# A mechanized program logic for concurrent programs with the weak memory model of Multicore OCaml

Glen Mével

Université Paris Cité

École doctorale de sciences mathématiques de Paris-Centre (ED 386)

Inria

# A mechanized program logic for concurrent programs with the weak memory model of Multicore OCaml

**par Glen Mével**

Thèse de doctorat d'informatique

*dirigée par* François Pottier

*présentée et soutenue publiquement le 14 décembre 2022*

*devant un jury composé de :*

| | | |
|---|---|---|
| Aleksandar Nanevski | associate research professor, IMDEA, Madrid | (rapporteur) |
| Mark Batty | professor, University of Kent | (rapporteur) |
| Azalea Raad | assistant professor, Imperial College, Londres | (examinatrice) |
| Stephen Dolan | software developer, Jane Street, Londres | (examinateur) |
| Ralf Treinen | professeur des universités, Université Paris-Cité | (examinateur) |
| François Pottier | directeur de recherches, Inria, Paris | (directeur) |
| Jacques-Henri Jourdan | chargé de recherches, CNRS & Laboratoire Méthodes Formelles, Gif-sur-Yvette | (co-encadrant, membre invité) |

# Description en français

## Titre

*Une logique de programmes mécanisée pour les programmes concurrents dans le modèle mémoire faible de Multicore OCaml*

## Résumé

Multicore OCaml ajoute au langage de programmation OCaml le support de la concurrence à mémoire partagée. Ce langage étendu obéit à un modèle faible de la mémoire dont une sémantique opérationnelle a été publiée. On peut alors se demander de quels principes de raisonnement on dispose pour s'assurer de la correction d'un programme écrit en Multicore OCaml.

Pour y répondre, on instancie Iris, un descendant moderne de la Logique de Séparation Concurrente, pour Multicore OCaml. On obtient une logique de programme de bas niveau, dont les règles de raisonnement exposent les détails techniques du modèle mémoire. Au-dessus de cette logique, on construit Cosmo, une logique de plus haut niveau dans laquelle on jouit de règles plus simples, au prix d'une légère limitation concernant les programmes qu'on peut vérifier. Cosmo offre des raisonnements naturels à propos des variables non-atomiques si elles sont exemptes de courses de données ; à propos des variables atomiques ; et à propos de la synchronisation en deux temps (acquisition/relâchement, ou *release/acquire*) que ces dernières réalisent. La synchronisation entre fils d'exécution est transcrite de façon concise par un mécanisme de vues de la mémoire, qui permet de s'abstraire du modèle sous-jacent.

On illustre l'emploi de cette logique de programme par plusieurs études de cas. On vérifie plusieurs implémentations de verrous vis-à-vis d'une spécification classique. On spécifie et vérifie également une structure de données concurrente élaborée et réaliste : une file concurrente multi-écrivains et multi-lecteurs. Dans chacun de ces cas, la spécification de la structure de données considérée décrit son comportement de synchronisation indépendamment de son implémentation et du modèle mémoire sur laquelle cette implémentation repose. On parvient à ce résultat par l'emploi combiné de vues et de « triplets logiquement atomiques ». Ainsi, cette approche de la vérification est modulaire vis-à-vis du modèle mémoire : une application qui utilise ces structure de données comme seul moyen de synchronisation peut être vérifiée sans aucune connaissance du modèle mémoire de Multicore OCaml.

## Mots-clé

programmation, concurrence, mémoire faible, vérification, logique de séparation, OCaml

# Description in English

## Title

*A mechanized program logic for concurrent programs with the weak memory model of Multicore OCaml*

## Abstract

Multicore OCaml extends OCaml with support for shared-memory concurrency. It is equipped with a weak memory model, for which an operational semantics has been published. This begs the question: what reasoning rules can one rely upon while writing or verifying Multicore OCaml code?

To answer it, we instantiate Iris, a modern descendant of Concurrent Separation Logic, for Multicore OCaml. This yields a low-level program logic whose reasoning rules expose the details of the memory model. On top of it, we build a higher-level logic, Cosmo, which trades off some expressive power in return for a simple set of reasoning rules that allow accessing non-atomic locations in a data-race-free manner, exploiting the sequentially-consistent behavior of atomic locations, and exploiting the release/acquire behavior of atomic locations. Cosmo allows both low-level reasoning, where the details of the Multicore OCaml memory model are apparent, and high-level reasoning, which is independent of this memory model.

We illustrate this claim via a number of case studies. We verify several implementations of locks with respect to a classic specification. We also specify and verify a realistic, sophisticated concurrent data structure: namely, a multiple-producer multiple-consumer concurrent queue. In each case, the specification describes the memory behavior of the data structure independently of its implementation—and of the underlying memory model. We achieve this through the joint use of so-called "logically atomic triples" and of Cosmo's views. Thus, Cosmo's approach to verification is modular with respect to the memory model: a coarse-grained application that uses these data structures as the sole means of synchronization can be verified without any knowledge of the weak memory model.

## Keywords

programming, concurrency, weak memory, verification, separation logic, OCaml

# Contents

# List of Figures

Il fait beau voir Jean de Paris
Avec ses douze méharis.
Il fait beau voir Jean de Bordeaux
Avec ses quatorze chameaux.
Mais j'aime mieux Jean de Madère
Avec ses quatre dromadaires.

Bien loin d'ici Jean de Madère
Voyage avec Robert Macaire
Et leur ami Apollinaire
Qui, de son temps, a su bien faire
Avec les quatre dromadaires.

---

Robert Desnos, *Chantefables*

# Résumé substantiel dans la langue de Desnos

## Vérification de programmes concurrents en mémoire faible

Ces vingt dernières années, les progrès technologiques concernant les ordinateurs multicœurs ont accentué le besoin d'outils puissants permettant d'écrire du code multicœur efficace et sûr. Cela comprend des langages de programmation bien conçus, leurs compilateurs, des bibliothèques concurrentes efficaces, ainsi que des logiques de programme expressives, qui permettent de prouver la correction desdites bibliothèques et du code qui les utilise. Quoique certains outils de vérification existent déjà, de nombreuses questions de recherche subsistent quant à l'emploi de ces outils pour spécifier et vérifier, modulairement, des bibliothèques réalistes de concurrence à grains fins.

Dans la plupart des langages de programmation multicœurs, les fils d'exécution communiquent via une mémoire partagée. Pour des raisons de performance cependant, les accès concurrents à cette mémoire partagée obéissent rarement à une sémantique *séquentiellement cohérente* (Lamport, 1979), selon laquelle un programme se comporterait comme un entrelacement des actions de ses fils, qui auraient tous la même vue de la mémoire à chaque instant. Pour bénéficier d'optimisations agressives effectuées par le matériel et permettre au compilateur d'en effectuer également, de nombreux langages parmi les plus utilisés adoptent des modèles plus relâchés de la mémoire, dit *faibles*, dont la description est subtile. Par exemple C, C++ et Rust suivent le modèle mémoire C11 (Batty et al., 2011; Lahav et al., 2017) ; Java dispose également d'un modèle mémoire faible (Manson et al., 2005; Lochbihler, 2012; Bender and Palsberg, 2019).

Multicore OCaml (Dolan et al., 2018, 2020) s'inscrit dans cette tendance. Il s'agit d'une extension du langage au chameau (Leroy et al., 2019) auquel elle ajoute un support de la concurrence à mémoire partagée. Dolan et al. donnent une sémantique à Multicore OCaml, et en particulier un modèle mémoire. Ce modèle mémoire est faible. Vu la subtilité de tels modèles, le besoin se fait sentir d'un ensemble de règles de raisonnement — autrement dit, d'une logique de programme — pour comprendre et vérifier du code concurrent écrit dans ce langage. Une telle logique de programme doit permettre un raisonnement compositionnel : une fois définie la spécification d'une bibliothèque $B$, la vérification de $B$ doit pouvoir se faire indépendamment de la vérification d'un code qui utilise $B$. Ainsi, un programmeur peut utiliser $B$ en boîte noire, sans connaissance sur son implémentation. De plus, la majorité du code qu'on écrit reste séquentiel et, lorsqu'on écrit du code concurrent, c'est souvent à travers l'emploi de structures de données concurrentes implémentées par des experts plutôt qu'en utilisant directement les primitives du langage. Il est donc souhaitable que la logique permette d'ignorer les détails du modèle mémoire de Multi-

core OCaml quand ceux-ci ne sont pas pertinents. Un programme séquentiel doit pouvoir être vérifié, idéalement, comme si le modèle mémoire était séquentiellement cohérent. Le comportement d'une structure de données concurrente doit pouvoir être spécifié indépendamment des primitives qui sont employées pour l'implémenter.

Toujours ces vingt dernières années, afin de spécifier et de vérifier formellement des programmes concurrents à mémoire partagée, se sont développées un grand nombre de *logiques de séparation concurrentes* dans le sillage des travaux de Brookes (2004) et O'Hearn (2007). L'idée-clé de la logique de séparation est qu'une assertion de la logique ne se contente pas d'affirmer un fait mathématique : elle reflète la possession d'une *ressource*, dont les zones de la mémoire sont l'archétype. Dans sa forme la plus élémentaire, la possession d'une ressource est unique. Cette idée, déjà fructueuse pour la vérification de programmes impératifs séquentiels, se révèle particulièrement précieuse pour la vérification de programmes concurrents. Des logiques de séparation d'ordre supérieur et de plus en plus expressives ont émergé (Brookes and O'Hearn, 2016), à l'instar d'Iris (Jung et al., 2015, 2018b). Iris est mécanisé et prouvé sûr dans Coq, supporte la concurrence à grains fins, et peut décrire des protocoles concurrents complexes grâce à deux mécanismes essentiels : l'*état fantôme* et les *invariants*. L'état fantôme en Iris permet de réaliser la notion de ressource et est très expressif, puisque sa structure peut être n'importe quel monoïde commutatif partiel, au choix. Iris est paramétrable par le langage de programmation mais suppose toutefois une sémantique séquentiellement cohérente. Une nouvelle génération de logiques lève cette restriction pour divers modèles faibles : iCAP-TSO (Sieczkowski et al., 2015), GPS (Turon et al., 2014), iGPS (Kaiser et al., 2017) et iRC11 (Dang et al., 2020) supportent des fragments du modèle C11.

Dans cette thèse, nous proposons une logique de séparation pour Multicore OCaml. Comme iGPS et iRC11, le travail que nous présentons est construit sur Iris. Toutes nos contributions (implémentation des logiques BaseCosmo et Cosmo, preuve de leur sûreté, et études de cas) sont mécanisées en Coq avec l'infrastructure Iris. Nos preuves sont consultables dans notre dépôt (Mével et al., 2021).

## Le modèle mémoire de Multicore OCaml

Par rapport à d'autres modèles mémoire actuels, celui de Multicore OCaml est relativement simple. Il distingue deux types de cases mémoire, *atomique* et *non-atomique*. L'ensemble des comportements possibles d'un programme est toujours bien défini. Les courses de données sur des cases non-atomiques sont permises et ont un « effet limité dans l'espace et dans le temps » (Dolan et al., 2018). Ceci différencie le modèle de Multicore OCaml du modèle C11, dans lequel toute course de données est un « comportement indéfini » qu'une logique de programme sûre doit donc bannir.

Dolan et al. décrivent le modèle de Multicore OCaml au moyen d'une sémantique opérationnelle (Figure 2.4), dans laquelle l'exécution d'un programme apparaît bel et bien comme un entrelacement des exécutions de chaque fil. Cependant, les interactions de ces fils avec la mémoire partagée (Figure 2.2) ont un comportement plus complexe que celui qu'on aurait dans l'habituel cadre séquentiellement cohérent. La description de ce comportement fait intervenir un *historique* pour chaque case non-atomique de la mémoire partagée, et des *vues* de cette mémoire partagée (Figure 2.1). L'historique $h$ d'une case non-atomique est l'ensemble des événements d'écriture qui ont eu lieu sur cette case, ordonnés par des horodatages $t$ (dont les valeurs sont

arbitraires et propres à chaque case, seul l'ordre étant significatif). Chaque fil d'exécution $\langle e, \mathcal{W} \rangle$ possède sa propre vue $\mathcal{W}$ de la mémoire partagée, qui donne, pour chaque case non-atomique $a$, l'horodatage $\mathcal{W}(a)$ du plus récent événement d'écriture dont ce fil a connaissance. Cette vue restreint les comportements possibles d'un accès à une case non-atomique.

Une lecture de la case non-atomique $a$ (BASE-NA-READ dans la Figure 2.2) ne peut pas lire un événement d'écriture obsolète, et doit donc lire un événement dont l'horodatage $t$ est au moins $\mathcal{W}(a)$, le plus récent horodatage connu du fil courant. Les deux cas de figure $\mathcal{W}(a) = t$ et $\mathcal{W}(a) < t$ sont permis : le second cas autorise un fil à lire un événement d'écriture dont il n'a pas encore été officiellement informé. Dans ce cas, le fil ne met pas à jour sa vue courante.

Une écriture dans la case non-atomique $a$ (BASE-NA-WRITE dans la Figure 2.2) ne peut pas produire un événement d'écriture obsolète, et doit donc utiliser un horodatage $t$ strictement supérieur à $\mathcal{W}(a)$... mais pas nécessairement supérieur à $\max(\mathrm{dom}(h))$ ; autrement dit, si le fil n'a pas connaissance du plus récent événement d'écriture, le comportement n'est pas déterministe, car le nouvel événement d'écriture peut s'intercaler soit avant soit après les événements dont le fil n'a pas encore connaissance (ce cas de figure correspond à une course de données sur $a$). L'historique $h$ de la case et la vue $\mathcal{W}$ du fil sont mis à jour pour prendre en compte ce nouvel événement d'écriture. Cependant, la vue des autres fils reste inchangée : les autres fils ne sont pas informés de ce nouvel événement d'écriture.

Contrairement aux cases non-atomiques, les cases atomiques ont bien une valeur unique, vue à l'identique par tous les fils. Cependant, leur fonction ne s'arrête pas là : les accès aux cases atomiques effectuent de la *synchronisation*. Cela se traduit par le fait qu'une case atomique $A$, en plus de sa valeur $v$, mémorise une vue $\mathcal{V}$. Un fil qui écrit dans $A$ (BASE-AT-WRITE dans la Figure 2.2) déverse sa connaissance $\mathcal{W}$ dans $A$. Par la suite, un fil qui lit cette même case $A$ (BASE-AT-READ dans la Figure 2.2) reçoit la connaissance emmagasinée dans $A$ et l'ajoute à sa propre vue. Les cases atomiques sont ainsi le moyen primitif en Multicore OCaml par lequel les fils d'exécution s'échangent la connaissance de certains événements d'écriture.

## BaseCosmo, une logique bas niveau pour Multicore OCaml

Dans cette thèse, on s'appuie sur la sémantique opérationnelle de Dolan et al. pour construire une première logique de programme sûre et mécanisée dans Coq, baptisée BaseCosmo, qu'on obtient en instanciant Iris. En substance, on fournit à Iris deux ingrédients (Jung et al., 2018b, §7.3).

- La sémantique opérationnelle de notre langage, qui se présente comme une relation de réduction locale à un fil, de la forme suivante :

  $$\sigma; \langle e, \mathcal{W} \rangle \Longrightarrow \sigma'; \langle e', \mathcal{W}' \rangle, p$$

  Cette relation décrit, partant d'un certain état partagé $\sigma$, la prochaine étape de calcul que peut effectuer un fil $\langle e, \mathcal{W} \rangle$ et la façon dont cette étape modifie l'état partagé et le fil considéré, créant éventuellement de nouveaux fils $p$. Dans un modèle séquentiellement cohérent, un fil est simplement représenté par son code source (c'est-à-dire une expression du langage) restant à évaluer ; cependant, dans Multicore OCaml, un fil est en fait un couple $\langle e, \mathcal{W} \rangle$ où $e$ est une expression restant à évaluer et $\mathcal{W}$ est la vue courante du fil.

- Une « interprétation d'état », c'est-à-dire un prédicat $S : \textsc{Store} \to \textsc{iProp}$ (où Store est le type représentant la mémoire partagée, défini dans la Figure 2.1, et iProp est le type des assertions d'Iris). Cette interprétation d'état exprime un invariant global sur l'état partagé. En l'occurrence, on a besoin d'exprimer un invariant concernant les vues emmagasinées dans les cases atomiques : ces vues doivent être *valides*, c'est-à-dire que l'horodatage qu'elles donnent pour chaque case non-atomique $a$ ne doit pas dépasser le plus récent événement d'écriture dans l'historique de $a$. L'interprétation d'état $S(\sigma)$ traduit l'état partagé $\sigma : \textsc{Store}$ en un état fantôme dont on a soigneusement choisi la structure afin de pouvoir séparer chacune des cases de la mémoire en une ressource individuelle, représentée par une assertion *points-to* qui satisfait certaines propriétés souhaitées.

On obtient une logique de séparation qui, outre les fonctionnalités de base d'Iris (§2.3.2), inclut une assertion $\{P\}\ \langle e, \mathcal{W}\rangle\ \{\Phi\}$. Celle-ci représente le triplet de Hoare à propos du fil $\langle e, \mathcal{W}\rangle$ ayant pour précondition $P : \textsc{iProp}$ et pour postcondition $\Phi : \textsc{Val} \times \textsc{View} \to \textsc{iProp}$. Un tel triplet exprime la correction partielle d'un programme vis-à-vis d'une spécification. Ceci se formalise dans un théorème d'adéquation qui dérive de celui fourni par Iris (Jung et al., 2018b, §6.4, §7.4) : si un programme $e$ satisfait un triplet $\{\mathsf{True}\}\ \langle e, \varnothing\rangle\ \{\Phi\}$ alors, d'une part, ce programme est sûr, c'est-à-dire que son exécution ne mènera jamais à une configuration bloquée ; d'autre part, ses éventuelles configurations finales $\langle v', \mathcal{W}'\rangle$ satisferont la postcondition $\Phi$. Les triplets satisfont également des règles de raisonnement usuelles et indépendantes du langage (Figure 2.5 et §2.3.2), telles que la règle de conséquence (CONSEQUENCE), la règle de séquence (BIND), la règle de passage au contexte (FRAME) et la règle d'ouverture d'un invariant (HoareInv).

Pour compléter notre logique de programme pour Multicore OCaml, on définit alors, au sein de la logique obtenue, des assertions spécifiques au langage. Elles revêtent la forme traditionnelle d'assertions *points-to* avec des permissions fractionnaires (Boyland, 2003; Bornat et al., 2005), à la différence que les cases ne contiennent pas une simple « valeur ».

- Une assertion $q \cdot a \rightsquigarrow_{\mathrm{na}} h$ représente la possession d'une fraction $q \in \mathbb{Q} \cap (0, 1]$ de la case non-atomique $a$, dont l'historique est $h$. La fraction 1 représente une possession unique.

- Une assertion $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle$ représente la possession d'une fraction $q$ de la case atomique $A$, dont la valeur est $v$ et qui a emmagasiné la vue $\mathcal{V}$ (au moins). La fraction 1 représente une possession unique.

Ces assertions satisfont entre autres les propriétés suivantes :

$$\frac{q \cdot a \rightsquigarrow_{\mathrm{na}} h \quad * \quad S(\sigma)}{\sigma(a) = h \quad * \quad 0 < q \leq 1} \qquad \frac{q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \quad * \quad S(\sigma)}{\exists \mathcal{W}.\, \sigma(A) = \langle v, \mathcal{W}\rangle \quad * \quad \mathcal{V} \sqsubseteq \mathcal{W} \ * \ 0 < q \leq 1}$$

$$q_1 \cdot a \rightsquigarrow_{\mathrm{na}} h_1 \ * \ q_2 \cdot a \rightsquigarrow_{\mathrm{na}} h_2 \ \dashv\vdash \ (q_1 + q_2) \cdot a \rightsquigarrow_{\mathrm{na}} h_1 \ * \ h_1 = h_2$$

$$q_1 \cdot A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}_1\rangle \ * \ q_2 \cdot A \rightsquigarrow_{\mathrm{at}} \langle v_2, \mathcal{V}_2\rangle \ \dashv\vdash \ (q_1 + q_2) \cdot A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}_1 \sqcup \mathcal{V}_2\rangle \ * \ v_1 = v_2$$

Pour des raisons techniques, on a aussi besoin de définir une assertion $\mathsf{valid}\,\mathcal{V}$ qui affirme que la vue $\mathcal{V}$ est valide vis-à-vis de l'état partagé $\sigma$, via l'invariant global sur les vues, évoqué précédemment et garanti par l'interprétation $S(\sigma)$.

Enfin, on prouve un triplet de Hoare pour chaque étape élémentaire de calcul du langage, et en particulier pour chaque opération d'accès à la mémoire (Figure 3.2). Les triplets pour ces opérations comportent en précondition et en postcondition la possession (unique ou fractionnaire) de la case mémoire concernée. Ces triplets reflètent fidèlement la sémantique opérationnelle et le modèle mémoire de Multicore OCaml.

On obtient ainsi un ensemble de règles de raisonnement permettant de vérifier un programme Multicore OCaml étape par étape. Utiliser cette logique de programme plutôt que de travailler directement avec la sémantique opérationnelle offre de clairs avantages.

- On dispose d'un langage de spécification riche, qui contient la logique hôte (celle de Coq) et une infrastructure d'état fantôme expressive.
- Un système de preuve sûr — et mécanisé — nous permet de prouver la sûreté d'un programme et sa correction fonctionnelle partielle.
- Grâce à la logique de séparation concurrente, les spécifications sont réduites à leur empreinte minimale et leurs preuves sont composables : chaque morceau de programme peut ainsi être spécifié et vérifié en isolation. Il n'y a pas besoin de considérer les interférences causées par les autres morceaux de programme. En particulier, lorsqu'on vérifie un programme avec plusieurs fils d'exécution, il n'est pas nécessaire d'envisager les très nombreux entrelacements possibles des actions de chaque fil.

Cependant, cette logique de programme expose les moindres détails de Multicore OCaml. Elle est par conséquent de bas niveau et assez fastidieuse à utiliser. Plusieurs critiques peuvent lui être faites.

- La vue $\mathcal{W}$ du fil courant doit être explicitée et il faut constamment s'assurer de sa validité, condition de bord qui apparaît dans la plupart des préconditions et postconditions de nos règles de raisonnement (Figure 3.2). On aimerait rendre tout cela implicite.
- L'assertion $q \cdot a \rightsquigarrow_{\mathrm{na}} h$ qui représente une case non-atomique $a$ expose le fait que cette case contienne un historique $h$ plutôt qu'une valeur unique. Les règles BASE-NA-READ et BASE-NA-WRITE paraphrasent la sémantique opérationnelle dans toute sa technicité, qui fait intervenir la notion d'horodatage. Il reste ainsi difficile de raisonner à propos d'une case non-atomique. Pourtant, en l'absence de courses de données sur cette case, on aimerait pouvoir raisonner plus naturellement, comme si la case contenait une valeur unique.

## Cosmo, une logique de plus haut niveau

On exauce maintenant ces deux souhaits en construisant, sur notre logique de bas niveau Base-Cosmo, une logique de plus haut niveau baptisée Cosmo. Dans Cosmo, l'assertion qui représente une case non-atomique revêt l'apparence traditionnelle $q \cdot a \rightsquigarrow_{\mathrm{na}} v$ comme dans un cadre séquentiellement cohérent. Cette assertion garantit que lire $a$ renverra la valeur $v$. L'assertion signifie donc, intuitivement, que la case $a$ contient la valeur $v$ *aux yeux du fil courant*. En termes plus techniques, l'assertion garantit que le plus récent événement d'écriture dans $a$ est une écriture de la valeur $v$ et que le fil courant connaît cette écriture.

En cas de course de données sur une case non-atomique, l'hypothèse sur la vue courante n'est pas vérifiée ; ainsi, représenter une case non-atomique par cette assertion simplifiée empêche de

raisonner à propos de certains programmes. Nous croyons raisonnable de supposer que la plupart des programmes (corrects) sont exempts de telles courses de données, de sorte que cette perte de généralité constitue rarement un problème.

La signification de l'assertion $q \cdot a \rightsquigarrow_{\mathrm{na}} v$ est relative au fil courant. Plus précisément, elle dépend de sa vue, puisqu'elle affirme que ladite vue contient un certain événement d'écriture. Pour donner un sens à cette assertion simplifiée, les assertions de notre logique doivent donc être paramétrées par une vue. La vue à laquelle s'applique une assertion Cosmo est la vue du possesseur de cette assertion. Ce changement de perspective résout les deux objections que nous avons formulées plus haut.

Un utilisateur de Cosmo n'a pas besoin de savoir comment cette logique est construite au-dessus de BaseCosmo : ni l'implémentation des vues, ni celle des assertions de Cosmo ; les règles de raisonnement de Cosmo peuvent lui être présentées directement. Toutefois, une preuve dans Cosmo donne une preuve dans BaseCosmo, il est donc aisé de combiner des efforts de preuve conduits dans les deux logiques, si jamais Cosmo n'était pas assez expressif.

## Subjectivité

La construction de Cosmo par-dessus BaseCosmo est montrée dans la Figure 4.1. Comme suggéré, une assertion de Cosmo est un prédicat (de type VPROP) qui, à une vue, associe une assertion de BaseCosmo. Entre autres pour garantir la sûreté de la règle de passage au contexte (FRAME) dans Cosmo, on impose (Kaiser et al., 2017; Dang et al., 2020) que ce prédicat soit *croissant*, les vues étant ordonnées point-à-point $\sqsubseteq$ par l'ordre sur les horodatages (une vue plus grande est plus récente, ou mieux informée, et contraint davantage les comportements possibles lors d'un accès non-atomique) et les assertions de BaseCosmo par l'implication. On porte dans Cosmo tous les connecteurs logiques de BaseCosmo. On a également un moyen simple de convertir une assertion $P$ de BaseCosmo en une assertion $\lceil P \rceil$ de Cosmo, qui consiste à ignorer la vue donnée en paramètre. Ainsi, Cosmo contient BaseCosmo. Une assertion de la forme $\lceil P \rceil$ est dite *objective*, en ce que sa validité ne dépend pas de la vue courante.

L'apport de Cosmo est justement de permettre de se référer à la vue courante. Pour cela, on introduit une assertion de la forme $\uparrow \mathcal{V}$ qui affirme simplement que la vue courante, qui reste implicite, contient la vue $\mathcal{V}$. Cette assertion est donc éminemment *subjective*. On l'utilise par exemple pour définir notre assertion simplifiée $q \cdot a \rightsquigarrow_{\mathrm{na}} v$ (Figure 4.1). En fait, plus généralement, les assertions de la forme $\uparrow \mathcal{V}$ suffisent à exprimer n'importe quelle assertion subjective. En effet (grâce à la croissance des assertions Cosmo vis-à-vis de la vue courante), pour affirmer un fait à propos de la vue courante, il suffit d'en capturer une sous-vue, c'est-à-dire un $\mathcal{V}$ tel que $\uparrow \mathcal{V}$, et de formuler le fait désiré à propos de $\mathcal{V}$ au moyen d'une assertion objective de BaseCosmo. Cette idée est formalisée dans la règle de raisonnement SPLIT-SUBJECTIVE-OBJECTIVE (Figure 4.2). Pour écrire cette règle, on introduit une assertion de la forme $P \mathbin{@} \mathcal{V}$ qui, étant donnée une assertion arbitraire $P$ de Cosmo et une vue $\mathcal{V}$, exprime l'assertion $P$ dans laquelle la vue courante a été remplacée par la vue $\mathcal{V}$ ; cette assertion ne dépend plus de la vue courante et est donc objective (autrement dit, c'est une assertion de BaseCosmo).

En Cosmo, plutôt que de cacher les vues, on fait le choix d'en faire une notion de base de la logique. On peut alors se demander ce qu'on y gagne par rapport à BaseCosmo. Premièrement, en tant qu'utilisateur de la logique, il n'est plus nécessaire de savoir qu'une vue est une fonction des cases non-atomiques vers les horodatages ; la notion d'horodatage est escamotée. Les vues

peuvent être rendues abstraites du point de vue de l'utilisateur, qui doit seulement savoir qu'elles sont munies d'une relation d'ordre — en fait, d'une structure de demi-treillis borné. Deuxièmement, grâce à la croissance des assertions vis-à-vis de la vue courante, on n'a jamais besoin de garder trace de la vue *exacte* d'un fil, ou de la vue exacte qui est emmagasinée dans une case atomique : on peut se contenter d'une sous-vue qui contient la connaissance pertinente. On peut rapprocher cette idée de celle à l'origine de la logique de séparation : en logique de séparation, grâce à la règle FRAME, on peut limiter l'empreinte mémoire d'une spécification à la portion pertinente de la mémoire, et ne pas mentionner le reste, ce qui offre un cadre plus modulaire pour la vérification. De même en Cosmo, on n'a pas à se soucier des événements d'écriture hors des zones de la mémoire qui nous intéressent. Dans le cas le plus extrême, lorsqu'on n'est pas intéressé par la synchronisation entre fils (par exemple lorsqu'on vérifie du code séquentiel), on peut se contenter de vues vides $\varnothing$, ce qui revient à omettre complètement les vues, et donne comme un cas particulier de Cosmo une logique de programme identique à celle qu'on aurait dans un cadre séquentiellement cohérent.

### Preuve de programme en Cosmo

Pour pouvoir spécifier et vérifier des programmes dans cette logique de séparation avec vues, on porte (§4.3) les triplets de BaseCosmo en des triplets $\{P\}\, e\, \{\Phi\}$, où un fil est représenté simplement par son expression $e$ et où les précondition et postcondition sont des assertions de Cosmo ($P : \textsc{vProp}$ et $\Phi : \textsc{Val} \to \textsc{vProp}$). La définition rend implicite la vue courante $\mathcal{W}$ du fil et les conditions de bord concernant sa validité. Les triplets dans Cosmo satisfont encore un théorème d'adéquation (§4.4). On peut ensuite porter dans Cosmo les spécifications de chaque opération du langage qu'on avait établies dans BaseCosmo (Figures 3.1 et 3.2). La Figure 4.3 montre les règles de raisonnement obtenues. La vue du fil avant exécution ($\mathcal{W}$) et après ($\mathcal{W}'$) n'est plus explicitée, ni les conditions de validité associées. Une assertion de la forme $\uparrow \mathcal{V}$ est employée en précondition quand le fil déverse sa vue courante dans une case atomique, ou en postcondition quand le fil ajoute à sa vue courante de nouvelles informations obtenues en lisant une case atomique.

Les opérations sur les cases non-atomiques (NA-READ et NA-WRITE) sont spécifiées avec l'assertion simplifiée $q \cdot a \rightsquigarrow_{\mathrm{na}} v$ plutôt qu'avec l'assertion d'historique $\lceil q \cdot a \rightsquigarrow_{\mathrm{na}} h \rceil$. Ces règles de raisonnement sont donc moins générales que celles de BaseCosmo — elles ne permettent pas de raisonner à propos de courses de données sur les cases non-atomiques — mais, en contrepartie, semblent bien plus naturelle : en effet, elles sont identiques en apparence aux règles d'accès qu'on aurait dans un modèle séquentiellement cohérent. La différence réside dans la nature de l'assertion *points-to* : en Cosmo, l'assertion qui représente une case non-atomique est subjective.

Comme mentionné plus tôt, les règles de raisonnement se simplifient quand on n'est pas intéressé par la synchronisation : dans les règles gouvernant les opérations sur les cases atomiques (AT-READ, AT-WRITE, CAS-FAILURE, CAS-SUCCESS), on peut prendre pour chacune des vues mentionnées ($\mathcal{V}$ et $\mathcal{V}'$) la vue vide $\varnothing$, et l'on obtient alors les règles simplifiées montrées dans la Figure 4.3c. Ces règles sont exactement celles qui gouvernent une case mémoire séquentiellement cohérente, d'autant plus que l'assertion qui représente une case atomique, elle, est bien objective.

La synchronisation effectuée par les accès atomiques, telle que décrite par les règles non simplifiées, est de type relâchement/acquisition (*release/acquire*). Elle permet la transmission d'une vue depuis un premier fil vers un second. En exploitant ce mécanisme conjointement à

la règle SPLIT-SUBJECTIVE-OBJECTIVE et à l'utilisation d'un invariant Iris, on peut réaliser le transfert d'une assertion $P$ arbitraire, même lorsque l'empreinte de $P$ contient des cases non-atomiques : l'invariant Iris transfère le support objectif $P \mathbin{@} \mathcal{V}$ de l'assertion $P$, tandis que la case atomique transmet la connaissance $\uparrow \mathcal{V}$ des événements d'écriture utiles. Il s'agit d'un mécanisme-clé, qu'on utilisera dans toutes nos études de cas. Il faut noter que lorsque $P$ est subjectif, on ne peut pas se contenter d'un invariant Iris pour effectuer le transfert : en effet, une ressource mise dans un invariant Iris est rendue disponible pour tous les fils sans distinction, et ne peut donc pas dépendre de la vue d'un fil donné ; dans Cosmo les invariants Iris sont donc par nature restreints aux assertions objectives. On retrouve donc dans la logique la nécessité d'effectuer une synchronisation.

# Études de cas

## Verrous et exclusion mutuelle

Pour éprouver notre logique de programme Cosmo, cette thèse s'attache ensuite à vérifier quelques structures de données concurrentes simples : un verrou tournant (§5.2), un verrou à ticket (§5.3), les algorithmes d'exclusion mutuelle de Dekker (§5.5) et de Peterson (§5.6).

Chacune de ces études de cas met en œuvre de façon cruciale la méthode décrite précédemment pour transférer une assertion au moyen de la règle SPLIT-SUBJECTIVE-OBJECTIVE. Les preuves de ces structures de données en Cosmo calquent les preuves qui auraient été faites dans un cadre séquentiellement cohérent, auxquelles on ajoute des vues (abstraites) qui matérialisent l'information transmise d'un fil à un autre, lorsqu'il est nécessaire de préciser les synchronisations qui ont lieu. Ainsi, nous croyons que Cosmo offre une façon à la fois expressive et naturelle de vérifier des programmes Multicore OCaml, malgré la subtilité du modèle mémoire faible.

