(r) = \vec{\sigma}'(r). \end{array}$$

The $\mathbf{C}_2$ transitions are of the form

$$(\sigma_1, \sigma_2, \vec{\sigma}, \sigma_F) \xrightarrow{m:\mathbf{C}_2} (\sigma_1, \sigma_2', \vec{\sigma}', \sigma_F)$$

with symmetric conditions. Finally, the $\mathbf{F}$ transitions are of the form

$$(\sigma_1, \sigma_2, \vec{\sigma}, \sigma_F) \xrightarrow{m:\mathbf{F}} (\sigma_1, \sigma_2, \vec{\sigma}', \sigma_F')$$

such that they correspond to a transition in $\pm_{\mathrm{S}}$ as above, and such that

$$\forall \ell \notin \mathsf{wr}(m),\ \sigma_F(\ell) = \sigma'_F(\ell) \qquad \mathsf{wr}(m) \cup \mathsf{rd}(m) \subseteq \mathsf{dom}(\sigma_F)$$
$$\mathsf{lk}(m) \subseteq \mathsf{dom}(\vec{\sigma}) \cup \vec{\sigma}^{-1}(\mathbf{F}) \qquad \forall r \notin \mathsf{lk}(m), \vec{\sigma}(r) = \vec{\sigma}'(r).$$

We can now define the internal J-opcategory $\pm^{\Vert}_{\mathrm{Sep}}$, where J is the set of ordered pairs of formulas of CSL. As for $\pm_{\mathrm{Sep}}$, we define $\pm^{\Vert}_{\mathrm{Sep}}[1]$ to be $\pm^{\circ\circ\bullet}_{\mathrm{Sep}}$, and $\pm^{\Vert}[0,(P,Q)]$ is the sub-asynchronous graph of $\pm^{\circ\circ\bullet}_{\mathrm{Sep}}$ induced by keeping all $\mathbf{F}$ transitions (and tiles over them) between three-player separated states $(\sigma_1, \sigma_2, \vec{\sigma}, \sigma_F)$ such that $\sigma_1 \vDash P$ and $\sigma_2 \vDash Q$.

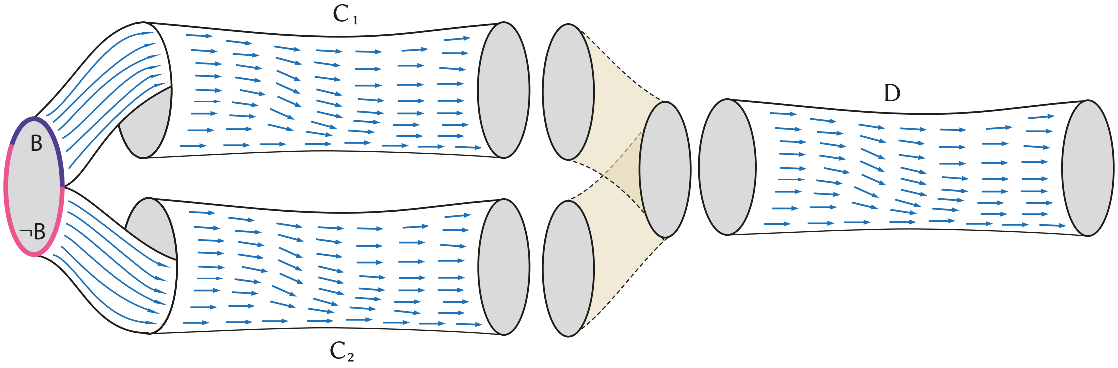
The internal functor pince has the following action on the supports of the internal J-categories: it maps a three-player state to the (two-player) separated state $(\sigma_1 * \sigma_2, \vec{\sigma}', \sigma_F)$, where $\vec{\sigma}'$ is the same as $\vec{\sigma}$ where C_1 and C_2 have been remapped to C. It maps $\pm^{\Vert}_{\mathrm{Sep}}[0,(P,Q)]$ to $\pm_{\mathrm{Sep}}[0, P*Q]$ by restriction of the above asynchronous morphism. The morphism pick maps that three-player state to the pair $\langle(\sigma_1, \vec{\sigma}_1, \sigma_2 * \sigma_F), (\sigma_2, \vec{\sigma}_1, \sigma_1 * \sigma_F)\rangle$. The unit $\pm^{\Vert\eta}_{\mathrm{Sep}}$ of the structure is the subgraph of $\pm_{\mathrm{Sep}}[1]$ with only Frame moves.

3.4 Generalized 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, because the initial and the final states of a program are part of its internal state. For example, the two cobordisms corresponding to the two programs being sequentially composed in the following program

$$(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2)\ \mathtt{;}\ D$$

could look like this:



In summary, the two cobordisms we wish to compose generally look like:

$$\begin{tikzcd}
A \arrow[r, "\shortmid" marking] \arrow[d] & B \arrow[d, "\lambda_{\mathrm{out}}"] \\
\pm[0,i] \arrow[r, "{\pm[1,ij]}", "\shortmid" marking] & \pm[0,j]
\end{tikzcd}
\qquad
\begin{tikzcd}
B' \arrow[r, "\shortmid" marking] \arrow[d, "\lambda_{\mathrm{in}}"'] & C' \arrow[d] \\
\pm[0,i'] \arrow[r, "{\pm[1,i'j']}", "\shortmid" marking] & \pm[0,j']
\end{tikzcd}$$

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 $(B,\lambda_{\mathrm{out}})$ and $(B',\lambda_{\mathrm{in}})$, we will use a ***filling system*** over the internal J-opcategory $\pm$. With each pair of interfaces $\lambda_{\mathrm{out}} : B \to \pm[0,j]$ and $\lambda_{\mathrm{in}} : B' \to \pm[0,i']$, the filling system associates a cobordism

$$\mathsf{fill}\,((B,\lambda_{\mathrm{out}}),(B',\lambda_{\mathrm{in}})) \quad : \quad (B,\lambda_{\mathrm{out}}) \quad \not\to \quad (B',\lambda_{\mathrm{in}})$$

from $(B,\lambda_{\mathrm{out}})$ to $(B',\lambda_{\mathrm{in}})$. 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 3.4.1. *Suppose that for all games* $\lambda, \lambda', \mu, \mu'$ *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 $\mathcal{C}$ 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:

$$\begin{tikzcd}
& B_j \times_{\pm[1,jj']} A_{j'} \arrow[dl] \arrow[dr] & \\
B_j \arrow[r] \arrow[d, "\lambda"] & B_j \cup_{\pm[1,jj']} A_{j'} \arrow[dd, dashed] & A_{j'} \arrow[l] \arrow[d, "\lambda'"] \\
\pm[0,j] \arrow[dr, "\mathrm{in}_{jj'}"] & & \pm[0,j'] \arrow[dl, "\mathrm{in}_{jj'}"'] \\
& \pm[1,jj'] &
\end{tikzcd}$$

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.

Lemma 3.4.2. *The hypothesis of Proposition 3.4.1 holds in the case of the stateful and stateless templates $\pm_{\mathrm{S}}$ and $\pm_{\mathrm{L}}$.*

Proof. Let $\pm$ be either $\pm_{\mathrm{L}}$ or $\pm_{\mathrm{S}}$. Given two games $\lambda : A \to \pm[0]$ and $\mu : B \to \pm[0]$ formulated as asynchronous morphisms, let us describe $\mathsf{fill}\,(\lambda, \mu)$. Every node x of A comes with a state $\mathrm{in}_{\lambda(x) \in} \pm [1]$ which we call the underlying state of x. Similarly, every node y of B comes with an underlying state $\mathrm{in}_{\lambda(y)} \in \pm[1]$. The pullback $A \times_{\pm[1]} B$ 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 just above satisfies the property that there exists a map:

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

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 $\mathrm{pince}[1] : \pm^{\parallel}[1] \to \pm[1]$ 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 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 $([s], [s])$.

□

Interestingly, this is not necessarily the case for the template ナ_{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 $\mathbf{C}, \mathbf{F}$ in $\text{ナ}^{\circ\bullet}_{\mathrm{Sep}}$ into a separated state between three players $\mathbf{C}_1, \mathbf{C}_2, \mathbf{F}$ in $\text{ナ}^{\circ\circ\bullet}_{\mathrm{Sep}}$.

3.5 Change of locks

In the language we consider, locks names follow lexical scoping: A lock r is introduced with the $\texttt{resource}\ r\ \texttt{do}\ C$ construction, and is only available inside of the command C. This would be formalized syntactically by a judgment $\mathfrak{L} \vdash C$ which states that all the locks in C are in the set $\mathfrak{L}$, with the expected introduction and elimination rules:

$$\frac{\mathfrak{L} \uplus \{r\} \vdash C}{\mathfrak{L} \vdash \texttt{resource}\ r\ \texttt{do}\ C} \qquad\qquad \frac{\mathfrak{L} \vdash C}{\mathfrak{L} \uplus \{r\} \vdash \texttt{with}\ r\ \texttt{do}\ C}$$

Correspondingly, the three machine models $\text{ナ} = \text{ナ}_{\mathrm{Sep}}, \text{ナ}_{\mathrm{S}}, \text{ナ}_{\mathrm{L}}$ considered in this paper are parameterized by the set of locks that the programs can access. In the case of the two internal opcategories ナ_{S} and ナ_{L}, the free locks are simply described by a set $\mathfrak{L}$ of lock names. Then, a program $\mathfrak{L} \vdash C$ is interpreted as a cobordism in $\mathbf{Cob}(\text{ナ}(\mathfrak{L}))$.

In the case of the internal J-opcategories ナ_{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. A proof π of a Hoare triple $\Gamma \vdash \{P\}\ C\ \{Q\}$ is interpreted as cobordism in $\mathbf{Cob}(\text{ナ}_{\mathrm{Sep}}(\Gamma))$. The rules Res and With, which manipulate the context Γ of locks and invariants refine the two rules defined above.

As we wish to consider both cases at once, 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. This change-of-basis technique is similar in nature to the way categorical models of Algol such as (Reynolds, 1997) deal with lexical scoping of variables. Hence, we call the two operations defined in this section ***change of locks*** operations.

The change-of-basis operations are induced by the two acute spans below:

$$\begin{tikzcd}
\pitchfork(\Gamma) \arrow[r, bend left, "\shortmid"{marking}, "\mathsf{when}[r]"] & \pitchfork(\Gamma, r) \arrow[l, bend left, "\shortmid"{marking}, "\mathsf{hide}[r]"]
\end{tikzcd}$$

where we write explicitly the dependence of the models $\pitchfork(\Gamma)$ 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:

$$\begin{tikzcd}
\Gamma \arrow[r, bend left, "\iota_r^\Gamma"] & \Gamma, r \arrow[l, bend left, "\pi_r^\Gamma"]
\end{tikzcd}$$

Consider $\mathbf{LockGraph}^*$ 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 double functor from this bicategory see as a double category to $\mathbf{AcuteSpan}(\mathbb{S})$, the double category of internal J-opcategories and acute spans we defined in Section 3.3.2. 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 $\pitchfork = \pitchfork_{\mathrm{Sep}}, \pitchfork_{\mathrm{S}}, \pitchfork_{\mathrm{L}}$ 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 double functor $\mathbf{Cob}(\cdot)$, we are able to transport a cobordism defined on the internal category $\pitchfork(\Gamma)$ to one defined on the internal category $\pitchfork(\Gamma, r : I)$, to interpret critical sections, and back, for resource introduction. We abuse the notation and write these lax double functors as follows:

$$\begin{tikzcd}
\mathbf{Cob}(\pitchfork(\Gamma)) \arrow[r, bend left, "\mathsf{when}[r]"] & \mathbf{Cob}(\pitchfork(\Gamma, r)) \arrow[l, bend left, "\mathsf{hide}[r]"]
\end{tikzcd}$$

The fact that it is possible to rename bound lock names at the syntactic level is reflected in the semantics by the fact that the target of $\mathsf{hide}[r]$ does not contain the name r of the hidden lock.

3.5.1 Hiding

To hide a lock r, we proceed in two steps for all the templates that we consider $\pitchfork = \pitchfork_{\mathrm{L}}, \pitchfork_{\mathrm{S}}, \pitchfork_{\mathrm{Sep}}$. 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 $\mathsf{hide}[r]$:

$$\begin{tikzcd} \pitchfork(\Gamma, r) & \pitchfork^{\langle r\rangle}(\Gamma, r) \arrow[l, "\mathsf{inj}_C"'] \arrow[r, "\mathsf{hide}_C"] & \pitchfork(\Gamma). \end{tikzcd}$$

The support $\pitchfork^{\langle r\rangle}(\Gamma, r)$ of the span describes programs where the lock r is hidden from the environment, which means that only the Code can use it. Hence, $\pitchfork^{\langle r\rangle}(\Gamma, r)[1]$ is defined to be the same as the template $\pitchfork(\Gamma, r)$, except that all Environment transitions $P(r), V(r)$ are deleted, and, in the case of the template of separated states, we remove the states of the form $(\sigma_C, \vec{\sigma}, \sigma_F)$, where the lock r is held by the Environment (that is: $\vec{\sigma}(r) = \mathbf{F}$).

Moreover, we know that at the beginning and at the end of the execution of C in $\mathtt{resource}\, r\, \mathsf{do}\, C$, the resource r is unlocked. This is why we define $\pitchfork^{\langle r\rangle}(\Gamma, r)[0]$ to only contain states where the resource r is unlocked. Note that it means that $\pitchfork^{\langle r\rangle}(\Gamma, r)[0]$ is isomorphic to $\pitchfork(\Gamma)[0]$; a fact which will prove useful later.

By definition of $\pitchfork^{\langle r\rangle}(\Gamma, r)$ as a restriction of $\pitchfork(\Gamma, r)$, there is a canonical injection $\mathsf{inj}_C : \pitchfork^{\langle r\rangle}(\Gamma, r) \to \pitchfork(\Gamma, r)$. 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 $\pitchfork_{\mathrm{L}}$ and $\pitchfork_{\mathrm{S}}$, we map the states $L \subseteq \mathfrak{L}$ to $L \setminus \{r\}$, and $\mathfrak{s} = (\mu, L)$ to $(\mu, L \setminus \{r\})$ respectively. In the case of the template of separated states $\pitchfork_{\mathrm{Sep}}$ used to interpret proofs, it is defined as follows on separated states:

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

Intuitively, this means that after the lock is hidden, the resource it protects belongs to the Code. The fact that the lock has been hidden from the Environment ensures that the mapping above defines an asynchronous morphism.

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.

3.5.2 Critical sections

The case of critical sections is more delicate, as the definition differs between $\pitchfork_{\mathrm{L}}$, $\pitchfork_{\mathrm{S}}$ on the one hand, and $\pitchfork_{\mathrm{Sep}}$ 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 they happen 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 (they are not semaphores). This difference of requirements is reflected by differences in the definition of $\mathsf{when}[r]$.

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

$$\textit{Nodes:} \quad (\mu, L) \longmapsto (\mu, L \cap \mathfrak{L})$$

$$\textit{Edges:} \quad \begin{cases} P(r){:}\mathbf{F} & \longmapsto \mathsf{nop} \\ V(r){:}\mathbf{F} & \longmapsto \mathsf{nop} \\ m & \longmapsto m \end{cases}$$

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

$$\pitchfork(\mathfrak{L}) \xleftarrow{\quad\nabla\quad} \pitchfork_{\langle r\rangle}(\mathfrak{L} \uplus \{r\}) \xrightarrow{\quad\text{incl}\quad} \pitchfork(\mathfrak{L} \uplus \{r\})$$

where the right leg is the obvious inclusion. Intuitively, the operation of pulling along ∇, which defines the $\mathsf{when}[r]$ 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 switching between the copies, 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

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

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

3.6 Sum of cobordisms

We define the disjoint sum of two cobordisms. This will be useful as a building block for interpreting conditionals, and for the CSL rule for disjunction.

First, we remark that the category $\mathbf{opCat}(\mathbb{S})$ of internal J-opcategories has binary sums: the sum of an internal J-opcategory $\pm$ and an internal J'-opcategory $\pm'$ is an internal $J \times J'$-opcategory $\pm + \pm'$. For example, for $j \in J$ and $j' \in J'$, define

$$(\pm + \pm')[0,(j,j')] \coloneqq \pm[0,j] + \pm[0,j'].$$

Commutation of colimits ensures that this operation is well defined.

As usual, this operation on cobordisms is defined using a structure at the level of internal J-opcategories.

Definition 3.6.1. An ***internal J-opcategory with sums*** $(\pm, \vee)$ is an internal J-opcategory $\pm$ together with a symmetric injections $\vee : J \times J \to J$ in $\mathbb{S}$ and an internal functor of J-opcategories $\nabla : \pm + \pm \to \pm$ whose action on colors is the map $\vee$.

In particular, every internal opcategory has a canonical structure of internal opcategory with sums.

Given two cobordisms $\mathbf{C}$ and $\mathbf{D}$ over an internal J-opcategory with sums, their sum $\mathbf{C} \oplus \mathbf{D}$ is defined as the following cobordism, with the obvious notations:

$$\begin{tikzcd}
I + I' \arrow[r, "\mathrm{in}+\mathrm{in}"] \arrow[d, "\lambda_I+\lambda_{I'}"'] & S + S' \arrow[d, "\lambda_\sigma+\lambda_{\sigma'}"] & O + O' \arrow[l, "\mathrm{out}+\mathrm{out}"'] \arrow[d, "\lambda_O+\lambda_{O'}"] \\
\pm[0,i] + \pm[0,i'] \arrow[r, "\mathrm{in}_{ij}+\mathrm{in}_{i'j'}"] \arrow[d, "\nabla"] & \pm[1,ij] + \pm[1,i'j'] \arrow[d, "\nabla"] & \pm[0,j] + \pm[0,j'] \arrow[l, "\mathrm{out}_{ij}+\mathrm{out}_{i'j'}"'] \arrow[d, "\nabla"] \\
\pm[0,i \vee j] \arrow[r, "\mathrm{in}_{i\vee i' j\vee j'}"] & \pm[1,i \vee i'j \vee j'] & \pm[0,j \vee j'] \arrow[l, "\mathrm{out}_{i\vee i' j\vee j'}"']
\end{tikzcd}$$

3.7 Interpretation of codes and proofs

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

3.7.1 Stateful and stateless interpretations of the code

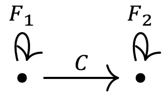
We begin by describing how the stateful and stateless interpretations $[\![C]\!]_{\mathrm{S}}$ and $[\![C]\!]_{\mathrm{L}}$ of a given code C are computed in our template game model. Since the interpretation is uniform in $\pitchfork_{\mathrm{S}}$ and $\pitchfork_{\mathrm{L}}$ and only depends on the combinators of C, we find it convenient to write $\pitchfork$ alternatively for $\pitchfork_{\mathrm{S}}$ or $\pitchfork_{\mathrm{L}}$. The code C with free locks $\mathfrak{L}$ is interpreted by structural induction as a cobordism of the form

$$[\![C]\!] \quad = \quad \begin{tikzcd}
[\![C]\!]_{in} \arrow[r] \arrow[d, "\lambda_{\mathrm{in}}"'] & [\![C]\!]_{support} \arrow[d] & [\![C]\!]_{out} \arrow[l] \arrow[d, "\lambda_{\mathrm{out}}"] \\
\pitchfork(\mathfrak{L})[0] \arrow[r] & \pitchfork(\mathfrak{L})[1] & \pitchfork(\mathfrak{L})[0]. \arrow[l]
\end{tikzcd} \tag{3.4}$$

The interpretation of every non-leaf command of the language corresponds to an operation on cobordisms already defined, and it is thus straightforward.

Instructions We explain first how to define the cobordism that interprets a single instruction m. 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 $\pitchfork(\mathfrak{L})[0]$, while its support $[\![m]\!]_{support}$ consists of the disjoint union of $[\![m]\!]_{in}$ and $[\![m]\!]_{out}$ augmented with an edge $s_1 \to s_2$ from $s_1 \in [\![m]\!]_{in}$ to $s_2 \in [\![m]\!]_{out}$ 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 $[\![m]\!]_{out}$ are Code transitions, with the state s_2 potentially equal to the error state.

This definition can be formulated using a well-chosen pullback: Consider the following asynchronous graph

$A =$ 

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 and Ω : **Set** → **AsyncGraph** is the functor we used to define $\pitchfork^{\circ\bullet}$ page 72.

$$\begin{tikzcd}
G(m) \arrow[r, hook] \arrow[d] & A \times \pitchfork^{\bullet} \arrow[d, "f"] \\
\Omega(\{F, m\}) \arrow[r, hook] & \Omega(\{F\} \cup \mathbf{Instr})
\end{tikzcd}$$

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 of the cobordism to its support using the universal property of the pullback.

Leaf codes Except for `malloc` which is treated below, every leaf command $x \coloneqq E \mid x \coloneqq [E] \mid [E] \coloneqq E' \mid \texttt{skip} \mid \texttt{dispose}(E)$ of the language corresponds to a specific machine instruction m which is interpreted as the cobordism $[\![m]\!]$ we have just defined. The case of `malloc` is different because it needs to choose non-deterministically the address which is allocated; as such, we take the disjoint sum of all the semantics of the basic instructions $\texttt{alloc}(E, \ell)$:

$$[\![\texttt{malloc}(E)]\!] \quad \coloneqq \quad \mathsf{fill}\left(\mathrm{Id}_{\pitchfork[0]}, \ldots\right) \ ; \bigoplus_{\ell \in \mathbf{Loc}} [\![\texttt{alloc}(E, \ell)]\!]$$

where we pre-compose with a filling system to normalize the initial states (the second component of fill is determined by the cobordism on its right.)

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 $[\![C]\!]_{\mathrm{S}}$ and $[\![C]\!]_{\mathrm{L}}$ in the same way.

Conditionals Conditional branching is interpreted as

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

defined as

$$[\![\texttt{test}(B)]\!]; [\![C_1]\!] \oplus [\![\texttt{test}(\neg B)]\!]; [\![C_2]\!] \tag{3.5}$$

precomposed with

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

as depicted at the beginning of Section 3.4, page 84.

Here, the purpose of precomposing with the filling is to identify the two copies of the input $\pitchfork[0]$ appearing on each sides of the disjoint sum (3.5).

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

$$[\![C_1 \| C_2]\!] \ = \ [\![C_1]\!] \| [\![C_2]\!] \qquad [\![C_1; C_2]\!] \ = \ [\![C_1]\!] \mathbin{⨟} [\![C_2]\!].$$

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:

$$[\![P(r)]\!] \mathbin{⨟} \mathsf{when}[r]\big([\![C]\!]\big) \mathbin{⨟} [\![V(r)]\!].$$

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

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

seen as an endofunctor on the category of arrows of the double category $\mathbf{Cob}(\pm)$ 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.

3.7.2 Interactive and separated interpretations of the proofs

Our purpose is to interpret by structural induction every CSL proof $\pi : \Gamma \vdash \{P\}\, C\, \{Q\}$ as a cobordism of the form

$$[\![\pi]\!]_{\mathrm{Sep}} \quad = \quad \begin{tikzcd} {[\![\pi]\!]_{Sep,in}} \arrow[r] \arrow[d] & {[\![\pi]\!]_{Sep,support}} \arrow[d] & {[\![\pi]\!]_{Sep,out}} \arrow[l] \arrow[d] \\ \pm_{\mathrm{Sep}}(\Gamma)[0,P] \arrow[r] & \pm_{\mathrm{Sep}}(\Gamma)[1] & \pm_{\mathrm{Sep}}(\Gamma)[0,Q]. \arrow[l] \end{tikzcd} \tag{3.6}$$

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. 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 $[\![\pi]\!]_{\mathrm{Sep}}$ 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 $[\![\pi]\!]_{\mathrm{Sep}} \parallel \pm_{\mathrm{Sep}}[0,R]$, where the asynchronous graph $\pm_{\mathrm{Sep}}[0,R]$, which contains only states which statisfy the predicate R and Environment transitions, is lifted to the identity cobordism defined in the expected way in the double category $\mathbf{Cob}(\pm_{\mathrm{Sep}})$.

The rules that correspond to machine instructions $m \in \mathbf{Instr}$ (such as Ld) are interpreted in a way which is similar to the interpretation of the corresponding codes, always preserving the permission associated with affected locations. For instance, the axiom rule

$$\frac{\qquad}{\Gamma \vdash \{((\mathrm{Own}_{\mathrm{T}}(x) * P) \land E{=}w){*}w \overset{p}{\mapsto} v\}\; x := [E]\; \{((\mathrm{Own}_{\mathrm{T}}(x) * P) \land x{=}v){*}w \overset{p}{\mapsto} v\}}\;\textsc{Ld}$$

is interpreted as a cobordisms whose Code edges are all the edges from an initial state to an final state of the form

$$((s,h),\vec{\sigma},\sigma_F) \quad \xrightarrow{x:=[E]:\mathrm{C}} \quad ((s[x \mapsto v],h),\vec{\sigma},\sigma_F)$$

where h contains a mapping $[w \to v]$, and where $E = w$ holds in h.

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

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

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

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

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

$$\left[\!\!\left[\dfrac{\overset{\vdots\ \pi}{\Gamma, r : J \vdash \{(P * J) \land B\}\, C\, \{Q * J\}}}{\Gamma, r : J \vdash \{P\}\, \mathtt{with}\ r\ \mathsf{do}\ C\, \{Q\}}\right]\!\!\right]_{\mathrm{Sep}} \quad = \quad \begin{array}{l} \mathrm{acquire}[r]; \\ \mathsf{when}[r]([\![\pi]\!]_{\mathrm{Sep}})\,; \\ \mathrm{release}[r] \end{array}$$

$$\left[\!\!\left[\dfrac{\overset{\vdots\ \pi_1}{\Gamma \vdash \{P_1\}\, C\, \{Q_1\}} \quad \overset{\vdots\ \pi_2}{\Gamma \vdash \{P_2\}\, C\, \{Q_2\}}}{\Gamma \vdash \{P_1 \lor P_2\}\, C\, \{Q_1 \lor Q_2\}}\right]\!\!\right]_{\mathrm{Sep}} \quad = \quad [\![\pi_1]\!]_{\mathrm{Sep}} \oplus [\![\pi_2]\!]_{\mathrm{Sep}}$$

In the definition of the interpretation of With, the cobordisms acquire$[r]$ and release$[r]$ are the counterparts of $[\![P(r)]\!]_{\mathrm{S}}$ and $[\![V(r)]\!]_{\mathrm{S}}$ at the level of separated states.