Ces exemples illustrent une autre force de Cosmo : on a déjà vu que la logique offre un fragment sans vues, dans lequel le raisonnement est indépendant du modèle mémoire, c'est-à-dire où les règles sont celles de la Logique de Séparation Concurrente traditionnelle. Ce fragment comporte les règles relatives aux cases non-atomiques. Les structures de données concurrentes étudiées ici, dont la correction est vérifiée avec Cosmo, ont des spécifications (Figures 5.1 et 5.5) qui s'inscrivent également dans ce fragment indépendant du modèle mémoire : la spécification d'un verrou en Cosmo (Figure 5.1) est la spécification traditionnelle en Logique de Séparation Concurrente (Gotsman et al., 2007; Hobor et al., 2008). Ainsi, Cosmo contient la Logique de Séparation Concurrente traditionnelle, et permet de raisonner à propos de programmes concurrents à gros grains où tous les accès à des objets partagés sont protégés par des verrous. Ces verrous eux-mêmes sont implémentés en mémoire faible et leur vérification en Cosmo dépend du modèle mémoire, mais leur utilisation en est indépendante.

Par ailleurs, l'exemple de l'algorithme de Peterson illustre l'intérêt d'avoir réduit la subjectivité des assertions de Cosmo à une classe simple d'assertions $\uparrow \mathcal{V}$ qui sont *persistantes*, c'est-à-dire qui n'assument la possession d'aucune ressource. Dans l'algorithme de Peterson en effet, la synchronisation nécessaire pour transférer la ressource protégée par le verrou se fait via l'une de deux cases atomiques distinctes, mais on ne sait pas encore laquelle sera utilisée par la prochaine acquisition au moment où la ressource est relâchée.

### Une file concurrente multi-producteurs multi-consommateurs

Afin de démontrer l'applicabilité de Cosmo à la vérification modulaire de bibliothèques plus complexes, la dernière contribution de cette thèse est la spécification et la vérification en Cosmo d'une structure de données concurrente non triviale : une file multi-producteurs multi-consommateurs (Chapitre 6). Spécifier une telle structure de données (§6.1) soulève des questions intéressantes et démontre l'expressivité de notre logique.

Pour commencer, la spécification indique que la file se comporte comme si toutes ses opérations agissaient atomiquement sur un état commun quand, en réalité, elles accèdent à des parties distinctes de la mémoire et requièrent de nombreuses étapes de calcul. On utilise pour cela le concept d'*atomicité logique* (Jung et al., 2015; Jung, 2019; da Rocha Pinto et al., 2014) transposé dans Cosmo. Il s'agit, à notre connaissance, de la première utilisation de l'atomicité logique dans un modèle mémoire faible. Il faut noter que dans ce cadre, l'atomicité logique n'implique pas la « linéarisabilité » au sens traditionnel : elle implique bien l'existence d'un historique linéaire des opérations de la structure de données, mais ne garantit pas automatiquement de synchronisation entre opérations successives.

Or, dans un modèle mémoire faible, la spécification d'une structure de données doit décrire non seulement le résultat de ses opérations, mais également la façon dont celles-ci se synchronisent. Cette information supplémentaire permet de raisonner à propos d'accès à des zones de la mémoire hors la structure de données elle-même. Dans le cas d'une file, qui peut être utilisée pour transférer une donnée en mémoire d'un producteur à un consommateur, il est crucial que l'enfilage d'un élément se synchronise avec le défilage du même élément. De façon intéressante, la spécification qu'on donne (Figure 6.4) garantit certaines synchronisations — indépendamment de l'implémentation — mais elle en laisse d'autres non spécifiées. Par exemple, il n'est pas requis qu'un défilage se synchronise avec un enfilage ultérieur. Bien que toutes les opérations de la file se comportent comme si elles étaient atomiques, et s'ordonnent dans un historique linéaire, on ne garantit pas que tout couple d'opérations consécutives dans cet historique se synchronise. Alors que dans une implémentation à gros grains, le verrou induirait des synchronisations entre toutes les opérations, notre spécification faible permet des implémentations à grains fins, plus relâchées et donc plus efficaces. Finalement, l'expressivité de Cosmo permet d'écrire une spécification plus lâche que celle qui aurait été écrite dans un formalisme par raffinement. L'implémentation qu'on vérifie (§6.2) tire parti de ce relâchement.

Par ailleurs, pour s'assurer que cette spécification faible est suffisante pour un client de la file concurrente, on vérifie une application simple qui applique en parallèle une chaîne de traitements à un flux de données, en utilisant une file pour transmettre les valeurs intermédiaires d'un traitement au suivant (§6.4).

Alors même que l'implémentation de la file exploite le modèle mémoire faible de Multicore OCaml et que sa correction est vérifiée avec Cosmo, sa spécification n'est pas liée aux détails du modèle mémoire : elle se suffit à elle-même. Au moyen de vues abstraites qui matérialisent la notion empirique de connaissance sur l'état partagé, la spécification explique à l'utilisateur les synchronisations effectuées par la file sans qu'il soit besoin de connaître les primitives sous-jacentes de Multicore OCaml, ni le modèle mémoire qui s'y attache.

# Chapter 1

# Introduction

Advances in multicore hardware during the last two decades have created a need for powerful tools for writing efficient and trustworthy multicore software. These tools include well-designed programming languages and their compilers, efficient thread-safe libraries, and expressive program logics for proving the correctness of these and of the applications that exploit them. While some such verification tools already exist, researchers are only beginning to explore whether and how these tools can be exploited to modularly specify and verify realistic libraries that support fine-grained shared-memory concurrency.

Most of the programming languages that support multicore programming present shared memory as the primitive means of communication between threads. Although this design choice offers the greatest flexibility for writing efficient programs, it comes at an important cost: in order to achieve maximal efficiency, shared-memory concurrency cannot follow an intuitive *sequentially consistent* semantics (Lamport, 1979), according to which a program would behave as an interleaving of the actions of its threads, all having the same view of memory at every time. To allow compilers to perform more aggressive software optimizations and to better exploit hardware optimizations, many mainstream programming languages adopt some subtle *weak memory model*. This is the case of low-level languages such as C, C++ and Rust, which share the *C11 memory model* (Batty et al., 2011; Lahav et al., 2017); and of higher-level languages such as Java and other languages based on the JVM (Manson et al., 2005; Lochbihler, 2012; Bender and Palsberg, 2019).

Multicore OCaml (Dolan et al., 2018, 2020) follows this trend. It extends the OCaml programming language (Leroy et al., 2019) with support for shared-memory concurrency. Dolan et al. give it a well-defined semantics and, in particular, a *memory model*, which specifies how threads interact through shared memory locations. Therefore, one may already ask: what reasoning rules can or should a Multicore OCaml programmer rely upon in order to verify their code? Furthermore, as mentioned, Multicore OCaml's memory model is *weak*: it does not enforce sequential consistency. Although it is expected that most application programmers will not need to worry about weak memory, because they will rely on a library of concurrent data structures written by domain experts, adopting a weak memory model allows said experts to write more efficient code—if they know what they are doing, that is!

Because shared-memory concurrency is subtle, we believe that there is a need for a set of reasoning rules—in other words, a program logic—that both populations of programmers can rely upon. Concurrency experts, who wish to implement efficient concurrent data structures,

must be able to verify that their code is correct, even though they have removed as many synchronization operations as they thought possible. Furthermore, they must be able to verify that their code implements a simple, high-level abstraction, thereby allowing application programmers, in turn, to use it as a black box and reason about application code in a simple manner. In short, the system must allow compositional reasoning, and must allow both low-level reasoning, where the details of the Multicore OCaml memory model are apparent, and high-level reasoning, which is independent of this memory model.

Just in the past twenty years, a large variety of concurrent separation logics have been designed in order to meet the challenge of formally specifying and verifying programs that exploit shared-memory concurrency. Brookes (2004) and O'Hearn (2007) introduced Concurrent Separation Logic (Brookes and O'Hearn, 2016), which supported coarse-grain sharing of resources via mutexes. Their approach was gradually improved over the years, leading to expressive, higher-order separation logics, such as Iris (Jung et al., 2015, 2018b). Iris is able to express complex concurrent protocols, thanks to mechanisms such as *ghost state* and *invariants*, and supports reasoning about fine-grain concurrency, at the level of individual memory accesses. Concurrent data structures, such as mutexes, need not be considered primitive any more; they can be implemented and verified. Still, plain Iris is restricted to sequentially consistent semantics: it does not support weak memory models. A new generation of logics remove this restriction, for various memory models: iCAP-TSO (Sieczkowski et al., 2015), GPS (Turon et al., 2014), iGPS (Kaiser et al., 2017) and iRC11 (Dang et al., 2020) target fragments of the C11 memory model. iGPS, iRC11 and now this work are based on Iris.

For instance, Sieczkowski et al. (2015) propose iCAP-TSO, a variant of Concurrent Separation Logic that is sound with respect to the TSO memory model. While iCAP-TSO allows explicit low-level reasoning about weak memory, it also includes a high-level fragment, the "SC logic", whose reasoning rules are the traditional rules of Concurrent Separation Logic. These rules, which are independent of the memory model, require all primitive accesses to memory to be data-race-free. Therefore, they require synchronization to be performed by other means. A typical means of achieving synchronization is to implement a lock, a concurrent data structure whose specification can be expressed in the SC logic, but whose proof of correctness must be carried out at the lower level of iCAP-TSO. As another influential example, Kaiser et al. (2017) follow a similar approach: they instantiate Iris (Jung et al., 2018b) for a fragment of the C11 memory model. This yields a low-level "base logic", on top of which Kaiser et al. proceed to define several higher-level logics, whose reasoning rules are easier to use. Our aim, in this work, is analogous. We wish to allow the verification of a low-level concurrent data structure implementation, such as the implementation of a lock. Such a verification effort must take place in a program logic that exposes the details of the Multicore OCaml memory model. At the same time, we would like the program logic to offer a high-level fragment that is independent of the memory model and in which data-race-free accesses to memory, mediated by locks or other concurrent data structures, are permitted.

Compared with other memory models in existence today, the Multicore OCaml memory model is relatively simple. Only two access modes, known as "non-atomic" and "atomic", are distinguished. Every program has a well-defined set of permitted behaviors. In particular, data races on non-atomic memory locations are permitted, and have "limited effect in space and in time" (Dolan et al., 2018). This is in contrast with the C11 memory model, where racy programs are deemed to have "undefined behavior", therefore must be ruled out by a sound program logic.

Dolan et al. (2018) describe the Multicore OCaml memory model via an operational semantics, where the execution of the program is in fact an interleaving of the executions of its threads. These threads interact with a memory whose behavior is more complex than usual, and whose description involves concepts such as *timestamps*, *histories* and *views* of the shared memory (§2.2.1).

In this work, we take Dolan *et al.*'s operational semantics, which we recall in Section 2.2, as a foundation. We instantiate Iris for this operational semantics. This yields a low-level logic, BaseCosmo (Chapter 3), whose reasoning rules expose the details of the Multicore OCaml memory model. Because of this, these rules are not very pleasant to use. In particular, the rules that govern access to non-atomic memory locations are rather unwieldy, as they expose the fact that the store maps each non-atomic location to a history, a set of write events. In order to facilitate reasoning, on top of BaseCosmo, we build a higher-level logic, Cosmo (Chapter 4), whose main features are as follows.

- Cosmo forbids data races on non-atomic locations. Data races on atomic locations remain permitted: atomic locations are in fact the sole primitive means of synchronization. This design decision allows Cosmo to offer a simplified set of reasoning rules, including:
  - standard, straightforward rules for data-race-free access to non-atomic locations;
  - standard, straightforward rules for possibly racy access to atomic locations, with the ability of exploiting the sequentially-consistent behavior of these locations; and
  - nonstandard yet arguably simple rules for reasoning about the release/acquire behavior of atomic locations.

  The last group of rules allow transferring a "view" of the non-atomic memory from one thread to another. By exploiting this mechanism, one can arrange to transfer an arbitrary assertion $P$ from one thread to another, even when the footprint of $P$ involves non-atomic memory locations. This feature is used in all of our case studies (Chapters 5 and 6).

  Although views have appeared in several papers (Kaiser et al., 2017; Dang et al., 2020), letting the user reason about release/acquire behavior in terms of abstract views seems novel and simpler than previous approaches. In particular, we claim that combining objective invariants and the rule Split-Subjective-Objective yields a simple reasoning scheme, which could be exploited in logics for other weak memory models.

- Cosmo offers a high-level fragment where reasoning is independent of the memory model, that is, a fragment whose reasoning rules are those of traditional Concurrent Separation Logic. In this fragment, there is no notion of view. This fragment consists at least of the rules that govern access to non-atomic locations and can be extended by allowing the use of concurrent data structures whose specification is independent of the memory model and whose correctness has been verified using full Cosmo. The spin lock and ticket lock (Chapter 5) are examples of such data structures: their specification in Cosmo is the traditional specification of a lock in Concurrent Separation Logic (Gotsman et al., 2007; Hobor et al., 2008). Thus, Cosmo contains traditional Concurrent Separation Logic, and allows reasoning about coarse-grained concurrent programs where all accesses to memory locations are mediated via locks.

We illustrate the use of Cosmo via several case studies. Chapter 5 shows the specification and verification of typical, elementary synchronization libraries: a spin lock, a ticket lock, Dekker's mutual exclusion algorithm and Peterson's one. Chapter 6 aims to demonstrate the applicability of Cosmo to the modular verification of larger multicore programs. We specify a multi-producer multi-consumer queue, a more involved data structure that is archetypal of concurrent programming; we then prove the correctness of a realistic implementation. Specifying such a concurrent data structure raises interesting questions, and demonstrates the expressiveness of our logic. For one, the specification indicates that the queue behaves as if all of its operations acted atomically on a common shared state, even though in reality they access distinct parts of the memory and require many machine instructions; this uses the concept of *logical atomicity* (Jung et al., 2015; Jung, 2019; da Rocha Pinto et al., 2014), transported to the setting of Cosmo. Besides, in the weak memory setting, the specification of the data structure describes not only the result of its operations, but also the manner in which these are synchronized. Two operations synchronize when the thread performing the second operation obtains the view of memory which was that of the thread that performed the first operation. This is crucial if a queue is used to transfer ownership of a piece of memory from a producer to a consumer. Interestingly, the specification guarantees some amount of synchronization—regardless of the implementation—but it leaves others unspecified. Even though all operations of the queue behave as if they were atomic, and are ordered in a linear history, not all pairs of successive operations in that history are guaranteed to be synchronized. For instance, it is not required that a dequeuing operation synchronizes with the later enqueuing of an element. This weak specification allows for more relaxed and thus more efficient implementations than a coarse-grained one.

All of our results, including the soundness of Cosmo and the verification of our case studies, are machine-checked in Coq. Our proofs are available from our repository (Mével et al., 2021).

# Chapter 2

# Background

We aim at verifying the functional correctness of *concurrent* programs, that is, programs in which several threads run over overlapping periods of time. Threads then compete for shared resources such as computing power or memory. This situation may arise because an operating system schedules several threads to run in alternation on a given processor; or because parallel hardware—such as a multicore processor—runs several threads truly simultaneously; or a combination of both.

It has long been noted that the possibility of interaction between concurrent threads—for instance via signals or shared memory— adds complexity to the study of concurrent programs, by comparison with that of sequential programs. First, the order in which instructions pertaining to different threads are executed becomes relevant, and there is a considerable number of possible orderings. Second, with concurrent interactions come new kinds of programming bugs, such as deadlocks—a situation in which a set of threads are blocked because they are waiting for one another—and starvation—a situation in which a thread is constantly denied access to a shared resource. Lastly, ensuring that threads see the shared resources consistently is costly performance-wise, so that hardware and compiler have implemented optimizations which complicate significantly the notion of consistency used—giving birth to so-called weak models of consistency.

To be able to study programs written in a concurrent programming language, we first need to model what behaviors are allowed, that is, give a semantics to the said language. It is worth noting that such a semantics is given at the level of the source language, and that the system which has the responsibility of implementing it is the combination of all actors that play a role in the execution of a program, from hardware up to the compiler of the source language. Thus, designing a semantics is guided by the kind of optimizations we want from processors and compilers: a more permissive semantics allows for more optimizations, at the expense of more complexity for the programmer.

## 2.1 An overview of weak memory

A traditional description of the behavior of concurrent systems is the *sequential consistency* model (Lamport, 1979). In this model, the behavior of the concurrent system is some interleaving of the executions of its components. Even this simple model proves challenging for verification, as the number of possible interleavings is exponential in the number of instructions of each

thread.

For instance, below is a simple concurrent program with two threads. The shared memory contains two cells $x$ and $y$, with initial value 0. The left thread writes 1 to $x$ then reads $y$, whereas the right thread writes 1 to $y$ then reads $x$. The output of the program is the pair of integers $(a, b)$.

$$\begin{array}{c} \text{initially } [x] = [y] = 0 \\ \begin{array}{l|l} [x] \leftarrow 1 & [y] \leftarrow 1 \\ a \leftarrow [y] & b \leftarrow [x] \end{array} \end{array}$$

In the sequentially consistent model, since both the left and right threads have two instructions, there are $\binom{2+2}{2,2} = 6$ possible interleavings, producing varying outputs:

$$\begin{array}{lll} [x] \leftarrow 1;\ a \leftarrow [y];\ [y] \leftarrow 1;\ b \leftarrow [x] & \text{output is } (0,1) \\ [x] \leftarrow 1;\ [y] \leftarrow 1;\ a \leftarrow [y];\ b \leftarrow [x] & \text{output is } (1,1) \\ [x] \leftarrow 1;\ [y] \leftarrow 1;\ b \leftarrow [x];\ a \leftarrow [y] & \text{output is } (1,1) \\ [y] \leftarrow 1;\ [x] \leftarrow 1;\ a \leftarrow [y];\ b \leftarrow [x] & \text{output is } (1,1) \\ [y] \leftarrow 1;\ [x] \leftarrow 1;\ b \leftarrow [x];\ a \leftarrow [y] & \text{output is } (1,1) \\ [y] \leftarrow 1;\ b \leftarrow [x];\ [x] \leftarrow 1;\ a \leftarrow [y] & \text{output is } (1,0) \end{array}$$

Thus the sequentially consistent model allows three possible outcomes: $(0, 1)$ or $(1, 0)$ or $(1, 1)$. The outcome $(0, 0)$ is excluded.

Enforcing sequential consistency, however, has a cost: it precludes many desirable optimizations. If $x$ and $y$ are disjoint memory locations, then there are no dependencies between the write of $x$ and the read of $y$, and an optimizing compiler fed with the source code of the left thread may be tempted to swap both instructions. In a sequential context, this change does not alter the observable behaviors of a program, and it might result in a significant performance gain—be it by reducing memory latency or by triggering further optimizations, such as redundant read elimination. In the face of concurrency however, the change results in other behaviors than those previously listed; for instance, $(0, 0)$ becomes a possible outcome:

$$a \leftarrow [y];\ [y] \leftarrow 1;\ b \leftarrow [x];\ [x] \leftarrow 1 \quad \text{output is } (0,0)$$

This new behavior violates sequential consistency, as it cannot be described as the outcome of any interleaving of the instructions of the source code of both threads.

Yet we do not want to give up on optimizing sequential programs. Even in a concurrent program, a significant part of the work of a thread may consist in local state mutation that is unaffected by other threads. Thus, rather than enforcing sequential consistency strictly, it is often more pragmatic to allow for more optimizations, which preserve semantics in a sequential context, and to let the programmer restrict behaviors by means of inter-thread synchronization idioms—such as locks or fences—when concurrency is involved. Therefore, *even without true parallelism*, optimization puts sequential consistency in peril. Multicore architectures, featuring cache layers, performing write buffering, make violations of sequential consistency unavoidable.

The broad problem of consistency of a shared state has been attacked from various angles. Historically, weaker models of consistency have been developed in the context of distributed systems, where several machines have different views of distributed data. For instance, whereas sequential consistency guarantees that there is a total order on memory accesses on which all

threads agree, "causal consistency" merely ensures order within sets of causally-related accesses, for some notion of causality.

Transactional memory models are an alternative to lock-based synchronization that originates from the world of distributed databases. In such models, blocks of operations called transactions are to be committed atomically: they can perform multiple updates to the shared state, but these effects only become visible to other processes all at once, at the final commit point. When a conflict is detected, a tentative transaction is aborted and re-tried. This kind of models aims to achieve serializability, a strong form of sequential consistency between transactions, while preventing deadlocks and relieving the programmer from manipulating locks. By contrast with lock-based programming, which is pessimistic in that it uses systematic synchronization, transactional memory can reduce the cost of parallelism when there is little contention. However, even transactional memory does not evade the desire for optimization: database systems often allow to relax the atomicity of transactions by reducing their so-called "isolation level", which breaks serializability and opens the door to inconsistent database accesses.

In the domain of programming languages, from the 1990s onward there has been a surge of interest for weak memory models. The most notable research efforts have been devoted to giving a memory model to Java (§2.1.1), then to C/C++ (§2.1.2). Drawing from the experience of these predecessors, Multicore OCaml also comes with its own weak memory model (§2.2.1). A comparison between Multicore OCaml's model and these two predecessors is sketched in §2.2.1.

### 2.1.1 Java

In the 1980s, memory models were limited to informal descriptions by hardware vendors, often incomplete and subject to backwards-incompatible changes (Adve and Boehm, 2010). Parallel computing was therefore reserved to programmers with an expert knowledge not only of their target architecture, but also of the specific optimizations that their compiler of choice would or would not do. At a time when multicore architectures were becoming more and more common, lack of a clearly-defined semantics at the software level was impeding broader usage of parallel features by programmers—and conversely, hardware vendors could only guess what programmers would expect or use. In 1995 Java—then a new language, whose design had included multithreading from the beginning—made an attempt at describing its memory guarantees; however this initial specification was unclear, buggy, and prevented valuable optimizations (Pugh, 2000). Thus, in the 2000s, considerable effort has been put into designing a clear and workable specification for the memory model of a high-level programming language. This has led to a new memory model, adopted in Java 5.0 (Manson et al., 2005).

The Java memory model is described in terms of relations between memory events in a given execution; which executions are allowed is then specified as a set of constraints on these relations—typically, some relations are required to be acyclic. This kind of formalism is called an *axiomatic model*. At the heart of Java's model is a partial order called the *happens-before* relation.

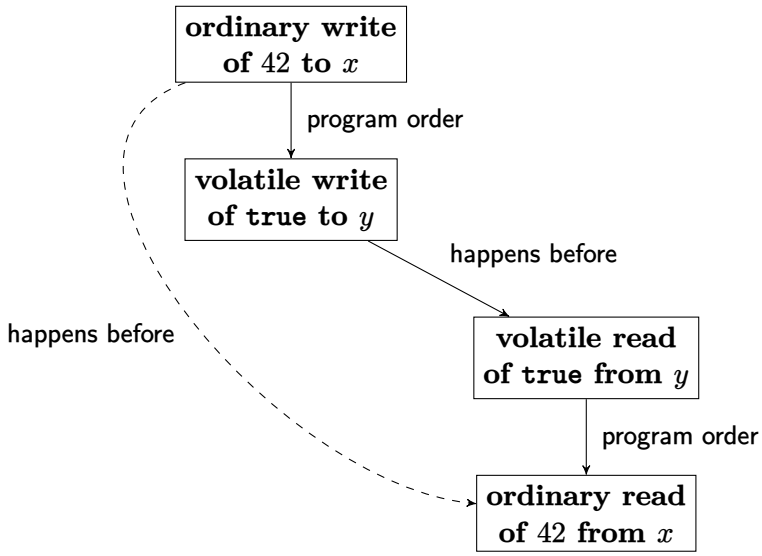
Java features ordinary variables, *volatile* variables and locks (also called "monitors"). By contrast with ordinary variables, accesses to volatile variables perform synchronization. Technically speaking, for a given volatile variable, there is a happens-before relation from a write event to a read event pulling the value of that write. Furthermore, the happens-before relation is transitive and contains the program order (that is, the order in which instructions are written in

source code). This allows the following concurrent programming idiom, called *message passing*.

$$
\begin{array}{c}
x \text{ is an ordinary variable} \\
y \text{ is a volatile variable} \\
\text{initially } [x] = 0 \text{ and } [y] = \texttt{false}
\end{array}
$$

$$
\begin{array}{l|l}
[x] \leftarrow 42 & a \leftarrow [y] \\
[y] \leftarrow \texttt{true} & b \leftarrow [x] \\
 & \texttt{assert } (a = \texttt{false} \vee b = 42)
\end{array}
$$

In the left thread, the write to $x$ is program-ordered before the write to $y$. In the case when the right thread reads `true` in $y$, the write to $y$ happens-before the read of $y$, which is program-ordered before the read of $x$. Thus, the write of 42 to $x$ happens-before the read of $x$. Schematically, the graph of memory events of such an execution may be represented as follows.




In this situation, the right thread indeed observes the value 42 in $x$, because the Java memory model mandates that an ordinary read pulls its value from the last write that happens-before this read.

For a given execution, we formally define a *data race* as a pair of two accesses to the same variable, at least one of them being a write, which are not ordered by the happens-before relation. Data races on non-volatile variables capture a class of concurrency bugs: they cannot be given a reliable behavior and are the sign of a lack of synchronization, a mistake from the programmer. A program is said to be *correctly synchronized* when no sequentially-consistent execution of the program exhibits a data race. The Java memory model enjoys the *data-race-freedom (DRF) property*: if a program is correctly synchronized, then any execution of the program is equivalent to a sequentially consistent execution (Manson et al., 2005, §9.2.1). This property allows programmers to reason about the correctness of their code: to prove that a program is correct, it is enough to prove it in a memory model where data races have undefined behavior (Aspinall and Ševčík, 2007).

Some memory models have a "catch-fire" approach to data races: racy programs are considered incorrect and absolutely nothing is guaranteed about their behavior. However, in Java,

even racy programs must observe minimal safety and security guarantees. For instance, the memory model strives to disallow "out-of-thin-air" values, that is, the fact that a location ends up storing a value that has not been produced by any expression of the source code. The possibility of such a situation is regarded as a serious flaw, for one, because it could produce an invalid pointer, thus breaking memory safety (Manson et al., 2005, §2.2), type safety, and object invariants. Hence, part of the challenge in designing a memory model for Java has consisted in giving semantics to racy programs as well, based on a notion of causality (Manson et al., 2005, §4).

### 2.1.2 C11

Another prominent memory model is that commonly referred to as the "C11 memory model", which is used by C, C++ and Rust at least. It was initially proposed for the 2011 revision of the C and C++ standards (Batty et al., 2011). Not unlike in Java's history, this initial model was later recognized to present several flaws: notably, it was plagued by the out-of-thin-air problem (Vafeiadis and Narayan, 2013), it prevented sensible optimizations (Vafeiadis et al., 2015) and, worse, proposed compilation schemes to some architectures were unsound (Lahav et al., 2017). Thus the model has been amended; an improved model is known as the "repaired C11 model" (Lahav et al., 2017).

As in Java, the C11 memory model is described axiomatically. Here, rather than diving into the details of this axiomatic model, we give some intuitions. The C11 model distinguishes between *atomic* and *non-atomic* locations. There are several modes of access to atomic locations, from stronger to weaker: *sequentially consistent (SC), release/acquire (RA), relaxed.*[1] The semantics of these access modes can be summarized as follows.

- SC accesses are the strongest and most costly. Atomic locations which are used with only SC accesses should behave as a sequentially consistent piece of memory, that is, at any time, they store single values that are seen coherently by all threads. This is summarized by the *DRF-SC theorem*, a variant of the data-race-freedom theorem (§2.1.1) which applies to C programs where atomic locations are used only through SC accesses.

- Release/acquire accesses are weaker, but suffice to implement the message-passing idiom shown in §2.1.1. They can be described in terms of allowed reorderings: an *acquire read* cannot be moved after a subsequent read, and a *release write* cannot be moved before a preceding write.

- Relaxed accesses are the weakest possible form of atomic accesses. By itself, they do not prevent any reordering. Said otherwise, they guarantee no synchronization. Still, by contrast with a data race on a non-atomic location, a data race involving relaxed atomic accesses has a defined behavior.

Unlike in Java (and now Multicore OCaml), several access modes can be mixed on a given atomic location. The interplay between access modes on a same location is responsible for a good share of the complexity of the C11 model. For one, in the original model, it causes SC accesses to exhibit surprising behaviors, weaker than one might expect, and is a motivation for the repaired model (Lahav et al., 2017).

[1] The standard also defines "consume" reads, which we leave out of this short introduction.

$$
\begin{aligned}
\ell &\in \textsc{Loc} \\
c, a, A \in \textsc{CellAddr} &\triangleq \textsc{Loc} \times \mathbb{N} \\
\alpha \in \textsc{AccMode} &\triangleq \{\mathrm{na}, \mathrm{at}\} \\
\sigma \in \textsc{Store} &\triangleq \textsc{Loc} \xrightarrow{\text{fin}} \textsc{List}\,(\textsc{CellContents}_{\mathrm{na}} + \textsc{CellContents}_{\mathrm{at}}) \\
\textsc{CellContents}_{\mathrm{na}} &\triangleq \textsc{Hist} \\
h \in \textsc{Hist} &\triangleq \textsc{Time} \xrightarrow{\text{fin}} \textsc{Val} \\
t \in \textsc{Time} &\triangleq \mathbb{Q}_{\geq 0} \\
\textsc{CellContents}_{\mathrm{at}} &\triangleq \textsc{Val} \times \textsc{View} \\
\mathcal{V}, \mathcal{W}, \mathcal{G} \in \textsc{View} &\triangleq \textsc{CellAddr} \xrightarrow{\text{aez}} \textsc{Time}
\end{aligned}
$$

Figure 2.1: Semantic objects: locations, cells, stores, histories, timestamps, views.

Unlike in Java (and now Multicore OCaml), the semantics is "catch-fire": a program that exhibits a data race on a non-atomic location has an entirely undefined behavior, which means that, as per the standard, anything is allowed to happen.

The C11 memory model as a whole is deemed particularly complex and, so far, many verification frameworks for that model (e.g. Vafeiadis and Narayan (2013); Turon et al. (2014); Kaiser et al. (2017)) tackle only a fragment of it.

## 2.2 Multicore OCaml

In this section, we present the Multicore OCaml memory model as well as the syntax and operational semantics of a core calculus that is representative of the Multicore OCaml programming language. We start by describing interactions with the memory, isolated in a memory subsystem (§2.2.1), then we present the reduction rules of the programming language itself (§2.2.2).

### 2.2.1 The memory model

In this section, we replicate Dolan *et al.*'s definition of the Multicore OCaml memory model (Dolan et al., 2018) and extend it with memory allocation, CAS and blocks, which do not appear in Dolan *et al.*'s paper. The formalization presented here is a small-step *operational model*, contrasting with the axiomatic models that describe the memory models of Java (§2.1.1) and C11 (§2.1.2). Even though Multicore OCaml can also be given an axiomatic model (Dolan et al., 2018), in this dissertation we are interested in an operational semantics because it enables reasoning about the execution of a program step by step, in a program verification framework such as Iris; indeed, in a later chapter (Chapter 3) we will instantiate Iris with the operational semantics that we present now. We first define a number of semantic objects (Figure 2.1), then describe the behavior of the memory subsystem (Figure 2.2). We then draw comparisons between the described model and those of Java and C11.

At this point, we assume a set Val of values, which is defined later on (§2.2.2).

### Locations

A *location* $\ell$ represents the address of a contiguous block of memory. We assume a countably infinite set of locations Loc.

### Stores, blocks and cells

A *store* $\sigma$ is a mathematical object that represents the entirety of the allocated shared memory. It maps a finite set of block locations to block contents. A *block* represents a contiguous allocated area of memory, supporting efficient random accesses. It has a *length* $n \in \mathbb{N}$ and comprises that number of *cells*. The contents of a block is the list of the contents of its cells, in order. The shape of a cell's contents depends on the access mode of that cell.

### Access modes

Each cell is rigidly ascribed an *access mode* $\alpha$, either *atomic* ($\alpha = \text{at}$) or *non-atomic* ($\alpha = \text{na}$). The memory subsystem described later (Figure 2.2) enforces that this access mode is chosen when allocating the block and remains constant throughout execution.[2]

As far as the memory model is concerned, cells live entirely independent lives: it is not required in principle that two cells of a given block have the same access mode[3] and no synchronization is implied between two cells of a given block.

### Cell addresses

Following a standard approach to ruling out aliasing issues, blocks at different locations are disjoint, so that an individual cell is uniquely identified by the pair $(\ell, i)$ of its enclosing block's address $\ell$ and of the cell's offset $i$ within that block; the offset ranges from 0 to $n - 1$ where $n$ is the length of the block. In this dissertation, we denote this pair as $\ell[i]$, and we use $c$, $a$, $A$ as conventional names for such a "cell address"; we write $a$ (respectively, $A$) to mean that it is the address of a non-atomic (respectively, atomic) cell.

By uncurrying, a store $\sigma$, mapping locations to lists of cell contents, can also be seen as mapping cell addresses to cell contents. In that regard, given a cell $c = \ell[i]$, we allow ourselves to write $\sigma(c)$ for $\sigma(\ell)_i$, that is, the element of index $i$ in the list $\sigma(\ell)$. Likewise, we write $\sigma[c \mapsto x]$ for $\sigma[\ell \mapsto \sigma(\ell)[i \mapsto x]]$, that is, for the store $\sigma$ where the contents of cell $c$ has been updated to $x$.

### Histories

As said, a store maps cell addresses to cell contents, whose shape depends on the access mode. The contents of a non-atomic cell is a history of all write events at this cell. A *history* $h$ is a finite map of timestamps to values. If desired, a history can also be viewed as a set of pairs of a timestamp and a value, that is, a set of write events, under the constraint that no two events have the same timestamp.

[2] In the actual Multicore OCaml language, typing enforces this discipline statically.

[3] However, merely for simplicity of presentation, the allocation primitives of the memory subsystem presented later enforce this property.

### Timestamps

We use *timestamps* $t$ to describe the behavior of non-atomic accesses. They belong to $\mathbb{Q}_{\geq 0}$, the set of the nonnegative rational numbers. This set is chosen for a combination of properties:

- it is infinite: write operations cannot run short of timestamps;
- it is totally ordered: the operational semantics of the memory subsystem relies on comparing timestamps;
- its order is dense, that is, a timestamp can be found in between any two timestamps; this requirement, originally introduced by Kang et al. (2017) in their promising semantics (their operational semantics for the C11 model), is not actually used in our own proofs, but it adds no complexity by contrast with using $\textsc{Time} \triangleq \mathbb{N}$ and it sticks more closely to the memory model devised by Dolan et al. (2018);
- it admits a minimum element 0: this timestamp is used for initial writes, and only for initial writes.

It is worth noting that the meaning of timestamps is purely "per cell": that is, the timestamps associated with two write events at two distinct cells are never compared.

### Views

A *view* $\mathcal{V}$, referred to as a "frontier" by Dolan et al., is a total mapping of cell addresses to timestamps which is *almost everywhere zero* ("aez" for short), that is, whose value is zero everywhere except at a finite set of cell addresses. A timestamp is meaningful only for non-atomic cells; the timestamp of atomic cells are not used by the semantics.

Each thread has a view, which increases over time. This view gives, for each cell, the timestamp of the most recent write event at this cell that the thread is aware of. The view of the active thread imposes a constraint on the behavior of reads and writes at non-atomic cells (§2.2.2).

The order on timestamps gives rise to a partial order on views: by definition, the view inequality $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ holds if and only if, for every cell $c$, the timestamp inequality $\mathcal{V}_1(c) \leq \mathcal{V}_2(c)$ holds. Equipped with this order, views form a bounded join semilattice. Its minimum element is the *empty view* $\varnothing$, which maps every memory cell to the timestamp 0. Its join operation is the pointwise maximum: that is, $\mathcal{V}_1 \sqcup \mathcal{V}_2$ is $\lambda c.\ \max(\mathcal{V}_1(c), \mathcal{V}_2(c))$.

This order can be thought of as an *information* order: if $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ holds, then a thread with view $\mathcal{V}_2$ has more information (is aware of more write events) than a thread with view $\mathcal{V}_1$. As will be seen when presenting the memory subsystem, a non-atomic access issued by a thread with view $\mathcal{V}_2$ has *fewer* permitted behaviors than the same instruction issued by a thread with view $\mathcal{V}_1$. Indeed, the knowledge of a write event "masks" older (according to their timestamps) write events at the same non-atomic cell: they cannot be read from anymore, and new write events can only be inserted after the latest currently-known one.

### Contents of atomic cells

The contents of an atomic cell is a pair of a value and a view. Indeed, atomic cells in Multicore OCaml serve two purposes, which are conceptually independent.

$$\textsc{mem-na-Alloc}\quad \frac{\ell \notin \mathrm{dom}(\sigma) \qquad n \geq 0 \qquad h = \{0 \mapsto v\}}{\sigma; \mathcal{W} \xrightarrow{\mathsf{alloc}_{\mathrm{na}}(\ell,n,v)} \sigma \uplus \{\ell[i] \mapsto h \mid 0 \leq i < n\}; \mathcal{W}}$$

$$\textsc{mem-na-Read}\quad \frac{h = \sigma(a) \qquad t \in \mathrm{dom}(h) \qquad \mathcal{W}(a) \leq t \qquad v = h(t)}{\sigma; \mathcal{W} \xrightarrow{\mathsf{rd}_{\mathrm{na}}(a,v)} \sigma; \mathcal{W}}$$

$$\textsc{mem-na-Write}\quad \frac{h = \sigma(a) \qquad t \notin \mathrm{dom}(h) \qquad \mathcal{W}(a) < t \qquad h' = h[t \mapsto v]}{\sigma; \mathcal{W} \xrightarrow{\mathsf{wr}_{\mathrm{na}}(a,v)} \sigma[a \mapsto h']; \mathcal{W}[a \mapsto t]}$$

$$\textsc{mem-at-Alloc}\quad \frac{\ell \notin \mathrm{dom}(\sigma) \qquad n \geq 0}{\sigma; \mathcal{W} \xrightarrow{\mathsf{alloc}_{\mathrm{at}}(\ell,n,v)} \sigma \uplus \{\ell[i] \mapsto \langle v, \mathcal{W} \rangle \mid 0 \leq i < n\}; \mathcal{W}}$$

$$\textsc{mem-at-Read}\quad \frac{\sigma(A) = \langle v, \mathcal{V} \rangle}{\sigma; \mathcal{W} \xrightarrow{\mathsf{rd}_{\mathrm{at}}(A,v)} \sigma; \mathcal{W} \sqcup \mathcal{V}}$$

$$\textsc{mem-at-Read-Write}\quad \frac{\sigma(A) = \langle v, \mathcal{V} \rangle \qquad \mathcal{V}' = \mathcal{W}' = \mathcal{W} \sqcup \mathcal{V}}{\sigma; \mathcal{W} \xrightarrow{\mathsf{rdwr}_{\mathrm{at}}(A,v,v')} \sigma[A \mapsto \langle v', \mathcal{V}' \rangle]; \mathcal{W}'}$$

$$\textsc{mem-Length}\quad \frac{\sigma(\ell) = b \qquad |b| = n}{\sigma; \mathcal{W} \xrightarrow{\mathsf{len}(\ell,n)} \sigma; \mathcal{W}}$$

Figure 2.2: Operational behavior of the memory subsystem: $\sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'$

On the one hand, each atomic cell stores a value with a sequentially consistent semantics. In other words, all threads agree at all times on the current value of an atomic cell; hence, it suffices to keep track of that value, as opposed to a complete history.

On the other hand, each atomic cell acts as a synchronization medium with a "release/acquire" semantics. As explained by Dolan et al. (2018), "non-atomic writes made by one thread can become known to another by communicating via an atomic cell." Formally speaking, each atomic cell stores a view, that is, a certain amount of information about the non-atomic store. This view is obtained and updated by accesses to that atomic cell.

### The memory subsystem

To complete the definition of the Multicore OCaml memory model, there remains to give an operational description of the behavior of the memory subsystem. This is done via a labeled transition system, that is, a relation $\sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'$, which describes how the shared global store $\sigma$ and the view $\mathcal{W}$ of the active thread evolve through a memory event $m$, yielding an updated store $\sigma'$ and an updated thread view $\mathcal{W}'$. The syntax of memory events is as follows, where $\alpha$ is an access mode, $\ell$ is a location, $c$ is a cell, $a$ is a non-atomic cell, $A$ is an atomic cell, $v, v'$ are values and $n$ is a natural integer:

$$m ::= \varepsilon \mid \mathsf{alloc}_{\alpha}(\ell, v, n) \mid \mathsf{rd}_{\alpha}(c, v) \mid \mathsf{wr}_{\mathrm{na}}(a, v) \mid \mathsf{rdwr}_{\mathrm{at}}(A, v, v') \mid \mathsf{len}(\ell, n)$$

These memory events also appear in the definition of the semantics of Multicore OCaml expressions (§2.2.2). Thus, they form a language by which the expression and memory subsystems communicate.

The rules defining the relation $\sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'$ appear in Figure 2.2 (under the form of inference rules, where the premises and conclusion of a rule are written above and below a line, respectively); they can be briefly described as follows.

When a fresh block of non-atomic cells is allocated (MEM-NA-ALLOC) with initial value $v$, their respective histories consist of a single write event of that value at timestamp 0. This guarantees that a read of these cells will succeed, even if the reading thread has not synchronized with the thread where allocation was performed—which may happen in a racy program that would communicate the newly-allocated address through a non-atomic cell without proper synchronization.

Naturally, a read instruction at a non-atomic cell must read from one of the write events in the history of this cell, and a write instruction must extend the history of this cell with a write event at a previously-unused timestamp. In either case, which timestamp may be chosen is constrained by the "view" of the active thread.

A read instruction at cell $a$ (MEM-NA-READ) cannot read from an outdated write event, therefore must read from an event whose timestamp $t$ is no less than $\mathcal{W}(a)$. Both $\mathcal{W}(a) = t$ and $\mathcal{W}(a) < t$ are permitted: the latter case allows a thread to read from a write event of which it is not yet aware. A non-atomic read instruction does *not* update the view of the active thread.

A write instruction at cell $a$ (MEM-NA-WRITE) cannot produce an outdated write event, therefore must use a timestamp $t$ greater than $\mathcal{W}(a)$. The new timestamp must be fresh (that is, not in the domain of $h$), but there is no requirement for $t$ to be greater than every timestamp in the domain of $h$. The history of cell $a$ and the view of the active thread are updated so as to include the new write event.

Allocating a fresh block of atomic cells (MEM-AT-ALLOC) causes them to be initialized with the value $v$ provided by the allocation instruction and with the view $\mathcal{W}$ of the active thread.

When reading an atomic cell (MEM-AT-READ) the view $\mathcal{V}$ stored in this cell is merged into the view of the active thread, reflecting the idea that the active thread acquires information via this read operation.

Updating an atomic cell (MEM-AT-READ-WRITE) overwrites the value $v$ stored at this cell with the new value $v'$ and updates the view $\mathcal{V}$ stored at this cell by merging the view $\mathcal{W}$ of the active thread into it, reflecting the idea that the active thread releases information via this write operation. Also, in the same atomic step, the active thread acquires the stored view, as when reading.

As Multicore OCaml is equipped with a garbage collector, there is no deallocation event.

Lastly, the length of a block can be requested (MEM-LENGTH), without affecting the store nor the view of the active thread.

### Extensions to Dolan *et al.*'s memory model

The memory model proposed by Dolan et al. does not address memory allocation. We thus extend their model with suitable memory events and reduction rules. It is important to keep in mind that Multicore OCaml ensures memory safety. This implies that uninitialized memory must not be observable. As a consequence, an allocation event is conflated with an initial write

to the newly-allocated block. The fact that no thread can read an older state is ensured by the structure of our semantic objects: for a given non-atomic cell, any view gives at least the minimum timestamp 0 which corresponds to the initial write. The initial write is immediately known to all threads. Implementation-wise, this implies that the runtime system of Multicore OCaml must perform some amount of synchronization at some point—either when allocating, or when publishing the address of a block to the shared memory, or when reading from such an address.

Dolan et al. omits CAS instructions, but adding them to the model is straightforward. Since an atomic write in their model combines the effects of an "acquire" read and that of a "release" write, the same memory event $\mathsf{rdwr}_{\mathrm{at}}(A, v, v')$ can be used to model both an atomic write and a CAS instruction. By contrast with the original model, that event mentions the overwritten value $v$. It is ignored by atomic writes but matters for CAS instructions (the difference will become apparent later on in Figure 2.4c).

Dolan et al.'s original paper features only mutable *references*, that is, blocks of length one. We extend their model with blocks of arbitrary lengths. Our extension permits efficient random accesses in the programming language, which is needed for realistic implementations of some data structures. In practice, the present-day implementation of Multicore OCaml has atomic references, non-atomic references and blocks ("arrays") of non-atomic cells. It would not be hard to add language support for blocks of atomic cells, or for records with mixed atomic and non-atomic fields, as already supported by our model.

### Comparison with the memory model of Java

The definition just given of the memory model of Multicore OCaml is operational, that is, it consists of a (small-step) reduction relation with an interleaving semantics—but where the shared state has a more intricate mathematical structure than it would have in a sequentially consistent setting. By contrast, the memory model of Java is traditionally described axiomatically (as done in §2.1.1): a "candidate execution" takes the form of a set of memory events and relations between them, and an actual execution is a candidate execution that respects certain conditions, such as the acyclicity of certain relations. In fact, Dolan et al. also give an axiomatic account of their model, and provide a paper proof of the equivalence between their operational and axiomatic definitions.

The memory model of Multicore OCaml is most similar to that of Java. Both provide two access modes—"non-atomic" and "atomic" in Multicore OCaml terms, or ordinary and "volatile" in Java terms—and a given memory cell must only be accessed with one mode (a constraint that both languages enforce thanks to their typing discipline). While non-atomic variables have a relaxed behavior, atomic variables perform synchronization which allows the message-passing, or "release-acquire", pattern.

To quote Dolan et al., both models "bound data races in space"; that is, a race on one cell does not compromise accesses to other regions of memory. This matches the memory safety guarantees offered by the Java and Multicore OCaml languages, in which some catastrophic failures—such as segmentation faults—must not happen even in the event of a race. By contrast, in unsafe languages such as C, the presence of any race in a program makes the behavior of the entire program undefined.

The memory model of Multicore OCaml differs from that of Java on certain aspects. Contrary to Java, Multicore OCaml also "bounds data races in time", in the sense that a race on a given

memory cell does not affect earlier accesses to that cell, nor does it compromise future accesses for good: by performing enough synchronization, a program may recover from a racy state. Dolan et al. give the following two examples.

The first example illustrates a lack of coherence in Java, which results in a race that compromises later accesses even if subsequent synchronization is done. Consider the following program, with two shared cells $x$ and $y$ where only $y$ is atomic.

$$
\begin{array}{c}
x \text{ is a non-atomic cell} \\
y \text{ is an atomic cell} \\
\text{initially } [x] = 0 \text{ and } [y] = \texttt{false}
\end{array}
$$

$$
\begin{array}{l|l}
[x] \leftarrow 1 & [x] \leftarrow 2 \\
[y] \leftarrow \texttt{true} & a \leftarrow [y] \\
 & b \leftarrow [x] \\
 & c \leftarrow [x] \\
 & \texttt{assert } (a = \texttt{false} \lor b = c)
\end{array}
$$

This program exhibits a race between the two writes to $x$. However, since atomic accesses perform synchronization, a programmer might expect that, if the second thread has read `true` in the atomic cell $y$, then it has observed all the writes to $x$ and thus, at that point, reading $x$ twice would yield the same value—which one would be read depends on how the conflicting writes would have been ordered when synchronizing. In other words, when $a = \texttt{true}$, either $(b, c) = (1, 1)$ or $(b, c) = (2, 2)$.

This expectation is violated in the memory model of Java, which also allows $(b, c) = (1, 2)$ and $(b, c) = (2, 1)$. Indeed, Java might optimize the first read of $x$ to the constant 2 without doing likewise for the second read (if, for example, aliasing hides from the compiler the fact that the cell to be read is $x$).

On the other hand, the expectation holds in Multicore OCaml; its axiomatic model ensures it via an additional *coherence* property. In the operational model presented earlier, after it has read `true` in $y$, the right thread has updated its view to the maximum of the timestamps of both writes to $x$, hence the following reads are in fact not racy: they can only read from the last write. The race on $x$ has been resolved thanks to synchronization.

The second example illustrates *load buffering*, which is allowed by the Java model, but is hard to reason about and can lead to arguably surprising results: with load buffering, reads can be made inconsistent because of *future* data races. Consider the following example, where $x$ is a shared non-atomic cell.

$$
\begin{array}{c}
x \text{ is a non-atomic cell} \\
\text{initially } x \text{ contains some valid address}
\end{array}
$$

$$
\begin{array}{l|l}
y \leftarrow \texttt{ref}_{\text{na}}\ 0 & y \leftarrow [x] \\
[y] \leftarrow 1 & [y] \leftarrow 2 \\
a \leftarrow [y] & \\
[x] \leftarrow y & \\
\texttt{assert } (a = 1) &
\end{array}
$$

This program exhibits a data race on $x$. Still, a programmer might want to reason about an execution before the race occurs; one might expect that $a$ receives the value 1 when reading $y$,

because at that point cell $y$ has just been allocated and is not shared yet, so no other thread could possibly have written to it. This expectation is violated in Java: because the read of $y$ and the write to $x$ in the left thread operate on distinct cells, Java can reorder both operations, which can lead to $a$ reading the value 2.

One of the design goals of the memory model of Multicore OCaml was to forbid load buffering; in the example above, Multicore OCaml fulfills the programmer's expectation.

Reasoning about load buffering operationally is a challenging problem; Kang et al. (2017) devise the promising semantics for tackling it in the context of C11. The axiomatic model of Multicore OCaml carefully avoids load buffering, consequently its operational semantics does not need promises and is simpler than that of Kang et al.. The drawback, as can be seen, is that Multicore OCaml permits fewer optimizations than Java. Dolan et al. (§8) assess the performance penalty of this stricter model.

These authors' claim that Multicore OCaml "bounds data races in space and time" is made formal in a *local data-race-freedom (local DRF) theorem* (Dolan et al., 2018, §4–5), which is a stronger variant of Java's DRF theorem (§2.1.1), involving a notion of ongoing data races.

### Comparison with the memory model of C11

It is less straightforward to compare the memory models of C11 (§2.1.2) and of Multicore OCaml.

A C11 non-atomic location has no counterpart in Multicore OCaml, because data races on such a location have undefined behavior according to the "catch-fire" model of C11, whereas the model of Multicore OCaml enforces memory safety, type safety, and gives well-defined semantics to all memory accesses. For this reason, a Multicore OCaml non-atomic cell is rather comparable to a C11 atomic location with relaxed accesses. The latter is slightly stronger though: indeed, in the axiomatic model of C11, there is a "coherence order" that relates all accesses to a given atomic location (Lahav et al., 2017, §3), while in Multicore OCaml, non-atomic reads are not necessarily ordered. This is visible in the operational semantics, specifically in rule MEM-NA-READ: reading a non-atomic cell does not update the thread's current view $\mathcal{W}$, hence it is possible that a thread reads a write with some timestamp $t$, then reads another write with an *earlier* timestamp $t' < t$. This can only occur, though, when the reads are involved in a data race. For instance, for the racy program below, the semantics of Multicore OCaml allows the outcome $(a, b) = (1, 0)$, which violates the assertion.

$$
\begin{array}{c}
x \text{ is a non-atomic cell} \\
\text{initially } [x] = 0 \\
\begin{array}{l|l}
a \leftarrow [x] & [x] \leftarrow 1 \\
b \leftarrow [x] & \\
\texttt{assert } (a \leq b) &
\end{array}
\end{array}
$$

By contrast, the analogous program in C11 with relaxed atomic accesses is guaranteed to pass the assertion.

A Multicore OCaml atomic cell is comparable to a C11 atomic location whose access mode would be a middle ground between SC accesses and release/acquire accesses. This can be analyzed by separating the two features provided by a Multicore OCaml atomic cell: the value it stores and the synchronization it performs.

1. A Multicore OCaml atomic cell stores a single value at all times, that is seen identically by all threads. In this respect, it is similar to a C11 atomic location with SC accesses, and is stronger than one with release/acquire accesses. Indeed, the latter stores a history of values rather than a single value, and can be observed in different states by different threads in the event of a data race.

2. A Multicore OCaml atomic cell transmits knowledge about the *rest* of the memory from writers to readers of that cell, in a release/acquire fashion. There is no implied synchronization between accesses to distinct atomic cells. Multicore OCaml atomic cells are thus weaker than a C11 atomic location with SC accesses, because C11 enforces synchronization between all SC accesses to *all* atomic locations.

### 2.2.2 The programming language

We now present a core calculus that is representative of the Multicore OCaml programming language as regards shared-memory concurrency. It is equipped with a dynamic thread creation operation and with the two access modes presented earlier. For simplicity, we refer to this calculus as Multicore OCaml.

#### Syntax

The syntax of our idealized version of Multicore OCaml appears in Figure 2.3. It is untyped (contrary to the actual Multicore OCaml) and equipped with a standard call-by-value, left-to-right evaluation. It features recursive functions, primitive operations on Boolean and integer values, tuples, optional values, and standard control constructs, including conditionals and sequencing. The last group of constructs in this figure are syntactic sugar, easily implemented by other constructs, thus there is no need to give semantics to the derived constructs. This calculus is extended with shared-memory concurrency, as follows. First, the construct `fork` $e$ spawns a new thread. In the parent thread, this operation immediately returns the unit value `()`. In the new thread, the expression $e$ is executed, and its result is discarded. Second, the language supports the standard operations of reading and writing, on both non-atomic and atomic memory cells. In addition, atomic memory cells support a compare-and-set (CAS) operation.

The expression $\texttt{array}_\alpha[n]\ v$ allocates a block of $n$ cells whose access mode is $\alpha$ and whose initial value is $v$. There is no deallocation operation. Reading from a cell with access mode $\alpha$ at offset $k$ of location $\ell$ is written $\ell[k]_\alpha$. Writing a value $v$ to a cell with access mode $\alpha$ at offset $k$ of location $\ell$ is written $\ell[k]_\alpha \leftarrow v$. In addition, atomic cells support the usual compare-and-set operation: `CAS` $\ell[k]\ v_1\ v_2$ reads the atomic cell at offset $k$ of location $\ell$, tests whether its value is equal to $v_1$, overwrites it with $v_2$ if that is the case, and returns the Boolean result of the test; importantly, the read and the write operations happen *atomically*.

All memory reads and updates expect a pair of a block location and an offset: in other words, cells are not first-class values, and there is no pointer arithmetic. There is syntactic sugar for single-cell blocks, or "references": $\texttt{ref}_\alpha\ v$ allocates a block of length one, $!_\alpha\,\ell$ reads at offset zero, $\ell :=_\alpha v$ writes at offset zero, and `CAS` $\ell\ v_1\ v_2$ performs a compare-and-set operation at offset zero.[4]

[4]In the surface language, non-atomics have type `'a ref`. Their operations are `ref`, `!`, and `:=`. Atomics have type `'a Atomic.t`. Their operations are `Atomic.make`, `Atomic.get`, `Atomic.set`, and `Atomic.compare_and_set`.

| | | |
|---|---|---|
| $x$ | $\in$ VAR | — variables |
| $\ell$ | $\in$ LOC | — memory locations |
| $b$ | $\in \{\texttt{false}, \texttt{true}\}$ | — Boolean values |
| $n$ | $\in \mathbb{Z}$ | — integer values |
| $v$ | $::= \mu\, f.\, \lambda\, x.\, e \mid \ell \mid \texttt{()} \mid b \mid n$ | — values |
| | $\mid \texttt{None} \mid \texttt{Some}\ v \mid (v, \ldots, v)$ | |
| $\odot$ | $\in \{\wedge, \vee, +, \times, \texttt{mod}, \ldots\}$ | — operators |
| $e$ | $::=$ | — pure expressions: |
| | $x \mid \mu\, f.\, \lambda\, x.\, e \mid e\ e \mid e \odot e$ | – $\lambda$-calculus with recursive functions |
| | $\mid \ell \mid \texttt{()} \mid b \mid n \mid \texttt{None} \mid \texttt{Some}\ e \mid (e, \ldots, e)$ | – primitive values and constructors |
| | $\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$ | – conditional |
| | $\mid \texttt{match}\ e\ \texttt{with Some}\ x \to e\ \texttt{|}\ \texttt{None} \to e$ | – option matching |
| | $\mid \texttt{let}\ (x, \ldots, x) = e\ \texttt{in}\ e$ | – tuple matching |
| | | — impure expressions: |
| | $\mid \texttt{fork}\ e$ | – spawning a new thread |
| | $\mid \texttt{array}_\alpha[e]\ e$ | – allocating a block |
| | $\mid e[e]_\alpha \mid e[e]_\alpha \leftarrow e \mid \texttt{CAS}\ e[e]\ e\ e$ | – accessing a cell |
| | $\mid \texttt{length}\ e$ | – requesting the length of a block |
| | | — syntactic sugar: |
| | $\mid \texttt{let rec}\ x\ x\ \ldots\ x = e\ \texttt{in}\ e$ | – definitions of recursive functions |
| | $\mid \texttt{let}\ x\ \ldots\ x = e\ \texttt{in}\ e \mid e\,;e$ | – definitions, sequencing |
| | $\mid \texttt{while}\ e\ \texttt{do}\ e\ \texttt{done}$ | – while loops |
| | $\mid \texttt{for}\ x\ \texttt{from}\ e\ \texttt{to}\ e\ \texttt{do}\ e\ \texttt{done}$ | – for loops |
| | $\mid \texttt{ref}_\alpha\ e$ | – allocating a reference |
| | $\mid !_\alpha e \mid e :=_\alpha e \mid \texttt{CAS}\ e\ e\ e$ | – accessing a reference |

Figure 2.3: The syntax of (an idealized fragment of) Multicore OCaml

### Operational semantics

The operational semantics of Multicore OCaml is shown in Figure 2.4. It is defined in two layers.

The "per-thread" reduction relation $e \xrightarrow{m} e', e'_1, \ldots, e'_n$ defined in Figures 2.4b and 2.4c describes how an expression $e$ takes a step and reduces to a new expression $e'$, possibly interacting with the memory subsystem via an event $m$, and possibly spawning a number of new threads $e'_1, \ldots, e'_n$.[5] Figure 2.4b presents the pure reduction rules, that is, the rules which do not involve an interaction with the memory subsystem (they are annotated with the silent event $\varepsilon$) and do not spawn new threads. Together with the rule for reducing under evaluation contexts (which are defined in Figure 2.4a), these rules form a standard small-step reduction semantics for right-to-left call-by-value $\lambda$-calculus. Figure 2.4c presents the reduction rules for the memory operations, which interact with the memory subsystem via a memory event, and the reduction rule for "fork", which spawns one new thread. From this point on, when $c = \ell[i]$ is a cell address, we allow ourselves to write $!_\alpha\, c$ for $\ell[i]_\alpha$, $c :=_\alpha v$ for $\ell[i]_\alpha \leftarrow v$ and $\texttt{CAS}\ c\ v_1\ v_2$ for $\texttt{CAS}\ \ell[i]\ v_1\ v_2$. Recall that cell addresses are not first-class values in our language.

Note that a successful compare-and-set (CAS) instruction and an atomic write instruction are both modeled by a memory event of the shape $\mathsf{rdwr}_{\mathrm{at}}(\_, \_, \_)$. Indeed, as per Dolan et al.'s memory model, an atomic write has both a "release" and an "acquire" effect.

The "thread-pool" reduction relation $\sigma; p \Longrightarrow \sigma'; p'$ defined in Figure 2.4d describes how the machine steps between two configurations $\sigma; p$ and $\sigma'; p'$. In such a configuration, $\sigma$ is a store, while $p$ is a thread pool, that is, a list of threads, where each thread $\langle e, \mathcal{W} \rangle$ is a pair of an expression $e$ and a view $\mathcal{W}$. The left-hand rule allows the per-thread execution system and the memory subsystem to synchronize on an event $m$. The right-hand rule shows how new threads are spawned; a newly-spawned thread inherits the view of its parent thread. Because a per-thread reduction step cannot both access memory and spawn new threads, these two rules suffice.

The operational semantics of Multicore OCaml is the reflexive transitive closure of the thread-pool reduction relation. It is therefore an interleaving semantics, albeit not a sequentially consistent semantics, as it involves a store whose behavior is nonstandard.

A machine configuration $\sigma; p$ is considered *stuck* if the thread pool $p$ contains at least one thread that cannot take a step yet has not reached a value. A stuck configuration represents an undesirable event, a runtime error. There are many ways of constructing stuck configurations: examples include applying a primitive operation to arguments of incorrect nature, attempting to call a value other than a function, and so on. All such errors are ruled out by the OCaml type system:[6] the execution of a well-typed program cannot lead to a stuck configuration. Although this claim does not appear to have been formally established, it seems clear that a syntactic proof of type soundness for ML with references (Wright and Felleisen, 1994) can be adapted to the semantics of Multicore OCaml.

A careful reader might note that a non-atomic read instruction (MEM-NA-READ) *could* potentially be stuck under certain circumstances. This could be the case, for instance, if the history $h$ is empty, or if the timestamp $\mathcal{W}(a)$ is too high, causing all write events in $h$ to be considered outdated. In reality, though, neither of these situations can arise, because only a

[5] In fact, a reduction step can either emit a memory event or spawn new threads, but not both. Also, the number of newly spawned threads is always zero or one.

[6] No changes to the type system are required by the move from OCaml to Multicore OCaml. The type `'a Atomic.t` is a new (primitive) abstract type.

$$
\begin{array}{lll}
K ::= \Box & & \text{— pure contexts} \\
\quad \mid e\ K \mid K\ v \mid e \odot K \mid K \odot v & & \\
\quad \mid \texttt{Some}\ K \mid (e, \ldots, e, K, v, \ldots, v) & & \\
\quad \mid \texttt{if}\ K\ \texttt{then}\ e\ \texttt{else}\ e & & \\
\quad \mid \texttt{match}\ K\ \texttt{with Some}\ x \to e\ \texttt{|}\ \texttt{None} \to e & & \\
\quad \mid \texttt{let}\ (x, \ldots, x) = K\ \texttt{in}\ e & & \\
 & & \text{— impure contexts} \\
\quad \mid \texttt{array}_\alpha[e]\ K \mid \texttt{array}_\alpha[K]\ v & & \\
\quad \mid e[K]_\alpha \mid K[v]_\alpha \mid e[e]_\alpha \leftarrow K \mid e[K]_\alpha \leftarrow v \mid K[v]_\alpha \leftarrow v & & \\
\quad \mid \texttt{CAS}\ e[e]\ e\ K \mid \texttt{CAS}\ e[e]\ K\ v \mid \texttt{CAS}\ e[K]\ v\ v \mid \texttt{CAS}\ K[v]\ v\ v & & \\
\quad \mid \texttt{length}\ K & &
\end{array}
$$

(a) Evaluation contexts

$$\frac{e \xrightarrow{m} e', e_1', \ldots, e_n'}{K[e] \xrightarrow{m} K[e'], e_1', \ldots, e_n'} \qquad (\mu\ f.\ \lambda\ x.\ e)\ v \xrightarrow{\varepsilon} e[\mu\ f.\ \lambda\ x.\ e/f][v/x] \qquad \frac{v_1 \odot v_2 = v'}{v_1 \odot v_2 \xrightarrow{\varepsilon} v'}$$

$$\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \xrightarrow{\varepsilon} e_1 \qquad (\texttt{match Some}\ v\ \texttt{with Some}\ x \to e_1\ \texttt{|}\ \texttt{None} \to e_2) \xrightarrow{\varepsilon} e_1[v/x]$$

$$\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \xrightarrow{\varepsilon} e_2 \qquad (\texttt{match None with Some}\ x \to e_1\ \texttt{|}\ \texttt{None} \to e_2) \xrightarrow{\varepsilon} e_2$$

$$\texttt{let}\ (x_1, \ldots, x_n) = (v_1, \ldots, v_n)\ \texttt{in}\ e \xrightarrow{\varepsilon} e[v_1/x_1] \ldots [v_n/x_n]$$

(b) Thread-local reduction: $e \xrightarrow{m} e', e_1', \ldots, e_n'$ — pure steps

$$\texttt{array}_{\text{na}}[v]\ n \xrightarrow{\mathsf{alloc}_{\text{na}}(\ell,v,n)} \ell \qquad !_{\text{na}}\ a \xrightarrow{\mathsf{rd}_{\text{na}}(a,v)} v \qquad a :=_{\text{na}} v \xrightarrow{\mathsf{wr}_{\text{na}}(a,v)} ()$$

$$\texttt{array}_{\text{at}}[v]\ n \xrightarrow{\mathsf{alloc}_{\text{at}}(\ell,v,n)} \ell \qquad !_{\text{at}}\ A \xrightarrow{\mathsf{rd}_{\text{at}}(A,v)} v \qquad A :=_{\text{at}} v \xrightarrow{\mathsf{rdwr}_{\text{at}}(A,v_1,v)} ()$$

$$\frac{v_0 \neq v_1}{\texttt{CAS}\ A\ v_1\ v_2 \xrightarrow{\mathsf{rd}_{\text{at}}(A,v_0)} \texttt{false}} \qquad \texttt{CAS}\ A\ v_1\ v_2 \xrightarrow{\mathsf{rdwr}_{\text{at}}(A,v_1,v_2)} \texttt{true}$$

$$\texttt{length}\ \ell \xrightarrow{\mathsf{len}(\ell,n)} n \qquad \texttt{fork}\ e \xrightarrow{\varepsilon} (), e$$

(c) Thread-local reduction: $e \xrightarrow{m} e', e_1', \ldots, e_n'$ — impure steps

$$\frac{e \xrightarrow{m} e' \qquad \sigma; \mathcal{W} \xrightarrow{m} \sigma'; \mathcal{W}'}{\begin{array}{l}\sigma; p_1, \langle e, \mathcal{W}\rangle, p_2 \\ \implies \sigma'; p_1, \langle e', \mathcal{W}'\rangle, p_2\end{array}} \qquad \frac{e \xrightarrow{\varepsilon} e', e_1', \ldots, e_n'}{\begin{array}{l}\sigma; p_1, \langle e, \mathcal{W}\rangle, p_2 \\ \implies \sigma; p_1, \langle e', \mathcal{W}\rangle, p_2, \langle e_1', \mathcal{W}\rangle, \ldots, \langle e_n', \mathcal{W}\rangle\end{array}}$$

(d) Thread-pool reduction: $\sigma; p \Longrightarrow \sigma'; p'$

Figure 2.4: Small-step operational semantics of Multicore OCaml

subset of *well-formed* machine configurations can be reached. In a well-formed configuration, every non-atomic cell ever allocated must have a nonempty history, and no thread can get hold of an unallocated cell, thereby removing the concern that $h$ might be empty. Furthermore, a well-formed configuration must satisfy the following *global view invariant*: there exists a *global view* $\mathcal{G}$ such that:

1. every thread's view $\mathcal{W}$ is contained in $\mathcal{G}$, that is, $\forall \langle e, \mathcal{W} \rangle \in p.\ \mathcal{W} \sqsubseteq \mathcal{G}$ holds, where $p$ is the thread pool;
2. the view of every atomic cell is contained in $\mathcal{G}$, that is, $\forall A, v, \mathcal{V}.\ \sigma(A) = \langle v, \mathcal{V} \rangle \implies \mathcal{V} \sqsubseteq \mathcal{G}$ holds;
3. the global view maps every non-atomic cell to a timestamp that exists in the history of this cell, that is, $\forall a, h.\ \sigma(a) = h \implies \mathcal{G}(a) \in \mathrm{dom}(h)$ holds;
4. the global view maps every currently unallocated cell to the minimal timestamp, that is, $\forall c \notin \mathrm{dom}(\sigma).\ \mathcal{G}(c) = 0$ holds.