3.8 The asynchronous soundness theorem

We explain how to state the asynchronous soundness theorem of concurrent separation logic which we described in the previous chapter in Section 2.5.3 in the language of cobordisms.

3.8.1 Comparing the three interpretations

Given a code C and a proof π of $\Gamma \vdash \{P\}\,C\,\{Q\}$, the proof of the asynchronous soundness relies on the existence of a chain of *translations*

$$[\![\pi]\!]_{\mathrm{Sep}} \xrightarrow{\mathcal{S}} [\![C]\!]_{\mathrm{S}} \xrightarrow{\mathcal{L}} [\![C]\!]_{\mathrm{L}} \tag{3.7}$$

between cobordisms living respectively in $\pitchfork_{\mathrm{Sep}}(\Gamma)$, $\pitchfork_{\mathrm{S}}(\mathfrak{L})$ and $\pitchfork_{\mathrm{L}}(\mathfrak{L})$. Here, we write $\mathfrak{L}$ 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 internal J-opcategories:

$$\pitchfork_{\mathrm{Sep}}(\Gamma) \xrightarrow{u_{\mathcal{S}}} \pitchfork_{\mathrm{S}}(\mathfrak{L}) \xrightarrow{u_{\mathcal{L}}} \pitchfork_{\mathrm{L}}(\mathfrak{L}).$$

The first internal functor $u_{\mathcal{S}}$ transports every separated state $(\sigma_C, \vec{\sigma}, \sigma_F)$ into the machine state obtained by multiplying all its components as in Definition 2.5.4 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 $\pitchfork_{\mathrm{Sep}}$, $\pitchfork_{\mathrm{S}}$ and $\pitchfork_{\mathrm{L}}$ 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. This leads to the following definition.

Definition 3.8.1. Given two internal J-opcategories $\pitchfork$ and $\pitchfork'$, and two cobordisms

$$\mathbf{C} : A \;\not\to\; B \text{ in } \mathbf{Cob}(\pitchfork) \qquad \text{and} \qquad \mathbf{C}' : A' \;\not\to\; B' \text{ in } \mathbf{Cob}(\pitchfork')$$

where we write A to denote the 0-cell $A \to \pitchfork[0, i]$ of $\mathbf{Cob}(\pitchfork)$, etc, a ***map of cobordisms*** $(\mathcal{F}, \mathcal{F}_A, \mathcal{F}_B, \mathcal{F}_\pitchfork)$ from $\mathbf{C}$ to $\mathbf{C}'$ is the data of an internal functor of J-opcategories $\mathcal{F}_\pitchfork : \pitchfork \to \pitchfork'$ and of a 2-cell $\mathcal{F}$ and two vertical 1-cells $\mathcal{F}_A$ and $\mathcal{F}_B$ in $\mathbf{Cob}(\pitchfork')$ of the form:

$$\begin{tikzcd}
\mathsf{push}[\mathcal{F}_\pitchfork](A) \arrow[rr, "{\mathsf{push}[\mathcal{F}_\pitchfork](\mathbf{C})}"] \arrow[d, "\mathcal{F}_A"'] & {} \arrow[d, Rightarrow, "\mathcal{F}"] & \mathsf{push}[\mathcal{F}_\pitchfork](B) \arrow[d, "\mathcal{F}_B"] \\
A' \arrow[rr, "\mathbf{C}'"'] & {} & B'
\end{tikzcd}$$

This means that the translations in (3.7) are simulations:

$$\mathsf{push}[u_{\mathcal{S}}]([\![\pi]\!]_{\mathrm{Sep}}) \xrightarrow{\mathcal{S}} [\![C]\!]_{\mathrm{S}} \quad \text{and} \quad \mathsf{push}[u_{\mathcal{L}}]([\![C]\!]_{\mathrm{S}}) \xrightarrow{\mathcal{L}} [\![C]\!]_{\mathrm{L}}.$$

Because the semantic interpretations of programs and proofs are defined by structural induction, we need the semantic combinators we use to “preserve” maps of cobordisms. The following lemmas provide conditions for their preservation.

The definition of internal functor of J-opcategories 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 $\mathbb{S} = \mathbf{AsyncGraph}$. 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 following lemma gives a condition for the sequential product to preserve maps of cobordisms.

Lemma 3.8.2. *Let $\pm$ and $\pm'$ be two internal J-opcategories related by an internal functor of J-opcategories $u : \pm \to \pm'$. Consider two composable cobordisms $\mathbf{C} : I \to M$ and $\mathbf{D} : M \to O$ in $\mathbf{Cob}(\pm)$, and two composable cobordisms $\mathbf{C}' : I' \to M'$ and $\mathbf{D} : M' \to O'$ in $\mathbf{Cob}(\pm')$. Suppose finally that they are related by two maps of cobordisms $\mathcal{F} : \mathbf{C} \to \mathbf{C}'$ and $\mathcal{G} : \mathbf{D} \to \mathbf{D}'$ such that $u = \mathcal{F}_{\pm} = \mathcal{G}_{\pm}$ and $\mathcal{F}_M = \mathcal{G}_M$. Then there is a map of cobordisms*

$$\mathcal{F};\mathcal{G} \;:\; \mathbf{C};\mathbf{D} \longrightarrow \mathbf{C}';\mathbf{D}'$$

Proof. The desired 2-cell in $\mathbf{Cob}(\pm')$ is the following, where the top invertible 2-cell is the isomorphism given above.

$$\begin{tikzcd}
\mathsf{push}[u](I) \arrow[rrrr, "\mathsf{push}[u](\mathbf{C};\mathbf{D})", "|" marking] \arrow[d, equal] & & {} \arrow[d, Rightarrow, "\wr"] & & \mathsf{push}[u](O) \arrow[d, equal] \\
\mathsf{push}[u](I) \arrow[rr, "\mathsf{push}[u](\mathbf{C})", "|" marking] \arrow[d, "\mathcal{F}_I"'] & {} \arrow[d, Rightarrow, "\mathcal{F}"'] & \mathsf{push}[u](M) \arrow[rr, "\mathsf{push}[u](\mathbf{D})", "|" marking] \arrow[d, "\mathcal{F}_M"', "\mathcal{G}_M"] & {} \arrow[d, Rightarrow, "\mathcal{G}"] & \mathsf{push}[u](O) \arrow[d, "\mathcal{F}_O"] \\
I' \arrow[rr, "\mathbf{C}'"', "|" marking] & {} & M' \arrow[rr, "\mathbf{D}'"', "|" marking] & {} & O'
\end{tikzcd}$$

□

We defer to Lemma 3.9.2 the explanation of why the condition $\mathcal{F}_M = \mathcal{G}_M$ is always true for the cobordisms which we compose sequentially.

The preservation of the maps of cobordisms by the action of the double functor $\mathbf{Cob}$ of Lemma 3.3.4 is expressed in the following lemma.

Lemma 3.8.3. *Let* $\text{ⵜ}_1, \text{ⵜ}_2, \text{ⵜ}'_1$ *and* $\text{ⵜ}'_2$ *be internal* J*-opcategories,* $\mathbf{C} : A \mathrel{\rightarrow\!\!\!\!\!\!+\;} B$ *be a cobordism in* $\mathbf{Cob}(\text{ⵜ}_1)$ *and* $\mathbf{C}' : A' \mathrel{\rightarrow\!\!\!\!\!\!+\;} B'$ *be one in* $\mathbf{Cob}(\text{ⵜ}'_1)$*. Suppose moreover given two acute spans* (F, G) *and* (F', G') *related as follows in the double category* **AcuteSpan***:*

$$\begin{tikzcd}
\text{ⵜ}_1 \arrow[r, "{(F,G)}", "|" marking] \arrow[d, "u"'] & \text{ⵜ}_2 \arrow[d, "v"] \\
\text{ⵜ}'_1 \arrow[r, "{(F',G')}"', "|" marking] & \text{ⵜ}'_2
\arrow[from=1-1, to=2-1, phantom, shift left=8ex, "\Downarrow\alpha"]
\end{tikzcd}$$

Then a map of cobordisms $\mathcal{F} : \mathbf{C} \to \mathbf{C}'$ *such that* $\mathcal{F}_{\text{ⵜ}} = u$ *induces a map of cobordisms from* $\mathbf{Cob}((F, G))(\mathbf{C})$ *to* $\mathbf{Cob}((F', G'))(\mathbf{C}')$ *which is above* v*.*

Proof. The induced map of cobordism is given by the following vertical composition of 2-cells in $\mathbf{Cob}(\text{ⵜ}'_2)$, and the lax double functor structure of $\mathbf{Cob}(\cdot)$.

$$\begin{tikzcd}
\mathrm{push}[v](\mathbf{Cob}((F,G))(A)) \arrow[r, "{\mathrm{push}[v](\mathbf{Cob}((F,G))(\mathbf{C}))}", "|" marking] \arrow[d, "\mathbf{Cob}(\alpha)_A"'] & \mathrm{push}[v](\mathbf{Cob}((F,G))(B)) \arrow[d, "\mathbf{Cob}(\alpha)_B"] \\
\mathbf{Cob}((F',G'))(\mathrm{push}[u](A)) \arrow[r, "{\mathbf{Cob}((F',G'))(\mathrm{push}[u](\mathbf{C}))}", "|" marking] \arrow[d, "{\mathbf{Cob}((F',G'))(\mathcal{F}_A)}"'] & \mathbf{Cob}((F',G'))(\mathrm{push}[u](B)) \arrow[d, "{\mathbf{Cob}((F',G'))(\mathcal{F}_B)}"] \\
\mathbf{Cob}((F',G'))(A') \arrow[r, "{\mathbf{Cob}((F',G'))(\mathbf{C}')}"', "|" marking] & \mathbf{Cob}((F',G'))(B')
\arrow[from=1-1, to=2-1, phantom, shift left=20ex, "\Downarrow\mathbf{Cob}(\alpha)_{\mathbf{C}}"]
\arrow[from=2-1, to=3-1, phantom, shift left=20ex, "\Downarrow\mathbf{Cob}((F',G'))(\mathcal{F})"]
\end{tikzcd}$$

□

3.8.2 The asynchronous soundness theorem

As we have explained in the previous chapter, the asynchronous theorem of CSL uses the notion of fibration. Definition 2.4.6 introduces different notions of fibration for asynchronous morphisms. We can lift this notion to maps of cobordisms by considering the asynchronous morphism between the supports which is induced by such a map.

We can now state the asynchronous soundness theorem for CSL 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

$$[\![\pi]\!]_{\mathrm{Sep}} \xrightarrow{\quad \mathcal{S} \quad} [\![C]\!]_{\mathrm{S}} \xrightarrow{\quad \mathcal{L} \quad} [\![C]\!]_{\mathrm{L}}$$

discussed in §3.8.1. The theorem states:

Theorem 3.8.4 (Asynchronous soundness theorem)**.** *For every CSL proof π of $\Gamma \vdash \{P\}C\{Q\}$, the comparison map $\mathcal{S} : [\![\pi]\!]_{Sep} \to [\![C]\!]_S$ is a Code 1-fibration, and the comparison map $\mathcal{L} \circ \mathcal{S} : [\![\pi]\!]_{Sep} \to [\![C]\!]_L$ is a 2-fibration.*

As we explained in Section 2.5.3, the first part of the theorem ensures that a program specified in CSL does not crash. Indeed, the cobordism $[\![\pi]\!]_{\mathrm{Sep}}$ lives above $\pm_{\mathrm{Sep}}[1]$ which does not include the error state $\lightning$. Since every step performed by the Code in $[\![C]\!]_{\mathrm{S}}$ can be lifted to $[\![\pi]\!]_{\mathrm{Sep}}$ 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 $[\![C]\!]_{\mathrm{L}}$, which can be lifted by the 2-fibrational property to a tile in the cobordism $[\![\pi]\!]_{\mathrm{Sep}}$ of separated states. There, the very existence of the tile implies that these two instructions are independent and do not produce any data race.

3.9 Proof of the asynchronous soundness theorem

The general method for proving Theorem 3.8.4 is to prove by structural induction over the proof trees of CSL that the maps $\mathcal{S}$ and $\mathcal{L}$ exist, have the fibrational properties stated in the theorem for the axioms of the logic, and that the maps and these properties are preserved by the rules of the logic.

3.9.1 Well-formed cobordisms

To do so, we need to refine the kind of cobordisms we use to interpret programs and proofs: We require that they satisfy a number of properties.

Definition 3.9.1. A cobordism

$$\begin{tikzcd}
I \arrow[r, "\mathrm{in}"] \arrow[d, "\lambda_I"'] & S \arrow[d, "\lambda_\sigma"] & O \arrow[l, "\mathrm{out}"'] \arrow[d, "\lambda_O"] \\
\pm[0,i] \arrow[r, "\mathrm{in}_{ij}"] & \pm[1,ij] & \pm[0,j] \arrow[l, "\mathrm{out}_{ij}"']
\end{tikzcd}$$

in an ambient category $\mathbb{S}$ is said to be ***well-formed*** if:

1. the map $\lambda_I : I \to \pm[0,i]$ is an isomorphism;

2. the map λ_σ is an Environment 1-fibration;
3. the map $\mathrm{out} : O \to S$ is a mono;
4. the pullback of $\mathrm{in} : I \to S$ along $\mathrm{out} : O \to S$ is the initial object of $\mathbb{S}$.

The conditions 1. and 2. are related to receptivity in games semantics: the first means that any initial state in $\pm[0, i]$ must be "accepted" by the program or the proof, and the second means that the Frame can always play any valid transition, or move. Condition 3. and 4. are more technical and are needed to ensure that sequential composition behaves nicely with respect to fibrational properties.

The condition that λ_I be a monomorphism (since it is an isomorphism) is used to ensure that the maps of cobordisms$\mathcal{S}$ and $\mathcal{L}$ always exist.

Lemma 3.9.2. *Consider two composable cobordisms* $\mathbf{C} : I \to M$ *and* $\mathbf{D} : M \to O$ *in* $\mathbf{Cob}(\pm)$*, and two composable cobordisms* $\mathbf{C}' : I' \to M'$ *and* $\mathbf{D} : M' \to O'$ *in* $\mathbf{Cob}(\pm')$*. Suppose finally that they are related by two maps of cobordisms* $\mathcal{F} : \mathbf{C} \to \mathbf{C}'$ *and* $\mathcal{G} : \mathbf{D} \to \mathbf{D}'$ *such that* $\mathcal{F}_\pm = \mathcal{G}_\pm$*. If* $\mathbf{D}'$ *is well-formed, then* $\mathcal{F}_M = \mathcal{G}_M$*.*

Proof. This follows from $\lambda'_M \circ \mathcal{F}_M = u \circ \lambda_M = \lambda'_M \circ \mathcal{G}_M$, where $u = \mathcal{F}_\pm = \mathcal{G}_\pm$ and from the fact that λ'_M is a mono. □

We also require that the internal J-opcategories we use to interpret have good fibrational properties:

Definition 3.9.3. An internal J-opcategory $\pm$ is called ***$\mathcal{M}$-fibrational*** if the maps $\mathrm{in}_{ij} : \pm[0, i] \to \pm[1, ij]$ and $\mathrm{out}_{ij} : \pm[0, j] \to \pm[1, ij]$ are monos, $\mathcal{M}$-1-fibrations and 2-fibrations, and if the map $\pm[2, ijk] \to \pm[1, ik]$ are an $\mathcal{M}$-1-fibration. This induces the notion of ***Environment-fibrational*** internal J-opcategories.

Every internal J-opcategories $\pm_{\mathrm{Sep}}$, $\pm_{\mathrm{S}}$ and $\pm_{\mathrm{L}}$ we have seen so far is Environment-fibrational.

Before we proceed to proving that all operations we use on cobordisms preserve well-formedness, we characterize fibrations as maps which satisfy a right lifting property. Let us write $\cdot\!\to\!\cdot$ for the asynchronous graph with two nodes 0 and 1 an and a unique edge $0 \to 1$. We write $\to_C$ when this unique edge is considered a Code transition, and $\to_F$ when it is considered as a Frame transition. We write $\cdot$ for the asynchronous graph with a unique node, and no edge or tile. We denote by $s : \cdot \longrightarrow \cdot\!\to\!\cdot$ the asynchronous morphism which maps that single node to the node 0 of $\cdot\!\to\!\cdot$.

Lemma 3.9.4. *Let $f : G \to G'$ be an asynchronous morphism between asynchronous graphs with a notion of Code and Frame edge. Then f is a Code 1-fibration (respectively an Environment 1-fibration) iff, for all squares of polarity preserving morphisms as in the left-hand diagram below (respectively the right-hand diagram), there exists a map denoted by the dashed arrow which makes both triangles commute.*

$$\begin{tikzcd}
\cdot \arrow[r] \arrow[d, "s"'] & G \arrow[d, "f"] \\
\to_C \arrow[r] \arrow[ur, dashed] & G'
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r] \arrow[d, "s"'] & G \arrow[d, "f"] \\
\to_F \arrow[r] \arrow[ur, dashed] & G'
\end{tikzcd}$$

Proof. Follows easily from the definitions. □

We can characterize 2-fibrations in the same way. Let $\wedge$ be the asynchronous graph with three nodes $0, 1, 2$ and two edges $0 \to 1 \to 2$, and let $\blacklozenge$ be the asynchronous graph with 4 nodes $0, 1, 1', 2$ and four edges $0 \to 1 \to 2$ and $0 \to 1' \to 2$ and a unique tile which sits on the two paths of length two above. Write $u : \wedge \to \blacklozenge$ for the inclusion, mapping the path of length two in $\wedge$ to the "upper path" of the tile in $\blacklozenge$. Similarly, we have:

Lemma 3.9.5. *Let $f : G \to G'$ be an asynchronous morphism between two asynchronous graphs. Then f is a 2-fibration iff, for all squares as in the diagram below, there exists a map denoted by the dashed arrow which makes both triangles commute.*

$$\begin{tikzcd}
\wedge \arrow[r] \arrow[d, "u"'] & G \arrow[d, "f"] \\
\blacklozenge \arrow[r] \arrow[ur, dashed] & G'
\end{tikzcd}$$

Proof. Follows easily from the definitions. □

One immediate consequence of this diagrammatic characterization of fibrations is that they are stable under pullbacks, in the following sense.

Lemma 3.9.6. *Given an morphism $f : G \to G'$ in $\mathbb{S}$, suppose g its pullback along some map $u : H' \to G'$:*

$$\begin{tikzcd}
H \arrow[r] \arrow[d, "g"'] \arrow[dr, phantom, "p.b."] & G \arrow[d, "f"] \\
H' \arrow[r, "u"'] & G'
\end{tikzcd}$$

then g is a fibration of some type (Code or Environment 1-fibrations, or 2-fibrations) if f is.

Proof. Assume f is a fibration characterized by a right lifting property with respect to a map $i : X \to Y$

$$\begin{tikzcd}
X \arrow[r] \arrow[d, "i"'] & H \arrow[r] \arrow[d, "g"'] \arrow[dr, phantom, "p.b."] & G \arrow[d, "f"] \\
Y \arrow[r] & H' \arrow[r, "u"'] & G'
\end{tikzcd}$$

then there exists a map $a : Y \to G$ making the two triangles commute. The universal property of the pullback implies that there exists a unique map $a' : Y \to H$ which makes the two triangles commute, and therefore g is a fibration as well. □

Another important observation, especially when dealing with colimits, is that of all the "test" objects ($\cdot$, $\cdot \to \cdot$, $\wedge$, $\blacklozenge$) only $\wedge$ is not representable in **AsyncGraph**, and hence is not tiny, since, in presheaf categories, tiny objects are the retracts of representables.

Definition 3.9.7. An object A of a category $\mathcal{C}$ is called ***tiny*** if the functor $\mathcal{C}(A, -)$ preserves colimits.

We now introduce another notion, adhesivity, which is going to be useful to reason about sequential composition of cobordisms.

3.9.2 Adhesivity of AsyncGraph

There are several equivalent definitions of adhesivity, the one we present here, which is not the tersest, has been chosen because it is used directly in the sequel.

Definition 3.9.8. A category $\mathcal{C}$ is ***adhesive*** if all pushouts along monos exist, and if all such pushout squares are van Kampen squares. A commutative square

$$\begin{tikzcd}
A \arrow[r, "g"] \arrow[d, "f"'] & C \arrow[d, "v"] \\
B \arrow[r, "u"'] & D
\end{tikzcd}$$

is said to be a ***van Kampen square*** if, for every cubical commutative diagram of the form

$$\begin{tikzcd}
W \arrow[rr] \arrow[dr] \arrow[dd] & & Y \arrow[dr] \arrow[dd] & \\
& X \arrow[rr] \arrow[dd] & & Z \arrow[dd] \\
A \arrow[rr, "g"] \arrow[dr, "f"'] & & C \arrow[dr, "v"] & \\
& B \arrow[rr, "u"'] & & D
\end{tikzcd}$$

whose two "back faces" (W, A, X, B and W, A, Y, C) are pullbacks, the top face is a pushout if and only if the two front faces (X, Z, B, D and Y, Z, C, D) are pullbacks. We call such a commutative cube a ***van Kampen cube***.

The category **AsyncGraph** of asynchronous graphs is adhesive since it is a presheaf category, and therefore a topos.

The following consequences of adhesivity will be useful, see for example (Lack and Sobociński, 2004).

Lemma 3.9.9. *In an adhesive category, pushouts of monos are monos, and the resulting pushout squares are also pullback squares.*

Another useful fact is that a the coproducts are well-behaved as soon as an adhesive category has strict initial objects.

Lemma 3.9.10. *Any adhesive category $\mathbb{S}$ which has a strict initial object is extensive. In particular, this means that its coproducts are disjoint, which means that coprojections are monic, and their pullback is an initial object in $\mathbb{S}$.*

3.9.3 Preservation of well-formedness

We prove that the operations that we use to interpret programs and proofs preserve the well-formedness of cobordisms. The cases of operations such as the parallel product and the two change of locks operations follows from the lemma below, which relies on the following property of the acute spans we use.

Definition 3.9.11. An acute span **S** of internal J-opcategories

$$\begin{tikzcd}
ナ_1 & ナ_2 \arrow[l, "F"'] \arrow[r, "G"] & ナ_3
\end{tikzcd}$$

is called ***strict*** if the following squares are pullbacks:

$$\begin{tikzcd}
\mp_2[0,i] \arrow[r, "\mathrm{in}_i"] \arrow[d, "{F[0,i]}"'] & \mp_2[1] \arrow[d, "{F[1]}"] & \mp_2[0,j] \arrow[l, "\mathrm{out}_j"'] \arrow[d, "{F[0,j]}"] \\
\mp_1[0,i] \arrow[r, "\mathrm{in}_i"'] & \mp_1[1] & \mp_1[0,j] \arrow[l, "\mathrm{out}_j"']
\end{tikzcd}$$

Lemma 3.9.12. *Given a strict acute span* **S** *of internal J-opcategories*

$$\begin{tikzcd}
\mp_1 & \mp_2 \arrow[l, "F"'] \arrow[r, "G"] & \mp_3
\end{tikzcd}$$

the operation on cobordisms $X \mapsto$ **Cob**(**S**)(X) *preserves well-formedness (*except for condition 1 of Definition 3.9.1*) if $G[1]$ is an Environment 1-fibration and a mono. If moreover the maps $G[0,i]$ are isos, then Condition 1 is also preserved.*

Proof. Let the following be a well-formed cobordism over the internal J-opcategory $\mp_1$:

$$\begin{tikzcd}
I \arrow[r, "\mathrm{in}"] \arrow[d, "\lambda_I"'] & S \arrow[d, "\lambda_\sigma"] & O \arrow[l, "\mathrm{out}"'] \arrow[d, "\lambda_O"] \\
\mp_1[0,i] \arrow[r, "\mathrm{in}_{ij}"] & \mp_1[1,ij] & \mp_1[0,i] \arrow[l, "\mathrm{out}_{ij}"']
\end{tikzcd}$$

and let the following cobordism over $\mp_2$ be obtained by pulling back along the components of the internal functor F using the pull$[F]$ lax double functor of Lemma 3.3.1:

$$\begin{tikzcd}
I' \arrow[r, "\mathrm{in}'"] \arrow[d, "\lambda_{I'}"'] & S' \arrow[d, "\lambda_{\sigma'}"] & O' \arrow[l, "\mathrm{out}'"'] \arrow[d, "\lambda_{O'}"] \\
\mp_2[0,i] \arrow[r, "\mathrm{in}_{ij}"] & \mp_2[1,ij] & \mp_2[0,i] \arrow[l, "\mathrm{out}_{ij}"']
\end{tikzcd}$$

We need to check the four conditions of Definition 3.9.1.

(1) Isos are stable under pullback in **AsyncGraph**, so $\lambda_{I'}$ is an iso as well. If the condition that $G[0,i]$ is an iso holds, the operation push$[F]$ also preserves Condition 1.

(2) Fibrations are stable under pullback, so $\lambda_{\sigma'}$ is also a 1-fibration. The hypothesis on $G[1]$ implies that Condition (2) is preserved by post-composing.

(3) The map $\mathrm{out}' : O' \to S'$ is a mono because, according to the pasting lemma on pullbacks and the fact that $G[1]$ is a mono, it is the pullback of $\mathrm{out} : O \to S$ along the map $O' \to O$ which is part of the pullback defining O'.

(4) The universal property of pullbacks implies that there is a map from the pullback X of I' and O' to the pullback of I and O, which is an initial object of **AsyncGraph** since it is a well-formed cobordism. Since **AsyncGraph** has strong initial objects, X is an initial object as well.

□

All the operations on cobordisms, the parallel product, hiding and lifting of locks (except for Condition 1, see below), are defined using acute spans which satisfy the conditions of the lemma above, and therefore they preserve well-formedness.

To prove that the other operations also preserve well-formedness, we need to prove that sequential composition preserves well-formedness. This fact relies on the following result.