In fact, the global view $\mathcal{G}$ is the join of all thread's views, and maps every allocated non-atomic cell to the maximal timestamp of its history. In other words, there always exists a thread which has observed the most recent write to a given cell—the thread that performed that write, at least. This fact, however, is not needed in our proofs. Besides, it would not hold anymore if we allowed thread deletion.

One can check that the above set of properties is indeed an invariant (i.e., it is preserved by every reduction step) and that this invariant implies that a non-atomic read instruction at an allocated non-atomic cell cannot be stuck. In fact, in the next chapter, we must prove this claim, and exploit this invariant, otherwise we would be unable to prove that our logic is sound: a sound program logic must guarantee that a verified program cannot get stuck. Thus, when we instantiate Iris for Multicore OCaml, we build the global view invariant into our "state interpretation" invariant (§3.1), and when we establish the reasoning rule for non-atomic read instructions (rule BASE-NA-READ in Figure 3.2), we exploit it (§3.4).

## 2.3 Program verification

Program verification is the act of ensuring that a program behaves as intended. In this dissertation we are interested in proving the *functional correctness* of programs, that is, proving that they produce expected outputs in relation with their inputs according to a given formal *specification*. An even more fundamental property is program *safety*: a program $e$ is safe, with respect to an operational semantics, if no reduction chain starting from $e$ leads to a stuck program. A stuck program can be regarded as a crash, an error that definitely must not happen.

A specification is typically expressed as a *Hoare triple* (Hoare, 1969), that is, a logical assertion of the following form:[7]

$$\{P\}\, e\, \{\lambda x.\, Q\}$$

[7]Note that the formalism we use takes several now-standard departures from Hoare's original logic (Hoare, 1969): whereas Hoare logic was axiomatic, ours is derived from our programming language's operational semantics and ensures program safety even for non-terminating programs. Furthermore, to accommodate for a functional language, the subject of triples are expressions rather than statements, and the postcondition features a binder which captures the resulting value of that expression.

In this notation, $e$ is a program fragment—source code—and $P$ and $Q$ are logical assertions which describe what should hold of the computer state—typically, the contents of the memory—before and after program $e$ has run. The syntax $\lambda x.\, Q$ binds the logical variable $x$ in the formula $Q$. Such a triple is intuitively read as: if program $e$ runs from a state that satisfies *precondition* $P$, then $e$ is safe; furthermore, if it terminates, then its return value $x$ and the state it leaves the computer in satisfy *postcondition* $Q$.

In this reading, program termination is an assumption: in the situation when $e$ does not terminate, only safety is guaranteed. This is called *partial* correctness. *Total* correctness is an alternative, stronger reading which guarantees termination rather than assuming it. Proving program termination, however, is out of the scope of this dissertation, as is proving deadlock-freedom or starvation-freedom—the latter being a liveness property.

To prove a functional specification, we have at hand a set of reasoning rules about Hoare triples. The first two rules in Figure 2.5 (ignoring FRAME for now) are fundamental rules, that do not depend on the actual programming language. Rule CONSEQUENCE states that, from a specification, we can deduce any weaker specification, that is, one with a stronger precondition and a weaker postcondition. Rule BIND implies that, to prove the specification of a program, we can chain a specification for its head reduction step and a specification for the resultant program fragment; this allows to reason about an execution step by step. To this effect, in addition to these two structural rules, one has rules that describe the behavior of each reduction step of the programming language. These rules are guided by syntax and reflect the operational semantics. For example, assuming that our language has Boolean values and an "if-then-else" construct, one would typically have rules resembling these two:

$$\frac{\{P\}\, e_1\, \{\lambda x.\, Q\}}{\{P\}\, \texttt{if true then}\ e_1\ \texttt{else}\ e_2\, \{\lambda x.\, Q\}} \qquad \frac{\{P\}\, e_2\, \{\lambda x.\, Q\}}{\{P\}\, \texttt{if false then}\ e_1\ \texttt{else}\ e_2\, \{\lambda x.\, Q\}}$$

With an expressive enough logical language, one can describe a program's contract very precisely. Hence Hoare-style functional correctness is one of the strongest forms of program verification, and requires a great deal of human intervention. Thus it is a work *per se* to come up with a satisfying specification: one that both is provable and captures the important facts while, ideally, remaining simple enough to manipulate.

An important feature is *composability* of specifications. Indeed, software code bases are large and rely extensively on re-use of conceptually independent components, such as libraries and functions. To scale verification to these code bases, it is of utmost importance that we are able to specify and verify an independent component in isolation, once and for all, and then re-use its specification when verifying each part of the code that use it. In this work, we achieve composability thanks to (concurrent) separation logic, and also thanks to a mechanism of monotonic views of memory. We now introduce the former; an extensive presentation of the latter is done in Chapter 4.

### 2.3.1 Separation Logic

In the perspective of composability, the original Hoare logic is inappropriate when applied to programs with heap-allocated locations. Indeed, in Hoare's framework, the specification of a component must describe in its precondition and postcondition the entirety of memory, including the portion of memory that is never accessed by the component. Moreover, it is often necessary

$$\textsc{consequence}\ \frac{P \vdash P' \qquad \{P'\}\, e\, \{\lambda x.\, Q'\} \qquad \forall x.\, Q' \vdash Q}{\{P\}\, e\, \{\lambda x.\, Q\}} \qquad \textsc{bind}\ \frac{\{P\}\, e\, \{\lambda x.\, Q\} \qquad \forall x.\, \{Q\}\, K[x]\, \{\lambda y.\, R\}}{\{P\}\, K[e]\, \{\lambda y.\, R\}}$$

$$\textsc{frame}\ \frac{\{P\}\, e\, \{\lambda x.\, Q\}}{\{R * P\}\, e\, \{\lambda x.\, Q * R\}}$$

Figure 2.5: Structural reasoning rules about Hoare triples in Separation Logic

to assert that variables used in the program are disjoint, that is, they do not alias each other, which accounts for a number of side conditions that grows quadratically with the number of variables.

Separation Logic (Reynolds, 2002) constitutes an answer to this lack of composability. In Separation Logic, each assertion holds for a given heap *fragment*, called its footprint. Formally speaking, a Separation Logic assertion is a predicate over heap fragments. In addition to the classical conjunction $P \wedge Q$, Separation Logic features a *separating conjunction* $P * Q$ which asserts that $P$ and $Q$ hold on *disjoint* heap fragments; the footprint of $P * Q$ is then the union of the footprints of both conjuncts.

This layer of abstraction relieves the user from dealing with disjointness side conditions, and it enables local reasoning: rule FRAME (Figure 2.5) holds in Separation Logic. Thanks to this rule, one can prove a specification about a program $e$ with just the minimal footprint $(P, Q)$ that is relevant to $e$, then use this specification in varying contexts, $R$ representing the rest of the memory—the *frame*. Separation is crucial: it prevents the assertions $P$ and $Q$ from being falsified by the context. A similar rule stated with classical conjunctions instead of separating conjunctions would not be valid.

### Concurrent Separation Logic

We are now interested in proving the correctness of concurrent programs, that is, programs where several threads can access a data structure concurrently. Let us assume for now that the memory model is sequentially consistent (Lamport, 1979).

It turns out that the ideas of Separation Logic extend quite naturally to this setting. Separation achieves non-interference between threads, a feature which earlier attempts (Owicki and Gries, 1976) missed. For instance, in a concurrent setting, the validity of the following Hoare triple is not trivial, because another thread might modify location $x$ concurrently.

$$\{x \rightsquigarrow \_\}\, x \coloneqq v\, \{\lambda \_.\, x \rightsquigarrow v\}$$

In Separation Logic, the assertion $x \rightsquigarrow \_$ asserts the unique ownership of $x$; if we further arrange so that threads operate on separate portions of the memory, then no other thread is allowed to operate on $x$ when we own $x \rightsquigarrow \_$, so the triple above is valid.

In a seminal work that builds on this idea, Brookes (2004) and O'Hearn (2007) devised Concurrent Separation Logic, an extension of Separation Logic which enables to reason about programs where several threads can access a same piece of memory. The original Concurrent

Separation Logic achieves sharing through hard-wired "conditional critical regions", that is, code sections guarded by a global lock. However, it has spawned a variety of descendants lifting this limitation and pushing further the applicability of such separation logics. Brookes and O'Hearn (2016) provide a survey of these descendants. In this dissertation, we use one of them, called Iris.

### 2.3.2 Iris

Iris (Jung et al., 2018b) is a fine-grain concurrent separation logic framework. Its starting idea is the observation that ghost state, implemented by parameterizable partial monoids, gives rise to a particularly expressive logic, able to subsume many previous proposals. Thus, at the core of Iris is a notion of ghost state—initially partial commutative monoids, now a generalization called cameras.

Like earlier logics, Iris tackles fine-grain concurrency through the use of *invariants*, that is, assertions whose resources are available by all threads equally and, thus, can never be revoked.

A distinctive trait of Iris invariants is that they are impredicative: the fact that an invariant holds is itself a first-class assertion, which can be put inside an invariant. This feature has implications on the logic itself: to tackle impredicativity, Iris is *step-indexed*, which means that it features a "later" modality $\triangleright P$ (Jung et al., 2018b, §5.5). It is not necessary to understand the details of step-indexing for the scope of our work; when reading this dissertation, later modalities may be simply ignored.

A strength of Iris is that it is implemented and mechanically proven sound in Coq. Besides, the Coq development ships with a deep embedding of the logic and provides a "Proof Mode", an interactive way of carrying proofs within the Iris logic, in a manner similar to how one would prove pure Coq goals (Krebbers et al., 2017). Lastly, Iris can take advantage of the expressiveness of the host logic: there is an injection from the type PROP of Coq propositions—so-called *pure* assertions, which do not own any resource—to the type IPROP of Iris assertions. In this dissertation, we leave that injection implicit.

It is worth noting that Iris has a notion of Hoare triples as first-class assertions, which may own resources.

In this section, we give a simplified overview of the Iris notions that we use throughout this dissertation. For a complete presentation of Iris, we refer the interested reader to Jung et al. (2018b).

#### Ghost state and ghost updates

In the original Separation Logic, assertions are predicates over heap fragments. This idea can be generalized to other notions of ownable *resources*: file handlers, printers, permissions to perform some actions... Ghost state in Iris is a flexible tool for defining custom resources. In fact, even invariants and the heap are encoded as pieces of ghost state. Ghost state takes values from algebraic structures called *cameras*. In this dissertation, we only give an informal description of cameras, whose actual definition is quite technical; we refer the interested reader to Jung et al. (2018b, §2.1, §4.4).

Ghost values are assigned to *ghost variables*: for any ghost variable $\gamma$ and any element $x$ of any camera, there is a separation logic assertion $\boxed{x}^{\gamma}$, whose intuitive reading is that we own a fragment of $\gamma$ and that this fragment has value $x$. Unlike what happens with a traditional

points-to assertion, the ownership *and value* of a ghost variable can be split into fragments. A camera structure comes with an associative and commutative composition law $(\cdot)$ which dictates how ghost state can be split and combined, according to the following rule (where $\dashv\vdash$ is the logical equivalence):

$$\textsc{GhostOp}\quad \boxed{x \cdot y}^{\gamma} \dashv\vdash \boxed{x}^{\gamma} * \boxed{y}^{\gamma}$$

This composition law is partial, that is, the composition $x \cdot y$ is not necessarily *valid* for all pairs of elements $x$ and $y$. In other words, some pairs of resources may be incompatible.

We can update fragments of the ghost state but not in arbitrary ways: every update must preserve the validity of the whole ghost state, that is, the new value of the fragment must be compatible with any other fragment that might exist prior to the update; these other fragments are left unchanged. Thus, we say that a *frame-preserving update* is possible from $x$ to $y$ (both elements of some camera $M$), and we write $x \rightsquigarrow y$, when any element $z$ of $M$ that is compatible with $x$ (that is, such that the composition $x \cdot z$ is valid) is also compatible with $y$ (Jung et al., 2018b, §2.1). Elements $z$ are the possible "frames".

In the Iris logic, changes to the ghost state go through an *update modality*: the assertion $\Rightarrow\!\!\!|\, P$ is intuitively read as "after some ghost update is performed, the assertion $P$ will hold". We also define the notation $P \Rrightarrow Q$ as sugar for $P \mathrel{-\!\!*} \Rightarrow\!\!\!|\, Q$, that is, "we can update the assertion $P$ to the assertion $Q$". Iris has the following reasoning rules for manipulating ghost state within the logic:

$$\textsc{GhostAlloc}\quad \frac{x \in M}{\Rightarrow\!\!\!|\, \exists \gamma.\, \boxed{x}^{\gamma}} \qquad \textsc{GhostUpdate}\quad \frac{x, y \in M \qquad x \rightsquigarrow y}{\boxed{x}^{\gamma} \Rrightarrow \boxed{y}^{\gamma}}$$

Rule GhostAlloc says that, by performing an update to the ghost state, we can initialize a fresh ghost variable $\gamma$ to any ghost value $x$. Rule GhostUpdate says that, if there is a frame-preserving update from $x$ to $y$, then we can transform the value of a fragment of ghost state from $x$ to $y$, again by performing an update to the ghost state.

The Iris toolkit provides several useful camera constructions, presented now.

**Exclusive camera** The *exclusive camera* (Jung et al., 2018b, §3.1) $\textsc{Ex}(X)$ over some type[8] $X$ has $X$ as its carrier. We denote by $\mathsf{ex}$ the injection from $X$ to $\textsc{Ex}(X)$. The composition law is such that there cannot be more than one fragment of the resource in existence: that is, the composition $\mathsf{ex}(x) \cdot \mathsf{ex}(y)$ is never valid. It follows that elements of this camera can be updated freely: $\mathsf{ex}(x) \rightsquigarrow \mathsf{ex}(y)$ for any $x$ and $y$.

More generally, an element $x$ of a camera $M$ is said to be *exclusive* when it cannot be composed with any element of $M$. In that case, we have $x \rightsquigarrow y$ for any element $y \in M$.

**Agreement camera** The *agreement camera* (Jung et al., 2018b, §3.1, §4.3) $\textsc{Ag}(X)$ over some type[8] $X$ has $X$ as its carrier. We denote by $\mathsf{ag}$ the injection from $X$ to $\textsc{Ag}(X)$. The composition law achieves agreement on some element of $X$: that is, $\mathsf{ag}(x) \cdot \mathsf{ag}(x)$ is $\mathsf{ag}(x)$, and $\mathsf{ag}(x) \cdot \mathsf{ag}(y)$ is invalid if $x \neq y$. This camera allows no non-trivial update.

[8] Technically speaking, $X$ must be an OFE, short for *ordered family of equivalences*, which is a step-indexed analog of the usual notion of type (Jung et al., 2018b, §4.2).

**Authoritative camera** The *authoritative camera* (Jung et al., 2018b, §6.3.3) $\textsc{Auth}(M)$ over some (unitary) camera $M$ has both *authoritative elements* of the form $\bullet\, x$ and *fragmentary elements* of the form $\circ\, y$, where $x$ and $y$ are elements of $M$; it also has elements of the form $\bullet\circ\,(x, y)$, so as to represent compositions of elements of the other two forms. An authoritative element represents the full knowledge of something, and cannot be split: the composition $(\bullet\, x_1) \cdot (\bullet\, x_2)$ is never valid. A fragmentary element represents partial knowledge, and can be split and joined: the composition $(\circ\, y_1) \cdot (\circ\, y_2)$ is defined as $\circ\,(y_1 \cdot y_2)$. Because a fragment must be a part of the whole, the composition $(\bullet\, x) \cdot (\circ\, y)$ is valid if and only if $y$ is *included* in $x$, that is, if there exists $z$ such that $x = y \cdot z$ (which we denote as $y \preccurlyeq x$).

To describe allowed updates in an authoritative camera, it is useful to define a notion of *local update* (Iris developers and contributors, 2022, §4.8): we say that we can perform a local update from $x$ and $y$ to $x'$ and $y'$, and we write $(x, y) \rightsquigarrow_{\mathrm{L}} (x', y')$, when every element $z$ such that $x = y \cdot z$ satisfies $x' = y' \cdot z$. We then have the following rule for updating the authority along with a fragment:

$$\frac{(x, y) \rightsquigarrow_{\mathrm{L}} (x', y')}{\bullet\, x \cdot \circ\, y \rightsquigarrow \bullet\, x' \cdot \circ\, y'}\ \textsc{AuthUpdate}$$

In other words, we can update a part of the whole from $y$ to $y'$ if, while doing so, we preserve the remaining parts (the "frame") $z$.

**Map camera** The *finite map camera* $X \xrightarrow{\mathsf{fin}} M$ from some arbitrary type $X$ to some camera $M$ has finite maps from $X$ to $M$ as its elements; composition and updates are pointwise.

**Product camera** The *product camera* $M_1 \times M_2$ of two cameras $M_1$ and $M_2$ has pairs as its elements; composition and updates are pointwise.

**Sum camera** The *sum camera* $M_1 + M_2$ (Jung et al., 2018b, §3.1) of two (unitary) cameras $M_1$ and $M_2$ has the sum type as its carrier. We denote by $\mathsf{inl}$ and $\mathsf{inr}$ the two associated injections. Composition in the sum camera is given by those of the two underlying cameras: $\mathsf{inl}(x) \cdot \mathsf{inl}(y) = \mathsf{inl}(x \cdot y)$ and $\mathsf{inr}(x) \cdot \mathsf{inr}(y) = \mathsf{inr}(x \cdot y)$; the composition $\mathsf{inl}(x_1) \cdot \mathsf{inr}(x_2)$ is invalid.

This camera supports pointwise updates: if $x \rightsquigarrow x'$ then $\mathsf{inl}(x) \rightsquigarrow \mathsf{inl}(x')$, and similarly with $\mathsf{inr}$. Under certain circumstances, we can also switch from one branch to the other: if $x \in M_1$ is exclusive, then so is $\mathsf{inl}(x)$, hence $\mathsf{inl}(x) \rightsquigarrow \mathsf{inr}(y)$ for any $y \in M_2$; and similarly from $\mathsf{inr}$ to $\mathsf{inl}$.

**Fraction camera** The *fraction camera* $\textsc{Frac}$ has the fractions $q \in \mathbb{Q}_{>0}$ as its elements and addition as its composition law. Updates are unrestricted. Using that camera, we can encode fractional ownership of resources (Jung et al., 2018b, §6.3.3, §7.3). Note that the fraction 0 is excluded; hence, if our encoding of fractional ownership of a resource constrains the fraction to be at most 1, then a fragment of that resource with the fraction 1 cannot be composed with any other fragment. In other words, the fraction 1 represents exclusive ownership.

**The global ghost state** The Iris logical framework is parameterized by a single camera structure, called the *global ghost state* (Jung et al., 2018b, §3.2). However, thanks to the product

camera construction, we can build a global ghost state as the product of however many cameras we need. Hence, in practice, when we say that we use a given camera, we simply assert that the global ghost state is a product which contains at least the camera we are interested in.

### Persistence

Ghost state in Iris can encode ownership of *resources* that are not duplicable in general—that is, these resources can be owned only once and they can be revoked—but it is also able to represent shareable *knowledge*, that remains true forever. More precisely, some camera elements $x$ are idempotent, that is, they do satisfy the equation $x = x \cdot x$, which implies that the assertion $\boxed{x}^\gamma$ is duplicable, that is, the entailment $\boxed{x}^\gamma \vdash \boxed{x}^\gamma * \boxed{x}^\gamma$ holds. Examples include:

- $\mathsf{ag}(x)$ for any $x$;
- $\circ\, x$ when $x$ is idempotent;
- $\{k_1 \mapsto x_1; \ldots; k_n \mapsto x_n\}$ when $x_1, \ldots, x_n$ are idempotent;
- $(x, y)$ when $x$ and $y$ are idempotent.

More generally, Iris has a notion of *persistent* assertions (Jung et al., 2018b, §2.3). Once true, these assertions hold true forever. They are duplicable, that is, the entailment $P \vdash P * P$ holds for any persistent assertion $P$.[9] Examples of persistent assertions include:

- any pure Coq proposition, that is, $P$ for any proposition $P$ of type PROP; for instance, an equality between Coq values;
- ownership of idempotent ghost state, $\boxed{x}^\gamma$ when $x = x \cdot x$;
- knowledge of an invariant, $\boxed{P}$ for any assertion $P$ (see next subsection);
- any Hoare triple, $\{P\}\, e\, \{Q\}$;
- $P * Q$ when $P$ and $Q$ are persistent;
- $\exists x.\, P$ and $\forall x.\, P$ when, for all $x$, $P$ is persistent.

Besides, Iris has a logic modality for asserting that an assertion holds *persistently*, that is, it "holds without asserting any exclusive ownership" (Jung et al., 2018b, §5.3). For a given assertion $P$, this modality is denoted as $\Box P$. It satisfies several reasoning rules, here are a few:

$$\frac{\Box P}{P}\ \textsc{PersistElim} \qquad \frac{\Box P}{\Box\Box P}\ \textsc{PersistIdemp} \qquad \frac{(\Box P) \wedge Q}{(\Box P) * Q}\ \textsc{PersistConj}$$

$$\frac{\forall x.\ \Box P}{\Box \forall x.\, P}\ \textsc{PersistForall} \qquad \frac{\exists x.\ \Box P}{\Box \exists x.\, P}\ \textsc{PersistExists}$$

For any assertion $P$, the assertion $\Box P$ is persistent. Rule PERSISTELIM shows that $\Box P$ is stronger than $P$. In fact, $P$ is persistent if and only if the reciprocal entailment $P \vdash \Box P$ also holds. Adding the modality is helpful when $P$ only contains duplicable resources but it is not obvious from the syntax of $P$ that $P$ is persistent.

[9] Persistence in Iris is a strictly stronger notion than duplicability. See (Jung et al., 2018b, §2.3) for details.

For instance, Hoare triples in Iris are in fact syntactic sugar for weakest-preconditions assertions, defined like so (Jung et al., 2018b, §6):

$$\{P\}\, e\, \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$$

### Invariants

An invariant in Iris (Jung et al., 2018b, §2.2) is an assertion that shares the ownership of a given resource to all threads equally. This resource can be accessed by any thread, in some restricted way. Because it must remain available to other threads, an invariant can never be falsified: it holds true forever, that is, it is persistent. Selected rules about invariants are shown below.

$$\textsc{InvAlloc}\ \frac{\triangleright I}{\Rightarrow\!\boxed{I}} \qquad \textsc{HoareInv}\ \frac{\{\triangleright I * P\}\, e\, \{\lambda x.\, Q * \triangleright I\} \qquad e \text{ runs for at most one step of execution}}{\boxed{I} \vdash \{P\}\, e\, \{\lambda x.\, Q\}}$$

Rule InvAlloc shows how to form an invariant: for any assertion $I$, we can consume $I$ in order to form a new invariant that we denote as $\boxed{I}$. In other words, we give up on our exclusive ownership of $I$ and place it in an invariant where it is owned by everyone. Then, rule HoareInv shows how to use it: we can *open* it around a program fragment $e$ and obtain its resource $I$ (under a "later" modality) for the duration of the execution of $e$. We must reestablish the invariant after $e$ has completed.

To ensure soundness in the face of concurrency, Iris imposes a strict constraint: $e$ cannot run for more than one step of execution. It is as if there was a global runtime lock which would protect all elementary steps of computation, and whose associated invariant would be the conjunction of all Iris invariants in existence.

Here we have omitted a technical detail which is needed for soundness: in the actual logic, invariants and Hoare triples are annotated with "namespaces" in order to forbid that an invariant be opened several times simultaneously.

# Chapter 3

# A low-level logic: BaseCosmo

In this chapter, we set up a program logic for Multicore OCaml, based on the operational semantics presented in the previous chapter. To do so, we rely on Iris (Jung et al., 2018b). Iris is not tied to a particular programming language or calculus. Its lower layer, the Iris base logic (Jung et al., 2018b, §3–5), is a purely logical construction, of which we have given a short introduction in §2.3.2. Its upper layer, the Iris program logic (Jung et al., 2018b, §6–7), is parameterized with a programming language. In order to instantiate it, a client must provide the following information (Jung et al., 2018b, §7.3).

- A set of "expressions".
- A subset of "values".
- A set of machine "states". For instance, a state might be a store, that is, a map whose domain is the set of all currently allocated memory locations.
- An operational semantics, in the form of a "per-thread step relation". This relation relates an expression and a state to an expression and a state and a list of expressions, which represent newly-spawned threads.
- A "state interpretation" predicate $S : \textsc{State} \to \textsc{iProp}$.[1] This predicate represents a global invariant about the machine state. It typically relates the state with a piece of ghost state whose structure is chosen by the client so as to justify splitting the ownership of the machine state in certain ways.

Once the client has provided this information, the framework yields a program logic, that is,

- A weakest-precondition predicate $\mathsf{wp}\ e\ \{\Phi\}$.
- A Hoare triple predicate $\{P\}\, e\, \{\Phi\}$, which is just sugar for $\square\,(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Phi\})$.
- An adequacy theorem (Jung et al., 2018b, §6.4, §7.4), which roughly state that if a closed program $e$ satisfies $\mathsf{wp}\ e\ \{\Phi\}$ then it is safe to run, that is, its execution will not lead to a stuck configuration; moreover, its final configurations will satisfy the postcondition $\Phi$.

[1] Recall that iProp is the type of Iris assertions.

- A set of programming-language-independent deduction rules for triples (§2.3.1 and §2.3.2). These include the consequence rule, the frame rule, rules for allocating and updating ghost state, rules for setting up and exploiting invariants, and so on.

It is then up to the client to perform extra programming-language-specific work, namely:

- Define programming-language-specific assertions, such as "points-to" assertions.
- Prove entailment laws describing, e.g., how points-to assertions can be split and combined.
- Establish programming-language-specific deduction rules for triples, e.g., rules that give triples for reading and writing memory cells.

We now apply this recipe to Multicore OCaml. This yields BaseCosmo, a logic for Multicore OCaml.

## 3.1 Instantiating Iris for Multicore OCaml

We begin instantiating Iris as follows:

- An "expression" is a pair $\langle e, \mathcal{V} \rangle$ of a Multicore OCaml expression and a view.
- Accordingly, a "value" is a pair $\langle v, \mathcal{V} \rangle$ of a Multicore OCaml value and a view.
- A "state" is a store $\sigma$.
- The "per-thread step relation" is as defined in Figure 2.4c.

To complete this instantiation, there remains to define a suitable "state interpretation" $S$. In choosing this definition, we have a great deal of freedom. Our choice is guided by several considerations, including the manner in which we wish to allow splitting the ownership of the state, and the invariants about the state that we wish to keep track of. In the present case, we have the following three independent concerns in mind.

— We wish to allow splitting the ownership of memory cells (be it atomic or non-atomic) under a standard "fractional permissions" regime (Boyland, 2003).

— We wish to have a persistent assertion reflecting the knowledge that a given memory block has a certain length.

— We need to keep track of the global view invariant (§2.2.2) enjoyed by the operational semantics of Multicore OCaml, because this invariant is required to justify that a non-atomic read instruction at an allocated non-atomic cell cannot be stuck.

To achieve these goals, we define our state interpretation $S$ with well-chosen camera structures. We use standard camera constructions presented in §2.3.2. We delay an explanation of the definition to §3.3.

- Let $\gamma_{\text{store}}$ be a ghost variable storing an element of $\textsc{Auth}(\textsc{StoreCamera})$, where we define the camera of stores as:

$$\textsc{StoreCamera} \triangleq \textsc{CellAddr} \xrightarrow{\text{fin}} (\textsc{Ag}(\textsc{CellContents}_{\text{na}}) + \textsc{Ag}(\textsc{CellContents}_{\text{at}})) \times \textsc{Frac}$$

  We encode a physical store $\sigma$ as an element of this camera as follows:

$$\begin{array}{rcll} \text{st}(\sigma) & \triangleq & \{\ell[i] \mapsto (\mathsf{inl}(\mathsf{ag}(h)), 1) & \mid \ell \in \text{dom}(\sigma) \wedge 0 \le i < |\sigma(\ell)| \wedge \sigma(\ell[i]) = \mathsf{inl}(h)\} \\ & \uplus & \{\ell[i] \mapsto (\mathsf{inr}(\mathsf{ag}(\langle v, \mathcal{V}\rangle)), 1) & \mid \ell \in \text{dom}(\sigma) \wedge 0 \le i < |\sigma(\ell)| \wedge \sigma(\ell[i]) = \mathsf{inr}(\langle v, \mathcal{V}\rangle)\} \end{array}$$

  We use this piece of ghost state to implement points-to assertions (§3.3.1, §3.3.2).

- Let $\gamma_{\text{len}}$ be a ghost variable storing an element of $\textsc{Auth}(\textsc{Loc} \xrightarrow{\text{fin}} \textsc{Ag}(\mathbb{Z}))$. This camera allows us to encode the lengths of the blocks of the physical store $\sigma$ as follows:

$$\text{len}(\sigma) \triangleq \{\ell \mapsto \mathsf{ag}(|\sigma(\ell)|) \mid \ell \in \text{dom}(\sigma)\}$$

  We use this piece of ghost state to implement length assertions (§3.3.3).

- Let $\gamma_{\text{gv}}$ be a ghost variable storing an element of $\textsc{Auth}(\textsc{View})$, where we equip the type of views with a camera structure by taking the join operation $\sqcup$ as the composition. This composition law is idempotent. This piece of ghost state allows us to keep track of the global view $\mathcal{G}$, which we use to implement view validity (§3.3.4).

- We define the state interpretation as the following $\textsc{iProp}$ assertion:

$$S(\sigma) \triangleq \boxed{\bullet\, \text{st}(\sigma)}^{\gamma_{\text{store}}} * \boxed{\bullet\, \text{len}(\sigma)}^{\gamma_{\text{len}}} * \exists \mathcal{G}.\, * \begin{cases} \boxed{\bullet\, \mathcal{G}}^{\gamma_{\text{gv}}} \\ \forall A, v, \mathcal{V}.\, \sigma(A) = \langle v, \mathcal{V}\rangle \implies \mathcal{V} \sqsubseteq \mathcal{G} \\ \forall a, h.\, \sigma(a) = h \implies \mathcal{G}(a) \in \text{dom}(h) \\ \forall c \notin \text{dom}(\sigma).\, \mathcal{G}(c) = 0 \end{cases}$$

## 3.2 Soundness of BaseCosmo

Having instantiated Iris for the operational semantics of Multicore OCaml (§2.2.2) and for a state interpretation of our choosing (previous section), we obtain a weakest-precondition predicate $\mathsf{wp}\ \langle e, \mathcal{V}\rangle\ \{\Phi\}$; a triple $\{P\}\ \langle e, \mathcal{V}\rangle\ \{\Phi\}$, satisfying a set of programming-language-independent deduction rules (Figure 2.5); and an adequacy theorem, stated now, which guarantees that this triple is sound.

Triples conform to the informal interpretation given in §2.3. In a triple $\{P\}\ \langle e, \mathcal{W}\rangle\ \{\Phi\}$, the precondition $P$ describes the resources required in order to safely execute the expression $e$ on a thread whose view is $\mathcal{W}$. The postcondition $\Phi$, which takes a pair $\langle v', \mathcal{W}'\rangle$ of a value and a new view as an argument, describes the updated resources that exist at the end of this execution, if it terminates. This is thus a logic of partial correctness, as defined in §2.3. This claim is made formal by the following theorem, which we establish by relying on Iris's generic adequacy theorem (Jung et al., 2018b, §7.4).

A configuration $\sigma; p$ is said to be *safe* if there is no reduction sequence $\sigma; p \Longrightarrow^* \sigma'; p'$ such that $\sigma'; p'$ is stuck (as defined on p. 20). A configuration $\sigma; \langle e, \mathcal{V}\rangle$ is said to be *adequate* with

respect to some pure predicate $\varphi : \textsc{Val} \times \textsc{View} \to \textsc{Prop}$ if it is safe and, for any reduction sequence $\sigma; \langle e, \mathcal{V} \rangle \Longrightarrow^* \sigma'; \langle v', \mathcal{V}' \rangle, p'$ such that $v'$ is a value, the proposition $\varphi(v', \mathcal{V}')$ holds. Note that this assertion is *pure*, that is, it lives in the host logic $\textsc{Prop}$ (§2.3.2).

Recall that $\varnothing$ is the empty view and $\varnothing$ is the empty store.

**Theorem 1 (Adequacy of BaseCosmo)** *For any expression $e \in \textsc{Expr}$ and any pure predicate $\varphi : \textsc{Val} \times \textsc{View} \to \textsc{Prop}$, if the entailment $\vdash \mathsf{wp}\, \langle e, \varnothing \rangle\, \{\varphi\}$ holds, then the configuration $\varnothing; \langle e, \varnothing \rangle$ is adequate with respect to $\varphi$. In particular, this configuration is safe.*

## 3.3 Multicore OCaml-specific assertions

We now define four custom assertions, namely the non-atomic and atomic "points-to" assertions, the block length assertion a "valid-view" assertion. Next (§3.4), we give a set of reasoning rules where these assertions appear.

### 3.3.1 Non-atomic points-to

We wish to be able to split up the ownership of the non-atomic store under a fractional permission regime. As usual (Boyland, 2003; Bornat et al., 2005), the fraction 1 should represent exclusive read-write access, while a fraction $q < 1$ should represent shared read-only access. Furthermore, we wish to ensure that whoever owns a share of a non-atomic cell has full knowledge of its history.

For this purpose, we have placed the ghost-state-ownership assertion $\boxed{\bullet\, \mathrm{st}(\sigma)}^{\gamma_{\mathrm{store}}}$ in the state interpretation (§3.1), and we now define the non-atomic points-to assertion for a cell $a$, a fraction $q$ and a history $h$, as follows:

$$q \cdot a \rightsquigarrow_{\mathrm{na}} h \;\triangleq\; \boxed{\circ\, \{a \mapsto (\mathsf{inl}(\mathsf{ag}(h)), q)\}}^{\gamma_{\mathrm{store}}}$$

We omit the fraction $q$ when it is 1: we write $a \rightsquigarrow_{\mathrm{na}} h$ for $1 \cdot a \rightsquigarrow_{\mathrm{na}} h$.