Lemma 3.9.13. *Suppose given the following diagram:*

$$\begin{tikzcd}
 & Y \arrow[rrrr, "\lambda_2"] \arrow[dr, "f'"] & & & & C \arrow[dr, "i'"] & \\
 & po & Z \arrow[rrrr, dashed, "\mu"] & & & po & D \\
W \arrow[uur, "f"] \arrow[rrrr] \arrow[dr, "g"'] & & & & A \arrow[uur, "i"] \arrow[dr, "j"] & & \\
 & X \arrow[uur, "g'"] \arrow[rrrr, "\lambda_1"] & & & & B \arrow[uur, "j'"'] &
\end{tikzcd}$$

where λ_1, λ_2, i and j are Environment 1-fibrations, and i, j are monos. Then the map μ induced by the universality of the pushout is an Environment 1-fibration as well.

Proof. First, because **AsyncGraph** is adhesive, i' and j' are monos as well. 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. □

It is now easy to prove that sequential composition preserves well-formedness.

Lemma 3.9.14. *Let* $\mathbf{C} : I \nrightarrow I'$ *and* $\mathbf{D} : I' \nrightarrow O'$ *be two well-formed cobordisms over one of the internal* J*-opcategories we have defined above. Then* $\mathbf{C};\mathbf{D}$ *is well-formed. This is also the case if* $\mathbf{D}$ *does not satisfy Condition 1 of Definition 3.9.1.*

Proof. Condition 1 is obvious, Condition 2 is the lemma above, Condition 3 follows from the stability of monos under pullback. Condition 4 follows from the fact that pushouts along monos are pullbacks in the diagram

$$\begin{tikzcd}
& & X \arrow[d, "\wr"] \arrow[dddll, bend right] \arrow[dddrr, bend left] & & \\
& & \emptyset \arrow[dl] \arrow[dr] & & \\
& \emptyset \arrow[dl] \arrow[dr] & & \emptyset \arrow[dl] \arrow[dr] & \\
I \arrow[dr] & & I' \arrow[dl] \arrow[dr] & & O' \arrow[dl] \\
& C \arrow[dr] & & D \arrow[dl] & \\
& & C;D & &
\end{tikzcd}$$

where X is the pullback of the map $I \to C;D$ along $O' \to C;D$, and where the four "small" squares are pullbacks for the reason above. □

The operation for which we do not yet have the tools to prove that it preserves well-formedness is the generalized sequential composition with filling systems, because they are not well-formed cobordisms, as they fail to satisfy Conditions 3 and 4.

Lemma 3.9.15. *Given the following commutative diagram, where the square is a pushout* *and o' is a mono:*

$$\begin{tikzcd}
& M \arrow[dl, "o"'] \arrow[dr, "i'"] & & O' \arrow[dl, hook, "o'"'] \\
C \arrow[dr, "f"'] & & C' \arrow[dl, "g"] & \\
& D & &
\end{tikzcd}$$

if the pullback of i' and o' is the initial object, then the composite $o' \circ g$ is a mono. Moreover, the pullback of O' and C is the initial object.

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:

$$\begin{tikzcd}
M \arrow[d, "i'_1"'] & \emptyset \arrow[l] \arrow[d] \\
C'_M \arrow[d, hook, "i'_2"'] & X \arrow[l] \arrow[d] \\
C' & O' \arrow[l]
\end{tikzcd}$$

As epimorphisms are preserved under pullbacks in **AsyncGraph**, 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:

$$\begin{tikzcd}
C \arrow[d] & M \arrow[l, "o"'] \arrow[d, "i'_1"] & \\
A \arrow[d] & C'_M \arrow[l, "h"'] \arrow[d, "\iota_1"] & \\
A + O' \arrow[d, "k"'] & C'_M + O' \arrow[l, "h+\mathrm{Id}"'] \arrow[d, hook, "{[i'_2, o']}"] & O' \arrow[l, "\iota_2"'] \\
D & C' \arrow[l, "g"'] &
\end{tikzcd}$$

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

Finally, $C \times_D O' \cong \emptyset$ because $C \times_{A+O'} O' \cong \emptyset$ (coproducts are disjoint, because adhesive categories are extensive if they have a strict initial object as we noted in Lemma 3.9.10) and $k : A + O' \to D$ is a mono. □

Fibrational properties of the disjoint sum Suppose the ambient category $\mathbb{S}$ has disjoint sum, which is the case of **AsyncGraph**. Then

Lemma 3.9.16. *The sum of two fibrations is a fibration.*

The three family of template which we use have a natural structure of internal J-opcategory with sums: for the two template $\pm_S$ and $\pm_L$, this is automatic as they are internal opcategories, and for the separated states, the map $\vee$ simply computes the disjunction of the formulas. We note that in all cases, the map $\nabla : \pm[1] + \pm[1] \to \pm[1]$ is an Environment 1-fibration and a 2-fibration. This ensures in particular that the disjoint sum of cobordisms preserves well-formedness, except for Condition 1, which can be recovered by precomposing with a suitable filling system when adequate.

Putting all of this together, we prove:

Proposition 3.9.17. *All operations which are used to interpret programs preserve well-formedness.*

Proof. For sequential composition, while loops and conditionals, this follows from Lemma 3.9.14 and Lemma 3.9.15. Parallel composition and lock allocation follows from Lemma 3.9.12 about the action of acute spans, and critical sections follow from the aforementioned lemma and sequential composition (this is where the caveats about Condition 1 in the lemmas above are useful). □

3.9.4 Strict maps of cobordisms

The proof of the asynchronous soundness theorem relies on the fact that the maps of cobordisms $\mathcal{S}$ and $\mathcal{L}$ are strict in the following sense:

Definition 3.9.18. A map of cobordisms $(\mathcal{F}, \mathcal{F}_{\pitchfork})$ from $\mathbf{C} \in \mathbf{Cob}(\pitchfork_1)$ to $\mathbf{D} \in \mathbf{Cob}(\pitchfork_2)$ is ***strict*** if the two squares below, with the same notations as Definition 3.8.1, are pullbacks.

$$\begin{tikzcd}
I \arrow[r, "\mathrm{in}_{\mathbf{C}}"] \arrow[d, "\mathcal{F}_I"'] \arrow[dr, phantom, "pb"] & C \arrow[d, "\mathcal{F}_C"] \arrow[dr, phantom, "pb"] & O \arrow[l, "\mathrm{out}_{\mathbf{C}}"'] \arrow[d, "\mathcal{F}_O"] \\
I' \arrow[r, "\mathrm{in}_{\mathbf{C}}"'] & D & O' \arrow[l, "\mathrm{out}_{\mathbf{D}}"]
\end{tikzcd}$$

We prove in this section that the maps of cobordisms $\mathcal{S}$ and $\mathcal{L}$ are strict. We begin with operations based on the pull-and-push construction along acute spans

Lemma 3.9.19. *Let $\mathbf{C} \in \mathbf{Cob}(\pitchfork_1)$ and $\mathbf{C}' \in \mathbf{Cob}(\pitchfork_1')$ be cobordisms related by a strict map of cobordisms $\mathcal{F} : \mathbf{C} \to \mathbf{C}'$, and let*

$$\begin{tikzcd}
\pitchfork_1 & \pitchfork_2 \arrow[l, "F", "="'] \arrow[r, "G"] & \pitchfork_3
\end{tikzcd}
\qquad
\begin{tikzcd}
\pitchfork_1' & \pitchfork_2' \arrow[l, "F'", "="'] \arrow[r, "G'"] & \pitchfork_3'
\end{tikzcd}$$

be two strict acute spans related by three internal functors of J-opcategories u_1, u_2 and u_3 as in the following commutative diagram

$$\begin{tikzcd}
\pitchfork_1 \arrow[d, "u_1"'] & \pitchfork_2 \arrow[l, "F", "="'] \arrow[r, "G"] \arrow[d, "u_2"] & \pitchfork_3 \arrow[d, "u_3"] \\
\pitchfork_1' & \pitchfork_2' \arrow[l, "F'", "="'] \arrow[r, "G'"] & \pitchfork_3'
\end{tikzcd}$$

Then the induced map of cobordisms $\mathcal{F}' : \mathbf{Cob}((F, G))(\mathbf{C}) \to \mathbf{Cob}((F', G'))(\mathbf{C}')$ is a strict map of cobordisms.

Proof. Write the cobordism $\mathbf{C}$ as the diagram

$$\begin{tikzcd}
I \arrow[r, "\mathrm{in}"] \arrow[d, "\lambda_I"'] & S \arrow[d, "\lambda_\sigma"] & O \arrow[l, "\mathrm{out}"'] \arrow[d, "\lambda_O"] \\
\pitchfork_1[0, i] \arrow[r, "\mathrm{in}_{ij}"] & \pitchfork_1[1, ij] & \pitchfork_1[0, i] \arrow[l, "\mathrm{out}_{ij}"']
\end{tikzcd}$$

and write

$$\begin{tikzcd}
J \arrow[r, "\mathrm{in}_{pb}"] \arrow[d, "\lambda_I"'] & T \arrow[d, "\lambda_\tau"] & P \arrow[l, "\mathrm{out}_{pb}"'] \arrow[d, "\lambda_O"] \\
\pitchfork_2[0, i] \arrow[r, "\mathrm{in}_{ij}"] & \pitchfork_2[1, ij] & \pitchfork_2[0, i] \arrow[l, "\mathrm{out}_{ij}"']
\end{tikzcd}$$

the result of pulling back the cobordism **C** along the plain internal functor F, which is part of the definition of the action of the acute span (F, G) on **C**.

First, we claim that the following squares induced by the pullbacks is a pullback square itself.

$$\begin{tikzcd}
I \arrow[r, "\text{in}"] & S & O \arrow[l, "\text{out}"'] \\
J \arrow[u] \arrow[r, "\text{in}_{pb}"'] \arrow[ur, phantom, "pb"] & T \arrow[u] \arrow[ur, phantom, "pb"] & P \arrow[u] \arrow[l, "\text{out}_{pb}"]
\end{tikzcd}$$

Since the two cases are symmetric, we prove that the left square is a pullback. This follows from the pasting lemma of pullbacks in the following commutative cube

$$\begin{tikzcd}
I \arrow[rr] \arrow[dd] & & S \arrow[dd] & \\
& J \arrow[ul] \arrow[rr] \arrow[dd] & & T \arrow[ul] \arrow[dd] \\
\text{よ}_1[0] \arrow[rr] & & \text{よ}_1[1] & \\
& \text{よ}_2[0] \arrow[ul] \arrow[rr] & & \text{よ}_2[1] \arrow[ul]
\end{tikzcd}$$

since the two side faces are pullbacks by definition, and the bottom face is one because (F, G) is a strict acute span by assumption. We have the same result for the pullback of $\mathbf{C}'$ along F'.

Finally, we conclude by using again the pasting lemma for pullback in the cube

$$\begin{tikzcd}
I \arrow[rr] \arrow[dd, "\mathcal{F}_I"'] & & S \arrow[dd, "\mathcal{F}_S"] & \\
& J \arrow[ul] \arrow[rr] \arrow[dd] & & T \arrow[ul] \arrow[dd] \\
I' \arrow[rr] & & S' & \\
& J' \arrow[ul] \arrow[rr] & & T' \arrow[ul]
\end{tikzcd}$$

We have just proved that the top and the bottom faces are pullbacks, and the hypothesis that $\mathcal{F}$ is a strict map of cobordisms means that the back face is a pullback as well. We conclude that the front face is one as well, which is what we needed to prove. □

The second basic operation we need to prove preserves strictness of maps of cobordisms is sequential composition. This fact relies heavily on the fact that the ambient category **AsyncGraph** is adhesive.

Lemma 3.9.20. *Let* $\mathbf{C}, \mathbf{D} \in \mathbf{Cob}(\pm_1)$ *and* $\mathbf{C}', \mathbf{D}' \in \mathbf{Cob}(\pm_1')$ *be cobordisms related by two strict maps of cobordisms* $\mathcal{F} : \mathbf{C} \to \mathbf{C}'$ *and* $\mathcal{F} : \mathbf{D} \to \mathbf{D}'$. *Suppose moreover that* $\mathbf{C}'$ *and* $\mathbf{D}'$ *satisfy Condition 3 of Definition 3.9.1.*

Then the induced map of cobordisms $\mathcal{F}; \mathcal{G} : \mathbf{C}; \mathbf{D} \to \mathbf{C}'; \mathbf{D}'$ *is strict.*

Proof. The situation is described by the following diagram:

$$\begin{tikzcd}
I \arrow[d] \arrow[dr] & & M \arrow[dl] \arrow[d] \arrow[dr] & & O \arrow[dl, hook'] \arrow[d] \\
I' \arrow[dr] & C_1 \arrow[d] \arrow[dr] & M' \arrow[dl, hook'] \arrow[dr] & C_2 \arrow[dl] \arrow[d] & O' \arrow[dl, hook'] \\
& C_1' \arrow[dr] & C_1; C_2 \arrow[d, dashed] & C_2' \arrow[dl] & \\
& & C_1'; C_2' & &
\end{tikzcd}$$

where the non-dashed vertical arrows are given by $\mathcal{F}$ and $\mathcal{G}$, and the map $M' \to C_1'$ is a mono, since the cobordism $\mathbf{C}'$ is satisfies Condition 3.

We focus on the central commutative cube. The bottom face is a pushout along a mono, and therefore it is a van Kampen square in **AsyncGraph**. The two back faces are pullbacks since we assume that $\mathcal{F}$ and $\mathcal{G}$ are strict. Since the top face is a pushout by the definition of sequential composition, adhesivity of **AsyncGraph** implies, by definition, that the two front faces are pullbacks. We conclude by the pasting lemma of pullbacks. □

Strictness of maps of well-formed cobordisms is stable under generalized sequential composition with filling systems follows from the following lemma.

Lemma 3.9.21. *Let* $\lambda : A \to \pm[0, i]$ *and* $\mu : B \to \pm[0, j]$ *be two colored-games over an internal* J*-opcategory which is Environment-fibrational. If* μ *is a mono, then the cobordism* $\mathsf{fill}\,(\lambda, \mu)$ *satisfies Condition 3 of Definition 3.9.1.*

Proof. Follows from the definition of $\mathsf{fill}\,(\lambda, \mu)$, and form the fact that monos are stable under pullback and pushout in **AsyncGraph**. □

Similarly to the previous section, the lemmas above imply the following.

Proposition 3.9.22. *Given a proof π of some CSL Hoare triple $\Gamma \vdash \{P\}\, C\, \{Q\}$, the two maps*

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

are strict maps of cobordisms.

3.9.5 Proof of 2-dimensional correctness

We begin with the proof that the constructions on cobordisms which we use preserve 2-fibrations because it is simpler than proving that they preserve Code 1-fibrations.

We prove by induction on the structure of the derivation trees of the logic that the map $\mathcal{L} \circ \mathcal{S} : [\![\pi]\!]_{\mathrm{Sep}} \to [\![C]\!]_{\mathrm{L}}$ is a 2-fibration. The fact that the rule Par for the parallel product preserves 2-fibration in the sense above uses the following technical lemma.

Lemma 3.9.23. *Consider an asynchronous morphism $s : S_1 \to S_2$ which we see as defining a notion of fibration through a right-lifting property, and another asynchronous morphism $g : G \to H$ which satisfies the following property:*

$$\forall\, l, r : S_2 \to G,\ (rs = ls \wedge gr = gl) \implies l = r. \tag{3.8}$$

Then, for another map $f : K \to G$, if $g \circ f$ is a $S_1 \to S_2$ fibration, then so is f. Moreover, maps which satisfy Condition 3.8) are stable under pullback. Monos are a particular case of such maps.

Proof. The second statement follows from the uniqueness property of the universal property of pullbacks, similarly to preservation of monos under pullbacks. The third is easy.

Let us consider the first statement. This situation is depicted with the following diagram:

$$\begin{tikzcd}
K \arrow[d, "f"'] & S_1 \arrow[l, "w"'] \arrow[d, "s"] \\
G \arrow[d, "g"'] & S_2 \arrow[l, "t"'] \arrow[ul, dashed, "\ell"] \\
H &
\end{tikzcd}$$

where the full arrow commute and where the map ℓ is such that $s\ell = w$ and $gf\ell = gt$. Since moreover $ts = fw$, we can use the hypothesis to conclude that $f\ell = t$, which means that ℓ is indeed a lifting and that f is a fibration. □

Lemma 3.9.24. *Let $(F, G) : \pm_1 \nrightarrow \pm_3$ and $(F, G') : \pm_1' \nrightarrow \pm_3'$ be two acute spans related by a map of acute spans $\alpha : F \to G$. Suppose that $F : \pm_2 \to \pm_1$ is a 2-fibration, and that $F' : \pm_2 \to \pm_3$ satisfies Condition (3.8).*

If $\mathcal{L} \circ \mathcal{S} : [\![\pi]\!]_{Sep} \to [\![C]\!]_L$ is a 2-fibrations, then so is the induced map of cobordisms $\alpha(\mathcal{L} \circ \mathcal{S}) : \mathbf{Cob}(F)([\![\pi]\!]_{Sep}) \to \mathbf{Cob}(G)([\![C]\!]_L)$.

Proof. Consider the following diagram, where f is the component of $\mathcal{L} \circ \mathcal{S}$ between the supports:

$$\begin{tikzcd}
T \arrow[rrr, dashed, "\alpha f"] \arrow[dr] \arrow[ddd] & & & T' \arrow[dl] \arrow[ddd] \\
& S \arrow[r, "f"] \arrow[d] & S' \arrow[d] & \\
& \pm_1 \arrow[r, "\alpha_1"] & \pm_1' & \\
\pm_2 \arrow[ur, "pick"] \arrow[rrr, "\alpha_2"] \arrow[d, "pince"'] & & & \pm_2' \arrow[ul, "pick'"'] \arrow[d, "pince'"] \\
\pm_3 \arrow[rrr, "\alpha_3"] & & & \pm_3'
\end{tikzcd}$$

and where the two squares on the sides are pullbacks.

First, since the map $F : \pm_2 \to \pm_1$ is a 2-fibration, and because the fibrations are preserved under pullbacks, the map $T \to S$ is a 2-fibration as well. Similarly, the map $T' \to S'$ satisfies Condition (3.8). since it is the pullback of F'. We conclude using Lemma 3.9.23 above. □

Since 2-fibrations are closed under product, and $\mathrm{pick}[1] : \pm_{\mathrm{L}}^{\parallel}[1] \to \pm_{\mathrm{L}}[1] \times \pm_{\mathrm{L}}[1]$ is a mono, the lemma above can be applied to the parallel product.

Corollary 3.9.25. *If $\mathcal{L}_1 \circ \mathcal{S}_1 : [\![\pi_1]\!]_{Sep} \to [\![C_1]\!]_L$ and $\mathcal{L}_2 \circ \mathcal{S}_2 : [\![\pi_2]\!]_{Sep} \to [\![C_2]\!]_L$ are 2-fibrations, then so is the induced map of cobordisms $(\mathcal{L}_1 \circ \mathcal{S}_1) \parallel (\mathcal{L}_2 \circ \mathcal{S}_2) : [\![\pi_1]\!]_{Sep} \parallel [\![\pi_2]\!]_{Sep} \to [\![C_1]\!]_L \parallel [\![C_2]\!]_L$.*

Similarly, this lemma can be applied for the two change of locks operations as well.

The proof that sequential composition preserves 2-fibrations follows from the fact that van Kampen cubes preserve 2-fibrations in the following sense:

Lemma 3.9.26. *Consider the following diagram in* **AsyncGraph***:*

$$\begin{tikzcd}
 & Y \arrow[rrrr, "\lambda_2"] \arrow[dr, "f'"] & & & & C \arrow[dr, "i'"] & \\
 & po & Z \arrow[rrrr, dashed, "\mu"] & & & po & D \\
W \arrow[uur, hook, "f"] \arrow[rrrr] \arrow[dr, hook, "g"'] & & & & A \arrow[uur, "i"] \arrow[dr, "j"] & & \\
 & X \arrow[uur, "g'"] \arrow[rrrr, "\lambda_1"] & & & & B \arrow[uur, "j'"'] &
\end{tikzcd}$$

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. □

3.9.6 Proof of 1-dimensional correctness

Preservation of Code 1-fibrations by sequential composition follows directly from a variant of Lemma 3.9.13 where all occurrences of Environment 1-fibrations are replaced with Code 1-fibrations. The proof for change of lock operations follows from Lemma 3.9.24.

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, which means we cannot use Lemma 3.9.24. Instead, we need to rely on the fact that the map $S \to \pm_S[1]$ 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 3.9.27. *Given two maps of cobordisms*

$$F_i : \mathbf{C}_i \to \mathbf{C}'_i$$

for $i = 1, 2$, and if F_1 and F_2 are 1-fibrations on Code transitions, then the induced morphism

$$F_1 \parallel F_2 : \mathbf{C}_1 \parallel \mathbf{C}_2 \to \mathbf{C}'_1 \parallel \mathbf{C}'_2$$

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 F_1 and F_2:

$$f_1 : S_1 \longrightarrow S'_1 \qquad\qquad f_2 : S_2 \longrightarrow S'_2$$

Finally, we write pick 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:

$$\begin{tikzcd}
\cdot \arrow[rrr] \arrow[d] & & & \to_C \arrow[d] \\
S_1 \parallel S_2 \arrow[rrr, dashed, "f_1 \parallel f_2"] \arrow[dr, "pick"] \arrow[ddd] & & & S'_1 \parallel S'_2 \arrow[dl, "pick'"'] \arrow[ddd] \\
& S_1 \times S_2 \arrow[r, "f_1 \times f_2"] \arrow[d] & S'_1 \times S'_2 \arrow[d] & \\
& \pm \times \pm \arrow[r, "u \times u"] & \pm' \times \pm' & \\
\pm^{\parallel} \arrow[ur, "pick"] \arrow[rrr, "u^{\parallel}"] \arrow[d, "pince"] & & & \pm'^{\parallel} \arrow[ul, "pick'"'] \arrow[d, "pince'"] \\
\pm \arrow[rrr, "u"] & & & \pm'
\end{tikzcd}$$

The polarity of the image of the unique edge of $\to_C$ in $\pm'^{\parallel}$ is either C_1 or C_2. Because the situation is symmetric, we can suppose without loss of generality that this polarity is C_1. In that case, the edge $\rightarrow_C$ 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_1. Moreover, because $\pi \circ \mathsf{pick} : ⫱^{\parallel} \to ⫱$ is a 1-fibration on Code transitions as well, we can also lift $\rightarrow_C$ to $⫱^{\parallel}$. Therefore, we have the following situation:

$$\begin{tikzcd}
\cdot \arrow[rrr] \arrow[dddd] & & & S_1 \times S_2 \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] \arrow[dd, "\langle f_1, f_2 \rangle"] & \\
 & & S_1 \arrow[dd, "\lambda_1"'] & & S_2 \arrow[dd, "\lambda_2"] \\
 & \rightarrow_C \arrow[ur, dashed, "(1)"] & & ⫱[1] \times ⫱[1] \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\
 & & ⫱[1] & & ⫱[1] \\
\rightarrow_{C_1} \arrow[uur, bend left] \arrow[rrr] \arrow[drrr, bend right] & & & ⫱^{\parallel}[1] \arrow[ul, "\pi_1 \circ \mathsf{pick}"'] \arrow[ur, "\pi_2 \circ \mathsf{pick}"] & \\
 & & & \rightarrow_F \arrow[uur, bend right] \arrow[uuuur, dashed, bend right=60, "(2)"'] &
\end{tikzcd}$$

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 $\rightarrow_{C_1}$ to $S_1 \times S_2$. We conclude by using the fact that 1-fibrations on Code transitions are stable by pushout, and that $\mathsf{pick} : ⫱^{\parallel} \to ⫱ \times ⫱$ is a 1-fibration. □

Part II

Relational soundness in Iris

4 Background on Iris

Iris is a state of the art concurrent separation logic. Its goal was to unify the many variants of concurrent separation logic which were being developed to handle increasingly complicated concurrent algorithms or language features. The first step (Jung, Swasey, et al., 2015) was to axiomatize the notion of logical resource: all the resources the predicates are describing, such as the heap, or the heap with permission are cameras, a structure similar to a partial commutative monoid. Cameras can be combined to enable reasoning about complex concurrent data structures and algorithms with subtle synchronization patterns.

The Iris program logic is also parameterized by a programming language, which is expressed as a small step operational semantics. This allows users to prove properties of programs written in their language of choice without having to redefine and reprove the soundness of the logic from scratch.

Early versions of Iris were program logics, in that concepts such as Hoare triples or invariants were primitive notions. In later versions (Robbert Krebbers, Jung, et al., 2017), such notions became defined in terms of a more basic modal logic, which is called the *base logic*.

Importantly, the soundness of the program logic follows from an adequacy theorem of the base logic. This allows one to develop new program logics which provide other guarantees than the usual Hoare triples of Iris in a much simpler way than if one had to define it and prove its soundness from scratch. In this part of the thesis, we define a new notion of Hoare triple, which allows its users to construct simulations between a program and a state transition system which we see as an abstract specification for the program. First, we present in this Chapter the basics of the Iris logic.

All the result and the background in this part are formalized in the Coq proof assistant.

4.1 Overview of the approach

One way to explain Iris is to start from the semantic definition of safety which can be used to define the meaning of a Hoare triple $\{P\}\ C\ \{Q\}$ from Chapter 1 and which

$$
\begin{array}{c}
\mathfrak{s} \vDash Q \\
\hline\hline
\mathsf{safe}(\mathtt{skip}, \mathfrak{s}, Q)
\end{array}
\qquad\qquad
\begin{array}{c}
(C, \mathfrak{s}) \not\rightarrow \lightning \quad \forall C', \mathfrak{s}', (C, \mathfrak{s}) \rightarrow (C', \mathfrak{s}') \Rightarrow \mathsf{safe}(C', \mathfrak{s}', Q) \\
\hline\hline
\mathsf{safe}(C, \mathfrak{s}, Q)
\end{array}
$$

Figure 4.1: Semantic validity for Hoare logic

we reproduce here in Figure 4.1 (recall the double bar denotes a coinductive predicate) Recall that validity of a Hoare triple was defined as:

$$\vDash \{P\}\, C\, \{Q\} \quad\iff\quad \forall \mathfrak{s},\ \ \mathfrak{s} \vdash P \Rightarrow \mathsf{safe}(C, \mathfrak{s}, Q)$$

It is easy to check that defining a new predicate wp $C\,\{Q\}$ as follows:

$$\mathfrak{s} \vDash \mathsf{wp}\ C\,\{Q\} \quad\iff\quad \mathsf{safe}(C, \mathfrak{s}, Q)$$

defines a semantic weakest precondition for the Hoare triple. Iris proceeds by defining a semantic weakest precondition which follows the same intuition as the one in Figure 4.1. One difference is that, in Iris, the machine state $\mathfrak{s}$ is not a parameter of the predicate, instead it is part of the logical state, or world.