What is going on here? On the one hand, the points-to assertion claims the ownership of a fragmentary element of the camera $\textsc{Auth}(\textsc{StoreCamera})$. Indeed, $\{a \mapsto (\mathsf{inl}(\mathsf{ag}(h)), q)\}$ denotes a singleton map, which maps the cell $a$ to the pair $(\mathsf{inl}(\mathsf{ag}(h)), q)$ where $h$ is a history (recall that $\textsc{CellContents}_{\mathrm{na}} = \textsc{Hist}$) and $q$ is a fraction. On the other hand, the state interpretation owns the authoritative element $\bullet\, \mathrm{st}(\sigma)$. This ties the non-atomic fragment of the store $\sigma$, which is part of the physical machine state, with the state of the ghost variable $\gamma_{\mathrm{store}}$.

When such a points-to assertion $q \cdot a \rightsquigarrow_{\mathrm{na}} h$ is at hand, one has a fragmentary element $\circ\, \{a \mapsto (\mathsf{inl}(\mathsf{ag}(h)), q)\}$. By comparing it against the authoritative element $\bullet\, \mathrm{st}(\sigma)$, one deduces $\{a \mapsto (\mathsf{inl}(\mathsf{ag}(h)), q)\} \preccurlyeq \mathrm{st}(\sigma)$ where $\preccurlyeq$ is the ordering induced by the pointwise composition law of the finite map camera, which yields $(\mathsf{inl}(\mathsf{ag}(h)), q) \preccurlyeq \mathrm{st}(\sigma)(a)$, hence $\mathsf{inl}(\mathsf{ag}(h)) \preccurlyeq \mathsf{inl}(\mathsf{ag}(\sigma(a)))$ and $q \preccurlyeq 1$. Finally, by the laws of the sum and agreement cameras, we find that $\sigma(a) = h$, that is, in the physical store, $a$ is indeed a non-atomic cell whose history is exactly $h$. In short, the following holds, where the conclusion is pure:

$$\frac{q \cdot a \rightsquigarrow_{\mathrm{na}} h \quad * \quad S(\sigma)}{\sigma(a) = h \quad * \quad 0 < q \leq 1}$$

The composition law of the underlying product camera achieves agreement of the history components and addition of the fraction components. In other words, the following holds:

$$q_1 \cdot a \rightsquigarrow_{\mathrm{na}} h_1 \;*\; q_2 \cdot a \rightsquigarrow_{\mathrm{na}} h_2 \;\dashv\vdash\; (q_1 + q_2) \cdot a \rightsquigarrow_{\mathrm{na}} h_1 \;*\; h_1 = h_2$$

Thus, one can read the assertion $q \cdot a \rightsquigarrow_{\mathrm{na}} h$ as representing the knowledge that the cell $a$ is non-atomic and that its history is $h$, along with the ownership of a $q$-share of that cell. As explained in §2.3.2, a 1-share is exclusive: in the composition rule above, if $q_1 = 1$, then from the combined points-to assertion and the state interpretation we deduce $1 + q_2 \leq 1$, which contradicts $0 < q_2$.

This technique of encoding a (fractional) points-to assertion using these ghost state constructions is not original: we follow the pattern presented by Jung et al. (2018b, §6.3.3, §7.3). A departure from the well-established points-to assertion is that, in our memory model, the global store maps each non-atomic cell to an entire history, rather than a single value, and the points-to assertion of BaseCosmo reflects that fact. Indeed, unless some information is known about the view of the current thread, every write in the history of a non-atomic cell might be relevant.

### 3.3.2 Atomic points-to

Regarding the atomic points-to assertion, we proceed essentially in the same manner:

$$q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle \;\triangleq\; \exists \mathcal{W}.\; \mathcal{V} \sqsubseteq \mathcal{W} * \boxed{\circ \{A \mapsto (\mathsf{inr}(\mathsf{ag}(\langle v, \mathcal{W} \rangle)), q)\}}^{\gamma_{\mathrm{store}}}$$

We omit the fraction $q$ when it is 1: we write $A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle$ for $1 \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle$.

As a result of these definitions, the assertion $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle$ claims the ownership of a $q$-share of $A$ and guarantees that it is an atomic memory cell, that the value it stores is $v$, and that the view it stores is at least $\mathcal{V}$. Again, a 1-share is exclusive.

$$\frac{q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle \quad * \quad S(\sigma)}{\exists \mathcal{W}.\, \sigma(A) = \langle v, \mathcal{W} \rangle \quad * \quad \mathcal{V} \sqsubseteq \mathcal{W} \;*\; 0 < q \leq 1}$$

$$q_1 \cdot A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}_1 \rangle \;*\; q_2 \cdot A \rightsquigarrow_{\mathrm{at}} \langle v_2, \mathcal{V}_2 \rangle \;\dashv\vdash\; (q_1 + q_2) \cdot A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}_1 \sqcup \mathcal{V}_2 \rangle \;*\; v_1 = v_2$$

By requiring the view $\mathcal{W}$ stored at $A$ to satisfy $\mathcal{V} \sqsubseteq \mathcal{W}$, as opposed to the equality $\mathcal{V} = \mathcal{W}$, we make the points-to assertion anti-monotonic in its view parameter: that is, if $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ holds, then $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}_2 \rangle$ entails $q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}_1 \rangle$. This seems convenient in practice, as it gives the user a concise way of retaining partial knowledge of the view that is stored at cell $A$. On the other hand, we do lose the ability of expressing negative information about the view, that is, an upper bound on the view. This should not be a problem in practice, as one always reasons about events that must be part of a view, and never about events that must not. In fact, in the high-level logic presented in the next chapter, assertions are *monotonic* functions of a view, so this style of reasoning becomes built-in.

### 3.3.3 Block length

We have placed the assertion $\boxed{\bullet\, \mathrm{len}(\sigma)}^{\gamma_{\mathrm{len}}}$ in the state interpretation. This lets us define an assertion representing the knowledge of the length of a given block:

$$\ell \;\mathsf{has\ length}\; n \;\triangleq\; \boxed{\circ \{\ell \mapsto n\}}^{\gamma_{\mathrm{len}}}$$

This assertion is persistent (§2.3.2) and achieves agreement on the length of a block:

$$\frac{\ell \mathsf{\ has\ length\ } n \quad * \quad \ell \mathsf{\ has\ length\ } n'}{n = n'}$$

We can use it to define a points-to assertion for entire blocks as sugar, where the $b_i$ are either histories or pairs of a value and a view:

$$q \cdot \ell \rightsquigarrow^*_\alpha [b_0, \ldots, b_{n-1}] \triangleq \ell \mathsf{\ has\ length\ } n * \mathop{\Large *}_{0 \le i < n} q \cdot \ell[i] \rightsquigarrow_\alpha b_i$$

Again, when omitted the fraction $q$ is taken to be 1, i.e. full ownership. Thanks to the length information, this block-points-to assertion achieves agreement on the value of a block:

$$q \cdot \ell \rightsquigarrow^*_\alpha b \quad * \quad q' \cdot \ell \rightsquigarrow^*_\alpha b' \quad \dashv\vdash \quad (q + q') \cdot \ell \rightsquigarrow^*_\alpha b \quad * \quad b = b'$$

Without the length information, we would only be able to deduce that the shortest list among $b$ and $b'$ is a prefix of the longest list.

As will be seen later (rule BASE-LENGTH in Figure 3.2), the length assertion is what we need for giving a specification to the operation of the language that computes the length of a block. By using the frame rule (FRAME), we will be able to derive another specification for this operation, that takes as precondition a block-points-to assertion rather than a length assertion:

$$\{q \cdot \ell \rightsquigarrow^*_\alpha b\}\ \langle \mathtt{length}\ \ell, \mathcal{W} \rangle \left\{ \lambda \langle v', \mathcal{W}' \rangle . * \begin{cases} v' = |b| \\ \mathcal{W}' = \mathcal{W} \\ q \cdot \ell \rightsquigarrow^*_\alpha b \end{cases} \right\}$$

### 3.3.4 Validity of a view

The last part of the state interpretation (§3.1) asserts the existence of a view $\mathcal{G}$ such that items (2), (3) and (4) of the global view invariant hold (§2.2.2). It also includes the ghost-state-ownership assertion $\boxed{\bullet \mathcal{G}}^{\gamma_{\mathrm{gv}}}$. We now define the "valid-view" assertion as follows:

$$\mathsf{valid}\, \mathcal{V} \triangleq \boxed{\circ \mathcal{V}}^{\gamma_{\mathrm{gv}}}$$

The assertion $\mathsf{valid}\, \mathcal{V}$ guarantees that $\mathcal{V}$ is a fragment of, or lower bound on, the global view; that is, $\mathcal{V} \sqsubseteq \mathcal{G}$ holds. Because the camera of views is idempotent (that is, $\mathcal{V} \sqcup \mathcal{V} = \mathcal{V}$), this assertion is persistent (§2.3.2). Intuitively, updates to the authoritative fragment $\bullet \mathcal{G}$ must preserve the "frame", which consists of the existing assertions $\mathsf{valid}\, \mathcal{V}$. Hence, once $\mathsf{valid}\, \mathcal{V}$ holds, it holds forever. This conveys the idea that the global view $\mathcal{G}$ only grows over time, with more write events.

A validity assertion appears in almost all of the reasoning rules that we present in the next subsection (§3.4): the rules effectively require (and allow) every thread to keep track of the fact that its current view is valid. This encodes item (1) of the global view invariant. The root cause of this phenomenon lies in the rule BASE-NA-READ, where the global view invariant must be exploited in order to prove that a non-atomic read instruction can make progress. Then, the other rules must preserve that invariant.

In order to initiate the verification of a full program, a user then needs the validity of the initial (empty) view. This is easily obtained through the following theorem, whose proof is trivial.

**Theorem 2 (Validity of the empty view)** *The entailment* $\vdash \Rightarrow \mathsf{valid}\, \varnothing$ *holds in BaseCosmo.*

BASE-VALUE
$$\{\mathsf{True}\}\ \langle v, \mathcal{W}\rangle\ \{\lambda\,\langle v', \mathcal{W}'\rangle\,.\,v' = v * \mathcal{W}' = \mathcal{W}\}$$

BASE-PURE
$$\frac{\forall m, e'', e_1', \ldots e_n'.\,(e \xrightarrow{m} e'', e_1', \ldots, e_n') \implies (m = \varepsilon \wedge e'' = e' \wedge n = 0) \qquad e \xrightarrow{\varepsilon} e' \qquad \triangleright\{P\}\ \langle e', \mathcal{W}\rangle\ \{\Phi\}}{\{P\}\ \langle e, \mathcal{W}\rangle\ \{\Phi\}}$$

BASE-FORK
$$\frac{\triangleright\{P\}\ \langle e, \mathcal{W}\rangle\ \{\lambda\_.\,\mathsf{True}\}}{\{P\}\ \langle \texttt{fork}\ e, \mathcal{W}\rangle\ \{\lambda\,\langle v', \mathcal{W}'\rangle\,.\,v' = \texttt{()} * \mathcal{W}' = \mathcal{W}\}}$$

Figure 3.1: BaseCosmo rules for pure steps and the "fork" operation

## 3.4 Multicore OCaml-specific rules

There remains to give a set of Multicore OCaml-specific deduction rules that allow establishing Hoare triples, reflecting the operational semantics of Multicore OCaml. Figure 3.1 shows standard rules (Jung et al., 2018b, §6.2), that do not interact with the shared memory, while Figure 3.2 shows the rules that govern the memory access operations. In these rules, some preconditions are prefixed with the "later" modality of Iris. In a first reading, we can ignore them. See Jung et al. (2018b, §6.2) for details.

A program that is already a value performs no computation at all (BASE-VALUE), and its return value is itself.

Running a pure step of computation (BASE-PURE) affects neither the ghost state nor the thread's view. Therefore, if $e \xrightarrow{\varepsilon} e'$ is the unique possible reduction step starting from $e$, then proving a specification about $e'$ with some view $\mathcal{W}$ suffices to prove the same specification about $e$ with the same view $\mathcal{W}$.

Running a "fork" operation (BASE-FORK) returns the unit value and spawns a new thread whose return value is ignored. Still, safety of the full program requires safety of the newly-spawned thread $e$, hence the rule requires proving a triple about $e$ with a trivial postcondition. The newly-spawned thread inherits the view $\mathcal{W}$ of its parent thread.

Rules in Figure 3.2 are "small axioms" (O'Hearn, 2019), that is, triples that describe the minimum resources required by each operation. Each of them is just a single triple $\{P\}\ \langle e, \mathcal{W}\rangle\ \{\Phi\}$, which, for greater readability, we display vertically:

$$\begin{array}{l}\left\{P\quad\right\}\\ \langle e, \mathcal{W}\rangle\\ \left\{\Phi\quad\right\}\end{array}$$

BASE-NA-ALLOC

$$\left\{n \geq 0\right\}\ \langle \mathtt{array}_{\mathrm{na}}[n]\ v, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, \exists \ell.\, * \left\{\begin{array}{l} v' = \ell \\ \mathcal{W}' = \mathcal{W} \\ \ell \rightsquigarrow^*_{\mathrm{na}} [\{0 \mapsto v\}, \ldots, \{0 \mapsto v\}] \quad (n \text{ cells}) \end{array}\right.\right\}$$

BASE-NA-ALLOC-ONE

$$\left\{\mathsf{True}\right\}\ \langle \mathtt{ref}_{\mathrm{na}}\ v, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, \exists \ell.\, * \left\{\begin{array}{l} v' = \ell \\ \mathcal{W}' = \mathcal{W} \\ \ell \rightsquigarrow_{\mathrm{na}} \{0 \mapsto v\} \end{array}\right.\right\}$$

BASE-NA-READ

$$\left\{q \cdot a \rightsquigarrow_{\mathrm{na}} h\ *\ \mathsf{valid}\, \mathcal{W}\right\}\ \langle !_{\mathrm{na}}\, a, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, \exists t.\, * \left\{\begin{array}{l} t \in \mathrm{dom}(h) \\ \mathcal{W}(a) \leq t \\ v' = h(t) \\ \mathcal{W}' = \mathcal{W} \\ q \cdot a \rightsquigarrow_{\mathrm{na}} h \end{array}\right.\right\}$$

BASE-NA-WRITE

$$\left\{a \rightsquigarrow_{\mathrm{na}} h\ *\ \mathsf{valid}\, \mathcal{W}\right\}\ \langle a :=_{\mathrm{na}} v, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, \exists t.\, * \left\{\begin{array}{l} t \notin \mathrm{dom}(h) \\ \mathcal{W}(a) < t \\ v' = () \\ \mathcal{W}' = \mathcal{W}[a \mapsto t] \\ a \rightsquigarrow_{\mathrm{na}} h[t \mapsto v] \\ \mathsf{valid}\, \mathcal{W}' \end{array}\right.\right\}$$

BASE-AT-ALLOC

$$\left\{n \geq 0\ *\ \mathsf{valid}\, \mathcal{W}\right\}\ \langle \mathtt{array}_{\mathrm{at}}[n]\ v, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, \exists \ell.\, * \left\{\begin{array}{l} v' = \ell \\ \mathcal{W}' = \mathcal{W} \\ \ell \rightsquigarrow^*_{\mathrm{at}} [\langle v, \mathcal{W}\rangle, \ldots, \langle v, \mathcal{W}\rangle] \quad (n \text{ cells}) \end{array}\right.\right\}$$

BASE-AT-ALLOC-ONE

$$\left\{\mathsf{valid}\, \mathcal{W}\right\}\ \langle \mathtt{ref}_{\mathrm{at}}\ v, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, \exists \ell.\, * \left\{\begin{array}{l} v' = \ell \\ \mathcal{W}' = \mathcal{W} \\ \ell \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{W}\rangle \end{array}\right.\right\}$$

BASE-AT-READ

$$\left\{q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle\ *\ \mathsf{valid}\, \mathcal{W}\right\}\ \langle !_{\mathrm{at}}\, A, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, * \left\{\begin{array}{l} v' = v \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \\ \mathsf{valid}\, \mathcal{W}' \end{array}\right.\right\}$$

BASE-AT-WRITE

$$\left\{A \rightsquigarrow_{\mathrm{at}} \langle \_, \mathcal{V}\rangle\ *\ \mathsf{valid}\, \mathcal{W}\right\}\ \langle A :=_{\mathrm{at}} v, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, * \left\{\begin{array}{l} v' = () \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \sqcup \mathcal{W}\rangle \\ \mathsf{valid}\, \mathcal{W}' \end{array}\right.\right\}$$

BASE-CAS-FAILURE

$$\left\{v_0 \neq v_1\ *\ A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle\ *\ \mathsf{valid}\, \mathcal{W}\right\}\ \langle \mathtt{CAS}\ A\ v_1\ v_2, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, * \left\{\begin{array}{l} v' = \mathtt{false} \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle \\ \mathsf{valid}\, \mathcal{W}' \end{array}\right.\right\}$$

BASE-CAS-SUCCESS

$$\left\{A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}\rangle\ *\ \mathsf{valid}\, \mathcal{W}\right\}\ \langle \mathtt{CAS}\ A\ v_1\ v_2, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, * \left\{\begin{array}{l} v' = \mathtt{true} \\ \mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}' \\ A \rightsquigarrow_{\mathrm{at}} \langle v_2, \mathcal{V} \sqcup \mathcal{W}\rangle \\ \mathsf{valid}\, \mathcal{W}' \end{array}\right.\right\}$$

BASE-LENGTH

$$\left\{\ell\ \mathsf{has\ length}\ n\right\}\ \langle \mathtt{length}\ \ell, \mathcal{W}\rangle\ \left\{\lambda\, \langle v', \mathcal{W}'\rangle\,.\, * \left\{\begin{array}{l} v' = n \\ \mathcal{W}' = \mathcal{W} \end{array}\right.\right\}$$

Figure 3.2: BaseCosmo triples for the memory access operations

### 3.4.1 Operations on non-atomic cells

**Allocation of non-atomic cells** Allocating a non-atomic memory block (BASE-NA-ALLOC) returns a memory location $\ell$ and does not change the view of the current thread. The points-to assertion $\ell \rightsquigarrow^*_{\mathrm{na}} [\{0 \mapsto v\}, \ldots, \{0 \mapsto v\}]$ in the postcondition represents the full ownership of all cells of the newly-allocated memory block and guarantees that their histories contain a single write of the value $v$ at timestamp 0.

From the previous rule, it is easy to derive a simplified rule for the allocation of non-atomic references (BASE-NA-ALLOC-ONE).

**Non-atomic read** Reading a non-atomic memory cell (BASE-NA-READ) requires (possibly shared) ownership of this memory cell, which is why the points-to assertion $q \cdot a \rightsquigarrow_{\mathrm{na}} h$ appears in the precondition. What can be said of the value $v'$ produced by this instruction? A non-atomic read can read from any write whose timestamp is high enough, according to this thread's view of the cell $a$. Thus, for some timestamp $t$ such that $t \in \mathrm{dom}(h)$ holds (i.e., $t$ is a valid timestamp) and $\mathcal{W}(a) \leq t$ holds (i.e., the view $\mathcal{W}$ allows reading from this timestamp), the value $v'$ must be the value that was written at time $t$, that is, $h(t)$. The thread's view is unaffected, and the points-to assertion is preserved.

In order to justify a non-atomic read, the validity of the current view is required: the assertion $\mathsf{valid}\,\mathcal{W}$ appears in the precondition of BASE-NA-READ. Without this requirement, we would not be able to establish this triple. Indeed, we must prove that this read instruction can make progress, that is, it cannot be stuck. In other words, we must prove that the history $h$ contains a write event whose timestamp is at least $\mathcal{W}(a)$. Quite obviously, in the absence of any hypothesis about the view $\mathcal{W}$, it would be impossible to prove such a fact. Thanks to the validity hypothesis $\mathsf{valid}\,\mathcal{W}$, we find that that $\mathcal{W}$ must be a fragment of the global view $\mathcal{G}$. This implies $\mathcal{W}(a) \leq \mathcal{G}(a)$. Furthermore, the state interpretation guarantees $\mathcal{G}(a) \in \mathrm{dom}(h)$. Therefore, this read instruction can read (at least) from the write event whose timestamp is $\mathcal{G}(a)$. Therefore, it is not stuck.

**Non-atomic write** Writing a non-atomic memory cell (BASE-NA-WRITE) requires exclusive ownership of this memory cell, which is expressed by the points-to assertion $a \rightsquigarrow_{\mathrm{na}} h$. It also requires the validity of the current view $\mathcal{W}$, as this information is needed to prove that the updated view $\mathcal{W}'$ is valid. In accordance with the operational semantics, the history $h$ is extended with a write event at some timestamp $t$ that is both fresh for the history $h$ and permitted by the view $\mathcal{W}$. The updated points-to assertion $a \rightsquigarrow_{\mathrm{na}} h[t \mapsto v]$ reflects this updated history. The thread's new view $\mathcal{W}'$ is obtained by updating $\mathcal{W}$ with a mapping of $a$ to the timestamp $t$.

### 3.4.2 Operations on atomic cells

The rules that govern atomic memory cells are a little heavy, due to the fact that atomic memory cells play two independent roles: they store a value and a view. Regarding the "value" aspect, these rules are identical to the standard rules of Concurrent Separation Logic with fractional permissions, which reflects the sequentially consistent behavior of atomic memory cells. Regarding the "view" aspect, these rules describe how views are written and read by the atomic memory instructions. As can be seen, the stored view accumulates the knowledge of the writers and transmits it to the readers. This reflects the "release/acquire" behavior of these

instructions and is analogous to the fact that, in axiomatic memory models such as that given by Dolan et al. for Multicore OCaml, there is a happens-before relation from a write operation to a read operation which reads from it.

**Allocation of atomic cells** When an atomic memory block is allocated (BASE-AT-ALLOC), its cells are initialized with the view $\mathcal{W}$ of the current thread. This is expressed by the points-to assertion $\ell \rightsquigarrow^*_{\text{at}} [\langle v, \mathcal{W}\rangle, \ldots, \langle v, \mathcal{W}\rangle]$ in the postcondition. Maintaining the global view invariant (§2.2.2) requires that the views of the new atomic cells be contained in the global view, that is, that these views are valid; for this reason, $\mathsf{valid}\,\mathcal{W}$ is a precondition of this rule.

Again it is easy to derive a simplified rule for the allocation of atomic references (BASE-AT-ALLOC-ONE).

**Atomic read** When an atomic memory cell is read (BASE-AT-READ), the view stored at this cell is merged into the view of the current thread: this is an "acquire read". This is expressed by the inequality $\mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}'$. We cannot expect to obtain an equality $\mathcal{V} \sqcup \mathcal{W} = \mathcal{W}'$ because, according to our definition of the atomic points-to assertion (§3.3), $\mathcal{V}$ is only a fragment of the view that is stored at cell $A$. Thus, we get only a lower bound on the thread's new view $\mathcal{W}'$. This does not hinder practical expressiveness. Nevertheless, we can prove that $\mathcal{W}'$ is valid.

**Atomic write** When performing an atomic write (BASE-AT-WRITE), the same "acquisition" phenomenon occurs: we get $\mathcal{V} \sqcup \mathcal{W} \sqsubseteq \mathcal{W}'$. Furthermore, this thread's view is merged into the view stored at this cell: this is a "release write". This is expressed by the updated points-to assertion $A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \sqcup \mathcal{W}\rangle$.

**(Atomic) Compare-and-set** The two rules associated with the CAS instruction respectively describe the case where this instruction fails (BASE-CAS-FAILURE) and the case where it succeeds (BASE-CAS-SUCCESS). These rules express the idea that a CAS behaves as a read followed (in case of success) with a write.

In this chapter, we have built a program logic for reasoning about Multicore OCaml programs. Compared to working with the bare operational semantics (§2.2), the benefits of BaseCosmo are clear. We have at our disposal a rich specification language which contains the host logic (that of Coq) and an expressive ghost state framework. A sound—and mechanized—proof system lets us prove program safety and partial functional correctness. By the virtue of Concurrent Separation Logic, specifications are reduced to their minimal footprints and their proofs are composable: a program fragment is specified and verified in isolation. Interference by other program fragments need not be considered. In particular, when verifying a program with several threads, we do not need to consider the many possible interleavings of the instructions of concurrent threads.

However, the program logic that we have just presented is still low-level, as it exposes tedious details of the Multicore OCaml memory model. Several aspects of it can be criticized:

1. The assertion $q \cdot a \rightsquigarrow_{\text{na}} h$ exposes the fact that a non-atomic memory cell stores a history $h$, as opposed to a single value $v$. The rules BASE-NA-READ and BASE-NA-WRITE paraphrase the operational semantics and reveal the timestamp machinery. This makes it difficult to reason about non-atomic memory cells. Yet, at least in the absence of data races on these

cells, one would like to reason in a simpler way. Is it possible to offer higher-level points-to assertions and rules, so that a non-atomic cell appears to store a single value?

2. The view $\mathcal{W}$ of the current thread is explicitly named in every triple $\{P\}\ \langle e, \mathcal{W}\rangle\ \{\Phi\}$, and its validity is typically explicitly asserted as part of every pre- and postcondition. This seems heavy. Is it possible to make this view everywhere implicit by default, and to have a way of referring to it only where needed?

In the next chapter, we answer these questions in the affirmative.

# Chapter 4

# A higher-level logic: Cosmo

In this chapter, we address both of the concerns just outlined by building a higher-level logic, Cosmo, on top of the low-level logic BaseCosmo. In Cosmo, the points-to assertion for non-atomic cells takes the traditional form $q \cdot a \rightsquigarrow_{\mathrm{na}} v$, as in Concurrent Separation Logic with fractional permissions (Boyland, 2003; Bornat et al., 2005). This assertion is strong enough to guarantee that reading $a$ will yield the value $v$. Therefore, at an intuitive level, one can take it to mean that "the cell $a$ currently contains $v$", or slightly more accurately, "in the eyes of this thread, the cell $a$ currently contains $v$". In more technical terms, this assertion guarantees that *the most recent write to the cell $a$ is a write of the value $v$* and that *this thread is aware of this write.*

The meaning of this simplified points-to assertion is relative to "this thread", that is, to a certain thread about which one is presently reasoning. More precisely, its meaning depends on "this thread's view", as the assertion claims that a certain write event is part of this thread's view. Therefore, to give meaning to this simplified points-to assertion, we find that we must parameterize every assertion with a view: in other words, a Cosmo assertion must denote *a function of a view to a BaseCosmo assertion.* This change in perspective not only seems required in order to address concern 1 above, but also addresses concern 2 at the same time.

Following Kaiser et al. (2017) and Dang et al. (2020), we require every Cosmo assertion to denote a *monotone* function with respect to the information ordering $\sqsubseteq$ (§2.2.1). This guarantees that, as new memory events become visible to this thread, the validity of every assertion is preserved. This condition makes the frame rule sound at the level of Cosmo.

In the following sections, we describe how Cosmo is constructed on top of BaseCosmo. A user of Cosmo need not be aware of this construction: the assertions and reasoning rules of Cosmo can be presented to her directly. Nevertheless, a Cosmo proof is a BaseCosmo proof, and it is therefore easy to combine proofs carried out in the two logics, should Cosmo alone not be expressive enough.

## 4.1 Language-independent Cosmo assertions and rules

Figure 4.1a defines a number of ways of constructing Cosmo assertions. As explained, Cosmo assertions, whose type is VPROP, are monotonic functions from views to BaseCosmo assertions.

We lift the standard connectives of Separation Logic, including $\vee$, $*$, $-\!\!*$, $\exists$, $\forall$, from BaseCosmo up to Cosmo. In the definition of the magic wand, a universal quantification over a future

$$\textsc{vProp} \triangleq \textsc{View} \xrightarrow{\text{mon}} \textsc{iProp}$$

$$\begin{aligned}
P \vee Q &\triangleq \lambda \mathcal{W}.\, P(\mathcal{W}) \vee Q(\mathcal{W}) \\
P * Q &\triangleq \lambda \mathcal{W}.\, P(\mathcal{W}) * Q(\mathcal{W}) \\
P \mathrel{-\!\!*} Q &\triangleq \lambda \mathcal{W}.\, \forall \mathcal{V} \sqsupseteq \mathcal{W}.\, P(\mathcal{V}) \mathrel{-\!\!*} Q(\mathcal{V}) \\
\exists x.\, P &\triangleq \lambda \mathcal{W}.\, \exists x.\, P(\mathcal{W}) \\
\forall x.\, P &\triangleq \lambda \mathcal{W}.\, \forall x.\, P(\mathcal{W}) \\
\lceil P \rceil &\triangleq \lambda \mathcal{W}.\, P \quad \text{where } P \in \textsc{iProp}
\end{aligned}$$

$$\begin{aligned}
\uparrow \mathcal{V} &\triangleq \lambda \mathcal{W}.\, \mathcal{V} \sqsubseteq \mathcal{W} \\
\textsc{iProp} \ni P \mathbin{@} \mathcal{V} &\triangleq P(\mathcal{V}) \quad \text{where } P \in \textsc{vProp}
\end{aligned}$$

(a) Language-independent

$$\begin{aligned}
q \cdot a \rightsquigarrow_{\text{na}} v &\triangleq \exists h.\, \exists t. \\
&\quad * \begin{cases} \lceil q \cdot a \rightsquigarrow_{\text{na}} h \rceil \\ t = \max(\mathrm{dom}(h)) \\ h(t) = v \\ \exists \mathcal{V}.\ \mathcal{V}(a) = t * \uparrow \mathcal{V} \end{cases} \\
q \cdot A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \rangle &\triangleq \lceil q \cdot A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \rangle \rceil \\
q \cdot A \rightsquigarrow_{\text{at}} v &\triangleq q \cdot A \rightsquigarrow_{\text{at}} \langle v, \varnothing \rangle \\
&\dashv\vdash \exists \mathcal{V}.\, q \cdot A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \rangle \\
\ell \text{ has length } n &\triangleq \lceil \ell \text{ has length } n \rceil
\end{aligned}$$

(b) Multicore OCaml-specific

Figure 4.1: Cosmo assertions

| Seen-Zero | Seen-Two | Split-Subjective-Objective |
|---|---|---|
| $\vdash \uparrow \varnothing$ | $\uparrow \mathcal{V}_1 * \uparrow \mathcal{V}_2 \dashv\vdash \uparrow(\mathcal{V}_1 \sqcup \mathcal{V}_2)$ | $P \dashv\vdash \exists \mathcal{V}.\, (\uparrow \mathcal{V} * P \mathbin{@} \mathcal{V})$ |

Figure 4.2: Cosmo view rules

view $\mathcal{V}$ that contains $\mathcal{W}$ is used so as to obtain a monotone function of $\mathcal{W}$; this is a standard technique. We also define the lifting of a BaseCosmo assertion $P$ up to a Cosmo assertion $\lceil P \rceil$. This provides a means of communication between BaseCosmo and Cosmo. The definition is simple: $\lceil P \rceil$ is the constant function $\lambda \mathcal{W}.\, P$. It is the archetypical example of an *objective* assertion, that is, an assertion whose meaning is independent of this thread's view. We often write just $P$ for $\lceil P \rceil$. Here are several typical examples of BaseCosmo assertions that can be usefully lifted up, yielding objective Cosmo assertions. In each case, we omit the brackets $\lceil \cdot \rceil$.

- A pure (Coq) proposition $P$ where $P \in \textsc{Prop}$ (§2.3.2).
- The ownership of a piece of ghost state $\boxed{m}^{\gamma}$ (§2.3.2).
- The knowledge of an invariant $\boxed{I}$ where $I \in \textsc{iProp}$ (§2.3.2).
- A Cosmo assertion at a fixed view $P \mathbin{@} \mathcal{V}$ (to be defined shortly).

Regarding invariants, we emphasize that $\boxed{I}$ is an ordinary BaseCosmo assertion, hence $I$ must also be a BaseCosmo assertion. That is, the logic Cosmo is subject to a restriction: *in every invariant $\boxed{I}$, the assertion $I$ must be objective.* In short, because an invariant represents knowledge that is shared by all threads, it cannot depend on some thread's view. At first, this restriction might seem problematic: in general, an arbitrary Cosmo assertion $P$ cannot appear in an invariant, therefore cannot be transmitted from one thread to another. Fortunately, in many situations, it is possible to work around this limitation. Before explaining how, we define two more forms of assertions, that allows manipulating views from the logic.

Beyond lifted assertions, Cosmo features two kinds of assertions to deal with views: $\uparrow \mathcal{V}$ and $P \mathbin{@} \mathcal{V}$.

We define the assertion $\uparrow \mathcal{V}$, pronounced "*I have* $\mathcal{V}$", which asserts that this thread's view contains the view $\mathcal{V}$. It is simply the function $\lambda \mathcal{W}.\, \mathcal{V} \sqsubseteq \mathcal{W}$, where the formal parameter $\mathcal{W}$ represents this thread's view. It is easy to see that this is a monotone function of $\mathcal{W}$. The assertion $\uparrow \mathcal{V}$ is the archetypical example of a *subjective* assertion, that is, an assertion whose meaning depends on this thread's view. It satisfies the first two rules in Figure 4.2. SEEN-ZERO states that having nothing is the same as having the empty view. SEEN-TWO states that having two views $\mathcal{V}_1$ and $\mathcal{V}_2$ separately is the same as having their join $\mathcal{V}_1 \sqcup \mathcal{V}_2$. The latter implies that the assertion $\uparrow \mathcal{V}$ is anti-monotone with respect to $\mathcal{V}$ (if $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ then $\uparrow \mathcal{V}_2 \vdash \uparrow \mathcal{V}_1$), and that it is duplicable ($\uparrow \mathcal{V} \vdash \uparrow \mathcal{V} * \uparrow \mathcal{V}$). In fact, as can be seen from its definition, the assertion $\uparrow \mathcal{V}$ does not assert the ownership of any resource, hence it is persistent.