4.1.1 Iris predicates

The Iris base logic propositions are predicates over *worlds*, and the Iris logic is parameterized over the set $\mathcal{W}$ of words. Iris propositions are predicates over worlds, that is, functions of type *iProp* $:= \mathcal{W} \rightarrow \mathbb{P}$, where $\mathbb{P}$ is the two-element set $\mathbb{2}$ or the type `Prop` in Coq. It is conventional to call the world $w \in \mathcal{W}$ which is passed to a predicate the *current world*.

Predicates can access the current state (that is, the state contained in the current world) using the predicate `state_interp` $\mathfrak{s}$ which states that the current memory state is equal to $\mathfrak{s}$. In other words, it is the predicate:

$$\mathtt{state\_interp}\ \mathfrak{s} \quad := \quad \lambda w.\ \text{“the memory state in } w \text{ is } \mathfrak{s}\text{”}$$

Using this predicate, we can already express that an expression must not be stuck:

$$\mathtt{not\_stuck}\ e \ := \ \exists \mathfrak{s},\ \mathtt{state\_interp}\ \mathfrak{s} \land \ulcorner\, \exists e', \mathfrak{s}',\ (e, \mathfrak{s}) \rightarrow (e', \mathfrak{s}')\, \urcorner$$

where $\ulcorner \cdot \urcorner : \mathbb{P} \rightarrow$ *iProp* injects propositions from the ambient logic into Iris propositions. This proposition expresses that the term e is reducible in the current memory state.

Iris is a separation logic, which means that there is a notion of the “part of the world” which verifies some predicate P. This is achieved by requiring that the set $\mathcal{W}$ of worlds have a partial commutative monoid structure (actually, as we will see in the sequel, a camera structure.) This partial commutative monoid structure induces an order relation on elements of $\mathcal{W}$:

$$w \preccurlyeq w' \quad \Longleftrightarrow \quad \exists w_f,\ w' = w \cdot w_f$$

We model the partiality of $\mathcal{W}$ by choosing a subset $\mathcal{V} \subseteq \mathcal{W}$ of valid elements. Intuitively, two subworlds w_1 and w_2 are compatible if $(w_1 \cdot w_2) \in \mathcal{V}$. This defines a separating conjunction of predicates: $w \vDash P * Q$ if w is the product of two worlds which satisfy P and Q respectively.

This generalizes our notion of logical state in Part I: The $*$ product which we defined on **LogState** gave a partial commutative monoid structure to logical states.

4.1.2 The Iris standard weakest precondition

Until the end of this thesis, we consider expression-based languages: there is no distinction between commands and expressions, which can also have side effects.

Definition 4.1.1 (Language)**.** We consider languages with expressions denoted by $e \in$ **Expr**, values $v \in$ **Val** with **Val** $\subseteq$ **Expr**. A language has a notion of state $\mathfrak{s} \in$ **State**, and a (thread-local) small-step operational semantics

$$\rightarrow \quad \subseteq \quad (\mathbf{Expr} \times \mathbf{State}) \times (\mathbf{Expr} \times \mathbf{State} \times \mathbf{List}(\mathbf{Expr}))$$

A step $(e, \mathfrak{s}) \rightarrow (e', \mathfrak{s}', \vec{e}_f)$ means that the expression e reduces to e' while changing the state from $\mathfrak{s}$ to $\mathfrak{s}'$, and creating the threads in the list $\vec{e}_f$. A value must not be reducible, and an non-value expression which cannot reduce is how Iris characterizes runtime errors.

This thread-local reduction relation is lifted to a threadpool reduction relation

$$\rightarrow_{\mathrm{tp}} \quad \subseteq \quad (\mathbf{List}(\mathbf{Expr}) \times \mathbf{State}) \times (\mathbf{List}(\mathbf{Expr}) \times \mathbf{State})$$

defined using the following rule:

$$\frac{(\mathrm{tp}[k], \mathfrak{s}) \rightarrow (e'_k, \mathfrak{s}', e_f)}{(\mathrm{tp}, \mathfrak{s}) \xrightarrow{k}_{\mathrm{tp}} (\mathrm{tp}[k \coloneqq e'_k] +\!\!+ e_f, \mathfrak{s}')}$$

where $\mathrm{tp}[k]$ denotes the kth element of the list tp, and $\mathrm{tp}[k \coloneqq e'_k]$ denotes tp where the kth element has been replaced by e'_k. The threadpool is the list of all the threads of the system which have been spawned. The thread ID of a thread, denoted in general with the variable *tid* is the index of the thread in the threadpool. This index is stable because irreducible programs are not removed from the threadpool. One reducible thread is chosen non-deterministically, and is reduced for one step. The threads it has spawned are added to the threadpool.

The next step to restate the definition of weakest precondition given by Figure 4.1 is to be able to *update* the memory state so that we can state that the weakest precondition holds in the new state $\mathfrak{s}'$ after the program has reduced. To do so, we use the viewshift $\Rrightarrow$ connector of the logic. The proposition $P \Rrightarrow Q$ asserts that we can replace any part of the world which satisfies P by a new subworld which satisfies Q in a way which "does not disturb" the rest of the world.

Remark 4.1.2. Readers familiar with Kripke semantics should keep in mind that the $\Rrightarrow$ connector is not related to the notion of "future world". As we will see, however, the Iris model has a notion of future world for the step-indexing, and propositions are closed under that relation.

Given an Iris predicate $\Phi : \mathbf{Val} \to \mathit{iProp}$, we can define a weakest precondition as the following recursive predicate:

$$\begin{aligned}
\mathsf{wp}\; e\, \{Q\} \;\coloneqq\;& (\ulcorner\, e \in \mathit{Val}\, \urcorner \wedge Q\; e) \;\vee \\
& (\exists \mathfrak{s}, \mathtt{state\_interp}\; \mathfrak{s} \;\mathrel{-\!\!*}\; \ulcorner\, \exists e', \mathfrak{s}', (e, \mathfrak{s}) \to (e', \mathfrak{s}', \vec{e}_f)\, \urcorner \wedge \\
& \;\forall e', \mathfrak{s}', \vec{e}_f, \ulcorner\, (e, \mathfrak{s}) \to (e', \mathfrak{s}', \vec{e}_f)\, \urcorner \Rrightarrow\; \mathtt{state\_interp}\; \mathfrak{s}' * \\
& \quad \triangleright \mathsf{wp}\; e'\, \{Q\} * \mathop{\circledast}\limits_{e_f \in \vec{e}_f} \mathsf{wp}\; e_f\, \{\top\})
\end{aligned}$$

Note that, because e is an expression, which evaluates to a value, the postcondition Q takes a program value as an argument: it is of type $Q : \mathbf{Val} \to \mathit{iProp}$.

If e is already a value, the predicate is true if e satisfies the postcondition Q. Otherwise, if the current memory state is $\mathfrak{s}$, the program e must be reducible at that state, and, for all possible reductions $(e, \mathfrak{s}) \to (e', \mathfrak{s}', \vec{e}_f)$, where $\vec{e}_f$ is the list of threads which are spawned during the reduction steps, we can update the world in such a way that the new memory state is $\mathfrak{s}'$, the weakest precondition holds for the reduct e' and all the spawned threads are safe.

The $\triangleright$ modality in front of the recursive occurrence of wp is needed because, the Iris weakest precondition is defined as a guarded fixpoint instead of coninductively. A recursive definition is guarded if each recursive occurrence is under a $\triangleright$ modality. This modality was introduced by Nakano (2000), and its relation with step-indexing was discovered by Appel et al. (2007).

Because Iris is a separation logic, the proposition

$$\mathrm{wp}\ e_1\ \{P_1\} * \mathrm{wp}\ e_1\ \{P_2\}$$

means that that the resources needed to guarantee the safety of e_1 and those for e_2 must be "disjoint", and e_1 and e_2 are safe to execute in parallel. This is why this is enough to requires the separated product of the weakest preconditions of the reduct of e and of every forked thread in the definition of the weakest precondition.

4.2 The Iris model

One requirement of the Iris logic is that the world be able to contain Iris predicates. This is particularly useful to implement invariants, which are a generalization of lock invariants in the version of CSL we have studied in Part I. Let us make the simplifying assumption that we want our world to only contain one Iris predicate. Then, we have the following isomorphisms:

$$\begin{array}{rcl} \mathit{iProp} & \cong & \mathcal{W} \to \mathbb{P} \\ \mathcal{W} & \cong & iProp \end{array}$$

The first isomorphism comes from the definition of Iris proposition as predicate over worlds. If we rewrite the second isomorphism into the first, we get:

$$\mathit{iProp} \quad \cong \quad \mathit{iProp} \to \mathbb{P}$$

This kind of recursive equation with a negative occurrence (on the left of an arrow) does not have a solution in most domains.

The solution Iris uses to solve this problem is to use step-indexing: intuitively, every set X is replaced by a sequence $(X_n)_{n\in\mathbb{N}}$ of sets. The unsolvable equation above is replaced with a system of equations which uses these indices break the cyclic nature of the previous equation:

$$\begin{array}{rcl} \mathit{iProp}_0 & \cong & \mathbb{P} \\ \mathit{iProp}_{n+1} & \cong & \mathit{iProp}_n \to \mathbb{P} \end{array}$$

We can use operators on such indexed sets to simplify its presentation: Given a set X, we write $\mathbf{G}X$ the constant sequence $(X)_{n\in\mathbb{N}}$. We can also define a "next" operator ▶ which takes care of decreasing the step-indices:

$$\begin{array}{rcl} (\blacktriangleright X)_0 & := & \{\star\} \\ (\blacktriangleright X)_{n+1} & := & X_n \end{array}$$

Using these two operators, we can rewrite the system of equations above as:

$$iProp \quad \cong \quad \blacktriangleright iProp \to \mathbf{G}\mathbb{P}$$

This presentation of the model of Iris has so far been informal, as we have not defined the notion of maps, etc. One way to formalize this is to work in the category of presheaves over the ordinal ω, seen as a category; this category is also known as the topos of trees (L. Birkedal et al., 2012).

4.2.1 Ordered families of equivalences

In the actual model of Iris, sequences of sets are replaced by sets equipped with a sequence of equivalence relations:

Definition 4.2.1. An ***ordered family of equivalences (OFE)*** is the data $(X, (\overset{n}{=})_n)$ of a set X and a sequence of equivalence relations over X, such that:

1. $\overset{n}{=} \subseteq \overset{m}{=}$ for $n \geq m$;
2. $x = y$ iff $\forall n,\ x \overset{n}{=} y$.

The first condition states that the equivalence relation becomes more discriminating as n grows, and the second states that the limit of $(\overset{n}{=})$ is the equality relation.

An OFE X induces a sequence of sets as in the informal overview above by quotienting by the equivalence relation at each step-index:

$$X \quad \longmapsto \quad \left(X / \overset{n}{=}\right)_n$$

There are two useful notions of maps of OFEs:

Definition 4.2.2. Given two OFEs X and Y, a map $f : X \to Y$ between the underlying sets is said to be:

- ***non-expansive*** if, for all $x, y \in X$, and step-index n, $x \overset{n}{=} y \Rightarrow f(x) \overset{n}{=} f(y)$;
- and it said to be ***contractive*** if, for all $x, y \in X$, $x \overset{n}{=} y \Rightarrow f(x) \overset{n+1}{=} f(y)$.

OFEs and non-expansive maps form a Cartesian-closed category **OFE**. This vocabulary comes from the intuition that X is an ultrametric space, and that $x \overset{n}{=} y$ means that the points x and y are at a distance $\leq 2^{-n}$. An OFE which will be useful to define the Iris model is the the OFE *SProp* of ***step-indexed propositions***: its underlying set *SProp* is the set

$$\{A \subseteq \mathbb{N} \mid \forall n \leq m,\ m \in A \Rightarrow n \in A\}$$

of downward closed subsets of ℕ, and two step-indexed propositions are related by $\neq n$ if

$$\forall m \leq n, \quad x \in A \quad \Longleftrightarrow \quad x \in B$$

An intuitive reading of $n \in A$ is that it is not possible to disprove the proposition A in the first n steps of "execution". Now that we have a step-indexed notion of proposition, it remains to define a step-indexed notion of partial commutative monoid.

4.2.2 RAs and Cameras

Iris is defined in terms of an extension of the notion of partial commutative monoid known as *resource algebras* (RA). Instead of having a neutral element, an RA M contains a partial map $| \quad | : M \rightharpoonup M$ called the *core*, which associates to some element a its duplicable part, if it exists; in which case we have:

$$|a| \cdot a = a \qquad \text{and} \qquad ||a|| = |a|$$

which does imply that the core is duplicable:

$$|a| \cdot |a| = ||a|| \cdot |a| = |a|$$

The step-indexed version of RAs are Cameras:

Definition 4.2.3. A ***camera*** M is the data $(M, \mathcal{V}, |\cdot|, \cdot)$ of:

- an OFE M of elements;
- a "step-indexed subset" of valid elements $\mathcal{V} : M \to SProp$ expressed as a non-expansive map of OFEs;
- a partial map $|\cdot| : M \to M^?$, where $M^? \coloneqq M \uplus \bot$ adjoins a new neutral element;
- and a commutative and associative non-expansive operation $\cdot : M \times M \to M$

which satisfy a number of axioms given in (Iris Team, 2021). This definition induces a step-indexed order on elements of M with $a \overset{n}{\preccurlyeq} b \Leftrightarrow \exists a',\ b \overset{n}{=} a \cdot a'$.

4.2.3 The Iris base logic

The Iris logic is parameterized by a camera M; we will explain later how the choice of M is done in a way that is modular.

The formulas of Iris base logic comprise higher-order BI

$$P ::= P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \exists x : \tau .\, P \mid \forall x : \tau .\, P \mid x$$

where the type τ of the quantified variable x includes Iris propositions *iProp* and the camera parameter M; as well as the later modality and a guarded fixpoint operator:

$$P ::= \cdots \mid \triangleright P \mid \mu x : \tau .\, P$$

where every occurrence of x in P must appear under a $\triangleright$ modality; and a number of modalities and atomic propositions to reason about resources:

$$P ::= \cdots \mid \mathsf{Own}(a) \mid \mathcal{V}(a) \mid \square\, P \mid \Rrightarrow P \mid \ulcorner\, \Phi\, \urcorner$$

where a in an element of M, and $\Phi \in \mathbb{P}$ is a proposition in the ambient logic. The formula $\mathsf{Own}(a)$ expresses the ownership of the resource a, and $\mathcal{V}(a)$ expresses that it is valid. The formula $\square\, P$ (read *persistently* P) asserts that the persistent part of P holds, and $\ulcorner\, \Phi\, \urcorner$ states that the proposition Φ is true.

Finally, the update modality $\Rrightarrow P$ is related to the viewshift connector $\Rrightarrow$ which we have seen before:

$$P \Rrightarrow Q \quad := \quad P \mathrel{-\!\!*} \Rrightarrow Q$$

The formula $\Rrightarrow P$ means intuitively that the current world (an element of M) can be updated so that the new world satisfies P The condition that this update must not disturb the rest of the world is formalized using the notion of ***frame-preserving update***: Given $a \in M$ and $B \subseteq M$, there is a frame-preserving update from a to B, written $a \rightsquigarrow B$ if:

$$\forall n, a_f^?,\ n \in \mathcal{V}(a \cdot a_f^?) \quad \Rightarrow \quad \exists b \in B,\ n \in \mathcal{V}(b \cdot a_f^?)$$

where $a_f^?$, which represents the rest of the world, or, in other words, what is owned by the environment, is an element of $M^? := M \uplus \bot$. One way to interpret it is that, whatever the frame $a_f^?$, which is assumed to be compatible with a, we can find an element $b \in B$ which can replace a without conflicting with the environment.

One example which we have already encountered is the allocation of a new location in the partial monoid **LogState** of logical states in Part II. The fact that there is always a free location in a logical state can be expressed as the following frame preserving update:

$$\emptyset \quad \rightsquigarrow \quad \{[\ell := (v, 1)] \mid \ell \in \mathbf{Loc}\}$$

The fact that full ownership of a memory location allows one to update its contents corresponds to the followoing frame-preserving update:

$$[\ell \coloneqq (v_1, 1)] \rightsquigarrow [\ell \coloneqq (v_2, 1)]$$

where we write $a \rightsquigarrow b$ for $a \rightsquigarrow \{b\}$.

As we will see, these two frame-preserving updates corresponds, respectively, to the two laws in the logic:

$$\top \mathrel{-\!\!*} \mathrel{|\!\!\!\Rrightarrow} \exists \ell.\, \ell \overset{1}{\mapsto} v \qquad\qquad \ell \overset{1}{\mapsto} v_1 \mathrel{-\!\!*} \mathrel{|\!\!\!\Rrightarrow} \ell \overset{1}{\mapsto} v_2$$

where we define $\ell \overset{p}{\mapsto} v$ as $\mathsf{Own}([\ell \coloneqq (v, p)])$.

4.2.4 Interpretation of Iris predicates

Iris predicates P are interpreted as monotone maps $P : M \to \mathit{SProp}$, where the order on M is given by divisibility, and the order on SProp is given by containment. Hence, a predicate P is upward-closed with respect to the order on M (which corresponds to the fact that Iris is an affine logic) and downward-closed with respect to step-indices (from the definition of SProp).

Writing $a, n \vDash P$ for $n \in P(a)$, we define the interpretations of the main constructions of the base logic:

$$\begin{aligned}
a, n \vDash P * Q &\iff \exists b_1, b_2,\ a \overset{n}{=} b_1 \cdot b_2 \ \wedge\ b_1, n \vDash P \ \wedge\ b_2, n \vDash Q \\
a, n \vDash P \mathrel{-\!\!*} Q &\iff \forall m, b,\ m \leq n \ \wedge\ m \in \mathcal{V}(a \cdot b) \ \wedge\ b, m \vDash P \ \Rightarrow\ a \cdot b, m \vDash Q \\
a, n \vDash \square P &\iff |a|, n \vDash P \\
a, n \vDash \triangleright P &\iff n = 0 \ \vee\ a, n - 1 \vDash P \\
a, n \vDash \mathrel{|\!\!\!\Rrightarrow} P &\iff \forall m, a',\ m \leq n \ \wedge\ m \in \mathcal{V}(a \cdot a') \ \Rightarrow\ \exists b,\ m \in \mathcal{V}(b \cdot a') \ \wedge\ b, m \vDash P \\
a, n \vDash \ulcorner\, \Phi\, \urcorner &\iff \Phi \\
a, n \vDash \mathsf{Own}(b) &\iff b \overset{n}{\preccurlyeq} a \\
a, n \vDash \mathcal{V}(b) &\iff n \in \mathcal{V}(b)
\end{aligned}$$

The guarded fixed-points are interpreted using a fixed point operator defined on contractive maps in the category **OFE**, as introduced by America and Rutten (1989) and generalized by Lars Birkedal, Støvring, and Thamsborg (2010).

Definition 4.2.4. A predicate P is valid, which we write $\vDash P$ if it is valid for all element of the camera and all the step-indices:

$$\vDash P \qquad \iff \qquad \forall a \in M, n \in \mathbb{N},\ \ a, n \vDash P.$$

4.2.5 Logical rules

This model satisfies a number of logical properties. We write here a few of them, and refer the reader to (Iris Team, 2021) for an exhaustive list.

The separating conjunction reflects the operation of the camera M, and an owned element is always valid:

$$\mathsf{Own}(a) * \mathsf{Own}(b) \Leftrightarrow \mathsf{Own}(a \cdot b) \qquad\qquad \mathsf{Own}(a) \Rightarrow \mathcal{V}(a)$$

The update modality behaves like a strong monad

$$P \Rightarrow \mathrel{\models\!\!\!\Rrightarrow} P \qquad\qquad \mathrel{\models\!\!\!\Rrightarrow}\mathrel{\models\!\!\!\Rrightarrow} P \Rightarrow \mathrel{\models\!\!\!\Rrightarrow} P \qquad\qquad Q * \mathrel{\models\!\!\!\Rrightarrow} P \Rightarrow \mathrel{\models\!\!\!\Rrightarrow} Q * P$$

and it reflects frame-preserving updates:

$$\frac{a \rightsquigarrow B}{\mathsf{Own}(a) \;\Rightarrow\; \mathrel{\models\!\!\!\Rrightarrow} \exists b,\ \mathsf{Own}(b)}$$

The persistence modality behaves like a comonad and corresponds to the core operation

$$\square\, P \Rightarrow P \qquad\qquad \square\, P \Rightarrow \square\,\square\, P \qquad\qquad \mathsf{Own}(a) \Rightarrow \mathsf{Own}(|a|)$$

and it makes predicate duplicable:

$$\square\, P \Leftrightarrow \square\, P * \square\, P$$

A predicate P is called *persistent* if it is equivalent to $\square\, P$. Note that this is stronger than being duplicable: the predicate $\exists p,\ \ell \overset{p}{\mapsto} 1$ is duplicable but not persistent. Of course, the laws of higher-order bunched implications also hold, such as $-\!\!*$ being a right adjoint to $*$, etc.

The later modality allows to define recursive predicates using guarded recursion. Such predicates come with a powerful induction principle which allows resonning about them, called *Löb induction*:

$$\frac{\triangleright P \;\Rightarrow P}{P}$$

This corresponds to an induction on the step-index in the semantics of the predicates. In the setting of a program logic, this means that the induction hypothesis can be used as soon as the program reduces by on step, because the definition of wp contains a later modality, tying logical steps with program steps. This allows for eliminating a later modality guarding an hypothesis using the rule:

$$\frac{P \Rightarrow Q}{\triangleright P \Rightarrow \triangleright Q}$$

In particular, the proof of a recursive function is able to use its induction hypothesis to reason about recursive calls because the execution of a recursive function always starts with the beta reduction step which substitute its argument.

Adequacy of the base logic

The adequacy of the Iris base logic states that the modalities of the Iris base logic do not allow to prove false statements in the meta-logic.

Theorem 4.2.5 (Adequacy)**.** *Suppose* $\vDash m_1 \cdots m_k \ulcorner \Phi \urcorner$*, where each* m_i *is an Iris modality such as* $\triangleright, \mathrel{\vDash\!\!\!\Rrightarrow}, \Box, \ldots$*, and* $\Phi \in \mathbb{P}$ *is a proposition, then* Φ *holds.*

4.3 High level logic

The logic we have described so far is not usable to prove programs. For instance, it is parameterized by a single camera, whereas programs most often need several kinds of resources to be specified and proved. Another missing feature are invariants, which are the way concurrent separation logics allow reasoning about how threads communicate and synchronize with each others. These features can be defined using the base logic above. We sketch in this section how they are defined and how they can be used.

4.3.1 Combining cameras

The mechanism with which several cameras can be used is simple: Given a list $M_1, \ldots, M_n$ of cameras, we define a new camera

$$\mathit{Res} \quad := \quad \prod_{i \in \{1,\ldots,n\}} \mathit{GName} \rightharpoonup_{\mathit{fin}} M_i$$

where $\mathit{GName} := \mathbb{N}$ are the type of ghost (or logical) variables. The camera structure induced by the product is straightforward, and the product of two partial finite maps $f_1, f_2 : X \rightharpoonup_{\mathit{fin}} M$ is defined as:

$$(f_1 \cdot f_2)(x) \quad := \quad \begin{cases} f_1(x) & \text{if } x \in \mathsf{dom}(f_1) \setminus \mathsf{dom}(f_2) \\ f_2(x) & \text{if } x \in \mathsf{dom}(f_2) \setminus \mathsf{dom}(f_1) \\ f_1(x) \cdot f_2(x) & \text{if } x \in \mathsf{dom}(f_1) \cap \mathsf{dom}(f_2) \end{cases}$$

and a finite partial map is valid iff all its component are valid. This definition is the same as the product of logical states from Definition 2.5.3 in Part I for $X := \mathbf{Loc}$ and $M := \mathbf{Val} \times \mathbf{Perm}$.

We can define the predicate

$$\boxed{m : M_i}^{\gamma} \quad := \quad \mathsf{Own}((\emptyset, \ldots, \emptyset, [\gamma := m], \emptyset, \ldots, \emptyset))$$

which expresses the ownership of an n-tuple of empty partial finite maps, except for the kth component which contains the singleton finite map associating the element m of M_i to the ghost variable γ.

This new predicate admits a rule for allocation, and the same rules as the primitive Own predicate for updating and combining:

$$\frac{a \in \mathcal{V}_{M_i}}{\Rrightarrow \exists \gamma.\ \boxed{a : M_i}^{\gamma}} \qquad \frac{a \rightsquigarrow B}{\boxed{a : M_i}^{\gamma} \Rrightarrow \exists b \in B.\ \boxed{b : M_i}^{\gamma}} \qquad \boxed{a}^{\gamma} * \boxed{b}^{\gamma} \Leftrightarrow \boxed{a \cdot b}^{\gamma}$$

4.3.2 Invariants

We will not explain how invariants are constructed, but the gist of it is that, as we sketched at the beginning of this chapter, Iris predicates can be endowed with a camera structure. To be useful, invariants cannot always hold, there needs to be a mechanism for opening invariants and then requiring they be reestablished.

This control is achieved in Iris using the so-called ***fancy update*** modalities ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$, which are defined using the update modality $\Rrightarrow$. This modality is parameterized by a masks $\mathcal{E}_1$ and $\mathcal{E}_2$ which denote sets of invariant names. The predicate ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ means that we can update from a world where every invariant in $\mathcal{E}_1$ holds to a wolds where every invariant in $\mathcal{E}_2$ hold and where P is verified. For instance ${}^{\top}{\Rrightarrow}^{\emptyset} P$ means that one is able to prove P while opening all the invariants (the mask $\top$ denotes the set of all invariant names).

The existence of an invariant is claimed using the predicate $\mathsf{inv}^{\mathcal{N}}(P)$, where $\mathcal{N}$ is the name of the invariant. It is a persistent predicate, which means in particular that it is duplicable, and that it can be shared between threads. The rules for creating and for opening invariants are the following:

$$\triangleright P \Rightarrow {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\cup\mathcal{N}} \mathsf{inv}^{\mathcal{N}}(P) \qquad \frac{\mathcal{N} \subseteq \mathcal{E}}{\mathsf{inv}^{\mathcal{N}}(P) \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * \square(\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \top)}$$

The second rule deserves an explanation: If the invariant $\mathcal{N}$ contains predicate P, and if it currently holds ($\mathcal{N} \subseteq \mathcal{E}$), then is it possible to update to a world where the invariant $\mathcal{N}$ no longer hold but where $\triangleright P$ holds. To close the invariant and go back to a situation where $\mathcal{N}$ holds, one can use the implication which consumes the resource $\triangleright P$. The later modalities $\triangleright$ in these rules come from the later operator in the domain equation at the beginning of Section (4.2).

4.3.3 The standard Iris weakest precondition

We now have the tools to define the standard Iris weakest precondition, which guarantees the safety of programs. It differs from the one sketched at the beginning of the section by the masks and the fancy updates which need to be added to be able to use invariants when proving programs.

$$\begin{aligned}
\mathsf{wp}_{\mathcal{E}}\, e\, \{Q\} \;\coloneqq\; & \mu\, \mathsf{wp}_0. \\
& (\ulcorner\, e \in \mathit{Val}\,\urcorner \wedge {}^{\mathcal{E}}{|\!\!\Rrightarrow}^{\mathcal{E}} Q\; e) \;\vee \\
& (\exists \mathfrak{s}, \mathtt{state\_interp}\; \mathfrak{s} \;-\!\!*\; {}^{\mathcal{E}}{|\!\!\Rrightarrow}^{\emptyset} \ulcorner\, \mathtt{reducible}\; e\; \mathfrak{s}\,\urcorner \wedge \\
& \quad \forall e', \mathfrak{s}', \vec{e}_f.\, \ulcorner\, (e, \mathfrak{s}) \to (e', \mathfrak{s}', \vec{e}_f)\,\urcorner \;-\!\!*\; {}^{\emptyset}{|\!\!\Rrightarrow}^{\emptyset} \rhd {}^{\emptyset}{|\!\!\Rrightarrow}^{\mathcal{E}} \mathtt{state\_interp}\; \mathfrak{s}' * \\
& \quad\; \mathsf{wp}_{0\;\mathcal{E}}\, e'\, \{Q\} * \mathop{\circledast}_{e_f \in \vec{e}_f} \mathsf{wp}_{0\;\top}\, e_f\, \{\top\})
\end{aligned}$$

All the expected rules about this weakest precondition are proved using the rules of Iris logic. The soundness theorem states that if we can prove $\mathsf{wp}_{\top}\, e\, \{v.\, \ulcorner\, \Phi\; v\,\urcorner\}$ from any world where the initial machine state is $\mathfrak{s}$, then all executions which start from $\mathfrak{s}$ are safe, and if they terminate, their final value satisfies Φ. This is stated formally as follows:

Theorem 4.3.1 (Soundness of the standard weakest precondition)**.** *Given a program e, a machine state $\mathfrak{s}$ and a postcondition $\Phi : \mathbf{Val} \to \mathbb{P}$, if*

$$\models {}^{\top}{|\!\!\Rrightarrow}^{\top} \mathtt{state\_interp}\; \mathfrak{s} \;*\; \mathsf{wp}_{\top}\, e\, \{v.\, \ulcorner\, \Phi\; v\,\urcorner\}$$

then for all execution paths $([e], \mathfrak{s}) \to^{*}_{\mathrm{tp}} (\mathrm{tp}', \mathfrak{s}')$, for all $e' \in \mathrm{tp}'$, either e' is a value, or $(e', \mathfrak{s}')$ is reducible. If moreover the main thread has terminated (that is, $e' = \mathrm{tp}'[0]$ is a value), then $\Phi\; e'$ holds.

The proof proceeds by Löb induction and uses the adequacy theorem of the Iris base logic.

The soundness theorem above does not allow to prove intensional properties of the program; the next chapter presents a new weakest precondition which allows to specify more accurately the behavior of the program.

5 A program logic to relate traces

Program logics describe program behaviors through their soundness theorems: the fact that some predicate about a program is provable inside the program logic implies that the semantics of this program satisfies some property in the ambient logic. This means that the expressiveness of the logic is limited by its soundness theorem.

5.1 Introduction

The Soundness Theorem 4.3.1 for the standard Iris weakest precondition is, essentially, able to express that all vertices in the graph of all possible executions of a program e starting at machine state $\mathfrak{s}$ satisfy some property, which corresponds to invariants being always satisfied.

To see why one would want a finer characterization of the behavior of programs, let us consider a program `hanoi` which solves the Tower of Hanoi problem. Recall that, in the Tower of Hanoi problem, there are three rods, A, B and C and a number n of disks of different diameters $\{1, \dots, n\}$. Each disk has a hole in its center so that it can slide onto any rod A, B or C. In this initial state, all disks are on rod A, in increasing diameter from the top. The goal is to move this stack from rod A to rod B, with the following constraints:

1. a disk must never lie on a smaller disk;
2. at most one disk can be missing from the stacks at any time;
3. each move consists in moving one disk from the top of a stack to the top of another stack (including the empty stack).

It is a standard exercise for students learning recursive programming to write a program which solves the Hanoi problem. One way to encode the Towers of Hanoi as a program specification is as follows: each disk is represented as its diameter, and each stack is represented as a linked list of the diameters of the disks (the head of the list contains the disk on the top of the stack.)

Conditions 1 and 2 can be expressed as invariants of the program logic: at each instant, each linked list is sorted, and the multiset of all elements of every list contains all disks except for at most one (which is being held in the hands of the player of the game) and no disk has a multiplicity greater than 1.

The problem with this specification is that the following program satisfies it:

$$\texttt{hanoi}\ \ell_A\ \ell_B\ \ell_C\ \coloneqq\ \texttt{swap}\ \ell_A\ \ell_B$$

indeed, at the end of the execution, all the disks have been moved to the second rod, and the invariant described above was never invalidated. The problem is that we are not able to express the third condition above, which restricts the *steps* the program can take.

5.1.1 Relating STSs to programs in Iris

More generally, the Towers of Hanoi problem described above is an instance of a more general pattern: there is a state transition system (STS) $(\mathcal{M}, \rightarrow)$ which is an abstract specification of the behavior the program should follow. It can be seen as a specification of the problem to solve as in the example above, or it can represent the algorithm the program should implement.

In the case of the Towers of Hanoi, $\mathcal{M} \subseteq \mathbb{N}^* \times \mathbb{N}^* \times \mathbb{N}^*$ represents the legal states of the three stacks of disks, and the transition relation $\rightarrow \subseteq \mathcal{M} \times \mathcal{M}$ expresses the allowed transitions of the game.

A common strategy, used for example by Robbert Krebbers, Timany, and Lars Birkedal (2017) to encode binary logical relations in Iris, is to state the following invariant

$$\texttt{inv}^{\mathcal{N}}\ (\exists s \in \mathcal{M}.\, \texttt{model\_is}\ s \ast \ulcorner\, s_0 \rightarrow^* s\, \urcorner \ast I\ s)$$

where s_0 is the initial state of the STS —for Hanoi, $s_0 = ([1, \ldots, n], \varepsilon, \varepsilon)$— and $\texttt{model\_is}$ is a predicate which states that the "current state" of the STS is equal to s. The Iris predicate $I : \mathcal{M} \rightarrow \mathit{iProp}$ expresses that the memory of the program corresponds to the state $s \in \mathcal{M}$ of the STS: In the Towers of Hanoi case, it states that the three linked lists contains the same integer sequences as the components of the model.

Using the soundness theorem of the logic, we can prove that at every step k of the execution of the program, there is a state s_k of the STS which is reachable from the initial state and which matches the state of the program according to the predicate I. What is lacking is that the sequence of witnesses $s_1, \ldots, s_k$ is a valid run of the STS $\mathcal{M}$. The technical objective of this work is to construct a program logic using the Iris base logic which does provide such a guarantee.

5.1.2 Simulations

The formal notion of correspondence between executions of the program and the runs of the STS which we use is simulation. This makes sense because a small step semantics of a programming language in the sense of Definition 4.1.1 defines an STS $\mathcal{L}$ over the states $(\mathrm{tp}, \mathfrak{s})$ where tp is a threadpool and $\mathfrak{s}$ is a machine state. The transition relation is $\rightarrow_{\mathrm{tp}}$, ignoring the label. A simulation is a relation between states of two STSs which is, in a sense, stable under transitions:

Definition 5.1.1. Given two STSs $(\mathcal{M}_1, \rightarrow_1)$ and $(\mathcal{M}_2, \rightarrow_2)$, a ***simulation*** is a relation $\phi \subseteq \mathcal{M}_1 \times \mathcal{M}_2$ which satisfies the following property. For all $(s_1, s_2) \in \mathcal{M}_1 \times \mathcal{M}_2$, if $\phi(s_1, s_2)$, and for all $s_1' \in \mathcal{M}_1$ such that $s_1 \rightarrow s_1'$, there exists s_2' such that $s_2 \rightarrow s_2'$ and $\phi(s_1', s_2')$.

Any STS induces another STS whose states are its runs:

Definition 5.1.2. Given an STS $\mathcal{M} = (\mathcal{M}, \rightarrow)$, a *run*, *execution* or *trace* of $\mathcal{M}$ is a finite or infinite sequence $t = \mathfrak{s}_1, \dots, \mathfrak{s}_n, \dots \in \mathcal{M}$ of states such that, for all $1 \leq i$, $\mathfrak{s}_i \rightarrow \mathfrak{s}_{i+1}$. We write $\mathbf{Runs}(\mathcal{M})$ the set of runs of $\mathcal{M}$, and $\mathbf{FRuns}(\mathcal{M})$ for finite runs. If t is a finite run of length n with the notations above, and if $\mathfrak{s}_{n+1}$ is such that $\mathfrak{s}_n \rightarrow \mathfrak{s}_{n+1}$, then $t \cdot \mathfrak{s}_{n+1}$ denotes the run $\mathfrak{s}_1, \dots, \mathfrak{s}_{n+1}$.

Given an STS $\mathcal{M}$, the STS $\mathfrak{R}(\mathcal{M})$ of runs of $\mathcal{M}$ is the STS $\mathfrak{R}(\mathcal{M}) = (\mathbf{FRuns}(\mathcal{M}), \rightarrow)$ where

$$s_1 \cdots s_n \rightarrow s_1 \cdots s_n s_{n+1} \qquad \Longleftrightarrow \qquad s_n \rightarrow s_{n+1}$$

This allows us to define the notion of ***history-sensitive simulation*** as follows: a history-sensitive simulation between two STSs $\mathcal{M}_1$ and $\mathcal{M}_2$ is a simulation between $\mathfrak{R}(\mathcal{M}_1)$ and $\mathfrak{R}(\mathcal{M}_2)$. Such a simulation relates pairs of "executions", and may examine the history of those computations.

This notion of simulation is too strict for our purposes however: each step of the program would need to be related to a step of the STS $\mathcal{M}$, but, because $\mathcal{M}$ is meant to be an abstract description of a computational process, it is natural to expect that it will take fewer steps than the program. For example, for the Towers of Hanoi, all the steps of the program which do not change the state of the memory are not matched by steps in the STS representing the rules of the Hanoi game.

The solution, of course, is to allow the simulation to stutter. In this chapter, we allow infinite stuttering, and define

Definition 5.1.3. Given two state transition systems $\mathcal{M}_1$ and $\mathcal{M}_2$, a ***(history-sensitive termination-insensitive) stuttering simulation*** is a history-sensitive simulation between $\mathcal{M}_1$ and $\mathcal{M}_2^{=}$, where $\mathcal{M}_2^{=}$ is the reflexive closure of $\mathcal{M}_2$.

In this chapter, we will define a program logic which, given a proof of trwp $e\,\{Q\}$, allows for easily proving that some user-chosen relation ϕ contains a stuttering simulation which relates an abstract model $\mathcal{M}$ and a program e, seen as an STS.

More precisely, we define

Definition 5.1.4. Given two STSs $\mathcal{M}_1$ and $\mathcal{M}_2$ and a relation $\phi \subseteq \mathcal{M}_1 \times \mathcal{M}_2$, we define ***$\phi$-similarity*** as the greatest simulation included in ϕ. Two states $s_1 \in \mathcal{M}_1$ and $s_2 \in \mathcal{M}_2$ are called ***ϕ-similar*** if they are related by the ϕ-similarity relation.

This is well defined because simulations are closed under union, and so are the relations $\subseteq \phi$. Intuitively, s_1 and s_2 are ϕ-similar when they are related by ϕ and for any step of s_1, there exists a step of s_2 so that the reducts are related by ϕ. This can be made formal with the following characterization of ϕ-similarity as the greatest fixpoint of the following monotone map over relations in $\wp(\mathcal{M}_1 \times \mathcal{M}_2)$:

$$\mathcal{R} \mapsto \{(x_1, x_2) \mid \phi(x_1, x_2) \ \land \ \forall x_1',\ x_1 \to x_1' \Rightarrow \exists x_2',\ x_2 \to x_2' \ \land \ \mathcal{R}(x_1', x_2')\}$$

This induces the notion of ***ϕ-history sensitive similarity*** and of ***ϕ-stuttering similarity*** since the corresponding notions of simulation are defined as simulations on transformed STSs.

5.1.3 Motivation

This goal of this logic is to be able to state formally that some program module e implements some algorithm $\mathcal{A}$, described as a state transition system $\mathcal{M}$. The usual program logic of Iris allows, in a sense, to prove that e satisfies properties which are expected of $\mathcal{A}$ by specifying e with Hoare triples. The relevance of these Hoare triples is supported by writing clients which use these specifications and whose safety depend on the program module e having the properties expected of it.

Practically speaking, one advantage of our approach is that one can use an off-the-shelf formal description of the algorithm $\mathcal{A}$. For example, we took a TLA+ specification, written by experts, of the single decree Paxos algorithm of L. Lamport (2001), and we proved that some program implements this algorithm. This also allows to split the proof of this subtle program in two independent parts: fist we prove that e implements $\mathcal{M}$ dealing with issues related to the implementation, and then we prove that $\mathcal{M}$ does solve the consensus problem in an idealized setting. See (Timany, Gregersen, et al., 2021) for more details.

As we will see in the next chapter, this technique can be adapted to prove termination of concurrent programs under fair schedulers by building a fair simulation between the program and a well-behaved abstract model.

5.2 The trace weakest-precondition

The new weakest precondition trwp which we introduce in this chapter relates executions of the program with traces in the model. Therefore, we generalize the `state_interp` predicate to take both the memory state $\mathfrak{s}$ and the model state s as parameter. Using the so-called authoritative camera construction which we will not detail here, the `state_interp` and `model_is` predicates can be defined in such a way that they follow the law:

$$\texttt{state\_interp}\,\mathfrak{s}\,s \ast \texttt{model\_is}\,s' \quad \mathrel{-\!\!*} \quad \ulcorner s = s' \urcorner$$

Only the `model_is` predicate is seen by the user of the logic during the proof of the program; the `state_interp` predicate is only used in the definition of the weakest precondition and in the proof of lemmas about trwp and `model_is`.

The logic is parameterized by a predicate `state_evol` $\mathfrak{s}_1\ s_1\ \mathfrak{s}_2\ s_2$ which expresses which transitions in the model are allowed, a notion which can depend on the old memory state $\mathfrak{s}_1$ and the new memory state $\mathfrak{s}_2$ of the program.

To make trwp a strict generalization of the standard Iris weakest precondition wp, in that every rule which is valid for the latter is also valid for the former, it suffices to choose a `state_evol` predicate which satisfies:

$$\forall \mathfrak{s}_1, \mathfrak{s}_2, s, \ \ \texttt{state\_evol}\,\mathfrak{s}_1\ s\ \mathfrak{s}_2\ s.$$

The standard definition for this predicate, given a model STS $\mathcal{M}$, is:

$$\texttt{state\_evol}\,\mathfrak{s}_1\ s_1\ \mathfrak{s}_2\ s_2 \quad \coloneqq \quad \ulcorner s_1 \to s_2 \ \vee\ s_1 = s_2 \urcorner$$

which corresponds to ignoring the machine states and asking that each step of the program correspond to a step in $\mathcal{M}^{=}$, the reflexive closure of the model STS $\mathcal{M}$.

The new weakest precondition is defined as the following guarded fixpoint:

$$\begin{aligned}
\mathsf{trwp}_{\mathcal{E}}\, e_1\,\{Q\} \coloneqq \Big(\ulcorner e_1 \in \mathbf{Val} \urcorner \wedge {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}} Q\, e_1\Big) \ \vee\ \Big(\ulcorner e \notin \mathbf{Val} \urcorner \wedge\ & \\
\forall \mathfrak{s}_1, s_1,\ \texttt{state\_interp}\,\mathfrak{s}_1\ s_1 \mathrel{-\!\!*} {}^{\mathcal{E}}\!\!\Rrightarrow^{\emptyset} \ulcorner \texttt{reducible}\, e_1\, \mathfrak{s}_1 \urcorner \ast & \\
\forall e_2, \mathfrak{s}_2, \vec{e}_f.\ \ulcorner (e_1, \mathfrak{s}_1) \to (e_2, \mathfrak{s}_2, \vec{e}_f) \urcorner \mathrel{-\!\!*} {}^{\emptyset}\!\!\Rrightarrow^{\emptyset} \triangleright {}^{\emptyset}\!\!\Rrightarrow^{\mathcal{E}} \exists s_2,\ & \\
\ulcorner \texttt{state\_evol}\,\mathfrak{s}_1\ s_1\ \mathfrak{s}_2\ s_2 \urcorner \ast \texttt{state\_interp}\,\mathfrak{s}_2\ s_2 \ast & \\
\mathsf{trwp}_{\mathcal{E}}\, e_2\,\{Q\} \ast \mathop{\Large\circledast}_{e \in \vec{e}_f} \mathsf{trwp}_{\top}\, e\,\{\top\}\Big) &
\end{aligned}$$

The weakest precondition takes as input the "current" machine state $\mathfrak{s}_1$ and the current state s_1 of the STS $\mathcal{M}$. If the program e_1 is not a value, it must be reducible in state $\mathfrak{s}$, and for all possible reduction steps of the program $(e_1, \mathfrak{s}_1) \rightarrow (e_2, \mathfrak{s}_2, \vec{e}_f)$, there must exist a corresponding state s_2 of the STS such that the transition from $\mathfrak{s}_1, s_1$ to $\mathfrak{s}_2, s_2$ is allowed, and such that $\mathsf{trwp}_{\mathcal{E}}\, e_2\, \{Q\}$ holds in a world where the new states are $\mathfrak{s}_2$ and s_2 and where the new threads are safe.

Given that guarded recursion is closely related to condinduction, this definition is reminiscent of *similarity*.

The predicate trwp satisfies all the rules expected of a weakest precondition. In addition, it is possible to change the state of the STS while executing an atomic expression (by definition, an expression which reduces to a value in one step). The following rule allows to take a step in the model:

$$\frac{\ulcorner s \rightarrow s' \urcorner * \texttt{model\_is}\ s * \mathsf{trwp}_{\mathcal{E}}\, e\, \{v.\, \texttt{model\_is}\ s' \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\, Q\ v\}}{\mathsf{trwp}_{\mathcal{E}}\, e\, \{Q\}}$$

In other words, if we own the fact that the current models of the STS is s and that $s \rightarrow s'$, then we can assume that, after e has executed, the new state of the STS is s'.

5.2.1 Soundness of trwp

The soundness theorem provides a criterion, given a relation ϕ and a proof of $\mathsf{trwp}_{\top}\, e\{Q\}$ in a logical state where the current program and model states are respectively $\mathfrak{s}_0$ and s_0, for $([e], \mathfrak{s}_0)$ and s_0 to be ϕ-history sensitive similar. This criterion states —if we ignore histories— that $\phi\ \mathfrak{s}\ s$ holds if $\ulcorner\, \texttt{state\_interp}\ \mathfrak{s}\ s\, \urcorner$ does.

The reason why this criterion is easy to use is that this it needs to be satisfied inside of Iris, while being able to use every invariant; and in practice the invariant and ϕ state essentially the same property, the first from inside the Iris logic and the second from the outside. For example, in the case of the Towers of Hanoi, the invariant states that the three linked lists contain the same elements as the three components of the abstract model using points-to predicates, and ϕ states the same fact, in the form of a predicate over the memory and the state of the STS.

We need some notations to state the soundness theorem. Given a run t of an STS, we write

$$t \quad = \quad x \cdots y$$

to mean that the first state of the run t is x and the last one is y. Note that if t is of length 1, x and y are the same state. We write $|t|$ for the length of the run t. Finally, if $t = x_1 \cdots x_{n-1} x_n$ is a run of length > 1, we write $t_{[:-1]}$ for the trace $x_1 \cdots x_{n-1}$.

As we will explain in the next section, the following technical condition is required by the fact that the Iris logic is step-indexed.

Definition 5.2.1. Given relation ϕ between finite executions of $\mathcal{L}$ and finite runs of $\mathcal{M}$, the STS $\mathcal{M}$ is called ***ϕ-finitary*** if, for all s, t_1, t_2, the the following set is finite:

$$\{s' \mid s \xrightarrow{\ell} s' \land \phi\ (t_1 \cdot \mathfrak{s})\ (t_2 \cdot s')\}$$

Theorem 5.2.2. *Let ϕ be a relation between finite executions of $\mathcal{L}$ and finite runs of $\mathcal{M}$. Given a program e, an initial memory $\mathfrak{s}$ and a model STS $\mathcal{M}$ with an initial state s_0, if the Iris formula*

$$\begin{aligned}
&{}^{\top}\!\Rrightarrow^{\top} \mathtt{state\_interp}\ \mathfrak{s}_0\ s_0 * \mathrm{trwp}_{\top}\ e\ \{Q\} * \\
&\quad \big(\forall t_1, t_2, \mathrm{tp}, \mathfrak{s}, s,\ \ulcorner\ t_1 = ([e], \mathfrak{s}_0) \cdots (\mathrm{tp}, \mathfrak{s}) \land t_2 = s_0 \cdots s\ \urcorner \mathrel{-\!\!*} \\
&\qquad (\ulcorner\ |t_1| > 0 \land |t_2| > 0 \implies \phi\ t_{1[:-1]}\ t_{2[:-1]}\ \urcorner) \mathrel{-\!\!*} \\
&\qquad \mathtt{state\_interp}\ \mathfrak{s}\ s \mathrel{-\!\!*} \\
&\qquad {}^{\top}\!\Rrightarrow^{\emptyset} \ulcorner\ \phi\ t_1\ t_2\ \urcorner\big)
\end{aligned}$$

is valid, and if the STS $\mathcal{M}$ is ϕ-finitary, then the singleton traces $[([e], \mathfrak{s}_0)]$ of the program and $[s_0]$ of the STS are ϕ-history sensitive similar.

The implication

$$|t_1'| > 0 \land |t_2'| > 0 \implies \phi\ t_{1[:-1]}\ t_{2[:-1]}$$

is a terse way of stating that the user needs to establish that ϕ relates the initial machine and STS states, and that afterwards they can assume that ϕ relates the previous states.

In the standard instantiation explained above with the reflexive closure $\mathcal{M} \coloneqq \mathcal{M}_0^{=}$ of a STS $\mathcal{M}_0$, the conclusion states that the singleton executions $[([e], \mathfrak{s}_0)]$ and $[s_0]$ are ϕ-history sensitive similar in the STSs $\mathcal{M}^{=}$ and $\mathcal{L}$. By definition of stuttering simulation, this means that $[([e], \mathfrak{s}_0)]$ and $[s_0]$ are ϕ-stuttering similar.

Remark 5.2.3. We have not presented the soundness theorem of the logic in is full generality for clarity, instead we have presented it as it looks when instantiated for some programming language. We refer the reader to the Coq development for the more general definition, where the predicates `state_interp` and `state_evol` are existentially quantified, and where there are more hypothesis available to prove $\ulcorner\ \phi\ t_1\ t_2\ \urcorner$.

5.2.2 Proof of the soundness theorem

Because ϕ-similarity can be defined as a coinductive predicate, the proof of the soundness theorem boils down to the fact that a guarded fixed point trwp in the Iris logic implies a coinductive predicate in the ambient logic.

This proof proceeds by considering an intermediate "pure" predicate gsim which sits in between those two predicates. It states the definition of ϕ-similarity *expressed as a guarded fixpoint* which is independent of resources. Writing $\mathsf{Hyp}(\phi, et, mt)$ for the hypothesis in Theorem 5.2.2, where et and mt are singleton program trace and STS run respectively, the proof proceeds by proving the following two implications:

1. $\models \mathsf{Hyp}(\phi, et, mt) \twoheadrightarrow \mathsf{gsim}_\phi(et, mt)$
2. $(\models \mathsf{gsim}_\phi(et, mt)) \implies et$ and mt are ϕ-similar

Before we describe the proofs of these two implications in turn, let us define the $\mathsf{gsim}_\phi(et, mt)$ predicate as the Iris guarded fixpoint:

$$\begin{aligned} \mathsf{gsim}_\phi(et, mt) \;\coloneqq\; & \triangleright \ulcorner \phi\ et\ mt \urcorner \wedge \forall \mathsf{tp}, \mathfrak{s}, \mathsf{tp}', \mathfrak{s}'. \\ & \quad \ulcorner et = \cdots (\mathsf{tp}, \mathfrak{s}) \;\wedge\; (\mathsf{tp}, \mathfrak{s}) \rightarrow_{\mathsf{tp}} (\mathsf{tp}', \mathfrak{s}') \urcorner \;\longrightarrow \\ & \quad \triangleright \triangleright \big(\exists s'. \ulcorner \mathtt{state\_evol}\ \mathfrak{s}\ s\ \mathfrak{s}'\ s' \urcorner \wedge \mathsf{gsim}_\phi(et \cdot (\mathsf{tp}', \mathfrak{s}'), mt \cdot s')\big) \end{aligned}$$

where the first $\triangleright$ spans over the whole formula. Notice that all the atomic predicates used in this definition are of the form $\ulcorner \Phi \urcorner$. This implies that it does not depend on resources: for all resources a and a'

$$a, n \models \mathsf{gsim}_\phi(et, mt) \qquad \Longleftrightarrow \qquad a', n \models \mathsf{gsim}_\phi(et, mt)$$

The three later modalities in the definition of gsim are necessary to eliminate the laters and the fancy updates (which are defined using a later modality) in the definition of trwp during the proof of the first implication $\mathsf{Hyp}(\phi, et, mt) \twoheadrightarrow \mathsf{gsim}_\phi(et, mt)$.