The nature of views is hidden from a user of Cosmo. That is, views can be presented to the user as an abstract type equipped with a bounded semilattice structure, whose least element and join operation are denoted by $\varnothing$ and $\sqcup$, respectively. This means that the user need not think of views as "functions of non-atomic cells to timestamps", or as "sets of write events". Instead, the user should think of a view as a certain amount of "information" about the (non-atomic part of the) shared memory. The deduction rules of the logic allow the user to reason abstractly about the manner in which this information is acquired and transmitted and about the places where it is needed.

The last line in Figure 4.1a defines $P \mathbin{@} \mathcal{V}$ as sugar for the function application $P(\mathcal{V})$. This notation provides a means of communication in the reverse direction: if $P$ is a Cosmo assertion, then $P @ \mathcal{V}$ is a BaseCosmo assertion, which can be read as "$P$ *holds at* $\mathcal{V}$". As already mentioned, we lift this assertion to an objective Cosmo assertion. Said otherwise: $P \mathbin{@} \mathcal{V}$, as a Cosmo assertion, denotes the assertion $P$ where $\mathcal{V}$ has been substituted for the ambient view; as it does not depend on the ambient view anymore, it is objective.[1]

Thanks to the constructs that have just been introduced, it is possible to express the idea that any Cosmo assertion $P$ can be split into a subjective component and an objective component. This is stated by the rule SPLIT-SUBJECTIVE-OBJECTIVE in Figure 4.2. When read from left to right, the rule splits $P$ into a subjective component $\uparrow \mathcal{V}$ and an objective component $P \mathbin{@} \mathcal{V}$, for some view $\mathcal{V}$. (The witness for $\mathcal{V}$ is this thread's view at the time of splitting.) When read from right to left, the rule reunites these components and yields $P$ again. This decomposition is crucial: it allows to transmit any Cosmo assertion $P$ from one thread to another even though, in general, $P$ cannot appear in an invariant. We *can* let its objective part $P$ appear in a invariant, thereby allowing it to be shared between threads; and we can transmit its subjective part via explicit synchronization operations, typically writes and reads of atomic cells. We will use that idiom in all our case studies (Chapters 5 and 6); our spin lock implementation (§5.2), for one, offers a typical example.

[1] An equivalent definition is the following: asserting $P \mathbin{@} \mathcal{V}$ is asserting that, even without other knowledge, if we have the knowledge contained in $\mathcal{V}$, then $P$ holds. In other words, $P \mathbin{@} \mathcal{V}$ is objective and entails $\uparrow \mathcal{V} \mathrel{-\!\!*} P$.

## 4.2 Multicore OCaml-specific Cosmo assertions

Figure 4.1b defines Cosmo assertions that allow reasoning about Multicore OCaml programs. We want a fractional points-to assertion for non-atomic cells, a fractional points-to assertion for atomic cells, and a length assertion for blocks. The last two are lifted directly from BaseCosmo, omitting the brackets $\lceil\cdot\rceil$. Thus, $q \cdot A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V}\rangle$ and "$\ell$ $\mathsf{has\ length}$ $n$" are objective Cosmo assertions.

We may omit the view $\mathcal{V}$ carried by atomic cells when it is not relevant: we define $q \cdot A \rightsquigarrow_{\text{at}} v$ as sugar for $q \cdot A \rightsquigarrow_{\text{at}} \langle v, \varnothing\rangle$, which is logically equivalent to $\exists \mathcal{V}.\, q \cdot A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V}\rangle$, and is still objective. We thus recover the standard points-to predicate of Concurrent Separation Logic with fractional permissions.

Regarding the non-atomic points-to assertion, although we could lift the corresponding assertion from BaseCosmo as-is, there is little interest in doing so: our aim is to simplify the manipulation of non-atomic cells by not having to make their entire history explicit, nor to have to reason about timestamps. Rather, we adopt the definition shown in Figure 4.1b. As announced earlier, this assertion takes the traditional form $q \cdot a \rightsquigarrow_{\text{na}} v$, and means that *the most recent write to the cell $a$ is a write of the value $v$* and that *this thread is aware of this write.* Its definition, whose right-hand side is a conjunction of four assertions, reflects this:

1. $q \cdot a \rightsquigarrow_{\text{na}} h$ claims a fraction $q$ of the cell $a$ and guarantees that its history is $h$.
2. $t = \max\,(\mathrm{dom}(h))$ guarantees that $t$ is the timestamp of the most recent write event at $a$.
3. $h(t) = v$ indicates that $v$ is the value written by this write event.
4. $\exists \mathcal{V}.\ \lceil \mathcal{V}(a) = t\rceil * \uparrow \mathcal{V}$ guarantees that this write event is visible to this thread.

Because $\uparrow \mathcal{V}$ is subjective, the non-atomic points-to assertion $q \cdot a \rightsquigarrow_{\text{na}} v$ is itself subjective. Therefore, it cannot appear in an invariant. This is the price to pay for the apparent simplicity of this predicate.

As was done for BaseCosmo in §3.3.3, we can define a Cosmo points-to assertion for whole blocks of memory, $q \cdot \ell \rightsquigarrow^*_\alpha [b_0, \ldots, b_{n-1}]$, as syntactic sugar (recalling that the atomic points-to assertion and the knowledge of the length of a block are lifted from BaseCosmo to Cosmo as-is).

## 4.3 Cosmo weakest-precondition assertions

We have just presented the universe of Cosmo assertions, which have type VPROP, in contrast with BaseCosmo assertions, which have type IPROP. We now wish to define a Cosmo weakest-precondition assertion $\mathsf{wp}\ e\ \{\Psi\}$ whose postcondition $\Psi$ is a function of a value to a Cosmo assertion. This is in contrast with BaseCosmo's weakest-precondition assertion $\mathsf{wp}\ \langle e, \mathcal{V}\rangle\ \{\Phi\}$ (§3.3) where $\Phi$ is a function of a value and a view to a BaseCosmo assertion. We define the former on top of the latter, as follows:

$$\begin{aligned}\mathsf{wp}\ e\ \{\Psi\} \triangleq\ & \lambda \mathcal{W}.\\ & \quad \forall \mathcal{V} \sqsupseteq \mathcal{W}.\\ & \quad\ \mathsf{valid}\ \mathcal{V} \mathrel{-\!\!*} \mathsf{wp}\ \langle e, \mathcal{V}\rangle\ \{\lambda\, \langle v', \mathcal{V}'\rangle\, .\, \Psi(v')(\mathcal{V}') * \mathsf{valid}\ \mathcal{V}'\}\end{aligned}$$

Because $\mathsf{wp}\ e\ \{\Psi\}$ must have type VPROP, it must be a monotone function of this thread's view $\mathcal{W}$. In order to make it monotone, we quantify over a future view $\mathcal{V}$ that contains $\mathcal{W}$. We use a BaseCosmo weakest-precondition assertion to require that, from the view $\mathcal{V}$, executing the expression $e$ must be safe and must yield a value and a view that satisfy $\Psi$. As a final touch, we place validity assertions in the hypothesis and in the postcondition so as to maintain the invariant that "this thread's view is valid", thus removing from the user the burden of keeping track of this information.

The Cosmo triple is derived from the Cosmo weakest-precondition assertion in the usual way (Jung et al., 2018b, §6): $\{P\}\,e\,\{\Psi\}$ stands for $\Box(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\Psi\})$.

## 4.4 Soundness of Cosmo

Cosmo, equipped with the weakest-precondition assertion that was just defined, is adequate. This follows straightforwardly from the adequacy theorem of BaseCosmo (§3.2).

**Theorem 3 (Adequacy of Cosmo)** *For any expression $e \in$ EXPR and any pure predicate $\varphi$ : VAL $\to$ PROP, if the entailment $\vdash \mathsf{wp}\ e\ \{\varphi\}$ holds then the configuration $\varnothing; \langle e, \varnothing\rangle$ is adequate with respect to $\lambda\,\langle v, \mathcal{V}\rangle\,.\,\varphi(v)$. In particular, this configuration is safe.*

## 4.5 Cosmo rules

Each of the BaseCosmo rules described in §3.4 can now be used as a basis to establish a higher-level Cosmo rule, whose statement is often simpler. The resulting rules appear in Figure 4.3.

The three rules in Figure 4.3a, for language constructs that do not interact with the memory, reflect those in Figure 3.1. Being Cosmo assertions, the triples that appear in these inference rules depend on an ambient view. When both the premises and the conclusion contain a triple, they are to be interpreted at the same view. For rule PURE, this conveys the idea that when taking pure reduction steps from $e$ to $e'$, the thread's view is unchanged. Similarly, for rule FORK, this corresponds to the fact that a spawned thread $e$ inherits the view of its parent thread `fork` $e$.

### 4.5.1 Operations on non-atomic cells

The first four rules in Figure 4.3b describe the operations of allocating, reading, and writing non-atomic memory cells. Allocation (NA-ALLOC) requires nothing and produces points-to assertions $\ell[i] \rightsquigarrow_{\mathrm{na}} v$ for $0 \leq i < n$, which represent the exclusive ownership of the fresh memory cells $\ell[i]$. A simplified rule (NA-ALLOC-ONE) is easily derived for allocating a non-atomic reference. Reading (NA-READ) requires $q \cdot a \rightsquigarrow_{\mathrm{na}} v$, which represents possibly-shared ownership of the memory cell $a$, and preserves this assertion. Writing (NA-WRITE) requires $a \rightsquigarrow_{\mathrm{na}} \_$, which represents exclusive ownership of $a$, and updates it to $a \rightsquigarrow_{\mathrm{na}} v$.

We expect the reader to find these rules unsurprising: indeed, they are identical to the rules that govern access to memory in Concurrent Separation Logic with fractional permissions (Bornat et al., 2005). In the absence of a mechanism that allows a points-to assertion to be shared between several threads,[2] these rules forbid data races. In Cosmo, as explained earlier,

[2]Sharing an assertion between several threads is usually permitted either via a runtime synchronization mechanism, such as a critical region (O'Hearn, 2007, Section 4) or a lock (Gotsman et al., 2007; Hobor et al., 2008), or

PURE

$$\frac{\forall m, e'', e_1', \ldots e_n'.\ (e \xrightarrow{m} e'', e_1', \ldots, e_n') \implies (m = \varepsilon \wedge e'' = e' \wedge n = 0) \qquad e \xrightarrow{\varepsilon} e' \qquad \triangleright \{P\}\, e'\, \{\Phi\}}{\{P\}\, e\, \{\Phi\}}$$

VALUE

$$\{\mathsf{True}\}\ v\ \{\lambda v'.\ v' = v\}$$

FORK

$$\frac{\triangleright \{P\}\, e\, \{\lambda \_.\ \mathsf{True}\}}{\{P\}\ \texttt{fork}\ e\ \{\lambda v'.\ v' = ()\}}$$

(a) Cosmo rules for pure steps and the "fork" operation

NA-ALLOC

$$\left\{n \geq 0\right\}\ \texttt{array}_{\mathrm{na}}[n]\ v\ \left\{\lambda v'.\ \exists \ell.\ v' = \ell\ *\ \ell \rightsquigarrow^*_{\mathrm{na}} [v, \ldots, v] \quad (n \text{ cells})\right\}$$

NA-ALLOC-ONE

$$\left\{\mathsf{True}\right\}\ \texttt{ref}_{\mathrm{na}}\ v\ \left\{\lambda v'.\ \exists a.\ v' = a\ *\ a \rightsquigarrow_{\mathrm{na}} v\right\}$$

NA-READ

$$\left\{q \cdot a \rightsquigarrow_{\mathrm{na}} v\right\}\ !_{\mathrm{na}}\, a\ \left\{\lambda v'.\ v' = v\ *\ q \cdot a \rightsquigarrow_{\mathrm{na}} v\right\}$$

NA-WRITE

$$\left\{a \rightsquigarrow_{\mathrm{na}} \_\right\}\ a :=_{\mathrm{na}} v\ \left\{\lambda ().\ a \rightsquigarrow_{\mathrm{na}} v\right\}$$

AT-ALLOC

$$\left\{n \geq 0\ *\ \uparrow \mathcal{V}\right\}\ \texttt{array}_{\mathrm{at}}[n]\ v\ \left\{\lambda v'.\ \exists \ell.\ * \begin{cases} v' = \ell \\ \ell \rightsquigarrow^*_{\mathrm{at}} [\langle v, \mathcal{V}\rangle, \ldots, \langle v, \mathcal{V}\rangle] \quad (n \text{ cells}) \end{cases}\right\}$$

AT-ALLOC-ONE

$$\left\{\uparrow \mathcal{V}\right\}\ \texttt{ref}_{\mathrm{at}}\ v\ \left\{\lambda v'.\ \exists A.\ * \begin{cases} v' = A \\ A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \end{cases}\right\}$$

AT-READ

$$\left\{q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle\right\}\ !_{\mathrm{at}}\, A\ \left\{\lambda v'.\ * \begin{cases} v' = v \\ q \cdot A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V}\rangle \\ \uparrow \mathcal{V} \end{cases}\right\}$$

AT-WRITE

$$\left\{A \rightsquigarrow_{\mathrm{at}} \langle \_, \mathcal{V}\rangle\ *\ \uparrow \mathcal{V}'\right\}\ A :=_{\mathrm{at}} v\ \left\{\lambda ().\ * \begin{cases} A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \sqcup \mathcal{V}'\rangle \\ \uparrow \mathcal{V} \end{cases}\right\}$$

CAS-FAILURE

$$\left\{v_0 \neq v_1\ *\ A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle\right\}\ \texttt{CAS}\ A\ v_1\ v_2\ \left\{\lambda v'.\ * \begin{cases} v' = \texttt{false} \\ A \rightsquigarrow_{\mathrm{at}} \langle v_0, \mathcal{V}\rangle \\ \uparrow \mathcal{V} \end{cases}\right\}$$

CAS-SUCCESS

$$\left\{A \rightsquigarrow_{\mathrm{at}} \langle v_1, \mathcal{V}\rangle\ *\ \uparrow \mathcal{V}'\right\}\ \texttt{CAS}\ A\ v_1\ v_2\ \left\{\lambda v'.\ * \begin{cases} v' = \texttt{true} \\ A \rightsquigarrow_{\mathrm{at}} \langle v_2, \mathcal{V} \sqcup \mathcal{V}'\rangle \\ \uparrow \mathcal{V} \end{cases}\right\}$$

LENGTH

$$\left\{\ell\ \mathsf{has\ length}\ n\right\}\ \texttt{length}\ \ell\ \left\{\lambda v'.\ v' = n\right\}$$

(b) Cosmo triples for the memory access operations

AT-ALLOC-ONE-SC

$$\left\{\mathsf{True}\right\}\ \texttt{ref}_{\mathrm{at}}\ v\ \left\{\lambda v'.\ \exists A.\ v' = A\ *\ A \rightsquigarrow_{\mathrm{at}} v\right\}$$

AT-READ-SC

$$\left\{q \cdot A \rightsquigarrow_{\mathrm{at}} v\right\}\ !_{\mathrm{at}}\, A\ \left\{\lambda v'.\ v' = v\ *\ q \cdot A \rightsquigarrow_{\mathrm{at}} v\right\}$$

AT-WRITE-SC

$$\left\{A \rightsquigarrow_{\mathrm{at}} \_\right\}\ A :=_{\mathrm{at}} v\ \left\{\lambda ().\ A \rightsquigarrow_{\mathrm{at}} v\right\}$$

CAS-FAILURE-SC

$$\left\{v_0 \neq v_1\ *\ A \rightsquigarrow_{\mathrm{at}} v_0\right\}\ \texttt{CAS}\ A\ v_1\ v_2\ \left\{\lambda v'.\ v' = \texttt{false}\ *\ A \rightsquigarrow_{\mathrm{at}} v_0\right\}$$

CAS-SUCCESS-SC

$$\left\{A \rightsquigarrow_{\mathrm{at}} v_1\right\}\ \texttt{CAS}\ A\ v_1\ v_2\ \left\{\lambda v'.\ v' = \texttt{true}\ *\ A \rightsquigarrow_{\mathrm{at}} v_2\right\}$$

(c) Derived Cosmo triples for atomic memory cells ignoring views

Figure 4.3: Cosmo reasoning rules

a non-atomic points-to assertion cannot appear in an invariant, as it is not an objective assertion. Therefore, Cosmo forbids data races on non-atomic memory cells. In other words, for a program to be verifiable in Cosmo, all accesses to non-atomic memory cells must be properly synchronized via other means, such as reads and writes of atomic memory cells. By contrast, BaseCosmo allows to reason about all programs that are safe with respect to the operational semantics of Dolan et al., and the latter gives a well-defined (albeit nondeterministic) semantics to racy uses of non-atomic cells.

### 4.5.2 Operations on atomic cells

The next six rules in Figure 4.3b describe the operations on atomic memory cells, namely allocation, reading, writing, and CAS. They are analogous to their BaseCosmo counterparts (Figure 3.2), yet simpler, as the validity assertions have vanished, and this thread's view is no longer named: instead, assertions of the form $\uparrow \mathcal{V}$ are used to indicate partial knowledge of this thread's view.

These rules deal with two aspects of atomic memory cells, namely the fact that an atomic memory cell holds a value, and the fact that it holds a view. Fortunately, these two aspects are essentially independent of one another. Furthermore, the second aspect can be ignored when it is not relevant: indeed, from the rules of Figure 4.3b, one can easily derive a set of simplified rules, shown in Figure 4.3c. These derived rules are the standard rules that govern access to memory in Concurrent Separation Logic with fractional permissions.

In Cosmo, because an atomic points-to assertion is objective, it *can* be shared between several threads, via an Iris invariant. Therefore, the rules of both Figure 4.3b and Figure 4.3c allow data races on atomic memory cells. By using just the derived rules of Figure 4.3c, one can reason in Cosmo about atomic memory cells exactly in the same way as one reasons in Concurrent Separation Logic under the assumption of sequential consistency (Parkinson et al., 2007).

Allocating atomic memory cells (AT-ALLOC) requires this thread to have some view $\mathcal{V}$, as witnessed by the precondition $\uparrow \mathcal{V}$. In the postcondition, one obtains atomic points-to assertions $\ell[i] \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \rangle$ for $0 \leq i < n$, which represent the exclusive ownership of the fresh memory cells $\ell[i]$. Such an assertion states that a memory cell $\ell[i]$ holds the value $v$ and a view that is at least as good as the view $\mathcal{V}$. Again, a simplified rule (AT-ALLOC-ONE) can be derived for allocating an atomic reference. Notice that, since $\uparrow \mathcal{V}$ is persistent, it still holds in the postcondition with no need to repeat it. Besides, both these allocation rules offer flexibility regarding the choice of $\mathcal{V}$. Indeed, $\mathcal{V}$ need not reflect all of the information that is currently available to this thread: it can be a partial view. (Recall that $\uparrow \cdot$ is anti-monotone.) In fact, if desired, one can always take $\mathcal{V}$ to be the empty view. (Recall that $\uparrow \varnothing$ can be obtained out of thin air.) This is how the derived rule AT-ALLOC-ONE-SC is obtained.

Reading an atomic memory cell (AT-READ) requires having a fractional points-to assertion $q \cdot A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \rangle$ and preserves it. The last conjunct of the postcondition, $\uparrow \mathcal{V}$, reflects the fact that the view held at cell $A$ becomes part of this thread's view: this is an "acquire read". The derived rule AT-READ-SC is obtained by dropping this information.

Writing an atomic memory cell (AT-WRITE) requires having an exclusive points-to assertion $A \rightsquigarrow_{\text{at}} \langle \_, \mathcal{V} \rangle$ as well as (possibly partial) information about this thread's view $\uparrow \mathcal{V}'$. In the

by a purely ghost mechanism, such as an invariant that can be accessed for the duration of an atomic instruction, as in some variants of Concurrent Separation Logic (Parkinson et al., 2007) and in Iris (Jung et al., 2018b).

postcondition, the points-to assertion is updated to $A \rightsquigarrow_{\text{at}} \langle v, \mathcal{V} \sqcup \mathcal{V}' \rangle$, which reflects the fact that both the value $v$ and the view $\mathcal{V}'$ are written to the memory cell $A$: this is a "release write". Furthermore, the second conjunct of the postcondition, $\uparrow \mathcal{V}$, indicates that the view $\mathcal{V}$ is acquired by this thread; indeed, an atomic write has both "release" and "acquire" effects. Again, $\uparrow \mathcal{V}'$ need not be repeated in postcondition, the user can weaken the view $\mathcal{V}'$ at will, and it is possible to ignore these details when they are irrelevant. In particular, the rule AT-WRITE-SC is obtained by letting $\mathcal{V}$ remain undetermined, by letting $\mathcal{V}'$ be the empty view, and by dropping $\uparrow \mathcal{V}$ from the postcondition.

CAS-FAILURE and CAS-SUCCESS combine the rules for reading and writing an atomic cell. A failed CAS instruction does not affect the content of the memory cell, but still acquires a view $\mathcal{V}$ from it. A successful CAS instruction writes a value and a view to the memory cell and acquires a view from it. Again, if one does not care about these information transfers, then one can use the simplified rules CAS-FAILURE-SC and CAS-SUCCESS-SC in Figure 4.3c.

# Chapter 5

# Locks and mutual exclusion

We now illustrate the use of Cosmo by proving the functional correctness of several simple concurrent data structures: a spin lock (§5.2), a ticket lock (§5.3), a lock based on Dekker's algorithm (§5.5) and one based on the similar Peterson's algorithm (§5.6); the last two support at most two threads.

## 5.1 Specification of locks

The lock is perhaps the most basic and best known concurrent data structure. It supports three operations, namely `make`, `acquire`, and `release`. One way of describing its purpose is to say that it allows achieving *mutual exclusion* between several threads: that is, the `acquire` and `release` operations delimit *critical sections* in the code, and the purpose of the lock is to guarantee that no two threads can be in a critical section "at the same time". However, this is not a very good description, especially in a weak memory setting, where the intuitive notions of "time" and "simultaneity" do not make much sense. What matters, really, is that a lock mediates access to a shared resource, and does so in a correct manner. A thread that successfully acquires the lock expects the resource to be in a consistent state, and expects to be allowed to affect this state in arbitrary ways, as long as it brings the resource back to a consistent state by the time it releases the lock.

Concurrent Separation Logic can express that idea in a simple and elegant manner (O'Hearn, 2007). The classic specification of dynamically-allocated locks in Concurrent Separation Logic (Gotsman et al., 2007; Hobor et al., 2008; Buisse et al., 2011; Sieczkowski et al., 2015; Birkedal

$$\underbrace{\{P}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\underbrace{\}}$$
$$\texttt{make ()}$$
$$\underbrace{\left\{\lambda \ell.\ \exists \mathsf{locked}.\ *\left\{\begin{array}{l}\{\mathsf{True}\}\ \texttt{acquire}\ \ell\ \{\lambda().\ P * \mathsf{locked}\}\\ \{P * \mathsf{locked}\}\ \texttt{release}\ \ell\ \{\lambda().\ \mathsf{True}\}\end{array}\right.\right\}}$$

Figure 5.1: A specification of the "lock" data structure

and Bizjak, 2018), a version of which appears in Figure 5.1, allows the user to choose an *invariant* $P$ when a lock is allocated. This invariant is a Separation Logic assertion. It appears in the postcondition of `acquire` and in the precondition of `release`, which means that a thread that acquires the lock gets access to the invariant, and can subsequently break this invariant if desired, while a thread that releases the lock must restore and relinquish the invariant.

In Figure 5.1, the entire specification is contained in a triple for `make`, whose precondition requires the invariant to hold initially, and whose postcondition contains triples for `acquire` and `release`. Because a triple is persistent (Jung et al., 2018b), therefore duplicable, several threads can share the use of the newly-created lock.

In addition to the invariant $P$, an abstract assertion "locked" appears in the postcondition of `acquire` and in the precondition of `release`. Because it is abstract, it must be regarded as nonduplicable by the user. Therefore, it can be thought of as a "token", a witness that the lock is currently held. This witness is required and consumed by `release`. There are two reasons why it is desirable to let such a token appear in the specification. First, from a user's point of view, this is a useful feature, as it forbids releasing a lock that one does not hold, an error that could arise in the (somewhat unlikely) situation where several copies of the invariant $P$ can coexist. Second, from an implementor's point of view, this makes the specification easier to satisfy. In a ticket lock implementation (§5.3), for instance, only the thread that holds the lock can safely release it. Thus, a ticket lock does not satisfy a stronger specification of locks where the "locked" token is omitted.

If one can prove that an implementation of locks satisfies this specification, then (because Separation Logic is compositional) a user can safely rely on this specification to *reason* about her use of locks in an application program. This is a necessary and sufficient condition for an implementation of locks to be considered correct. Proving something along the lines of "no two threads can be in a critical section at the same time" is not necessary, nor would it be sufficient. It would not provide a means of reasoning about user programs. And, in a weak memory setting, a lock might actually guarantee mutual exclusion and nevertheless be broken, failing to achieve the necessary degree of synchronization.

We emphasize that, when this specification is understood in the setting of Cosmo, the user invariant $P$ is an arbitrary Cosmo assertion, thus possibly a subjective assertion. Indeed, the synchronization performed by the lock at runtime ensures that every thread has a consistent view of the shared resource. This is in contrast with Iris invariants, which involve no runtime synchronization, and therefore must be restricted to objective assertions (§4.1).

We must also emphasize that, because we work in a logic of partial correctness, this specification does not guarantee deadlock freedom, nor does it guarantee any form of fairness. As an extreme example, an implementation where `acquire` diverges satisfies the specification in Figure 5.1.

Proving the correctness of a lock implementation against the abstract lock interface shown in Figure 5.1 consists of proving a triple for `make`. In the body of `make`, once all the physical locations are allocated and are in scope, one would typically create the ghost state that the proof needs (if any), establish an invariant that governs the data structure, instantiate "locked", then prove the mandated triples for `acquire` and `release`.

```
let make () =
  | ref_at false

let acquire ℓ =
  | while ¬(CAS ℓ false true) do () done

let release ℓ =
  | ℓ :=_at false
```

Figure 5.2: Implementation of a spin lock

## 5.2 A spin lock

A spin lock is a simple implementation of a lock. It relies on a single atomic memory reference $\ell$, which holds a Boolean value. Its implementation in Multicore OCaml appears in Figure 5.2. The implementation of `acquire` involves busy-waiting in a loop, whence the name "spin lock".

An Iris proof of correctness of a spin lock (Birkedal and Bizjak, 2018, Example 7.36), in a traditional sequentially consistent setting, would rely on the following Iris invariant, which states that either the lock $\ell$ is currently available and its user assertion $P$ holds, or the lock is currently held:

$$\mathsf{isSpinLockSC} \triangleq \boxed{(\ell \rightsquigarrow \texttt{false} * P) \ \vee \ (\ell \rightsquigarrow \texttt{true})}$$

In our setting, however, such an invariant does not make sense, and cannot even be expressed, because the assertion $P$ is an arbitrary assertion of type VPROP, whereas Cosmo requires every Iris invariant $\boxed{I}$ to be formed with an *objective* assertion $I$ of type IPROP (§4.1).

We work around this restriction by reformulating this invariant under an objective form: "either the lock is currently available and the user assertion $P$ holds *in the eyes of the thread that last released the lock*, or the lock is currently held". Cosmo allows expressing this easily:

$$\mathsf{isSpinLock} \triangleq \boxed{(\exists \mathcal{V}.\, \ell \rightsquigarrow_{\mathrm{at}} \langle \texttt{false}, \mathcal{V} \rangle * P \mathbin{@} \mathcal{V}) \ \vee \ (\ell \rightsquigarrow_{\mathrm{at}} \texttt{true})}$$

The left-hand disjunct, which describes the situation where the lock is available, now involves an existential quantification over a view $\mathcal{V}$. The atomic points-to assertion $\ell \rightsquigarrow_{\mathrm{at}} \langle \texttt{false}, \mathcal{V} \rangle$ indicates that $\mathcal{V}$ is the view currently stored at reference $\ell$. The objective assertion $P \mathbin{@} \mathcal{V}$ indicates that $P$ holds at this view. That is, the assertion $P$ holds in the eyes of whomever last wrote the reference $\ell$. (We use the present-tense "holds", as opposed to the past-tense "held", because every Cosmo assertion is monotonic in its implicit view parameter.) In other words, $P$ holds in the eyes of the thread that last released the lock.

The right-hand disjunct, which describes the case where the lock is held, uses the simplified points-to assertion $\ell \rightsquigarrow_{\mathrm{at}} \texttt{true}$, which is sugar for $\ell \rightsquigarrow_{\mathrm{at}} \langle \texttt{true}, \varnothing \rangle$ (§4.5). In this case, the view stored at reference $\ell$ is irrelevant.

Proving that the spin lock satisfies the specification in Figure 5.1 then goes as follows. We prove the triple of function `make`. In the body of `make`, for the lock $\ell$ that has just been allocated and the user assertion $P$ that was chosen by the user, we establish the invariant isSpinLock shown above. We do not need any additional ghost state. Since a spin lock has no need for a "locked" token, we let "locked" be True. With the lock invariant in context, we then proceed to proving the triples for `acquire` and `release`. Their proofs are sketched in the following.

The proof of `acquire` amounts to establishing the following triple for the CAS instruction:

$$\{\mathsf{isSpinLock}\}\ \texttt{CAS}\ \ell\ \texttt{false}\ \texttt{true}\ \{\lambda b.\, b = \texttt{true} \implies P\}$$

This triple guarantees that if the CAS succeeds then one can extract the assertion $P$. To establish it, we must open the invariant isSpinLock for the duration of the CAS instruction (Jung et al.,

2018b, §2.2) and reason separately about the case where the lock is available and the case where it is held. In the latter case, we apply the reasoning rule CAS-FAILURE (Figure 4.3b) and conclude easily. In the former case, we have $\ell \rightsquigarrow_{\text{at}} \langle \texttt{false}, \mathcal{V} \rangle$ and $P \mathbin{@} \mathcal{V}$, for an unknown view $\mathcal{V}$. The rule CAS-SUCCESS (instantiated with the empty view for $\mathcal{V}'$) shows that the outcome of the CAS instruction in this case is $\ell \rightsquigarrow_{\text{at}} \texttt{true} * \uparrow \mathcal{V}$. In other words, the CAS instruction achieves the double effect of writing `true` to the reference $\ell$ and acquiring the view $\mathcal{V}$ that was stored at this reference by the last write. This is exactly what we need. Indeed, the axiom SPLIT-SUBJECTIVE-OBJECTIVE lets us combine $P \mathbin{@} \mathcal{V}$ and $\uparrow \mathcal{V}$ to obtain $P$. By performing an "acquire" read, we have ensured that $P$ holds in the eyes of the thread that has just acquired the lock. Furthermore, the points-to assertion $\ell \rightsquigarrow_{\text{at}} \texttt{true}$ allows us to establish the right-hand disjunct of the invariant isSpinLock and to close it again.

The proof of `release` exploits SPLIT-SUBJECTIVE-OBJECTIVE in the reverse direction. Per the precondition of `release`, we have $P$. We split it into two assertions $P \mathbin{@} \mathcal{V}$ and $\uparrow \mathcal{V}$, where $\mathcal{V}$ is technically a fresh unknown view, but can be thought of as this thread's view. Then, we open the invariant isSpinLock for the duration of the write instruction. We do not know which of the two disjuncts is currently satisfied (indeed, we have no guarantee that the lock is currently held), but we find that, in either case, we have $\ell \rightsquigarrow_{\text{at}} \_$. This allows us to apply the reasoning rule AT-WRITE (Figure 4.3b), which guarantees that, after the write instruction, we have $\ell \rightsquigarrow_{\text{at}} \langle \texttt{false}, \mathcal{V} \rangle$. In other words, because we have performed a "release" write, we know that, after this write, our view of memory is stored at reference $\ell$. Because we have $\ell \rightsquigarrow_{\text{at}} \langle \texttt{false}, \mathcal{V} \rangle$ and $P \mathbin{@} \mathcal{V}$, we are able to prove that the left-hand disjunct of isSpinLock holds and to close the invariant.

## 5.3 A ticket lock

The ticket lock is a variant of the spin lock where a "ticket dispenser" is used to serve threads in the order that they arrive, thereby achieving a certain level of fairness. A simple implementation of the ticket lock appears in Figure 5.3. A lock is a pair $(\texttt{served}, \texttt{next})$ of two atomic references `served` and `next`, each of which stores an integer value. The counter `served` displays the number of the ticket that is currently being served, or ready to be served. The counter `next` displays the number of the next available ticket.

A thread wishing to acquire the lock first obtains a unique number $n$ from the counter `next`, which it increments at the same time. This number is known as a *ticket*. Then, the thread waits for its number to be called: that is, it waits for the counter `served` to contain the value $n$. When it observes that this is the case, it concludes that it has acquired the lock.

A thread that wishes to release the lock simply increments the counter `served`, so that the next thread in line is allowed to proceed and take the lock. To do so, a CAS instruction is unnecessary: a sequence of a read instruction, an addition, and a write instruction suffices. Indeed, because the lock is held and can be released by only one thread, no interference from other threads is possible. This argument must be explicitly made somewhere in the proof, so an exclusive token "locked" is required.