The proof of this implication proceeds by Löb induction and it fairly similar to the proof of soundness of the usual Iris weakest precondition.

Extracting a trace from a guarded predicate

The more interesting part of the proof is the second implication:

$$(\models \mathsf{gsim}_\phi(et, mt)) \qquad \implies \qquad et \text{ and } mt \text{ are } \phi\text{-similar}$$

To emphasize the resemblance between the gsim predicate and ϕ-similarity, we express the latter in a language close to the Coq syntax for coinductive predicates:

$$\begin{aligned} & \textbf{Coinductive}\ \mathsf{gsim}_\phi(et, mt) \;\coloneqq \\ & \qquad \phi\ et\ mt \;\wedge\; \forall \mathsf{tp}, \mathfrak{s}, \mathsf{tp}', \mathfrak{s}'. \\ & \qquad\quad et = \cdots (\mathsf{tp}, \mathfrak{s}) \;\wedge\; (\mathsf{tp}, \mathfrak{s}) \rightarrow_{\mathsf{tp}} (\mathsf{tp}', \mathfrak{s}') \;\longrightarrow \\ & \qquad\quad \exists s'. \mathtt{state\_evol}\ \mathfrak{s}\ s\ \mathfrak{s}'\ s' \wedge \mathsf{gsim}_\phi(et \cdot (\mathsf{tp}', \mathfrak{s}'), mt \cdot s') \end{aligned}$$

They only differ syntactically by the modalities and the different nature of their fixpoint. The proof of this implication is not trivial however: the STS needs to be ϕ-finitary for it to hold. The reason is that the following result does not hold in general:

$$\vDash \exists x : A.\, P\ x \quad \Rightarrow \quad \exists x : A,\ \vDash P\ x \tag{5.1}$$

where the predicate P does not depend on resources. The reason is that, unfolding the definition of $\vDash \exists x : A.\, P\ x$ and of validity, this amounts to proving that

$$\left(\forall i : \mathbb{N}, \exists x_i : A,\ i \vDash P\ x_i\right) \quad \Rightarrow \quad \exists x : A, \forall i : \mathbb{N},\ i \vDash P\ x$$

which, not surprisingly, does not hold for all A (we omit the resource a since validity of these formulas do not depend on it). If A is finite however, if the LHS of the implication holds, by the pigeonhole principle and the axiom of choice, there exists x_∞ such that for infinitely many $i \in \mathbb{N}, i \vDash P\ x_\infty$.

Because, by definition, predicates are downward closed for step indices ($i \vDash P$ implies that, for any $j < i$, $j \vDash P$), this witness x_∞ holds for every step-index i, and the implication (5.1) holds.

The case of the universal quantification and conjunction is not problematic because this amounts to permuting two universal quantifiers. Disjunction is equivalent to existential quantification over Booleans which means that “disjunction distributes over $\vDash$” (though we do not need it here).

Using (5.1) to extract the witness s', the proof proceeds by a simple coinduction.

This technique was used by Tassarotti, Jung, and Harper (2017) and Spies et al. (2021) for similar purposes.

Remark 5.2.4. The condition that the STS $\mathcal{M}$ is finitely branching which we presented here is stronger than necessary. In the next chapter, we need the weaker but more complicated condition, that for all program configuration $(\mathrm{tp}, \mathfrak{s})$ and machine state $\mathfrak{s}'$, for all STS state $s \in \mathcal{M}$,

$$(\exists et, mt,\ et = \cdots (\mathrm{tp}, \mathfrak{s}) \land mt = \cdots s \land \phi\ et\ mt) \Rightarrow |\{s' \mid \mathtt{state\_evol}\ \mathfrak{s}\ s\ \mathfrak{s}'\ s'\}| < \infty$$

5.2.3 Infinite traces

The soundness theorem of this logic is a proof method to prove that a singleton program trace $[(\mathrm{tp}, \mathfrak{s})]$ and a singleton run $[s]$ of the STS $\mathcal{M}$ are ϕ-similar for some binary relation ϕ.

This implies in particular that given a possibly infinite trace et of the form $et = (\mathrm{tp}, \mathfrak{s}) \cdots$, there exists a possibly infinite run mt of $\mathcal{M}$ of the same length as et which begins by s, and such that:

$$\forall n < |et|,\ \phi\ et_{[:n]}\ mt_{[:n]}$$

where we write $et_{[:n]}$ for the prefix of et which consists in the first n-transitions.

The fact that this relates infinite traces will allow us to transport liveness properties from the STS run to the execution trace in the next chapter.

5.3 Related works

There have been several works which aim at proving refinement of programs using a concurrent separation logic such as Iris, often using logical relations: (Robbert Krebbers, Timany, and Lars Birkedal, 2017) explains how to define binary logical relations and prove them sound using Iris, (Timany, Léo Stefanesco, et al., 2017) defines a binary logical relation to prove that the ST monad preserves observational purity of the language, and ReLoC is a relational logic encoded in Iris (Dan Frumin, Robbert Krebbers, and Lars Birkedal, 2018). The goal of these logics is to prove contextual refinements, whereas our goal is to prove more intentional notions of refinements at the level of traces. As a consequence, they can use the usual Iris weakest precondition, and put the state of the program being refined in the logical state. They state in an Iris invariant the fact the current state of the refined program is reachable from its initial state, as we explained at the beginning of this chapter.

The goal of the Polaris logic, by Tassarotti and Harper (2019), is to reason about programs written in a probabilistic concurrent shared memory programming language. The nondeterminism of the scheduler mixed with the probabilistic behavior of the program is a difficult combination to reason about. Their solution is to relate such a program e with a simple probabilistic program which denotes an element of a monad for mixing non-determinism and probabilities defined by Varacca and Winskel (2006). Our method is similar since we relate a program with an abstract specification of its behavior, which is easier to study. Their weakest precondition is similar to ours, specialized to their notion of abstract model.

There are works aiming at proving non-interference properties of programs using program logics. The work closest to ours is SeLoC (D. Frumin, R. Krebbers, and L. Birkedal, 2021) a logic on top of Iris to prove strong low-bisimulations. Bisimulations are easier in some respect to deal with because they do not have existential quantifiers interacting with the step indexing, and they do not deal with stuttering.

6 Fair termination in Iris

Most concurrent programs need a fair scheduler to make progress. This is especially true when programs cannot rely on an operating system to put to sleep threads waiting for a lock to become unlocked and to wake them up when another thread unlocks this lock.

This means that whenever a thread A is waiting on another thread B to be able to make progress, for instance the following program which uses a lock ℓ:

$$\texttt{acquire}(\ell); \texttt{release}(\ell) \quad \| \quad \texttt{acquire}(\ell); \texttt{release}(\ell)$$

an unfair scheduler could first execute the left hand-side `acquire` operation, followed by only executing the right hand-side thread. Because the program cannot go to sleep, the implementation of `acquire` must spin and try to acquire the lock in a loop until it succeeds. This means that this program has infinite executions.

In most circumstances, it is not a reasonable behavior for the scheduler, and we only consider traces where the scheduler is behaving fairly:

Definition 6.0.1. Consider an execution trace of a program, of the form:

$$(\mathrm{tp}_1, \mathfrak{s}_1) \xrightarrow{tid_1}_{\mathrm{tp}} (\mathrm{tp}_2, \mathfrak{s}_2) \xrightarrow{tid_2}_{\mathrm{tp}} \cdots$$

This trace is called ***fair*** if, either the trace is finite, or, for all $n \in \mathbb{N} \setminus \{0\}$, if the thread at index *tid* of tp_n is enabled, then there exists $m \in \mathbb{N}$ such that either:

1. the thread at index *tid* is not enabled in tp_{n+m}, or
2. the thread *tid* is executed in step $n + m$, in that: $tid_{n+m} = tid$.

In words and in the language of linear temporal logic, it is *always* the case that if a thread is enabled, *eventually* it is disabled or it runs.

Note that, in our particular setting, a thread stops being enabled only after it takes at least one step, so the first disjunct of the definition could be removed. We choose however to use the usual definition.

It is easy to convince oneself that every fair execution of the simple program above is finite. By definition, this means that it is ***fairly terminating***.

Our goal in this chapter is to use the framework described in the previous chapter to prove the fair termination of concurrent programs in the Iris logic.

6.1 Termination and fairness preserving simulation

Proving termination of a program "directly" in Iris is made very difficult by the step-indexing underlying it: the validity of an Iris predicate states that it holds at every step-index. If, as it is the case for most weakest preconditions, each step of the program corresponds to at least one later modality $\triangleright$ in the weakest precondition, then we are limited to expressing properties about all finite prefixes for executions of the program. Termination being a liveness property, finite prefixes are not enough.

One solution, which is part of the Iris development, is to define a weakest precondition without later modalities. Such a weakest precondition guarantees termination, but does not allow to eliminate later modalities. As a consequence, not all predicates can be used in invariants, and Löb induction cannot be used.

6.1.1 Fairness models

Our approach is instead to relate the program to an abstract model expressed as an STS in such a way that if the model is fairly terminating, so is the program. This means that the traces of the STSs we consider need to come with a notion of fairness, which is not the case of the simple unlabeled STSs of the previous chapter. Instead, we use the following notion:

Definition 6.1.1. A ***fairness model*** $\mathcal{F}$ is the data of a set $\mathcal{F}$ of states, a set $\mathcal{R}$ of roles, and a transition relation $\rightarrow \subseteq \mathcal{F} \times \mathcal{R} \times \mathcal{F}$ labeled by roles. Moreover, it is equipped with a map $\texttt{enabled\_roles} : \mathcal{F} \rightarrow \wp_{\text{fin}}(\mathcal{R})$ which associates a finite set of *enabled roles* to each state $s \in \mathcal{F}$. It must approximate the set of outgoing roles:

$$\forall \rho \in \mathcal{R},\ \forall s, s' \in \mathcal{F},\ s \xrightarrow{\rho} s' \implies \rho \in \texttt{enabled\_roles}\ s.$$

and it must not disable other roles: if $s \xrightarrow{\rho} s'$, then

$$\forall \rho' \neq \rho, \quad \rho' \in \texttt{enabled\_roles}\ s \Rightarrow \rho' \in \texttt{enabled\_roles}\ s'$$

Finally, $\mathcal{F}$ comprises a map $\mathit{fuel\_limit} : \mathcal{F} \rightarrow \mathbb{N}$ which will be useful to ensure finite branching conditions.

A run of $\mathcal{F}$ is a non-empty finite or infinite sequence of the form:

$$s_1 \xrightarrow{\rho_1} s_2 \xrightarrow{\rho_2} \cdots$$

The notion of role plays the same role as thread identifiers in the STS of programs: they allow to talk about "who" is taking steps. In particular, this notion allows to define a notion of fair run for a fairness model:

Definition 6.1.2. Consider a run of a fairness model $\mathcal{F}$ of the form:

$$s_1 \xrightarrow{\rho_1} s_2 \xrightarrow{\rho_2} \cdots$$

This run is called ***fair*** if it is finite, or if, for all $n \in \mathbb{N}\backslash\{0\}$, for all $\rho \in \texttt{enabled\_roles}\ s_n$, there exists $m \in \mathbb{N}$ such that either $\rho \notin \texttt{enabled\_roles}\ s_{n+m}$, or $\rho = \rho_{n+m}$.

The fairness model $\mathcal{F}$ is called ***fairly terminating*** if all its fair runs are finite.

6.1.2 Running example

The running example in this chapter is a classical example of a simple program which fairly terminates. It has two threads which each flip k times a shared Boolean variable b. Thread Yes flips it from true to false, and thread No flips it from false to true. In code:

```
Yes () ≔ (if CAS(b, true, false) then n ≔ !n − 1); if !n > 0 then Yes ()
 No () ≔ (if CAS(b, false, true) then m ≔ !m − 1); if !m > 0 then No ()
```

where we use ML notations for references: !n reads the contents of the reference n, and n ≔ v updates it with a new value v. Note that only b is shared, the variables n and m are local to each thread, and initially hold k.

The CAS(ℓ, v_1, v_2), which stands for compare-and-swap, or compare-and-set, is an atomic primitive which updates the contents of ℓ with v_2 if it currently contains v_1, in which case it returns true; otherwise, if ℓ contains another value than v_1, it does not modify ℓ and returns false.

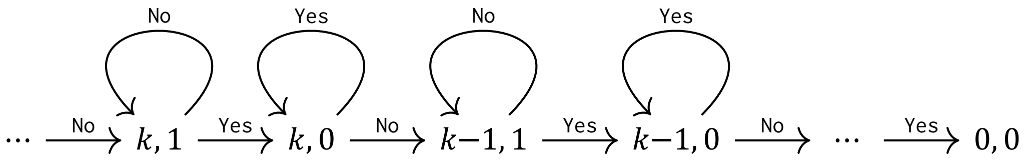
This program fairly terminates because, intuitively, at each stage of the computation, exactly one of the two threads can make progress. In a fair execution, each thread eventually gets executed, which means that the program makes progress eventually. This means that the contents of n and m always eventually decrease, and since they are always positive, the program terminates.

The model we will use, in a sense, formalizes the above reasoning by only counting the relevant steps, which are the `CAS` operations, which are less that one twentieth of the reduction steps in this simple example.

To define a small model, it is useful to note that the contents of `n` and of `m` are almost equal just before executing the `CAS` operation:

- if b is true, `m` and `m` are equal,
- if b is false, $\mathtt{m} = \mathtt{n} + 1$

this means that, in a way, the state of the program can be summarized by the value of `m` and the value of b. Hence, the states of the fairness model $\mathcal{F}_{\text{YN}}$ for the example above are the pairs (m, b) of a natural number $m \in \mathbb{N}$ and a Boolean $b \in \mathbb{2}$. There are two roles, `Yes` and `No`, which represent the actions of the corresponding threads of the program. The transitions are the following, labeled with their role:



The loops represent the failed CASes, and the horizontal transitions represent the successful CASes which make the program progress. For now, let us say that both roles are enabled in every state, that is, `enabled_roles` is the constant map equal to $\{\mathtt{Yes}, \mathtt{No}\}$. We will see in the sequel that, for rather unfortunate technical reasons, the model needs to be slightly modified.

Since our strategy to prove that the the parallel composition of `Yes` and of `No` is fairly terminating is to reduce it to the fact that $\mathcal{F}_{\text{YN}}$ is a fairly terminating fairness model, it is natural to ask how does one prove the latter.

6.1.3 A local criterion for the fair termination of models

Proving directly that a simple model such as $\mathcal{F}_{\text{YN}}$ is fairly terminating is very tedious, as one needs to do a lexicographic induction on the pairs which define the states, but also an induction on the "deadline" m which Definition 6.1.2 supplies.

Instead, one can use a local criterion which formalizes the intuition that at each step, there is a certain role ρ which will make progress when it gets to run. Given a fair run, we know that this role is eventually going to run, and therefore that progress will be made. The notion of progress is defined using the usual notion of well-founded order:

Definition 6.1.3. An ordered set $(A, \leq)$ is a *well-founded ordered set* if there does not exist any infinite decreasing sequences of elements of A.

The states of the fairness model $\mathcal{F}_{\text{YN}}$ equipped with the lexicographic order over $\mathbb{N} \times \mathbb{2}$ are well-founded. We can now state our criterion:

Definition 6.1.4. A fairness model $\mathcal{F}$ is called ***locally fairly terminating*** if there exists a well founded order $\leq$ over $\mathcal{F}$ and a map $\pi : \mathcal{F} \to \mathcal{R}$ from states to roles which satisfies the following conditions:

1. for all transitions $s \xrightarrow{\rho} s'$, $s' \leq s$;
2. for all states $s \in \mathcal{F}$ which are not dead ends, $\pi(s) \in \texttt{enabled\_roles}\ s$ and for all s' such that $s \xrightarrow{\pi(s)} s'$, $s' < s$;
3. for all transitions $s \xrightarrow{\rho} s'$, if $\rho \neq \pi(s)$, then $\pi(s') = \pi(s)$;

where a state s is called a dead end if there are no outgoing transitions from it.

We call this criterion *local* because it can be checked for each transition independently, without any reference to traces. This criterion is correct:

Lemma 6.1.5. *If a fairness model $\mathcal{F}$ is locally fairly terminating, then it is fairly terminating.*

Proof. The proof proceeds by proving the following statement by well-founded induction over $\bar{s}$: *For any run mt whose first state is $\leq \bar{s}$, if it is fair, then it is finite.*

Let $\bar{s}$ be a state and mt be such a fair model run; write s_k its kth state and ρ_k the role of its kth transition, pose $\rho := \pi(s_1)$. By definition of locally fair model, $\rho \in \texttt{enabled\_roles}\ s_1$, which means, since mt is fair, that there exists $m \in \mathbb{N}$ such that

$$\rho \notin \texttt{enabled\_roles}\ s_{1+m} \ \lor\ \rho = \rho_{1+m} \tag{6.1}$$

where we see s_k and ρ_k as depending on the trace mt. We proceed by induction over natural numbers m.

For $m = 0$, the left disjunct is in contradiction with $\rho \in \texttt{enabled\_roles}\ s_1$, therefore $\rho = \rho_1$ and, by definition of local fairness, $s_2 < s_1$, and we conclude with the "outer" induction hypothesis applied to the tail of the run mt.

For $m > 0$, we examine the first transition of mt:

1. If the trace is a singleton it is finite and we are done.

2. If it is of the form $s_1 \xrightarrow{\rho} s_2$, we proceed as above with the "outer" induction hypothesis.
3. Otherwise, $\rho_1 \neq \rho$, so $\pi(s_2) = \pi(s_1)$, and the tail of mt satisfies (6.1) for $m - 1$ instead of m, and so we can conclude using the "inner" induction hypothesis.

□

The criterion is enough to prove that $\mathcal{F}_{\text{YN}}$ is fairly terminating: define $\pi((m, b))$ to be Yes for $b = 1$ and No otherwise. Checking that this defines a locally fairly terminating model is immediate.

6.1.4 Termination and fairness preserving refinements

Now that we have explained how to establish the fair termination of a fairness model $\mathcal{F}$, it remains to define a notion of refinement *between traces* $\lesssim$ which preserves the properties we need: Given a program execution et and a fairness model run mt such that $et \lesssim mt$,

1. if et is a fair execution, then mt is a fair run,
2. if mt is a finite run, then et is a finite execution.

Given such a notion of refinement, to prove that a program e fairly terminates, it is sufficient to find a fairly terminating fairness model $\mathcal{F}$ such that, for all execution trace et of e, there exists a run mt of $\mathcal{F}$ such that $et \lesssim mt$. The existence of such a run will follow in great part from the corollary of the soundness of trwp which we stated in Section 5.2.3.

The main difficulty in defining the trace refinement relation $\lesssim$ is to enforce fairness preservation. The intuition is the following. If the execution trace et is fair, this means that "resources" are equitably allocated to each thread; and such a trace must only be related to run of $\mathcal{F}$ where resources are allocated equitably to each role. Keeping in mind that we want our proof method to be thread-local, we ask that each role ρ be the responsibility of a thread $\mathfrak{T}(\rho) \in \mathbb{N}$, and each thread tid needs to allocate the resources it gets from the scheduler fairly among the set $\mathfrak{T}^{-1}(tid)$ of roles it is in charge of.

More concretely, any infinite sequence of transitions of thread tid in et must correspond to an infinite number of transitions in mt of *every* role in $\mathfrak{T}^{-1}(tid)$. This is achieved by associating a "fuel amount" $\mathfrak{F}(\rho) \in \mathbb{N}$ to each role ρ. Each step of a thread tid must decrease (strictly) the amount of fuel of every role in $\mathfrak{T}^{-1}(tid)$, unless this step corresponds, according to $\lesssim$, to a step labeled with a role $\rho \in \mathfrak{T}^{-1}(tid)$, in which case the fuels of the roles in $\mathfrak{T}^{-1}(tid) \setminus \{\rho\}$ must decrease, and the fuel of ρ can be filled up.

Remark 6.1.6. This notion of fuel is, in a sense, dual to step-indexing: if the step-index reaches "-1", Prover wins the game (that is, $0 \vDash \triangleright \bot$ is true), in other words, Verifier needs to find a contradiction in a finite number of moves. Here, on the other hand, Prover needs to produce a witness before the fuel runs out.

This relation $\precsim$ over traces needs to keep track of these two mappings $\mathfrak{T}$ and $\mathfrak{F}$ which need to evolve at each step. To manage this complexity, we define $\precsim$ as the composition of to relations: $\precsim \coloneqq \precsim_\mathsf{s} \circ \precsim_\mathsf{f}$ which relate the former traces to an existentially quantifier trace of an STS $\mathfrak{Live}(\mathcal{F})$. In other words, for program execution et and a model run mt,

$$et \precsim mt \qquad \Longleftrightarrow \qquad \exists \mathsf{t} \in \mathbf{Runs}(\mathfrak{Live}(\mathcal{F})),\ \ et \precsim_\mathsf{f} \mathsf{t} \ \ \land \ \ \mathsf{t} \precsim_\mathsf{s} mt$$

The relation $\precsim_\mathsf{f}$ is the predicate on traces which is induced by a mild generalization of the notion of simulation of Definition 5.1.1, and $\precsim_\mathsf{s}$ is a very simple as well. Hence, the whole complexity of the accounting of fuel and of the correspondence between threads and roles (which can change dynamically) is confined in the definition of $\mathfrak{Live}(\mathcal{F})$.

6.1.5 The $\mathfrak{Live}$ construction

Given a fairness model $\mathcal{F}$, we define a (labeled) STS $\mathfrak{Live}(\mathcal{F})$ which keeps track of mapping between roles and threads, and of fuels, as suggested above.

Definition of the labeled STS

A state of $\mathfrak{Live}(\mathcal{F})$ is a triple $(s, \mathfrak{F}, \mathfrak{T})$ of a state $s \in \mathcal{F}$, together with two maps $\mathfrak{F} : \texttt{enabled\_roles}\ s \to \mathbb{N}$ and $\mathfrak{T} : \texttt{enabled\_roles}\ s \to \mathbb{N}$ which associate a fuel amount and a thread ID to each role ρ which is enabled in the current underlying state s.

The set of labels of $\mathfrak{Live}(\mathcal{F})$ is

$$\{\mathsf{Step}\ \rho\ tid \mid \rho \in \mathcal{R}, tid \in \mathbb{N}\} \quad \uplus \quad \{\mathsf{Silent}\ tid \mid tid \in \mathbb{N}\}$$

(recall that $\mathcal{R}$ is the set of roles of the fairness model $\mathcal{F}$) The intuition is that a step labeled by $\mathsf{Step}\ \rho\ tid$ corresponds to the situation where the thread tid takes a step in the program, and one of the roles ρ under its responsibility takes a step in the fairness model. A step labeled $\mathsf{Silent}\ tid$, on the other hand, corresponds to a step in the program which does not correspond to a step in the fairness model, in other words, a stuttering step.

We now describe the transitions in the labeled STS $\mathfrak{Live}(\mathcal{F})$. The idea is that there are two kinds of transitions

- A thread *tid* can take a step in $\mathfrak{Live}(M)$ of the form

  $$(s, \mathfrak{F}, \mathfrak{T}) \xrightarrow{\text{Silent } tid} (s, \mathfrak{F}', \mathfrak{T}')$$

  which does not corresponds to a step in the underlying fairness model $\mathcal{F}$ in exchange of consuming fuel: for *every* $\rho \in \mathfrak{T}^{-1}(tid)$ which the thread *tid* is in charge of, the corresponding fuel $\mathfrak{F}(\rho)$ must decrease strictly, in that $\mathfrak{F}'(\rho) < \mathfrak{F}(\rho)$.

- A thread *tid* can also take a step in the underlying model which corresponds to a role $\rho \in \mathfrak{T}^{-1}(tid)$

  $$(s, \mathfrak{F}, \mathfrak{T}) \xrightarrow{\text{Step } \rho \ tid} (s', \mathfrak{F}', \mathfrak{T}')$$

  This allows to refill the fuel of ρ up to the limit $\mathit{fuel\_limit}\ s'$, that is, $\mathfrak{F}'(\rho) \leq \mathit{fuel\_limit}\ s'$; this is required to keep the STS finitely branching. All the other roles which are associated with *tid* must decrease:

  $$\forall \rho' \in \mathfrak{T}(tid) \setminus \{\rho\}, \quad \mathfrak{F}'(\rho') < \mathfrak{F}(\rho')$$

  Roles which appear between s and s' can have any fuel $\leq \mathit{fuel\_limit}\ s'$ in $\mathfrak{F}'$. Of course, we also require there be a step

  $$s \xrightarrow{\rho} s'$$

  in the fairness model $\mathcal{F}$.

In addition to the two constraints above, in both cases, the fuel of the roles which are not associated with the thread *tid* must not increase:

$$\forall \rho \in \mathcal{R} \setminus \mathfrak{T}^{-1}(tid), \quad \mathfrak{F}'(\rho) \leq \mathfrak{F}(\rho)$$

and roles which change owners ($\mathfrak{T}'(\rho) \neq \mathfrak{T}(\rho)$) must have their fuel decrease strictly.

The trace refinement predicates $\lesssim_\mathrm{f}$ and $\lesssim_\mathrm{s}$

Given two labeled STS $\mathcal{M}_1$ and $\mathcal{M}_2$ with states S_1 and S_2 and labels L_1 and L_2 respectively, and relations $\phi_S \subseteq S_1 \times S_2$ and $\phi_L \subseteq L_1 \times L_2$, we say that a trace *et* of $\mathcal{M}_1$ and a trace *mt* of $\mathcal{M}_2$ are ***related by ϕ_S and ϕ_L*** if: they have the same size, and, for all n, the nth state of *et* and the nth state of *mt* are related by ϕ_S, and similarly ϕ_L relates the respective nth labels of the traces.

The refinement predicate $\precsim_{\mathrm{f}}$ is defined as follows:

$$et \precsim_{\mathrm{f}} \mathrm{t} \quad\Leftrightarrow\quad et \text{ and } \mathrm{t} \text{ are related by enabled\_tids and labels\_match}$$

where those two relations are defined as follow. Given a label tid of the labeled STS $\mathcal{L}$ of the programming language, and a label ℓ of the labeled STS $\mathfrak{Live}(\mathcal{F})$, labels_match $tid\,\ell$ holds iff

$$\ell = \mathsf{Silent}\ tid \quad\lor\quad \ell = \mathsf{Step}\ tid\,\rho \quad \text{(for some } \rho\text{)}$$

The predicate over states of the two STSs enforces that thread IDs which appear in the states of $\mathfrak{Live}(\mathcal{F})$ correspond to threads in the corresponding state of the program, and that no role is the responsibility of a thread which is not enabled. Hence we define enabled_tids$((\mathrm{tp}, \mathfrak{s}), (s, \mathfrak{F}, \mathfrak{T}))$ as

$$\begin{aligned}&(\forall \rho,\ tid,\ \mathfrak{T}(\rho) = tid \ \Rightarrow\ tid < |\mathrm{tp}|)\ \land\\ &(\forall tid,\ \mathrm{tp}[tid] \in \mathbf{Val}\ \Rightarrow\ (\forall \rho,\ \mathfrak{T}(\rho) \neq tid))\end{aligned} \tag{6.2}$$

The second trace refinement relation $\precsim_{\mathrm{s}}$ is simpler. There is forgetful map $\mathcal{S}$ from the states of $\mathfrak{Live}(\mathcal{F})$ to the sets of $\mathcal{F}$:

$$(s, \mathfrak{F}, \mathfrak{T}) \ \mapsto\ s$$

and a partial map $\mathcal{L}$ from the labels of of $\mathfrak{Live}$ to the sets of $\mathcal{F}$ which maps the labels of the form $\mathsf{Step}\ tid\,\rho$ to ρ and is undefined otherwise. This defines a map $\mathcal{D}$ from the traces of $\mathfrak{Live}(\mathcal{F})$ to the traces of $\mathcal{F}$ by disregarding transitions which are not in the domain of the label partial map above; this corresponds to destuttering of the trace.

This map is very well behaved:

Lemma 6.1.7. *If the image under $\mathcal{L}$ of the label of transition (trace of length 2) $(s, \mathfrak{F}, \mathfrak{T}) \xrightarrow{\ell} (s', \mathfrak{F}', \mathfrak{T}')$ is not defined, then $s = s'$.*

Moreover, there exists a map Ψ from the states of $\mathfrak{Live}(\mathcal{F})$ to $\mathbb{N}$ such that, for transitions as above:

$$\Psi(s', \mathfrak{F}', \mathfrak{T}') < \Psi(s, \mathfrak{F}, \mathfrak{T})$$

Proof. Take $\Psi(s, \mathfrak{F}, \mathfrak{T}) \coloneqq \sum_{\rho \in \mathtt{enabled\_roles}\ s} \mathfrak{F}(\rho)$. □

The existence of the decreasing potential Ψ means in particular that the destuttering map $\mathcal{D}$ can be defined in Coq. Finally, we define $\precsim_{\mathrm{s}}$ as the graph of $\mathcal{D}$:

$$\mathrm{t} \precsim_{\mathrm{f}} mt \quad\Leftrightarrow\quad \mathcal{D}(\mathrm{t}) = mt$$

Preservation of fairness and termination

There is a natural notion of fairness for runs of the STS $\mathfrak{Live}(\mathcal{F})$:

Definition 6.1.8. Consider a run of $\mathfrak{Live}(\mathcal{F})$, for a fairness model $\mathcal{F}$, of the form:

$$(s_1, \mathfrak{F}_1, \mathfrak{T}_1) \xrightarrow{\ell_1} (s_2, \mathfrak{F}_2, \mathfrak{T}_2) \xrightarrow{\ell_2} \cdots$$

This run is called ***fair*** if it is finite, or for all $n \in \mathbb{N} \setminus \{0\}$, for all $\rho \in \texttt{enabled\_roles}\ s_n$, then there exists $m \in \mathbb{N}$ such that either: $\rho \notin \texttt{enabled\_roles}\ s_{n+m}$, or $\ell_{n+m} = \mathrm{Step}\ \mathit{tid}\ \rho$ for some $\mathit{tid} \in \mathbb{N}$.

This amounts to defining a fair run of $\mathfrak{Live}(\mathcal{F})$ as being in the preimage under $\mathcal{D}$ of a fair run of $\mathcal{F}$.

We are going to prove the following:

$$\begin{tikzcd}
et \arrow[r, bend left, "\text{fair}"] \arrow[r, phantom, "\precsim_{\mathsf{f}}"] & \mathfrak{t} \arrow[l, bend left, "\text{finite}"] \arrow[r, bend left, "\text{fair}"] \arrow[r, phantom, "\precsim_{\mathsf{s}}"] & mt \arrow[l, bend left, "\text{finite}"]
\end{tikzcd}$$

where, for instance, the arrow from et to $\mathfrak{t}$ means that if et is fair, then $\mathfrak{t}$ is fair. The right hand-side arrows are easy to prove, because $\precsim_{\mathsf{s}}$ relates traces which are equal up to finite stuttering. The implication that, if $\mathfrak{t}$ is finite, then et is also finite is obvious because of the definition of $\precsim_{\mathsf{f}}$ states that they must have the same length.

The part which is more difficult to prove is that $\precsim_{\mathsf{f}}$ preserves fairness.

Lemma 6.1.9. *Given a execution et and a run of $\mathfrak{Live}(\mathcal{F})$, if et is a fair execution, then $\mathfrak{t}$ is a fair run.*

Idea of the proof. We write $(s_n, \mathfrak{F}_n, \mathfrak{T}_n) \xrightarrow{\ell_n} (s_{n+1}, \mathfrak{F}_{n+1}, \mathfrak{T}_{n+1})$ the n-th transition of $\mathfrak{t}$, and, similarly, we write $(\mathrm{tp}_n, \mathfrak{s}_n) \xrightarrow{\mathit{tid}_n} (\mathrm{tp}_{n+1}, \mathfrak{s}_{n+1})$ the nth transition of et.

Let ρ be a role, and assume without loss of generality that it is enabled in the first state of $\mathfrak{t}$. We need to prove that there exists a natural number $m \in \mathbb{N}$ such that $\rho \notin \texttt{enabled\_roles}\ s_{1+m}$ or such that $\ell_{1+m} = \mathrm{Step}\ \rho\ \mathit{tid}$ for some tid.

Let $\mathit{tid} = \mathfrak{T}_1(\rho)$, which is defined since we assume the role ρ is enabled at the beginning. Because et is fair, we know that there exists N such that $\mathit{tid}_N = \mathit{tid}$. We proceed by lexicographic induction on the pair $(\mathfrak{F}_1(\rho), N)$ of the fuel of ρ at the beginning of $\mathfrak{t}$ and this bound N. Let us consider the first transition of $\mathfrak{t}$.

1. If $\ell_1 = \mathsf{Step}\ \rho\ tid$, we are done.
2. If $\ell_1 = \mathsf{Step}\ \rho'\ tid'$ or $\mathsf{Silent}\ tid'$, with $tid' \neq tid$, we use the induction hypothesis with the tails of t and of et, with $N - 1$ and the same fuel.
3. If $\ell_1 = \mathsf{Step}\ \rho'\ tid$ with $\rho' \neq \rho$, then the definition of the transition in $\mathfrak{Live}(\mathcal{F})$ states that the fuel of ρ must decrease, and fairness of et gives us a new bound N before which thread tid will take a step, and we conclude using the induction hypothesis.

□

We have now proved everything we need about the trace refinement relations $\precsim_\mathsf{f}$ and $\precsim_\mathsf{s}$. It remains to explain how to prove that, given a program execution et, there exists runs t of $\mathfrak{Live}(\mathcal{F})$ andmt of $\mathcal{F}$ such that $et \precsim_\mathsf{f} \mathsf{t} \precsim_\mathsf{s} mt$. It is easy to see that it suffices to find a trace t, since we can then define $mt \coloneqq \mathcal{D}(\mathsf{t})$. The next section explains how the program logic presented in the previous chapter can be used to prove the existence of t.

6.2 A logic for proving fairness and termination preserving refinements

We use the logic we described in the previous chapter instantiated with the labeled STS $\mathfrak{Live}(\mathcal{F})$ which depends on a user chosen fairness model $\mathcal{F}$.

6.2.1 Changes to the logic

To accommodate the change from unlabeled to labeled state transition systems, we need to slightly change the definition of trwp. Another modification is that because we need to keep track of which thread takes which transition in the fairness model, we need to annotate the weakest precondition with the thread ID of the thread running the code. This new weakest precondition is defined as follows, with the changes highlighted in green:

$$
\begin{aligned}
&\mathrm{fwp}_{\mathcal{E}}\; e_1 \;@\mathit{tid}\,\{Q\} \;\coloneqq\\
&\quad \big(\ulcorner\, e_1 \in \mathbf{Val}\;\urcorner \wedge\; {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} Q\; e_1\big) \;\vee\; \big(\ulcorner\, e \notin \mathbf{Val}\,\urcorner \wedge\\
&\quad \forall \mathfrak{s}_1, s_1, n,\; \texttt{state\_interp}\;\mathfrak{s}_1\; s_1\; n \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\emptyset}\ulcorner\, \texttt{reducible}\; e_1\; \mathfrak{s}_1\;\urcorner \;*\\
&\qquad \forall e_2, \mathfrak{s}_2, \vec{e}_f.\, \ulcorner\, (e_1, \mathfrak{s}_1) \to (e_2, \mathfrak{s}_2, \vec{e}_f)\;\urcorner \mathrel{-\!\!*} {}^{\emptyset}{\Rrightarrow}^{\emptyset} \triangleright\; {}^{\emptyset}{\Rrightarrow}^{\mathcal{E}}\; \exists s_2, \ell.\\
&\qquad \ulcorner\; \texttt{state\_evol}\;\mathfrak{s}_1\; s_1\; \mathit{tid}\;\ell\;(n, |\vec{e}_f|)\;\mathfrak{s}_2\; s_2\;\urcorner \;*\; \texttt{state\_interp}\;\mathfrak{s}_2\; s_2\;(n + |\vec{e}_f|) \;*\\
&\qquad \mathrm{fwp}_{\mathcal{E}}\; e_2 \;@\mathit{tid}\,\{Q\} \;*\; \underset{k \mapsto e \in \vec{e}_f}{\bigcircledast}\; \mathrm{fwp}_{\top}\; e \;@n{+}k\;\{\texttt{fork\_post}\; n{+}k\}\big)
\end{aligned}
$$

In addition to the new *tid* parameter to fwp, the main change is that the `state_evol` predicate takes labels and the thread count into account. The pair $(n, |\vec{e}_f|)$ which is passed to it corresponds to the thread count before the reduction step is taken, and the number of threads created during the reduction step of the program. The total number of threads is retrieved using a new parameter to `state_interp` which corresponds to the total number of threads. Finally, threads which are spawned can have a non-trivial postcondition `fork_post` *tid* which may depend on their thread ID.

In the case where the labeled STS is $\mathfrak{Live}(\mathcal{F})$, the predicate `state_evol` is instantiated as follows:

$$
\begin{aligned}
&\texttt{state\_evol}\;\mathfrak{s}_1\; s_1\; \mathit{tid}\;\ell\;(n, k)\;\mathfrak{s}_2\; s_2 \coloneqq\\
&\qquad \mathrm{labels\_match}\;\mathit{tid}\;\ell \;\wedge\; s_1 \xrightarrow{\ell} s_2 \;\wedge\; (\mathrm{tids\_le}\; n\; s_1 \Rightarrow \mathrm{tids\_le}\;(n+k)\; s_1)
\end{aligned}
$$

where tids_le n s states that every *tid* in the $\mathfrak{T}$ component of s is $\le n$, and where the labels_match predicate is defined in Section 6.1.5, page 149.

The soundness theorem of the logic can also be slightly adapted to fit the labeled setting without any difficulty.

All that remains to do is to deduce a trace refinement from the ϕ-history sensitive similarity predicate which the soundness theorem provides.

Lemma 6.2.1. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two labeled STSs, and let $\phi \subseteq$ $\mathbf{FRuns}(\mathcal{M}_1) \times \mathbf{FRuns}(\mathcal{M}_2)$ a relation between finite runs. Let $\phi_S \subseteq \mathcal{M}_1 \times \mathcal{M}_2$ be a relation between their states, and ϕ_L be a relation between their labels. Then, if $s_1 \in \mathcal{M}_1$ and $s_2 \in \mathcal{M}_2$ are ϕ-history sensitive similar, then, for all finite or infinite trace $t_1 \in \mathbf{Runs}(\mathcal{M}_1)$ which starts with the state s_1, there exists a trace $t_2 \in \mathbf{Runs}(\mathcal{M}_1)$ such that t_1 and t_2 are related by ϕ_S and ϕ_L.*

Using this lemma, we can deduce the existence, for every execution et of a program from an initial state which has been proved with fwp, of a trace $\mathfrak{t}$ of $\mathfrak{Live}(\mathcal{F})$ such that $et \precsim_{\mathfrak{f}} \mathfrak{t}$. We now finish by explaining how the logical resources are setup to provide Iris predicates and reasoning rules which are easy to manipulate.

6.2.2 Logical resources

As expected, the Iris logical state contains Iris resources for each the three components of the states of $\mathfrak{Live}(\mathcal{F})$, and they are tied to the "raw" model state in the `state_interp` predicate.

First, as in the previous chapter, there is a predicate `model_is` s which state that the current state of the underlying fairness model $\mathcal{F}$ is equal to s. As before, it must agree with the `state_interp` predicate:

$$\texttt{state\_interp}\; \mathfrak{s}\; (s, \mathfrak{F}, \mathfrak{T})\; n \;*\; \texttt{model\_is}\; s' \quad -\!\!* \quad \ulcorner s = s' \urcorner$$

For each role ρ which is enabled in the current state of the fairness model $\mathcal{F}$, there is a predicate $\rho \mapsto_F \mathfrak{f}$ which states that the fuel amount associated to ρ is $\mathfrak{f}$. It satisfies the rule:

$$\texttt{state\_interp}\; \mathfrak{s}\; (s, \mathfrak{F}, \mathfrak{T})\; n \;*\; \rho \mapsto_F \mathfrak{f} \quad -\!\!* \quad \ulcorner \mathfrak{F}(\rho) = \mathfrak{f} \urcorner$$

The way we reflect mapping is less straightforward: instead of maps from roles to thread IDs, they are represented as maps from thread IDs to sets of roles. Formally, we represent the map $\mathfrak{T} : R \rightarrow \mathbb{N}$ as a map $M' : \mathbb{N} \rightarrow \wp_{\text{fin}}(R)$ such that:

$$M\; \rho = tid \qquad \Leftrightarrow \qquad \rho \in M'\; tid$$

where $R \subseteq \mathcal{R}$ is a set of roles of $\mathcal{F}$. We put an exclusive structure on the finite sets of roles in the codomain of the map. That is, the camera operation on $\wp_{\text{fin}}(R)$ which we use is never defined. We write $tid \mapsto_T \vec{\rho}$ to denote the exclusive ownership of the singleton map $[tid := \vec{\rho}]$, where $\vec{\rho} \subseteq \mathcal{R}$. The exclusive structure on sets of roles means that $tid \mapsto_T \vec{\rho} * tid' \mapsto_T \vec{\rho}'$ implies that $tid \neq tid'$.

The law which relates it to the states interpretation is the following:

$$\begin{aligned} &\texttt{state\_interp}\; \mathfrak{s}\; (s, \mathfrak{F}, \mathfrak{T})\; n \;*\; tid \mapsto_T \vec{\rho} \quad -\!\!* \\ &\qquad\qquad\qquad \ulcorner \forall \rho,\; \rho \in \vec{\rho} \quad \Leftrightarrow \quad \mathfrak{T}(\rho) = tid \urcorner \end{aligned} \tag{6.3}$$

Remark 6.2.2 (Why exclusive?)**.** The reason we need an exclusive structure on the set of roles above is that the resource corresponds to an *obligation*: when the thread tid makes a transition, the set of roles whose fuel decreases must include the set of roles associated with the thread *tid*. A simple way of achieving this is to associate *tid* with exactly its set of roles in the model.

If we had used a more natural camera structure on these sets, such as disjoint union, the reasoning principle (6.3) would have been too weak, and the equivalence on the right would have been a simple implication. Another way to see this is that the set of roles $\vec{\rho}$ in $tid \mapsto_T \vec{\rho}$ is *linear*. Of course the whole $tid \mapsto_T \vec{\rho}$ can be dropped since Iris is an affine logic, but we cannot drop one single $\rho \in \vec{\rho}$ like we would have been able to do if we had put a disjoint union structure on $\wp_{\text{fin}}(\mathcal{R})$.

The resources for the fuel and the mapping are often used in concert since they contain all the information about some thread. We define

$$\text{tid\_has\_fuels}\ tid\ \mathfrak{f}s \coloneqq tid \mapsto_T \text{dom}(\mathfrak{f}s) * \mathop{\circledast}_{\rho \mapsto \mathfrak{f} \in \mathfrak{f}s} \rho \mapsto_F \mathfrak{f}$$

where $\mathfrak{f}s$ is a finite partial map from roles ρ to fuel amounts $\mathfrak{f}$. This predicate states that the thread *tid* is associated to the roles which are in the domain of $\mathfrak{f}s$ and each of these roles ρ has fuel $\mathfrak{f}s(\rho)$.

6.2.3 Inference rules

Unlike in the previous chapter, where the standard instantiation of trwp allows for infinite stutter by choosing as STS the reflexive closure of some user-chosen STS $\mathcal{M}$, which allows the logic to be conservative over the usual Iris weakest precondition wp, fwp is *not* conservative of wp. Indeed, each “internal” step of of each thread *tid* which does not correspond to a step in the fairness model $\mathcal{F}$ consumes the fuel of the roles which are under its responsibility.

Silent steps

Iris has a useful concept of a pure reduction step of a program. Intuitively, for ML-style languages, this corresponds to steps of a pure λ-calculus.

Definition 6.2.3. A *pure reduction step* $e \to e'$ between two expressions is the data of two expressions which correspond to a reduction step which ignores the machine state

$$\forall \mathfrak{s},\quad (e, \mathfrak{s}) \;\to\; (e', \mathfrak{s})$$

and which is deterministic:

$$\forall \mathfrak{s}, \mathfrak{s}', e'' \quad (e, \mathfrak{s}) \;\to\; (e'', \mathfrak{s}') \quad\Rightarrow\quad e'' = e' \;\wedge\; \mathfrak{s}' = \mathfrak{s}$$

This definition allows to define a rule for pure reductions: if $e \to e'$ is a pure reduction, then:

$$\frac{\text{tid\_has\_fuels}\ \mathit{tid}\ (\mathfrak{fs} + 1) * (\text{tid\_has\_fuels}\ \mathit{tid}\ \mathfrak{fs} \mathrel{-\!\!*} \text{fwp}_{\mathcal{E}}\ e'\ @\mathit{tid}\,\{Q\}) \quad \mathfrak{fs} \neq \emptyset}{\text{fwp}_{\mathcal{E}}\ e\ @\mathit{tid}\,\{Q\}}$$

where $\mathfrak{fs} + 1$ adds 1 to each fuel in the finite partial map $\mathfrak{fs}$. This simply says that reducing by one step "costs" one unit of fuel for every role associated to *tid*. The standard instantiation of trwp in the previous chapter and the standard Iris weakest precondition wp, in contrast, have the simpler rule:

$$\frac{\text{wp}_{\mathcal{E}}\ e'\ \{Q\}}{\text{wp}_{\mathcal{E}}\ e\ \{Q\}}$$

The rule for fwp can be automated in a proof assistant so that the fuel is automatically decreased and the program e automatically reduced until it reaches an expression which is not purely reducible. (Currently, this tactic only supports the common case where $\mathfrak{fs}$ is a singleton, but this should not be too difficult to fix)

The other rules of the logic, for reading and writing references or for CAS operations, including the invariant opening, admit inference rules which are the same as for wp except for the fuel being decremented using the pattern above.

Steps in the fairness model

For the steps of a thread *tid* which correspond to a step in the fairness model $\mathcal{F}$, the rules are a little more complicated because the role ρ of the step in $\mathcal{F}$ is distinguished from the other roles associated to the thread *tid*. We give as an example the rule for a successful CAS, since it is a common point for externally visible actions, and with a role associated to the thread *tid*. The rule is expressed as a Hoare triple, which is defined using fwp as explained in Chapter 4.

The Hoare triple

$$\begin{gathered}\{\triangleright \ell \mapsto v_1 * \triangleright \texttt{model\_is}\ s_1 * \triangleright \text{tid\_has\_fuels}\ \mathit{tid}\ [\rho \coloneqq \mathfrak{f}_1]\} \\ \texttt{CAS}(\ell, v_1, v_2)\ @\ \mathit{tid} \\ \{\ell \mapsto v_2 * \texttt{model\_is}\ s_2 * \text{tid\_has\_fuels}\ \mathit{tid}\ [\rho \coloneqq \mathfrak{f}_2]\}\end{gathered}$$

holds when

- there is a transition $s_1 \xrightarrow{\rho} s_2$ in the underlying fairness model $\mathcal{F}$;
- the role ρ is still enabled in s_2: $\rho \in \texttt{enabled\_roles}\ s_2$;

- the new fuel associated to ρ is under the limit: $\mathfrak{f}_2 \leq \mathit{fuel\_limit}\ s_2$.

If the second condition had been false, the postcondition would have contained the predicate $\mathit{tid} \mapsto_T \emptyset$ instead of $\mathsf{tid\_has\_fuels}\ \mathit{tid}\ [\rho \coloneqq \mathfrak{f}_2]$.

Postcondition for forks

We need to maintain the invariant enabled_tids which is defined in equation (6.2), which states that only non-finished threads are associated to roles. To do this, we require the post-condition of every forked thread to be $\mathtt{fork\_post}\ \mathit{tid} \coloneqq \mathit{tid} \mapsto_M \emptyset$. The soundness theorem of the logic will also require the main thread to have this postcondition.

For this reason, in the example with the two threads Yes and No, the final step of each thread needs to take a step in the fairness model $\mathcal{F}_{\text{YN}}$ to move to a state where the corresponding role is not enabled. This has to be done at the last step because a thread which is associated to no role does not have the right to take any step. We describe how we handle this requirement in the sequel when we describe the proof of the example.

6.2.4 Soundness theorem

We are finally ready to state the soundness theorem of the logic when instantiated with a labeled STS of the form $\mathfrak{Live}(\mathcal{F})$ for some fairness model $\mathcal{F}$.

Theorem 6.2.4. *Given a program e, a finitely branching fairly terminating fairness model $\mathcal{F}$, a state s_0 in $\mathcal{F}$, if the following Iris predicate is valid*

$$\mathtt{model\_is}\ \sigma_0 \mathrel{-\!\!*} \mathsf{tid\_has\_fuels}\ 0\ \ \mathit{initial\_fuel} \mathrel{-\!\!*} {}^{\top}{\Rrightarrow}^{\top}\, \mathsf{fwp}_{\top}\ e\ @0\ \{0 \mapsto_T \emptyset\}$$

then e is fairly terminating.

In the Iris predicate above, 0 *is the thread ID of the main thread, and initial_fuel is the constant map defined on* $\mathtt{enabled\_roles}\ s_0$ *equal to fuel_limit* s_0*.*

Proof. The proof is a straightforward application of the soundness theorem of fwp and of the results in Section 6.1. One needs in addition to prove that, if $\mathcal{F}$ is finitely branching, then so is $\mathfrak{Live}(\mathcal{F})$, assuming a bound on the thead IDs which is given by the fact that tids_le holds.

Given a state $(s, \mathfrak{F}, \mathfrak{T})$, a successor state $(s', \mathfrak{F}', \mathfrak{T}')$ is the data of s' such that $s \to s'$, of which we assume there is a finite number, and of two maps defined on a fixed finite set $\mathtt{enabled\_roles}\ s'$ with finite ranges: for $\rho \in \mathtt{enabled\_roles}\ s'$,

$$\mathfrak{F}'(\rho) \leq \max(\max(\mathfrak{F}), \mathit{fuel\_limit}\ s') \qquad \mathfrak{T}'(\rho) \leq \mathit{maxtid}$$

where *maxtid* is the bound given by tids_le, and $\max(\mathfrak{F})$ is the maximal fuel value which appears in $\mathfrak{F}$. □

6.2.5 Back to the example

Recall that we considered two threads

$$\begin{aligned} \texttt{Yes}\ () &\coloneqq (\texttt{if}\ \texttt{CAS}(\texttt{b}, \texttt{true}, \texttt{false})\ \texttt{then}\ \texttt{n} \coloneqq\ !\texttt{n} - 1); \texttt{if}\ !\texttt{n} > 0\ \texttt{then}\ \texttt{Yes}\ () \\ \texttt{No}\ () &\coloneqq (\texttt{if}\ \texttt{CAS}(\texttt{b}, \texttt{false}, \texttt{true})\ \texttt{then}\ \texttt{m} \coloneqq\ !\texttt{m} - 1); \texttt{if}\ !\texttt{m} > 0\ \texttt{then}\ \texttt{No}\ () \end{aligned}$$

flipping a Boolean variable b back and forth a finite number of time, each corresponding to a role in the fairness model $\mathcal{F}_{\text{YN}}$. They do not form a complete program, since they rely on references to be allocated and initialized. We use a program Start which allocates the references, spawns the two threads and stops immediately. Roughly (ignoring lexical scoping issues):

$$\begin{aligned} &\texttt{Start}\ () \coloneqq \\ &\qquad \texttt{b} \coloneqq \texttt{alloc}(\texttt{true});\ \texttt{n} \coloneqq \texttt{alloc}(k);\ \texttt{m} \coloneqq \texttt{alloc}(k); \\ &\qquad \texttt{fork}(\texttt{Yes});\ \texttt{fork}(\texttt{No}) \end{aligned}$$

We do not need to give a role in $\mathcal{F}_{\text{YN}}$ to Start, instead, it is initially responsible of both Yes and No, until the first fork operation, after which it is only responsible of No, and finally it terminates and is responsible of no role. We are lucky, in that there is no operation after it has delegated its last role to the thread No.