We now sketch a proof of this ticket lock implementation. Our proof requires a straightforward modification of the proof carried out by Birkedal and Bizjak in a sequentially consistent setting (2018, §9). That is precisely our point: it is our belief and our hope that, in many cases, a traditional Iris proof can be easily ported to Cosmo. The process is mostly a matter of identi-

```
let make () =
  | let served = ref_at 0 in
  | let next = ref_at 0 in
  | (served, next)

let rec release ℓ =
  | let (served, next) = ℓ in
  | served :=_at (!_at served) + 1

let rec acquire ℓ =
  | let (served, next) = ℓ in
  | let n = !_at next in
  | if CAS next n (n + 1) then
  |   | while !_at served ≠ n do () done
  | else
  |   | acquire ℓ
```

Figure 5.3: Implementation of a ticket lock

$$\begin{array}{rcl@{\qquad}rcl}
\mathit{served} & : & \textsc{Auth}(\textsc{Ex}(\mathbb{N})) & \mathit{issued} & : & \textsc{Auth}(\textsc{DisjointSet}(\mathbb{N})) \\
\mathsf{locked} & \triangleq & \exists s.\, \boxed{\circ\, s}^{\mathit{served}} & \mathsf{ticket}\ n & \triangleq & \boxed{\circ\, \{n\}}^{\mathit{issued}}
\end{array}$$

$$\mathsf{isTicketLock} \triangleq \boxed{\exists s, n, \mathcal{V}.\, \ast \begin{cases} \mathtt{served} \rightsquigarrow_{\mathrm{at}} \langle s, \mathcal{V} \rangle \ \ast\ \mathtt{next} \rightsquigarrow_{\mathrm{at}} n \\ \boxed{\bullet\, s}^{\mathit{served}} \ \ast\ \boxed{\bullet\, [0, n)}^{\mathit{issued}} \\ (\mathsf{locked} \ast P \mathbin{@} \mathcal{V}) \vee \mathsf{ticket}\ s \end{cases}}$$

Figure 5.4: Internal invariant of a ticket lock

fying which atomic memory cells also serve as information transmission channels (thanks to the "release/acquire" semantics of atomic writes and reads) and of decorating every Iris invariant with explicit views, where needed, so as to meet the requirement that every invariant must be objective. Here, a moment's thought reveals that only the view stored at the reference `served` matters; the view stored at `next` is irrelevant.

The ghost state, the invariant and the definition of "locked"[1] used in the verification of the ticket lock appear in Figure 5.4.

- The ghost variable *served* stores an element of the camera $\textsc{Auth}(\textsc{Ex}(\mathbb{N}))$. The exclusive assertion $\boxed{\circ\, s}^{\mathit{served}}$ represents both the knowledge that ticket $s$ is currently being served and a permission to change who is being served. The exclusive assertion "locked", defined as $\exists s.\, \boxed{\circ\, s}^{\mathit{served}}$, represents a permission to change who is being served, therefore a permission to release the lock.

- The ghost variable *issued* keeps track of which tickets have been issued since the lock was created. It stores an element of $\textsc{Auth}(\textsc{DisjointSet}(\mathbb{N}))$, where $\textsc{DisjointSet}\ \mathbb{N}$ is the camera whose elements are finite sets of integers and whose partial composition law is disjoint set union. The assertion $\mathsf{ticket}\ n$ represents the ownership of the ticket numbered $n$. It is exclusive in the sense that $\mathsf{ticket}\ n_1 \ast \mathsf{ticket}\ n_2$ entails $n_1 \neq n_2$.

- The invariant $\mathsf{isTicketLock}$ synchronizes the physical state and the ghost state by mentioning the auxiliary variables $s$ and $n$ both in points-to assertions and in ghost state ownership

[1] Recall that these are established in the body of `make`, where the newly-allocated references `served` and `next` are in scope.

$$\{P\}$$
$$\texttt{make ()}$$
$$\left\{\lambda(\ell_0, \ell_1).\ \exists \mathsf{canLock}, \mathsf{locked}.\ \circledast_{i\in\{0,1\}} \circledast \left\{\begin{array}{c} \mathsf{canLock}\ i \\ \{\mathsf{canLock}\ i\}\ \texttt{acquire}\ \ell_i\ \{\lambda().\ P * \mathsf{locked}\ i\} \\ \{P * \mathsf{locked}\ i\}\ \texttt{release}\ \ell_i\ \{\lambda().\ \mathsf{canLock}\ i\} \end{array}\right.\right\}$$

Figure 5.5: A specification of a lock that can be used by two threads

assertions. The same technique as in the previous subsection (§5.2) is used to make this invariant objective. The last conjunct in its definition states that either no thread holds the lock and the user assertion $P$ holds *in the eyes of the thread that last released the lock*, or the invariant owns the ticket numbered $s$. This implies that, in order to acquire the lock while maintaining the invariant, a thread must present and relinquish the ticket numbered $s$.

A thread that wishes to enter the waiting line manufactures a fresh ticket by incrementing `next`, then is allowed to wait, as a specification for the waiting loop is:

$$\{\mathsf{ticket}\ n\}\ \texttt{while}\ !_{\mathrm{at}}\ \texttt{served} \neq n\ \texttt{do ()}\ \texttt{done}\ \{\lambda().\ \mathsf{locked} * P\}$$

This specification is proven as follows. If a thread can present the exclusive ticket numbered $s$ where $s$ is the value of `served`, then the invariant cannot simultaneously hold that ticket, hence the left-hand disjunct of the disjunction $(\mathsf{locked} * P \mathbin{@} \mathcal{V}) \vee \mathsf{ticket}\ s$ in the invariant currently holds.[2] This allows the thread to take ownership of that left-hand disjunct, and transition the invariant to the right-hand disjunct by giving up its ticket.

Conversely, a thread that releases the lock does not have the required ticket at hand, so must re-establish the invariant.

By comparison to a proof in a sequentially consistent setting, the only addition to account for weak memory is the fact that the read of `served` in `acquire` acquires a view and the write of `served` in `release` releases a view.

## 5.4 Specification of mutual exclusion for two threads

We now study a related but slightly differing class of data structures: that which implement mutual exclusion between two participants. For such data structures, we will consider the interface shown in Figure 5.5. Again it is presented in a higher-order style.

Like ordinary locks, such data structures guard an arbitrary, possibly subjective assertion $P$, and offer three operations, namely `make`, `acquire` and `release`. This time, `make` returns a pair of two handles, one for each participant. The specification is analogous to that of locks (Figure 5.1), but introduces a new token, $\mathsf{canLock}\ i$, so as to limit the number of participants to two: `make` produces only two tokens $\mathsf{canLock}\ 0$ and $\mathsf{canLock}\ 1$, and the user has no way to

[2] This is the "golden idol" metaphor of Kaiser et al. (2017).

```
let make () =
  | let (turn, flag_0, flag_1) = (ref_at 0, ref_at false, ref_at false) in
  | ((turn, 0, flag_0, flag_1), (turn, 1, flag_1, flag_0))

let acquire ℓ =
  | let (turn, myTid, myFlag, otherFlag) = ℓ in
  | myFlag :=_at true;
  | while !_at otherFlag = true do
  |   | if !_at turn ≠ myTid then {
  |   |   | myFlag :=_at false;
  |   |   | while !_at turn ≠ myTid do () done;
  |   |   | myFlag :=_at true
  |   |   }
  | done

let release ℓ =
  | let (turn, myTid, myFlag, otherFlag) = ℓ in
  | turn :=_at 1 − myTid;
  | myFlag :=_at false
```

Figure 5.6: Dekker's algorithm for two threads

forge more of them. A participant thread holds a token canLock $i$ when it is outside of a critical section, and trades it for locked $i$ and the user assertion $P$ when inside a critical section.

In the following sections, we verify two related implementations of that specification for mutual exclusion: Dekker's algorithm (§5.5) and Peterson's algorithm (§5.6).

## 5.5 Dekker's algorithm

Dekker's algorithm is among the first mutual exclusion algorithms to be invented. It ensures some form of fairness and its implementation requires only read and write operations on sequentially consistent registers: it does not require a CAS operation. Dekker's algorithm remains valid in Multicore OCaml, provided atomic references are used in its implementation. This is not an entirely trivial claim, as the user-chosen invariant $P$ may be subjective.

The code, shown in Figure 5.6, uses three atomic references:

- Each of the two threads $i \in \{0, 1\}$ uses a Boolean register $\texttt{flag}_i$ to indicate it currently holds the lock or intends to acquire it. Both threads read $\texttt{flag}_i$ but only thread $i$ writes it.
- An integer register `turn` indicates which thread has priority, should both threads simultaneously attempt to acquire the lock. It is read and written by both threads.

When, as one of the two participant threads, we want to acquire the lock, we signal it by setting our own flag to `true`. We then wait for the other thread to either release the lock or to

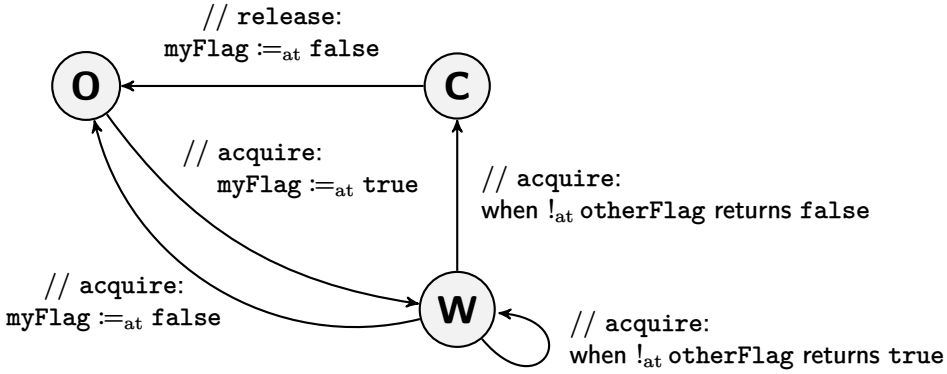



Figure 5.7: Internal (ghost) state of each thread participating in Dekker's algorithm

$$
\begin{array}{rcl}
f_0, f_1 & \in & \textsc{Bool} \\
s_0, s_1 & \in & \textsc{DekkerState} \triangleq \{\mathsf{O}, \mathsf{W}, \mathsf{C}\} \\
\gamma_0, \gamma_1 & : & \textsc{Auth}(\textsc{Ex}(\textsc{DekkerState})) \\
\omega_0, \omega_1 & : & \textsc{Auth}(\textsc{Ex}(\textsc{View}))
\end{array}
$$

$$
\mathsf{dekkerInv} \triangleq \exists f_0, f_1, s_0, s_1, \mathcal{V}_0, \mathcal{V}_1.\ *\begin{cases}
\texttt{turn} \rightsquigarrow_{\text{at}} \_ \\
\texttt{flag}_0 \rightsquigarrow_{\text{at}} \langle f_0, \mathcal{V}_0 \rangle * \boxed{\bullet\, s_0}^{\gamma_0} * \boxed{\bullet\, \mathcal{V}_0}^{\omega_0} * (s_0 = \mathsf{O} \vee f_0 = \texttt{true}) \\
\texttt{flag}_1 \rightsquigarrow_{\text{at}} \langle f_1, \mathcal{V}_1 \rangle * \boxed{\bullet\, s_1}^{\gamma_1} * \boxed{\bullet\, \mathcal{V}_1}^{\omega_1} * (s_1 = \mathsf{O} \vee f_1 = \texttt{true}) \\
(s_0 \neq \mathsf{C} \wedge s_1 \neq \mathsf{C}) \implies P \mathbin{@} (\mathcal{V}_0 \sqcup \mathcal{V}_1)
\end{cases}
$$

$$
\mathsf{canLock}\ i \triangleq \mathsf{dekkerInv} * \exists \mathcal{V}.\ \boxed{\circ\, \mathsf{O}}^{\gamma_i} * \boxed{\circ\, \mathcal{V}}^{\omega_i} * \uparrow \mathcal{V}
$$

$$
\mathsf{locked}\ i \triangleq \mathsf{dekkerInv} * \exists \mathcal{V}.\ \boxed{\circ\, \mathsf{C}}^{\gamma_i} * \boxed{\circ\, \mathcal{V}}^{\omega_i}
$$

Figure 5.8: Internal invariant of Dekker's algorithm

retract from its intention to acquire it, by waiting until that other thread's flag is `false`. While waiting, we check who has priority, as given by the reference `turn`. If priority goes to the other thread, then we retract by setting our flag back to `false`—thus letting the other thread proceed to the critical section—, we wait until priority switches to us, and we signal again our intention to acquire the lock.

When releasing the lock, we switch `turn` so as to give priority to the other thread, then we set our own flag to `false`.

We now outline our proof that the code in Figure 5.6 satisfies the specification in Figure 5.5. The algorithm's possible states are as follows: at any time, each of the participant threads is in one of three possible states: either it is outside of a critical section ($\mathsf{O}$), or it has set its flag to `true` and is now waiting ($\mathsf{W}$), or it is inside a critical section ($\mathsf{C}$). Figure 5.7 summarizes these states and the transitions between them. Figure 5.8 presents the algorithm's invariant. We use two ghost variables $\gamma_0$ and $\gamma_1$ to keep track of the logical state of each thread.

Note that, while the use of `turn` avoids deadlocks and ensures fairness, it plays no role in functional correctness. According to the invariant, neither the value of `turn` or, in a weak memory setting, the synchronization it performs, are relevant. Effectively, should this reference be removed and the outer loop in `acquire` be reduced to an empty body, just spinning until `otherFlag` is `false`, then the code would still satisfy the functional specification—but it may deadlock. Hence, it is important to emphasize that we use a logic of partial correctness, in which we do not prove deadlock-freedom or fairness. We only verify that this data structure behaves like a lock.

Let us sketch the crucial step in the proof of `acquire`. While waiting to acquire the lock, a thread knows that its own state is $\mathsf{W}$ (because it holds $\boxed{\circ\,\mathsf{W}}^{\gamma_i}$). As per the conjunct $(s_{1-i} = \mathsf{O} \vee f_{1-i} = \texttt{true})$ of the invariant, when the waiting thread reads `false` in `otherFlag`, it learns that the other thread's state is $\mathsf{O}$. Thus, at that moment, none of the two participants are in state $\mathsf{C}$ so, from the last conjunct of the invariant, the waiting thread can take $P \mathbin{@} (\mathcal{V}_0 \sqcup \mathcal{V}_1)$ by switching its own state to $\mathsf{C}$.

The key novelty, with respect to the proof that one could carry out in a sequentially consistent setting, is that the invariant must record the view $\mathcal{V}_i$ that was the view of thread $i$ when this thread last released the lock. This view is stored at the atomic reference $\texttt{flag}_i$. When the lock is available, because the lock could have been last released by either thread, the assertion $P$ holds either at $\mathcal{V}_0$ or at $\mathcal{V}_1$. Because Cosmo assertions are monotonic, this implies that $P$ holds at the combined view $\mathcal{V}_0 \sqcup \mathcal{V}_1$. The invariant records this fact.

Thus, when the outer loop in `acquire` terminates, the invariant can give us $P \mathbin{@} (\mathcal{V}_0 \sqcup \mathcal{V}_1)$. A key step is to argue that we also have $\uparrow \mathcal{V}_0$ and $\uparrow \mathcal{V}_1$ at this moment of the program, so as to then combine the pieces via Split-Subjective-Objective and obtain $P$. Let us sketch why this is the case.

- Establishing $\uparrow \mathcal{V}_i$ is easy enough, because $\mathcal{V}_i$ is a past view of this very thread. One way of recording this information is to let the token $\mathsf{canLock}\ i$ carry it; indeed, the role of this token is precisely to carry information from a `release` to the next `acquire`. We include in the definition of $\mathsf{canLock}\ i$ an assertion $\uparrow \mathcal{V}$ and set up a ghost variable $\omega_i$ whose purpose is to allow us to prove that $\mathcal{V}$ is in fact $\mathcal{V}_i$.

- The wait terminates immediately after reading $\texttt{flag}_{1-i}$ and getting the value `false`. Thus, thanks to the invariant, the thread acquires $\mathcal{V}_{1-i}$ via this read.

```
let make () =
    | let (turn, flag_0, flag_1) = (ref_at 0, ref_at false, ref_at false) in
    | ((turn, 0, flag_0, flag_1), (turn, 1, flag_1, flag_0))

let acquire ℓ =
    | let (turn, myTid, myFlag, otherFlag) = ℓ in
    | myFlag :=_at true;
    | turn :=_at 1 − myTid;
    | while (!_at otherFlag = true ∧ !_at turn = 1 − myTid) do () done

let release ℓ =
    | let (turn, myTid, myFlag, otherFlag) = ℓ in
    | myFlag :=_at false
```

Figure 5.9: Peterson's algorithm for two threads

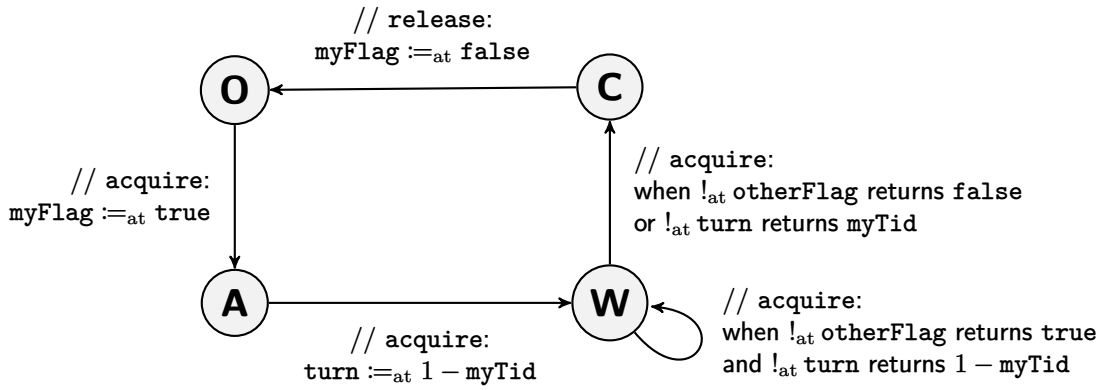



Figure 5.10: Internal (ghost) state of each thread participating in Peterson's algorithm

## 5.6 Peterson's algorithm

Peterson's algorithm (Raynal, 2013, §2.1.2) is an alteration of Dekker's (previous section) which also implements fair mutual exclusion of two participants. It still uses three atomic references $\texttt{flag}_1$, $\texttt{flag}_2$, `turn` with the same meaning. The code, shown in Figure 5.9, is shorter and apparently simpler than that of Dekker's algorithm. In spite of this (or perhaps because of this), it is harder to understand. Indeed, its proof of correctness, that we now outline, is more involved.

Instead of three, there are *four* possible states for each participant, summarized in Figure 5.10: either the participant is outside of a critical section (O), or it has just entered `acquire` and set its flag to `true` (A), or it has written `turn` and is now waiting (W), or it is inside a critical section (C). The algorithm's invariant appears in Figure 5.11.

Again, when the loop in `acquire` terminates, we need to argue that the other participant is not in state C (so that the invariant can give us $P \mathbin{@} (\mathcal{V}_0 \sqcup \mathcal{V}_1)$), and that we have $\uparrow \mathcal{V}_0$ and $\uparrow \mathcal{V}_1$.

$$
\begin{array}{rcl}
s_0, s_1 & \in & \textsc{PetersonState} \triangleq \{\mathsf{O}, \mathsf{A}, \mathsf{W}, \mathsf{C}\} \\
\gamma_0, \gamma_1 & : & \textsc{Auth}(\textsc{Ex PetersonState}) \\
\omega_0, \omega_1 & : & \textsc{Auth}(\textsc{Ex View})
\end{array}
$$

$$
\mathsf{petersonInv} \triangleq \boxed{\begin{array}{l}
\exists t \in \{0,1\}, f_0, f_1, s_0, s_1, \mathcal{V}, \mathcal{V}_0, \mathcal{V}_1. \\
\quad * \begin{cases}
\mathtt{turn} \rightsquigarrow_{\mathrm{at}} \langle t, \mathcal{V} \rangle \\
\mathtt{flag}_0 \rightsquigarrow_{\mathrm{at}} \langle f_0, \mathcal{V}_0 \rangle * \boxed{\bullet\, s_0}^{\gamma_0} * \boxed{\bullet\, \mathcal{V}_0}^{\omega_0} * (s_0 = \mathsf{O} \vee f_0 = \mathtt{true}) \\
\mathtt{flag}_1 \rightsquigarrow_{\mathrm{at}} \langle f_1, \mathcal{V}_1 \rangle * \boxed{\bullet\, s_1}^{\gamma_1} * \boxed{\bullet\, \mathcal{V}_1}^{\omega_1} * (s_1 = \mathsf{O} \vee f_1 = \mathtt{true}) \\
(t = 0 \wedge s_0 = \mathsf{W}) \implies (s_1 = \mathsf{W} \wedge \mathcal{V} = \mathcal{V}_1) \\
(t = 1 \wedge s_1 = \mathsf{W}) \implies (s_0 = \mathsf{W} \wedge \mathcal{V} = \mathcal{V}_0) \\
(s_0 \neq \mathsf{C} \wedge s_1 \neq \mathsf{C}) \implies P \mathbin{@} (\mathcal{V}_0 \sqcup \mathcal{V}_1)
\end{cases}
\end{array}}
$$

$$
\mathsf{canLock}\ i \triangleq \mathsf{petersonInv} * \exists \mathcal{V}. \boxed{\circ\, \mathsf{O}}^{\gamma_i} * \boxed{\circ\, \mathcal{V}}^{\omega_i} * \uparrow \mathcal{V}
$$

$$
\mathsf{locked}\ i \triangleq \mathsf{petersonInv} * \exists \mathcal{V}. \boxed{\circ\, \mathsf{C}}^{\gamma_i} * \boxed{\circ\, \mathcal{V}}^{\omega_i}
$$

Figure 5.11: Internal invariant of Peterson's algorithm

Thread $i$'s own view $\uparrow \mathcal{V}_i$ is obtained via the token $\mathsf{canLock}\ i$ as in Dekker's algorithm. Proving the other two facts requires a case analysis, because the waiting loop has two exit points, due to the conjunction in the conditional statement:

- The loop can terminate immediately after reading $\mathtt{flag}_{1-i}$ yields the value `false`. Then, as in Dekker's algorithm, the thread acquires $\mathcal{V}_{1-i}$ via this read, and learns that the other thread is in state $\mathsf{O}$ from the fact that its flag is `false`.

- Or, the loop terminates after reading `turn` yields the value $i$. Then, we must prove that the thread acquires $\mathcal{V}_{1-i}$ via this read. This is nonobvious: it requires us to argue that the view $\mathcal{V}$ stored at reference `turn` must in fact contain $\mathcal{V}_{1-i}$. The intuitive reason why this is true is that the two participants write `turn` in a polite way, always giving priority to the other thread. Thus, thread 0 writes only the value 1 to `turn` and vice versa. Therefore, when thread $i$ reads $i$ from `turn`, this value must have been written by thread $1-i$, which implies that this read allows thread $i$ to acquire the view $\mathcal{V}_{1-i}$. Technically, this argument is reflected in the invariant by the proposition $(t = 0 \wedge s_0 = \mathsf{W}) \implies (s_1 = \mathsf{W} \wedge \mathcal{V} = \mathcal{V}_1)$ and its symmetric counterpart. This proposition also lets thread $i$ learn that the other thread is still in state $\mathsf{W}$, which it entered when writing to `turn`.

Observe that, by contrast with Dekker's algorithm, the reference `turn` plays an active role in the functional correctness of Peterson's algorithm: not only its value is relevant, the synchronization it performs also is.

This last case study highlights what we believe is a strength of Cosmo: without loss of generality, we have reduced the problem of transferring arbitrary assertions to that of transferring a simple class of *persistent* assertions, that assert no ownership. This allows the verification of

programs, such as Peterson's algorithm, where a given resource may be transferred through varying channels: the ownership of the resource does not need to go through these channels, it is put in a single global invariant and only persistent knowledge goes through these channels.

To illustrate that point, we may define syntactic sugar for ownership of a Multicore OCaml atomic cell that is used to transfer an arbitrary assertion $P$:

$$A \rightsquigarrow_{\mathrm{at}} \langle v, P \rangle \triangleq \exists \mathcal{V}.\, A \rightsquigarrow_{\mathrm{at}} \langle v, \mathcal{V} \rangle * (P \mathbin{@} \mathcal{V})$$

This new points-to assertion is objective and enjoys the following reasoning rules:

$$\frac{\left\{A \rightsquigarrow_{\mathrm{at}} \langle v, P * Q \rangle\right\}}{\left\{\lambda v'.\, v' = v * A \rightsquigarrow_{\mathrm{at}} \langle v, P \rangle * Q\right\}}\ !_{\mathrm{at}}\, A \qquad \frac{\left\{A \rightsquigarrow_{\mathrm{at}} \langle \_, P * Q \rangle * R\right\}}{\left\{\lambda (). A \rightsquigarrow_{\mathrm{at}} \langle v, P * R \rangle * Q\right\}}\ A :=_{\mathrm{at}} v$$

However, in this new framework, when reading (or writing) cell $A$, we must choose immediately which portion $Q$ of the stored assertion we take for ourselves and which portion $P$ we leave in the cell; symmetrically, when writing, we must choose immediately the assertion $R$ to store. This restricts the reasoning that we are able to conduct. By contrast, with the view framework, one does not need to choose: an assertion $\uparrow \mathcal{V}$ is persistent, therefore duplicable.

# Chapter 6

# A bounded multiple-producer multiple-consumer queue

To confirm Cosmo as a practical tool, we now turn to studying a realistic concurrent data structure. We demonstrate that Cosmo lets us give a precise specification to such a data structure and lets us verify a non-trivial implementation of it. A sequential data structure implementation can always be made robust in the face of concurrency simply by guarding all of its opeations with a lock. While such a *coarse-grained* implementation of a concurrent data structure is certainly correct, there often exist implementations which yield better performance, especially under heavy contention, based on subtle *fine-grained* memory accesses. These implementations are delicate and often rely on subtle properties of the memory model. An informal correctness argument is difficult, likely unreliable, hence unconvincing. Thus, concurrent data structures are prime candidates for formal verification. Many machine-checked proofs of concurrent data structures have appeared in the literature already (Parkinson et al., 2007; Frumin et al., 2018, 2020; Zakowski et al., 2018), but relatively few verification efforts take place in a weak-memory setting (Lê et al., 2013b,a), and fewer still rely on a modular methodology, where a proof of a concurrent data structure and a proof of its client (perhaps a concurrent application, or another concurrent data structure) can be modularly combined.

A concurrent queue is an archetypal example of a realistic concurrent data structure: it is widely used in practice—for example to manage a sequence of tasks that are generated and handled by different threads—and it admits fine-grained implementations. In this chapter, we present a specification of a concurrent queue, and we formally verify that a particular fine-grained implementation satisfies this specification. While other such formalizations already exist in a sequentially-consistent setting (Vindum and Birkedal, 2021; Vindum et al., 2021), we consider a weak-memory setting. Such a formalization effort is innovative and challenging in several aspects.

First, weak memory models are infamous for the subtlety of the reasoning that they impose. In this regard, we believe that Cosmo and the Multicore OCaml memory model strike a good balance between the ease of reasoning enabled by the logic and the flexibility and performance allowed by the memory model.

Second, the specification of the concurrent queue should indicate that it behaves as if all of its operations acted atomically on a common shared state, even though in reality they access distinct parts of the memory and require many machine instructions. To address this challenge, we use

the recently-developed concept of *logical atomicity* (Jung et al., 2015; Jung, 2019; da Rocha Pinto et al., 2014), which we transport to the setting of Cosmo. To the best of our knowledge, this is the first use of logical atomicity in a weak-memory setting. This raises new questions: for instance, even though our implementation realizes a total order on the operations on a queue, it offers strictly weaker guarantees than would be offered by a coarse-grained implementation. Indeed, in the context of a weak memory model, the specification of a concurrent data structure must describe not only the result of its operations, but also the manner in which these are synchronized, that is, the *happens-before* relationships that exist between these. This additional information allows reasoning about accesses to areas of memory *outside* of the data structure itself. This is crucial, for example, if a queue is used to transfer the ownership of a piece of memory from a producer to a consumer: there must exist a happens-before relationship between the `enqueue` operation and the corresponding `dequeue` operation, so as to ensure that the consumer acquires the producer's view of this piece of memory. Our specification faithfully captures a subtle behavior of the implementation: even though operations are totally ordered by logical atomicity, not all operations are ordered by happens-before—but *some* are.

We believe our approach, whose key ingredients are Cosmo and logical atomicity, scales to other memory models and other data structures. Indeed, first, the core of Cosmo (beyond basic Separation Logic) is a logic for reasoning with *views*, an operational description of the memory model; other memory models than that of Multicore OCaml can also be termed in this fashion, as iGPS (Kaiser et al., 2017) and iRC11 (Dang et al., 2020) have demonstrated for C11. Second, logical atomicity has already successfully been used for various data structures in the Iris community (Iris developers and contributors, 2021; Frumin et al., 2020).

The chapter opens with an explanation of the specification of a concurrent queue (§6.1). Then, we present an implementation of the queue in Multicore OCaml (§6.2), and explain our proof of its correctness (§6.3). Next, we demonstrate that our specification is indeed usable, by exploiting it in the context of a simple piece of client code, where the concurrent queue is used to establish a pipeline between a set of producers and a set of consumers (§6.4).

## 6.1 Specification of a MPMC queue

A queue is a first-in first-out container data structure. At any time, it holds an ordered list of items. It supports two main operations: `enqueue` inserts an item at one extremity of the queue (the *head*); `dequeue` extracts an item—if there is one—from the other extremity (the *tail*).

In a concurrent setting, a legitimate question is whether several threads can operate the queue safely. The answer depends on the implementation. Possible restrictions include allowing only one thread to enqueue (single producer), or allowing only one thread to dequeue (single consumer), or both. In this section, we specify a *multiple-producer, multiple-consumer* (MPMC) queue, that is, a queue in which any number of threads are allowed to enqueue and dequeue.

### 6.1.1 Specification in a sequential setting

Let us start by assuming a sequential setting. We can then use standard Separation Logic (Reynolds, 2002; briefly presented in §2.3.1) to reason about programs and the resources they manipulate. In Separation Logic, a queue $q$ holding $n$ items $[v_0, ..., v_{n-1}]$, where the left extremity

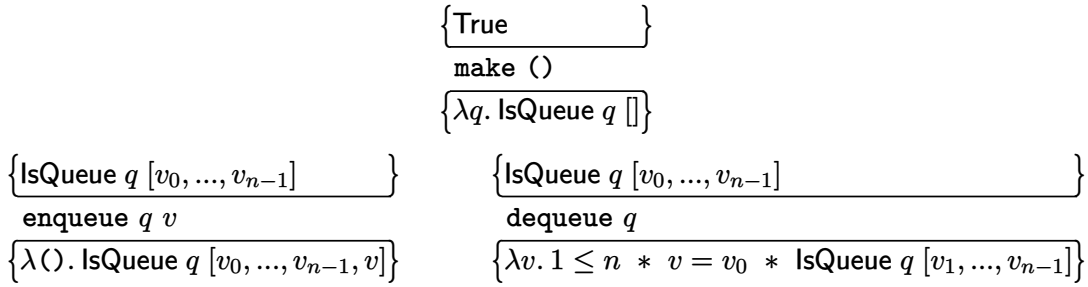



Figure 6.1: A specification of the "queue" data structure in a sequential setting

of the list is the tail and the right extremity is the head, can be represented with an assertion:

$$\mathsf{IsQueue}\ q\ [v_0, ..., v_{n-1}]$$

As is usual in Separation Logic, this *representation predicate* asserts the unique ownership of the entire data structure. It is exclusive. When holding it, we can safely manipulate the queue without risk of invalidating other assertions about resources that may alias parts of our queue. In particular, the representation predicate cannot be duplicated.

The operations of the queue admit a simple sequential specification which is presented in Figure 6.1.

- The function `make` has no prerequisite and gives us the ownership of a new empty queue.
- If we own a queue $q$, then we can `enqueue` some item $v$ into $q$; if this operation ever returns, then it must return the unit value `()` and give us back the ownership of $q$, where $v$ has been appended at the head.
- Conversely, if we own a queue $q$, then we can `dequeue` from $q$; if this operation ever returns, then it must return the first item $v_0$ found at the tail of $q$, and it gives us back the ownership of $q$, where that first item has been removed.

This specification implies that `dequeue` cannot possibly return when the queue is empty ($n = 0$); in this case, it must loop forever. This is pointless in a sequential setting, but becomes meaningful in the presence of concurrency, where it makes sense for `dequeue` to wait until an item is enqueued. This specification also applies to bounded queues, where (somewhat analogously) `enqueue` loops when the capacity is reached ($n = C$), waiting until room becomes available.

### 6.1.2 Specification under sequential consistency: logical atomicity

We now consider a situation where several threads can access the queue concurrently. Let us assume for now that the memory model is sequentially consistent (Lamport, 1979, presented in §2.1). A derivative of Concurrent Separation Logic (§2.3.1) accommodates for this situation; in this section; we adopt Iris (§2.3.2). In Iris, we may retain the exact same specification as is presented in Figure 6.1, recalling that $\mathsf{IsQueue}\ q\ [v_0, ..., v_{n-1}]$ is an exclusive assertion: a thread that has this assertion can safely assume to be the only one allowed to operate on the queue. A

$$\textsc{LAHoare}\ \frac{\langle x.\, P\rangle\ e\ \langle Q\rangle}{\forall x.\, \{P\}\ e\ \{Q\}} \qquad \textsc{LAInv}\ \frac{\langle x.\, \triangleright I * P\rangle\ e\ \langle Q * \triangleright I\rangle}{\boxed{I} \vdash \langle x.\, P\rangle\ e\ \langle Q\rangle}$$

Figure 6.2: Selected rules for logically atomic triples

client application can transfer this assertion between threads via some means of synchronization: for instance, it may use a lock to guard all operations on the shared queue, following the approach of Concurrent Separation Logic. However this coarse grain concurrency has a run-time penalty, and it also creates some contention on the use of the queue. These costs are often unnecessary, as many data structures are designed specifically to support concurrent accesses. In this chapter, as stated, we wish to prove the correctness of a MPMC queue implementation, which should thus ensure, by itself, thread safety. Hence we can achieve finer-grain concurrency, where operating on a queue does not require its exclusive ownership.