This is not the case for the threads Yes and No, because they must decrement their variable after their last CAS. The solution is to add flags to the states of $\mathcal{F}_{\text{YN}}$: a state is of the form

$$(m, b, ye, ne)$$

where the new flags $ye, ne \in \mathbb{2}$ mean that Yes, respectively No, is still executing. The enabled_roles predicate is defined using these flags using

$$\texttt{Yes} \in \texttt{enabled\_roles}\ (m, b, ye, ne) \iff ye = 1$$

and symmetrically for No. The transitions express that threads can shutdown only after their counter has reached zero.

This difficulty surmounted, the proof is fairly straightforward, using an invariant to related the values of b and m in $\mathcal{F}_{\text{YN}}$ to the values of the three references.

6.3 Related works

The most closely related work is a paper by Tassarotti, Jung, and Harper (2017) which uses Iris to construct a logic which allows to prove the existence of a fairness preserving termination preserving refinement between two programs. In their case, their goal is to prove that a simple compiler which replaces abstract channel operations by shared memory implementations preserves fair termination. Their soundness theorem is similar as well, in that it states that each fair infinite trace of the target program is refined by a fair trace of the source program.

They have a second example where they prove that a Craig-Landin-Hagersten queue lock refines a ticket-lock. The statement is made formal by implementing a compiler which translates a program using the first into one using the second.

The main technical difference between the two approaches is that they need to change the base model of Iris in two ways. First, to be able to talk about traces, they add a new piece of data to the definition of a camera: Each camera M has a step-indexed transition relation $(\curvearrowright_n)_{n\in\mathbb{N}}$. For every camera except for the camera which holds the state of the source program, this relation is the total relation.

The second modification to the Iris model is the addition of linear predicates, which they need in order to prevent the prover from dropping the obligation to take a step in the source language. The interpretation of an Iris predicate becomes a non-expansive map

$$M \times M \longrightarrow \mathbb{P}$$

which is upward closed only with respect to its first parameter, which corresponds to the usual Iris affine predicates; the second parameter corresponds to linear predicates. They have an "affinely" modality to express rules which only hold for affine propositions such as framing and weakening. Linear propositions cannot be put inside an invariant.

Our intuition for why we do not need the predicate $\mathit{tid} \mapsto_T \vec{\rho}$ to be linear is that the weakest precondition takes as input a resource of the form $\mathit{tid} \mapsto_T \vec{\rho}\,'$, which means that dropping this resource is not a risk. Moreover, as we mentioned in Remark 6.2.2, the exclusive structure on $\vec{\rho}$ makes the predicate behave somewhat linearly.

We think the main advantage of our approach, beyond the fact that changing the base logic of Iris is a fair amount of work, is that putting the role-to-thread mapping and the fuel accounting in the $\mathfrak{Live}$ construction clarifies the reasoning. Their construction of the camera which holds the state of the source program is fairly involved and takes around 7000 lines of Coq (for comparison the $\mathfrak{Live}$ construction and the proofs of Section 6.1 take 1350 lines).

Less closely related, there as several program logics to reason about fairness of concurrent programs, such as the Rely-Guarantee logic LiLi by Liang and Feng (2016), and the concurrent separation logic Tada Live described in a draft by D'Osualdo et al. (2019).

There also has been a lot of work on the topic of fairness preserving simulations from the model checking community, for example in the context of LTL (Kesten, Manna, and Pnueli, 1994), TLA+ (L. Lamport, 1994), games (Henzinger, Kupferman, and Rajamani, 1997), etc.

Another related line of work are the time credit-base logics by Guéneau, Charguéraud, and François Pottier (2018) and Mével, Jourdan, and François Pottier (2019). Their goal is to prove worst case complexity bounds using a program logic. They add a predicate $\$k$ to the logic to state that the program must finish in less than k steps. This is related to our $tid \mapsto_F \mathfrak{f}$ predicate. The main difference is that they do not keep track of the identity of the thread, and credits always decrease.

Conclusion

This thesis contains two works I was involved in during my PhD. This last chapter concludes this thesis with remarks about the two works which were presented, and future directions which I think are interesting to pursue.

Semantic models of CSL

In Part I of this thesis, I presented a semantic model of a simple concurrent separation logic and its associated programming language. Our goal was to give a semantic model which gives a semantic interpretation to the proof of CSL themselves, in the tradition of Curry-Howard, and which takes on data races head on.

The simplicity of the language, and the fact that the logic is strongly associated to the language compared to Iris which can be instantiated with any language written in the format of Definition 4.1.1, had the benefit, besides simplifying the model, to bring to light structures in the logic. For example, the change of lock operations on the interpretations of the proofs make sense because the logic follows the same lexical scoping of locks as the language.

Choice of the language

Looking back, I believe however that a better choice of programming language would have been a first order λ-calculus with ML-style references. The first advantage is that variables would have been immutable which can be substituted into in the specifications, instead of using the $\mathsf{Own}(x)$ predicates.

The other advantage is that it would have clarified a question we had as to what is the right shape of the cobordisms we use to interpret programs. In this thesis, the "input interface" I needs to be isomorphic to $\pm[0, i]$, but not the output interface, because we had not found this constraint to have been useful. In a language where programs reduce to values however, the natural solution is to interpret a program of type $C : A \to B$, where A and B denote sets if the language is first order, as a cobordism roughly of the form:

$$\begin{tikzcd}
A \times \pitchfork[0] \arrow[r] \arrow[d, "\pi_2"'] & \llbracket C \rrbracket_{support} \arrow[d] & B' \times \pitchfork[0] \arrow[l] \arrow[d, "\pi_2"] \\
\pitchfork[0] \arrow[r] & \pitchfork[1] & \pitchfork[0]. \arrow[l]
\end{tikzcd}$$

where $B' \subseteq B$, following the intuition that the input and output are "covering spaces" above base space $\pitchfork[0]$ such that the fiber above any memory state is the set of possible values.

Since, in the language we consider, every command can be considered as having the type $\mathbb{1} \to \mathbb{1}$, this suggests that the right definition is to ask that the map $\llbracket C \rrbracket_{out} \to \pitchfork[0]$ be an iso. Alas, this remark came to me too late.

Higher order

The variant of CSL we considered cannot reason about higher-order functions. There are two possible ways to scale our approach to this setting.

One possibility would be to do a construction similar to the Int construction of Joyal, Street, and Verity (1996), which constructs a compact-closed category from a traced symmetric monoidal category, which **Cob**($\pitchfork$) is, at least intuitively. The issue is that the resulting double category would not be closed with respect to the parallel product, but the disjoint union. I believe this approach could be made to work for a sequential separation logic.

Another possibility is to switch from our cospan-based semantics to the span-based template games of Melliès (2019). The approach would be to create a game model of the logic of bunched implications which would be parameterized by a partial commutative monoid (PCM). In his setting, the template $\pitchfork$ is an internal category which describes the polarities of the games and the strategies. The idea would be to annotate these polarities with elements of the PCM which would represent the resources owned by each of the players.

Weak memory

Another natural extension is giving a semantics to a concurrent separation logic for weak memory. This is natural because axiomatic semantics of weak memory models are expressed as partial orders over the events of computations, and asynchronous graphs are true concurrency structures which induce partial orders over events.

A counter argument is that most successful concurrent separation logics, eg. (Kaiser et al., 2017), for weak memory use the promising semantics of Kang et al. (2017), whose distinctive feature is that it is, essentially, expressed as a normal state transition system. The complexity of weak memory is largely put inside the state, which is a set of timestamped messages instead of a simple partial map from locations to values. Part of this success can be explained by the fact that separation logic was designed to deal with complex state rather than complex reduction semantics (for instance, separation logics for probabilistic languages are an active area of research).

Fairness in Iris

Part II of this thesis develops a framework on top of Iris for relating the behavior of a program and of an abstract model of computation in an intensional manner. This approach is quite flexible: From a single soundness theorem which asserts that the initial state of the program and of the model are similar in some sense, using constructions on the model we are able to, on the one hand, prove that distributed programs implement classical algorithms, and on the other hand, that concurrent programs fairly terminate.

Finite but unbounded stuttering

The genesis of Chapter 6 is a previous failed attempt at proving fair termination of programs. The goal was to avoid the need for a fixed (in our case, given by the current state of the fairness model) bound to impose finite stuttering. Inspired by the fact that finite stutter can be expressed as a nested coinductive-inductive predicate, our plan was to define a weakest precondition as a *least fixpoint* which would look schematically like this:

$$\begin{aligned}\mathrm{fwp}\ e\ \{Q\} \ &\coloneqq\ \text{“}e\text{ is a value”} \lor \\ &(\forall\ e \to e',\ \mathrm{fwp}\ e'\ \{Q\}) \lor (\forall\ e \to e',\ \text{“take a step in }\mathcal{M}\text{”} \land \triangleright \mathrm{fwp}\ e'\ \{Q\})\end{aligned}$$

Note the lack of later modality in the disjunct corresponding to not taking a step in the model. Because this is a least fixpoint, the program e would either eventually reduce to a value, or the proof of fwp e $\{Q\}$ would eventually take a step in the model.

The issue is that we did not manage to define an Iris predicate similar to gsim which does not depend on the resources and which expresses that *each thread* always eventually takes a step in the model without quantifying over ordinals larger than the indexing ordinal, which is needed to get a simulation in the ambient logic out.

This method works for single-threaded programs, as it was successfully used in (Spies et al., 2021) in version of Iris with transfinite step-indexing, and I think it would improve the usability of the logic significantly if this obstacle was cleared.

Liveness of distributed systems

I think the objective of proving the liveness of distributed systems, under fairness assumptions about the network, it reachable. The idea would be that, for each message which must be received, there is a role in an abstract model, which becomes disabled when the message is received. Locally, this means that each node must send it infinitely often until it receives an acknowledgment.

Liveness is more of an issue in distributed systems compared to typical shared memory algorithms because the nodes and the network are unreliable, so this would be a good case study.

Compositionality of models

More generally, the main limitation of Part II is that the whole program must be proved using one single abstract model $\mathcal{M}$. In the future, we need to develop ways of combining such models as well as hiding them.

Bibliography

America, Pierre and Jan Rutten (1989). “Solving reflexive domain equations in a category of complete metric spaces.” In: *Journal of Computer and System Sciences* (cit. on p. 126).

Appel, Andrew W., Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon (2007). “A very modal model of a modern, major, general type system.” In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007.* Ed. by Martin Hofmann and Matthias Felleisen (cit. on p. 121).

Balabonski, Thibaut, François Pottier, and Jonathan Protzenko (2014). “Type Soundness and Race Freedom for Mezzo.” In: *FLOPS* (cit. on p. 37).

Bénabou, Jean (1967). “Introduction to bicategories.” In: *Reports of the Midwest Category Seminar.* Berlin, Heidelberg, pp. 1–77. ISBN: 978-3-540-35545-8 (cit. on pp. 68, 69).

Birkedal, L., R. Møgelberg, J. Schwinghammer, and K. Støvring (2012). “First steps in synthetic guarded domain theory: step-indexing in the topos of trees.” In: *Logical Methods in Computer Science* (cit. on p. 123).

Birkedal, Lars, Kristian Støvring, and Jacob Thamsborg (2010). “The category-theoretic solution of recursive metric-space equations.” In: *Theoretical Computer Science* (cit. on p. 126).

Bodin, Martin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith (2014). “A Trusted Mechanised JavaScript Specification.” In: POPL ’14. San Diego, California, USA (cit. on p. 8).

Bornat, Richard, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson (2005). “Permission Accounting in Separation Logic.” In: *POPL* (cit. on pp. 24, 54).

Bornat, Richard, Cristiano Calcagno, and Hongseok Yang (2006). “Variables as Resource in Separation Logic.” In: *ENTCS* 155 (cit. on pp. 32, 54).

Boyland, John (2003). “Checking Interference with Fractional Permissions.” In: *Static Analysis.* Ed. by Radhia Cousot. Springer Berlin Heidelberg (cit. on pp. 24, 54).

Brookes, Stephen (2004). “A semantics for concurrent separation logic.” In: *CONCUR* (cit. on pp. 21, 29, 30, 32, 41, 54).

Brookes, Stephen (2011). “A Revisionist History of Concurrent Separation Logic.” In: *Mathematical Foundations of Programming Semantics, MFPS.* Ed. by Michael W. Mislove and Joël Ouaknine (cit. on p. 29).

Courser, Kenny Allen (2020). “Open Systems: A Double Categorical perspective.” PhD thesis. University of California Riverside (cit. on pp. 65, 68).

D’Osualdo, Emanuele, Azadeh Farzan, Philippa Gardner, and Julian Sutherland (2019). “TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs.” In: *CoRR* (cit. on p. 159).

da Rocha Pinto, Pedro, Thomas Dinsdale-Young, and Philippa Gardner (2014). “TaDA: A Logic for Time and Data Abstraction.” In: *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP’14)*. Ed. by Richard E. Jones (cit. on p. 33).

Day, Brian and Ross Street (1997). “Monoidal Bicategories and Hopf Algebroids.” In: *Advances in Mathematics* 129.1, pp. 99–157. ISSN: 0001-8708 (cit. on p. 81).

Dinsdale-Young, Thomas, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang (2013). “Views: compositional reasoning for concurrent programs.” In: *POPL* (cit. on p. 33).

Dodds, Mike, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal (2016). “Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic.” In: *ACM Trans. Program. Lang. Syst.* 38.2 (cit. on p. 36).

Ehresmann, Charles (1963). “Catégories structurées.” In: *Annales scientifiques de l’École Normale Supérieure* 80.4, pp. 349–426 (cit. on pp. 64, 66).

Filliâtre, Jean-Christophe and Andrei Paskevich (2013). “Why3 — Where Programs Meet Provers.” In: *Proceedings of the 22nd European Symposium on Programming*. Lecture Notes in Computer Science (cit. on p. 12).

Floyd, Robert (1967). “Mathematical Aspects of Computer Science.” In: Proceedings of Symposium on Applied Mathematics. American Mathematical Society. Chap. Assigning Meanings to Programs (cit. on p. 7).

Frumin, D., R. Krebbers, and L. Birkedal (2021). “Compositional Non-Interference for Fine-Grained Concurrent Programs.” In: *Proceedings of Security and Privacy* (cit. on p. 140).

Frumin, Dan, Robbert Krebbers, and Lars Birkedal (2018). “ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency.” In: LICS ’18 (cit. on p. 140).

Georges, Aïna Linn, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal (2021). “Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities.” In: *Proc. ACM Program. Lang.* 5.POPL (cit. on p. 24).

Giarrusso, Paolo G., Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers (2020). “Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris.” In: *Proc. ACM Program. Lang.* 4.ICFP (cit. on p. 26).

Gondelman, Léon, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal (2021). “Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic.” In: *Proc. ACM Program. Lang.* POPL (cit. on p. 24).

Gotsman, Alexey, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv (2007a). "Local Reasoning for Storable Locks and Threads." In: *Programming Languages and Systems*. Ed. by Zhong Shao (cit. on p. 37).

Gotsman, Alexey, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv (2007b). "Local Reasoning for Storable Locks and Threads." In: *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*. Ed. by Zhong Shao. Vol. 4807. Lecture Notes in Computer Science. Springer, pp. 19–37. DOI: `10.1007/978-3-540-76637-7\_3`. URL: `https://doi.org/10.1007/978-3-540-76637-7%5C_3` (cit. on p. 23).

Guéneau, Armaël, Arthur Charguéraud, and François Pottier (2018). "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification." In: *ESOP*. Ed. by Amal Ahmed (cit. on p. 159).

Hayman, Jonathan and Glynn Winskel (2008). "Independence and Concurrent Separation Logic." In: *LMCS* (cit. on p. 37).

Henzinger, Thomas A., Orna Kupferman, and Sriram K. Rajamani (1997). "Fair Simulation." In: *CONCUR '97*. Ed. by Antoni W. Mazurkiewicz and Józef Winkowski (cit. on p. 159).

Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (cit. on pp. 1, 2, 7, 9).

Iris Team (2021). *The Iris 3.4 Documentation*. https://plv.mpi-sws.org/iris/appendix-3.4.pdf (cit. on pp. 124, 127).

Ishtiaq, Samin S. and Peter W. O'Hearn (2001). "BI as an Assertion Language for Mutable Data Structures." In: *Conference Record of POPL 2001*. Ed. by Chris Hankin and Dave Schmidt. ACM (cit. on p. 17).

Jones, Cliff B. (1983). "Specification and Design of (Parallel) Programs." In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. A. Mason. North-Holland/IFIP, pp. 321–332 (cit. on p. 15).

Joyal, André, Ross Street, and Dominic Verity (1996). "Traced monoidal categories." In: *Mathematical Proceedings of the Cambridge Philosophical Society*, pp. 447–468 (cit. on p. 161).

Jung, Ralf, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer (2017). "RustBelt: Securing the Foundations of the Rust Programming Language." In: 2.POPL (cit. on pp. 8, 26, 36).

Jung, Ralf, Robbert Krebbers, Lars Birkedal, and Derek Dreyer (2016). "Higher-Order Ghost State." In: 51.9 (cit. on pp. 25, 36).

Jung, Ralf, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer (2018). "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." In: *Journal of Functional Programming* 28. DOI: `10.1017/S0956796818000151` (cit. on p. 36).

Jung, Ralf, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer (2015). “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning.” In: 50.1 (cit. on pp. 25, 36, 118).

Kaiser, Jan-Oliver, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis (2017). “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris.” In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Ed. by Peter Müller (cit. on p. 162).

Kang, Jeehoon, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer (2017). “A Promising Semantics for Relaxed-Memory Concurrency.” In: *SIGPLAN Not.* (Cit. on p. 162).

Kesten, Yonit, Zohar Manna, and Amir Pnueli (1994). “Temporal verification of simulation and refinement.” In: *A Decade of Concurrency Reflections and Perspectives*. Ed. by J. W. de Bakker, W. -P. de Roever, and G. Rozenberg (cit. on p. 159).

Krebbers, Robbert, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal (2017). “The Essence of Higher-Order Concurrent Separation Logic.” In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 696–723 (cit. on pp. 25, 36, 118).

Krebbers, Robbert, Amin Timany, and Lars Birkedal (2017). “Interactive Proofs in Higher-Order Concurrent Separation Logic.” In: *SIGPLAN Not.* (Cit. on pp. 26, 132, 140).

Krogh-Jespersen, Morten, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal (2020). “Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems.” In: *ESOP*. Ed. by Peter Müller (cit. on p. 24).

Lack, Stephen and Paweł Sobociński (2004). “Adhesive Categories.” In: *Foundations of Software Science and Computation Structures*. Ed. by Igor Walukiewicz (cit. on p. 103).

Lamport (1979). “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.” In: *IEEE Transactions on Computers* C-28.9, pp. 690–691 (cit. on p. 23).

Lamport, Leslie (1994). “The Temporal Logic of Actions.” In: *ACM Trans. Program. Lang. Syst.* (Cit. on pp. 7, 26, 159).

Lamport, Leslie (2001). “Paxos Made Simple.” In: *ACM SIGACT News (Distributed Computing Column)*, pp. 51–58 (cit. on p. 134).

Leino, Rustan (2010). “Dafny: An Automatic Program Verifier for Functional Correctness.” In: *16th International Conference, LPAR-16, Dakar, Senegal* (cit. on p. 12).

Leroy, Xavier (2009). “Formal verification of a realistic compiler.” In: *Communications of the ACM* 52.7, pp. 107–115 (cit. on p. 8).

Ley-Wild, Ruy and Aleksandar Nanevski (2013). “Subjective Auxiliary State for Coarse-Grained Concurrency.” In: *SIGPLAN Not.* 48.1, pp. 561–574 (cit. on p. 33).

Liang, Hongjin and Xinyu Feng (2016). “A Program Logic for Concurrent Objects under Fair Scheduling.” In: *SIGPLAN Not.* (Cit. on p. 159).

McCusker, Guy and David Pym (2007). "A Games Model of Bunched Implications." In: *Computer Science Logic*. Ed. by Jacques Duparc and Thomas A. Henzinger (cit. on p. 20).

Melliès, Paul-André (2019). "Categorical Combinatorics of Scheduling and Synchronization in Game Semantics." In: *Proc. ACM Program. Lang.* 3.POPL. ISSN: 2475-1421 (cit. on pp. 42, 54, 63, 81, 161).

Melliès, Paul-André and Léo Stefanesco (2017). "A Game Semantics for Concurrent Separation Logic." In: *MFPS* (cit. on p. 26).

Melliès, Paul-André 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* (cit. on p. 26).

Melliès, Paul-André and Léo Stefanesco (2020). "Concurrent Separation Logic Meets Template Games." In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20 (cit. on p. 26).

Melliès, Paul-André and Noam Zeilberger (2015). "Functors are Type Refinement Systems." In: *POPL 2015*. ACM, pp. 3–16 (cit. on p. 39).

Mellor-Crummey, John M. and Michael L. Scott (1991). "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." In: *ACM Trans. Comput. Syst.* 9.1 (cit. on p. 8).

Mével, Glen, Jacques-Henri Jourdan, and François Pottier (2019). "Time credits and time receipts in Iris." In: *ESOP*. Ed. by Luis Caires (cit. on p. 159).

Nakano, Hiroshi (2000). "A Modality for Recursion." In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000* (cit. on p. 121).

O'Hearn, Peter (2007). "Resources, Concurrency, and Local Reasoning." In: *TCS* 375 (cit. on p. 54).

O'Hearn, Peter W. (2004). "Resources, Concurrency and Local Reasoning." In: *CONCUR*. Ed. by Philippa Gardner and Nobuko Yoshida. Lecture Notes in Computer Science (cit. on p. 21).

O'Hearn, Peter W. and David J. Pym (1999). "The logic of bunched implications." In: *Bull. Symb. Log.* (Cit. on p. 19).

O'Hearn, Peter W., John C. Reynolds, and Hongseok Yang (2001). "Local Reasoning about Programs that Alter Data Structures." In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. Ed. by Laurent Fribourg. Lecture Notes in Computer Science (cit. on p. 17).

Owicki, Susan and David Gries (1978). "An Axiomatic Proof Technique for Parallel Programs." In: *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Ed. by David Gries. New York, NY: Springer New York, pp. 130–152.

ISBN: 978-1-4612-6315-9. DOI: 10.1007/978-1-4612-6315-9_12. URL: https://doi.org/10.1007/978-1-4612-6315-9_12 (cit. on p. 15).

Parkinson, Matthew, Richard Bornat, and Peter O'Hearn (2007). "Modular Verification of a Non-Blocking Stack." In: POPL '07. ISBN: 1595935754 (cit. on p. 23).

Pottier, François (2013). "Syntactic soundness proof of a type-and-capability system with hidden state." In: *J. Funct. Program.* (Cit. on p. 37).

Reddy, Uday S. (1996). "Global State Considered Unnecessary: An Introduction to Object-Based Semantics." In: *LISP Symb. Comput.* 9.1, pp. 7–76 (cit. on p. 40).

Reynolds, John C. (1997). "The Essence of Algol." In: *Algol-like Languages*. Ed. by Peter W. O'Hearn and Robert D. Tennent. Boston, MA: Birkhäuser Boston (cit. on p. 87).

Reynolds, John C. (2000). "Intuitionistic Reasoning about Shared Mutable Data Structure." In: *Millennial Perspectives in Computer Science*. Palgrave (cit. on p. 17).

Reynolds, John C. (2002). "Separation Logic: A Logic for Shared Mutable Data Structures." In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings* (cit. on p. 17).

Sergey, Ilya, James R. Wilcox, and Zachary Tatlock (2017). "Programming and Proving with Distributed Protocols." In: 2.POPL (cit. on p. 24).

Shulman, Michael (2008). "Framed bicategories and monoidal fibrations." In: *Theory and Applications of Categories [electronic only]* 20, pp. 650–738 (cit. on p. 67).

Spies, Simon, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal (2021). "Transfinite Iris: resolving an existential dilemma of step-indexed separation logic." In: *PLDI*. Ed. by Stephen N. Freund and Eran Yahav (cit. on pp. 139, 163).

Svendsen, Kasper and Lars Birkedal (2014). "Impredicative Concurrent Abstract Predicates." In: *Programming Languages and Systems*. Ed. by Zhong Shao (cit. on p. 36).

Tassarotti, Joseph and Robert Harper (2019). "A Separation Logic for Concurrent Randomized Programs." In: 3.POPL (cit. on p. 140).

Tassarotti, Joseph, Ralf Jung, and Robert Harper (2017). "A Higher-Order Logic for Concurrent Termination-Preserving Refinement." In: *ESOP*. Ed. by Hongseok Yang. Lecture Notes in Computer Science (cit. on pp. 139, 158).

Timany, Amin, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal (2021). *Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic*. arXiv: 2109.07863 [cs.PL] (cit. on pp. 27, 134).

Timany, Amin, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal (2017). "A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST." In: *Proc. ACM Program. Lang.* POPL (cit. on pp. 26, 140).

Turing, Alan (1949). "Checking a Large Routine." In: *Report of a Conference on High Speed Automatic Calculating Machines* (cit. on p. 7).

Turon, Aaron, Viktor Vafeiadis, and Derek Dreyer (2014). "GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation." In: (cit. on p. 24).

Vafeiadis, Viktor (2011). "Concurrent Separation Logic and Operational Semantics." In: *ENTCS* 276 (cit. on pp. 32–34, 41, 54).

Varacca, Daniele and Glynn Winskel (2006). "Distributing probability over non-determinism." In: *Math. Struct. Comput. Sci.* (Cit. on p. 140).

Xia, Li-yao, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic (2020). "Interaction trees: representing recursive and impure programs in Coq." In: *Proc. ACM Program. Lang.* 4.POPL, 51:1–51:32 (cit. on p. 33).

Zakowski, Yannick, Paul He, Chung-Kil Hur, and Steve Zdancewic (2020). "An equational theory for weak bisimulation via generalized parameterized coinduction." In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP*, pp. 71–84 (cit. on p. 33).