In this context, another option is for the client to share this ownership among several threads, logically. In Iris, one would typically place the exclusive ownership in an invariant (§2.3.2). Recall that an invariant is an assertion which is agreed upon by all threads, and is owned by anyone; it remains true forever. As the *public state* of the queue—the list $[v_0, ..., v_{n-1}]$ of currently stored items—would only be known from that invariant, the client would also express in there the properties about this state that their particular application needs. Then, when one of the threads needs to access the shared resource, it can *open* the invariant, get the assertions it contains, perform the desired operation on the shared state, reestablish the invariant, and finally close it. However recall that, to ensure soundness in the face of concurrency, the use of invariants in Iris obeys a strict constraint: they can remain open during at most one step of execution. Unfortunately, `enqueue` and `dequeue` are complex operations which, *a priori*, take several steps. Hence a client would be unable to open their invariant around the triples shown in Figure 6.1. Yet these operations are "atomic" in some empirical sense.

The concept of logical atomicity (Jacobs and Piessens, 2011; Jung, 2019; Jung et al., 2015, §7) aims at addressing that difficulty. To use it, we substitute ordinary Hoare triples with *logically atomic triples*. Two important reasoning rules for logically atomic triples are given in Figure 6.2.[1] A logically atomic triple is denoted with angle brackets $\langle \ldots \rangle$. Just like an ordinary triple, it specifies a program fragment with a precondition and a postcondition. In fact, as witnessed by rule LAHoare, one can deduce an ordinary Hoare triple from a logically atomic triple. The core difference resides in rule LAInv: it is most similar to rule HoareInv of regular triples (§2.3.2), except that the former does not require the program $e$ to run for at most one step. Thus, invariants can be opened around a logically atomic triple, regardless of the number of execution steps of the program fragment: in a sense, when a function is specified using a logically atomic triple, one states that said function behaves as if it were atomic. The definition of logically atomic triples is further discussed in §6.3.6 and given with detail in previous work (Jung, 2019; Jung et al., 2015, §7). We now try to give an intuition of that concept: a logically atomic triple $\langle P\rangle\ e\ \langle Q\rangle$ states, roughly, that the expression $e$ contains an atomic instruction, called the *commit*

[1] Recalling Iris notations, $\boxed{I}$ is an invariant whose content is the assertion $I$, and $\triangleright$ is a step-indexing modality, a technicality of Iris that we can ignore in this dissertation.

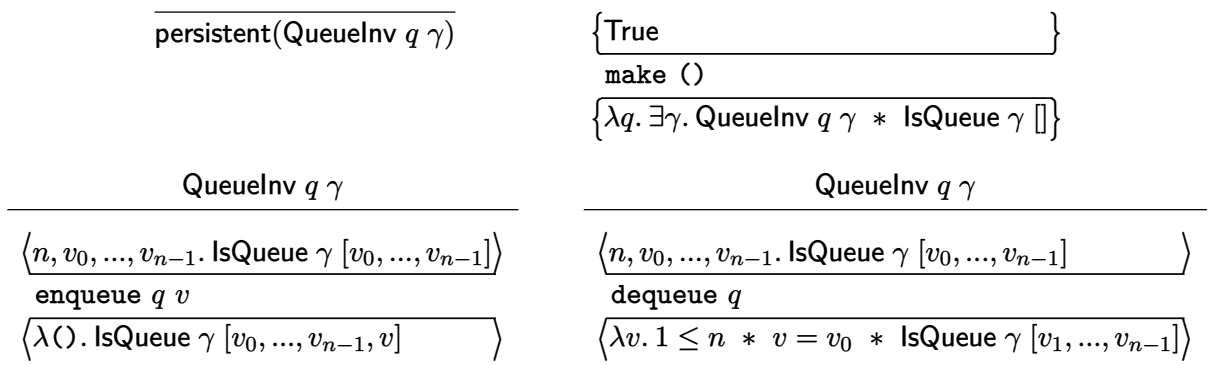



Figure 6.3: A specification of the "queue" data structure in a sequentially consistent memory model

*point*, which has $P$ as a precondition and $Q$ as a postcondition. Because it is atomic, invariants can be opened around that commit point.

Using logically atomic triples, the specification can be written as shown in Figure 6.3. It closely resembles that of the sequential setting (Figure 6.1). The first noticeable difference is the use of angle brackets $\langle \ldots \rangle$ denoting logically atomic triples instead of curly brackets $\{\ldots\}$ for ordinary Hoare triples.

Another difference is the presence, in the syntax of logically atomic triples, of an explicit binder for some variables $(n, v_0, ..., v_{n-1})$. This binder addresses a subtlety of logical atomicity: a client calling `enqueue` or `dequeue` does not know in advance the state of the queue at the commit point, which is when the precondition and postcondition are to be interpreted. Hence, both formulas have to be parameterized by said shared state. Said otherwise, a logically atomic triple provides a *family* of pre/postcondition pairs covering every possible shared state at the commit point.

The last departure from the sequential specification is that the representation predicate is now split into two parts: a persistent assertion QueueInv $q$ $\gamma$ and an exclusive assertion IsQueue $\gamma$ $[v_0, ..., v_{n-1}]$, connected by a ghost variable $\gamma$. That splitting is an artifact of our correctness proof technique, which we detail in §6.3. Note that this does not complicate the use of the queue by the client: both assertions are produced when creating the queue, and while the exclusive component can be put in an invariant as before, the persistent component can be directly duplicated and distributed to all threads.[2]

The use of such a specification in a concrete example will be detailed in §6.4.3. For now, we illustrate how a weaker specification can be easily deduced from this one.

[2] An Iris expert may want to conceal the queue invariant, QueueInv $q$ $\gamma$, inside IsQueue $\gamma$ $[v_0, ..., v_{n-1}]$. However, we need to access this invariant at various places other than the commit point. This is feasible with a more elaborate definition of logically atomic triples than the one given in this dissertation, so that they support *aborting* (Jung, 2019). Another drawback is that we would lose the timelessness of the representation predicate.

**A persistent specification**

If it were not for logical atomicity, and we still wanted to share the ownership of the queue, we would have little choice left than renouncing to an exclusive representation predicate. Only a persistent assertion would be provided, because the description of the public state has to be stable in the face of interference by other threads. The resulting specification would be much weaker. For example, we may merely specify that all of the elements stored in the queue satisfy some predicate $\Phi$. In doing so, we lose most structural properties of a queue: the same specification could also describe a stack or a bag.

$$\frac{}{\mathsf{persistent}(\mathsf{QueuePersistent}\ q\ \Phi)} \qquad \frac{\mathsf{QueuePersistent}\ q\ \Phi}{\{\Phi\, v\}\ \texttt{enqueue}\ q\ v\ \{\lambda().\ \mathsf{True}\}} \qquad \frac{\mathsf{QueuePersistent}\ q\ \Phi}{\{\mathsf{True}\}\ \texttt{dequeue}\ q\ \{\lambda v.\ \Phi\, v\}}$$

To derive these Hoare triples from the ones of Figure 6.3, one simply defines the persistent assertion as follows, where the boxed assertion is an Iris invariant:

$$\mathsf{QueuePersistent}\ q\ \Phi \triangleq \exists \gamma.\ \ast \left\{ \begin{array}{l} \mathsf{QueueInv}\ q\ \gamma \\ \boxed{\exists n, v_0, ..., v_{n-1}.\ \mathsf{IsQueue}\ \gamma\ [v_0, ..., v_{n-1}]\ \ast\ \Phi\, v_0\ \ast\ \cdots\ \ast\ \Phi\, v_{n-1}} \end{array} \right.$$

This assertion is trivial to produce at the creation of the queue, when `make` hands us the assertions $\mathsf{QueueInv}\ q\ \gamma$ and $\mathsf{IsQueue}\ \gamma\ []$. Then, for proving the weaker specification of `enqueue` and `dequeue`, one opens the invariant around the associated logically atomic triples.

### 6.1.3 Specification under weak memory: synchronization

Up to now, we have ignored the weakly consistent behavior of the semantics of Multicore OCaml (Dolan et al., 2018; described in §2.2.1). Starting in this section, we take this aspect into account and propose a refined specification, stated in our program logic Cosmo.

Because Cosmo is based on Iris, logically atomic triples can also be defined in Cosmo. In fact, the specification shown in Figure 6.3 still applies. Yet, as such, it is of little value in a weakly consistent context. Indeed, as explained in §6.1.2, it is designed so that $\mathsf{IsQueue}\ \gamma\ [v_0, ..., v_{n-1}]$ can be shared among threads by means of an invariant. But, in Cosmo, invariants are restricted to containing objective assertions. Hence, our first addition to the specification is to stipulate that the representation predicate is objective. This reflects the fact that there exists a total order on the updates to the logical state, on which all threads objectively agree.

Even with this addition, the specification given in §6.1.2 is not precise enough to verify interesting clients such as the one described in §6.4. Indeed, in a weakly consistent setting, one typically expects a concurrent data structure to establish synchronization between some of its concurrent accesses. For example, imagine that thread $A$ enqueues a pointer to a complex data structure (say, a hash table). Then, when thread $B$ dequeues this pointer, $B$ should obtain the unique ownership of the hash table and be able to access it accordingly. In a weakly consistent memory model, $B$ expects to see all of the changes that $A$ has made to the data structure. This is guaranteed only if there is a *happens-before* relationship from the enqueuing event to the dequeuing event.

One possibility would be to guarantee that our concurrent queue implementation behaves like its coarse-grained alternative, that is, a sequential implementation guarded by a lock. This would correspond to an intuitive definition of linearizability, even though this notion is difficult

$$\frac{}{\mathsf{persistent}(\mathsf{QueueInv}\,q\,\gamma)} \qquad \frac{}{\mathsf{objective}(\mathsf{IsQueue}\,\gamma\,\mathcal{T}\,\mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})])}$$

$$\{\uparrow\mathcal{V}_0\}$$
$$\texttt{make ()}$$
$$\{\lambda q.\ \exists\gamma.\ \mathsf{QueueInv}\,q\,\gamma\ *\ \mathsf{IsQueue}\,\gamma\,\mathcal{V}_0\,\mathcal{V}_0\,[]\}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\langle \mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad \mathsf{IsQueue}\,\gamma\,\mathcal{T}\ \ \mathcal{H}\ \ [(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})]\ \ *\ \uparrow\mathcal{V}\rangle\\ \texttt{enqueue}\ q\ v\\ \langle \lambda().\ \mathsf{IsQueue}\,\gamma\,\mathcal{T}\,(\mathcal{H}\sqcup\mathcal{V})\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}),(v,\mathcal{V})]\ *\ \uparrow\mathcal{H}\rangle\end{array}}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\langle \mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad \mathsf{IsQueue}\,\gamma\ \ \mathcal{T}\ \ \ \mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})]\ *\ \uparrow\mathcal{V}\rangle\\ \texttt{dequeue}\ q\\ \langle \lambda v.\ \mathsf{IsQueue}\,\gamma\,(\mathcal{T}\sqcup\mathcal{V})\,\mathcal{H}\,[(v_1,\mathcal{V}_1),\ldots,(v_{n-1},\mathcal{V}_{n-1})]\ *\ \uparrow\mathcal{T}\ *\ \uparrow\mathcal{V}_0\ *\ 1\le n\ *\ v=v_0\rangle\end{array}}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\langle \mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad \mathsf{IsQueue}\,\gamma\,\mathcal{T}\ \ \mathcal{H}\ \ [(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})]\ \ *\ \uparrow\mathcal{V}\rangle\\ \texttt{try\_enqueue}\ q\ v\\ \left\langle \lambda b.\ \vee\begin{bmatrix}\mathsf{IsQueue}\,\gamma\,\mathcal{T}\ \ \mathcal{H}\ \ [(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] & *\ b=\texttt{false}\\ \mathsf{IsQueue}\,\gamma\,\mathcal{T}\,(\mathcal{H}\sqcup\mathcal{V})\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}),(v,\mathcal{V})]\ *\ \uparrow\mathcal{H}\ *\ b=\texttt{true}\end{bmatrix}\right\rangle\end{array}}$$

$$\frac{\mathsf{QueueInv}\,q\,\gamma}{\begin{array}{l}\langle \mathcal{T},\mathcal{H},n,(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1}).\\ \quad \mathsf{IsQueue}\,\gamma\ \ \mathcal{T}\ \ \ \mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})]\ *\ \uparrow\mathcal{V}\rangle\\ \texttt{try\_dequeue}\ q\\ \left\langle \lambda v^?.\ \vee\begin{bmatrix}\mathsf{IsQueue}\,\gamma\ \ \mathcal{T}\ \ \ \mathcal{H}\,[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})] & *\ v^?=\texttt{None}\\ \mathsf{IsQueue}\,\gamma\,(\mathcal{T}\sqcup\mathcal{V})\,\mathcal{H}\,[(v_1,\mathcal{V}_1),\ldots,(v_{n-1},\mathcal{V}_{n-1})]\ *\ \uparrow\mathcal{T}\ *\ \uparrow\mathcal{V}_0\ *\ 1\le n\ *\ v^?=\texttt{Some}\,v_0\end{bmatrix}\right\rangle\end{array}}$$

Figure 6.4: A specification of the "queue" data structure in a weak memory model

to define precisely outside of the world of sequential consistency (Smith et al., 2019). However, the concurrent queue implementation (§6.2) that we aim to verify is weaker than that: it does guarantee *some* happens-before relationships, but not between all pairs of accesses. Namely, it guarantees a happens-before relationship:

1. from an enqueuer to the dequeuer that obtains the corresponding item;
2. from an enqueuer to the following enqueuers;
3. from a dequeuer to the following dequeuers.

The first one permits resource transfer through the queue as described in the example above.

In Cosmo, as seen in previous chapters, happens-before relationships can be expressed as transfers of views (denoted in this chapter by calligraphic capital letters, such as $\mathcal{T}, \mathcal{H}, \mathcal{V}, \mathcal{S}$): specifying a happens-before relationship between two program points can be done by giving the client the ability to transfer any assertion of the form $\uparrow \mathcal{V}$ between these two points: this corresponds to saying that the destination program point has all the knowledge the source program point had about the shared memory. Thanks to rule SPLIT-SUBJECTIVE-OBJECTIVE (§4.1), this is sufficient for transferring any subjective resource from a sender to a receiver, as will be confirmed later when verifying a client application of the queue (§6.4.3). This technique has already been illustrated by the case studies in Chapter 5.

In the specification of the queue, to express the happens-before relationships mentioned earlier, the representation predicate now takes more parameters:

$$\mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})]$$

1. For each item $v_k$ in the queue, we now have a corresponding view $\mathcal{V}_k$. This view materializes the flow of memory knowledge from the thread which enqueued the item, to the one which will dequeue it.
2. The *head* view $\mathcal{H}$ materializes memory knowledge accumulated by successive enqueuers.
3. The *tail* view $\mathcal{T}$ materializes memory knowledge accumulated by successive dequeuers.

The queue that we study, however, does not guarantee any happens-before relationship from a dequeuer to an enqueuer.[3] Hence, it provides fewer guarantees than a sequential queue guarded by a lock.

Interestingly, Cosmo is able to express this subtle difference between the behavior of our library and that of a lock-based implementation: the full specification under weak memory is shown in Figure 6.4. This specification extends the previous one (Figure 6.3) with views. The mentioned happens-before relationships are captured as follows.

1. When a thread with a local view $\mathcal{V}$ (in other words, with $\uparrow \mathcal{V}$ as a precondition) `enqueue`s an item $v$, it pairs it with the view $\mathcal{V}$. Afterwards, when another thread `dequeue`s that same item $v_0$, it merges the view $\mathcal{V}_0$ that was paired with it into its own local view (in other words, it obtains $\uparrow \mathcal{V}_0$ as a postcondition).

[3] This is not entirely true: the implementation shown in §6.2 does create a happens-before relationship from the dequeuer of rank $k$ to the enqueuer of rank $k + C$ (hence also to all enqueuers of subsequent ranks). We choose to not reveal this in the specification, since it reflects uninteresting implementation details.

2. When a thread `enqueue`s an item, it also obtains the head view $\mathcal{H}$ left by the previous enqueuer (in other words, it obtains $\uparrow\mathcal{H}$ as a postcondition), and it adds its own view $\mathcal{V}$ to the head view (which becomes $\mathcal{H} \sqcup \mathcal{V}$).

3. When a thread `dequeue`s an item, it also obtains the tail view $\mathcal{T}$ left by the previous dequeuer (in other words, it obtains $\uparrow\mathcal{T}$ as a postcondition), and it adds its own view $\mathcal{V}$ to the tail view (which becomes $\mathcal{T} \sqcup \mathcal{V}$).

## 6.2 Implementation of a MPMC queue using a ring buffer

We now present an implementation of a bounded MPMC queue, that satisfies the specification devised in §6.1. We show the code (§6.2.1, §6.2.2) and give intuitions about its mode of operation (§6.2.3, §6.2.4).

The presented implementation takes inspiration from a well-established algorithm (Rigtorp, 2021) that has been used in production in several applications. In addition to supporting multiple producers and multiple consumers, a feature of this queue implementation is that it is *bounded*, that is, it occupies no more than a fixed memory size. A motivation for that trait is that items may be enqueued more frequently than they are dequeued; in this situation, a bounded queue has no risk of exhausting system memory; instead, if the maximum size is reached, enqueuing either blocks or fails.

### 6.2.1 Overview of the data structure

The code of the concurrent queue library we consider appears in Figure 6.5. It uses a ring buffer of fixed capacity $C \geq 1$. The buffer is represented by two arrays of length $C$, `statuses` and `items`; in each slot (whose offset ranges from 0 included to $C$ excluded), the buffer thus stores a *status* in an atomic field and an *item* in a non-atomic field. The data structure also has two integers stored in atomic references, `head` and `tail`.

Items are identified by their *rank* (starting at zero) of insertion in the queue since its creation. The item of rank $k$ is stored in slot "$k \bmod C$", which from now on we denote as $\widehat{k}$. The reference `head` stores the rank of the next item to be enqueued, that is, the number of items that have been enqueued since the creation of the queue, including those that have since been dequeued. Similarly, `tail` stores the rank of the next item to be dequeued, that is, the number of items that have been dequeued.

Each slot is in one of two states: either it is *occupied*, meaning that it stores some item of the queue; or it is *available*, meaning that the value it stores is irrelevant. In addition, for the concurrent queue operations to work properly, we must remember for which rank each slot was last used.[4] The status encodes this information in a single integer, as follows:

- an even status $2k$ indicates that slot $\widehat{k}$ is available for storing a future item of rank $k$;
- an odd status $2k + 1$ indicates that slot $\widehat{k}$ is currently occupied by the item of rank $k$.

Let $h$ and $t$ be the value of `head` and `tail`, respectively. At any time, the ranks of items that are stored in the queue—or in the process of being stored—range from $t$ included to $h$ excluded,

[4] Actually, we need not remember the full rank $k$: only the *cycle*, $k \div C$, is needed.

```
let make () =
    let tail = ref_at 0 in
    let head = ref_at 0 in
    let items = array_na[C] () in
    let statuses = array_at[C] 0 in
    for i from 0 to C − 1 do
      statuses[i]_na ← 2i
    done;
    (tail, head, statuses, items)
```

```
let try_enqueue q v =
    let (tail, head,
         statuses, items) = q in
    let h = !_at head in
    let s = statuses[h mod C]_at in
    if s = 2h ∧ CAS head h (h + 1) then
      items[h mod C]_na ← v;
      statuses[h mod C]_at ← 2h + 1;
      true
    else
      false
```

```
let try_dequeue q =
    let (tail, head,
         statuses, items) = q in
    let t = !_at tail in
    let s = statuses[t mod C]_at in
    if s = 2t + 1 ∧ CAS tail t (t + 1) then
      let v = items[t mod C]_na in
      items[t mod C]_na ← ();
      statuses[t mod C]_at ← 2(t + C);
      Some v
    else
      None
```

```
let rec enqueue q v =
    if try_enqueue q v
      then ()
      else enqueue q v
```

```
let rec dequeue q =
    match try_dequeue q with
    | Some v → v
    | None   → dequeue q
```

Figure 6.5: Implementation of the bounded queue

and there cannot be more than $C$ such items. Thus, a property of the queue is:

$$0 \leq t \leq h \leq t + C$$

### 6.2.2 Explanation of the code

The function `enqueue` repeatedly calls `try_enqueue` until it succeeds; the latter can fail either because the buffer is full or because of a competing enqueuer.[5]

When calling `try_enqueue`, we start by reading the current value $h$ of the reference `head`. To check that slot $\widehat{h}$ is available for rank $h$, we read its status. If it differs from $2h$, we fail: a status less than $2h$ indicates that the buffer is full[6], a status greater than $2h$ indicates interference from another competing enqueuing thread, which has been attributed rank $h$ before we have.

If the status is $2h$, then we try to increment `head` from $h$ to $h + 1$. If the CAS fails, we fail: again, another competing enqueuer has been attributed rank $h$.

If the CAS succeeds, then we are attributed rank $h$ and can proceed to inserting an item into slot $\widehat{h}$. As we will explain later, this implies that its status has not changed since we read it: the slot is still available. We write the new item, and then we update the status accordingly. This update must come last, as it serves as a signal that the slot is now occupied with an item and ready for dequeuers. Notice that, under weak memory, another reason why the order of these two writes matters is that the atomic write to `statuses` must propagate the information that the non-atomic write to `items` has taken place. Thus, a thread which dequeues this item (after reading its status) is certain to read a correct value from the array `items`. This is a typical release/acquire idiom.

Similarly, `dequeue` repeatedly calls `try_dequeue` until it succeeds; the latter works analogously to `try_enqueue`,[7] and can fail either because the buffer is empty or because of a competing dequeuer.[5]

### 6.2.3 Monotonicity of the internal state of the queue

Once the queue has been created, the reference `head` is only accessed from function `try_enqueue`. The only place where it is modified is the compare-and-set operation in this function, which attempts to increment it by one, using a compare-and-set operation. Hence this counter is strictly monotonic, and we can regard a successful increment from $h$ to $h + 1$ as uniquely attributing rank $h$ to the candidate enqueuer. Only then it is allowed to write into slot $\widehat{h}$.

Similarly, `tail` is only accessed from function `try_dequeue`, is strictly monotonic, and a successful increment from $t$ to $t + 1$ uniquely attributes rank $t$ to the candidate dequeuer. Only then it is allowed to write into slot $\widehat{t}$.

The status of a given slot is strictly monotonic too. Indeed, there are two places where it is updated. As an enqueuer, when we write $2h + 1$, no other enqueuer updated the status since we read it to be $2h$, because only we have been attributed rank $h$. In particular, it remained even,

[5] The code does not distinguish between these two causes, but this is feasible with only one more test.

[6] Technically, the public state of the queue may contain less than $C$ elements in this case, so that we may consider it is not full. Here, by "full" we mean that the next buffer slot is either not reclaimed by anyone or still in the process of being emptied by another thread. Even though the buffer is "full", it may have available slots if dequeuing has completed more rapidly in these other slots.

[7] Overwriting the extracted value with a unit value `()` is unnecessary for functional correctness but it prevents memory leaks.

so no dequeuer tried to obtain rank $h$ and update the status of slot $\widehat{h}$. Hence, the status is still $2h$ when we overwrite it with $2h+1$. Symmetrically, the status is still $2t+1$ when a dequeuer overwrites it with $2(t+C)$.

### 6.2.4 Notes on contention in the queue

A noteworthy feature of this implementation is that it tries to limit competition between enqueuers and dequeuers. Indeed, enqueuers and dequeuers generally operate on separate references: enqueuers never access `tail` and dequeuers never access `head`. Hence in favorable situations—when the buffer is neither empty nor full—there are no enqueuer-dequeuer competitions beyond ones between an enqueuer and a dequeuer of the same rank.

A weakness of this implementation, however, is that it does not enjoy any non-blocking property (Fraser, 2004, Chapter 2): if an enqueuer or a dequeuer halts after it has been attributed a rank but before it updates the corresponding slot, then after some time, any other thread trying to enqueue or dequeue fails.

## 6.3 Proof of the specification for the ring buffer

We now turn to proving the following.

**Theorem 4** *There exist predicates* IsQueue *and* QueueInv *such that the implementation shown in Figure 6.5 (§6.2) satisfies the functional specification appearing in Figure 6.4 (§6.1.3).*

Following the Iris methodology, we describe the protocol that threads must follow to access the shared queue, by defining suitable ghost state and invariant. These definitions appear in Figure 6.6: the persistent predicate $\mathsf{QueueInv}\ q\ \gamma$ is in fact an invariant; the exclusive representation predicate $\mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})]$ is defined using ghost state, as are several internal resources. We detail these definitions in the following sections.

### 6.3.1 Public state

The assertion $\mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})]$, defined in Figure 6.6a, exposes to the user the public state of the queue. This public state, as motivated in §6.1.3, is composed of the tail view, the head view, and the list of current items with their views. It is tied to the internal state of the queue via the use of an authoritative ghost state, stored in a ghost variable $\gamma$. More precisely, the public state is kept in sync with the values which appear in an authoritative assertion $\boxed{\bullet\ (\mathcal{T}, \mathcal{H}, [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})])}^{\gamma}$, the latter being owned by the internal invariant.

The two assertions satisfy the properties shown in Figure 6.7a. Rule IsQueue-Agree asserts that the state known to the invariant (first premise) is identical to that known to the representation predicate (second premise). Rule IsQueue-Update asserts that, whenever we own both the representation predicate and its authoritative counterpart, we can update the public state to any other value by taking a ghost update step.

We achieve these properties by using an adequate camera for the values of the ghost variable $\gamma$. This camera is built by composing several classical Iris constructs (§2.3.2): the exclusive camera $\textsc{Ex}(S)$, and the authoritative camera $\textsc{Auth}(M)$.

It is worth remarking that this construction makes the representation predicate exclusive: it is absurd to own simultaneously two assertions of the form $\mathsf{IsQueue}\ \gamma\ -\ -\ -$.

$$\begin{array}{rcl} \gamma & : & \textsc{Auth}(\textsc{Ex}(\textsc{View} \times \textsc{View} \times \textsc{List}(\textsc{Val} \times \textsc{View}))) \\ \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})] & \triangleq & \boxed{\circ\ (\mathcal{T}, \mathcal{H}, [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})])}^{\gamma} \end{array}$$

(a) Ghost state describing the public state

$$\begin{array}{rcl} \textsc{StatusWithView} & \triangleq & (\mathbb{Z} \times \textsc{View}, \sqsubseteq, \sqcup) \\ (s_1, \mathcal{S}_1) \sqsubseteq (s_2, \mathcal{S}_2) & \triangleq & s_1 < s_2 \vee (s_1 = s_2 \wedge \mathcal{S}_1 \sqsupseteq \mathcal{S}_2) \\ (s_1, \mathcal{S}_1) \sqcup (s_2, \mathcal{S}_2) & \triangleq & \begin{cases} (s_2, \mathcal{S}_2) & \text{if } s_1 < s_2 \\ (s, \mathcal{S}_1 \sqcap \mathcal{S}_2) & \text{if } s_1 = s_2 = s \\ (s_1, \mathcal{S}_1) & \text{if } s_1 > s_2 \end{cases} \end{array}$$

(b) Semi-lattice defining an order on status-view pairs

$$\begin{array}{rcl} \gamma_{\mathrm{st}} & : & \mathbb{Z} \xrightarrow{\text{fin}} \textsc{Auth}(\textsc{StatusWithView}) \\ \mathsf{Witness}\ \gamma_{\mathrm{st}}\ i\ (s, \mathcal{S}) & \triangleq & \boxed{\{i \mapsto \circ\ (s, \mathcal{S})\}}^{\gamma_{\mathrm{st}}} \end{array}$$

(c) Ghost state reflecting the monotonicity of statuses

$$\begin{array}{rcl} \gamma_{\mathrm{tok}} & : & \textsc{Auth}\left(\mathbb{Z} \xrightarrow{\text{fin}} \textsc{Ex}(\textsc{Unit} \ +\ \textsc{Val} \times \textsc{View})\right) \\ \mathsf{TokenR}\ \gamma_{\mathrm{tok}}\ k & \triangleq & \boxed{\circ\ \{k \mapsto ()\}}^{\gamma_{\mathrm{tok}}} \\ \mathsf{TokenW}\ \gamma_{\mathrm{tok}}\ k\ (v, \mathcal{V}) & \triangleq & \boxed{\circ\ \{k \mapsto (v, \mathcal{V})\}}^{\gamma_{\mathrm{tok}}} \end{array}$$

(d) Ghost state implementing tokens

$$\begin{array}{rcl} \mathsf{QueueInv}\ q\ \gamma & \triangleq & \exists \gamma_{\mathrm{st}}, \gamma_{\mathrm{tok}}.\ \boxed{\mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\mathrm{st}}\ \gamma_{\mathrm{tok}}} \\ \mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\mathrm{st}}\ \gamma_{\mathrm{tok}} & \triangleq & \end{array}$$

$$\left\{\begin{array}{l} \exists \mathtt{tail}, \mathtt{head}, \mathtt{statuses}, \mathtt{items}. \\ \exists t, \mathcal{T}, h, \mathcal{H}, (v_t, \mathcal{V}_t), ..., (v_{h-1}, \mathcal{V}_{h-1}), (s_0, \mathcal{S}_0), ..., (s_{C-1}, \mathcal{S}_{C-1}). \\ \quad \left\{\begin{array}{l} q = (\mathtt{tail}, \mathtt{head}, \mathtt{statuses}, \mathtt{items}) \\ 0 \le t \le h \le t + C \\ \mathtt{tail} \rightsquigarrow_{\mathrm{at}} \langle t, \mathcal{T} \rangle \ *\ \mathtt{head} \rightsquigarrow_{\mathrm{at}} \langle h, \mathcal{H} \rangle \\ \mathtt{statuses} \rightsquigarrow^{*}_{\mathrm{at}} [(s_0, \mathcal{S}_0), ..., (s_{C-1}, \mathcal{S}_{C-1})] \\ \boxed{\bullet\ (\mathcal{T}, \mathcal{H}, [(v_t, \mathcal{V}_t), ..., (v_{h-1}, \mathcal{V}_{h-1})])}^{\gamma} \\ \boxed{\{i \mapsto \bullet\ (s_i, \mathcal{S}_i) \mid 0 \le i < C\}}^{\gamma_{\mathrm{st}}} \\ * \left\{\begin{array}{l} \boxed{\bullet \begin{pmatrix} \{k \mapsto () & | & h - C \le k < t\} \\ \uplus\ \{k \mapsto (v_k, \mathcal{V}_k) & | & t \le k < h\} \end{pmatrix}}^{\gamma_{\mathrm{tok}}} \\ \ast_{h-C \le k < t}\ \mathsf{Available}\ k\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}}) \quad \vee\ s_{\widehat{k}} = 2k + 1 \\ \ast_{t \le k < h}\ \mathsf{Occupied}\ k\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}})\ (v_k, \mathcal{V}_k) \quad \vee\ s_{\widehat{k}} = 2k \end{array}\right. \end{array}\right. \end{array}\right.$$

$$\begin{array}{rcl} \mathsf{Available}\ k\ (s, \mathcal{S}) & \triangleq & * \left\{\begin{array}{l} s = 2(k + C)\ *\ (\mathtt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} -) \mathbin{@} \mathcal{S} \\ \mathsf{TokenR}\ \gamma_{\mathrm{tok}}\ k \end{array}\right. \\ \mathsf{Occupied}\ k\ (s, \mathcal{S})\ (v, \mathcal{V}) & \triangleq & * \left\{\begin{array}{l} s = 2k + 1 \quad *\ (\mathtt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} v\ ) \mathbin{@} \mathcal{S} \\ \mathsf{TokenW}\ \gamma_{\mathrm{tok}}\ k\ (v, \mathcal{V})\ *\ \mathcal{V} \sqsubseteq \mathcal{S} \end{array}\right. \end{array}$$

(e) Internal invariant of the queue

Figure 6.6: Definitions of assertions intervening in the proof of the bounded queue

IsQueue-Agree

$$\frac{\boxed{\bullet\,(\mathcal{T},\mathcal{H},[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})])}^{\gamma} \;*\; \mathsf{IsQueue}\;\gamma\;\mathcal{T}'\;\mathcal{H}'\;[(v'_0,\mathcal{V}'_0),\ldots,(v'_{n-1},\mathcal{V}'_{n-1})]}{\mathcal{T}=\mathcal{T}'\;\wedge\;\mathcal{H}=\mathcal{H}'\;\wedge\forall i.\;\; v=v'_i\;\wedge\;\mathcal{V}_i=\mathcal{V}'_i}$$

IsQueue-Update

$$\frac{\boxed{\bullet\,(\mathcal{T},\mathcal{H},[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})])}^{\gamma} \;*\; \mathsf{IsQueue}\;\gamma\;\mathcal{T}\;\mathcal{H}\;[(v_0,\mathcal{V}_0),\ldots,(v_{n-1},\mathcal{V}_{n-1})]}{\Rightarrow\boxed{\bullet\,(\mathcal{T}',\mathcal{H}',[(v'_0,\mathcal{V}'_0),\ldots,(v'_{n-1},\mathcal{V}'_{n-1})])}^{\gamma} \;*\; \mathsf{IsQueue}\;\gamma\;\mathcal{T}'\;\mathcal{H}'\;[(v'_0,\mathcal{V}'_0),\ldots,(v'_{n-1},\mathcal{V}'_{n-1})]}$$

(a) Properties of the representation predicate

Witness-Persistent

$$\frac{}{\mathsf{persistent}(\mathsf{Witness}\;\gamma_{\mathrm{st}}\;i\;(s,\mathcal{S}))}$$

Witness-Order

$$\frac{\boxed{\{i\mapsto\bullet\,(s,\mathcal{S})\}}^{\gamma_{\mathrm{st}}} \;*\; \mathsf{Witness}\;\gamma_{\mathrm{st}}\;i\;(s',\mathcal{S}')}{(s',\mathcal{S}')\sqsubseteq(s,\mathcal{S})}$$

Witness-Update

$$\frac{\boxed{\{i\mapsto\bullet\,(s,\mathcal{S})\}}^{\gamma_{\mathrm{st}}} \;*\; (s,\mathcal{S})\sqsubseteq(s',\mathcal{S}')}{\Rightarrow\boxed{\{i\mapsto\bullet\,(s',\mathcal{S}')\}}^{\gamma_{\mathrm{st}}} \;*\; \mathsf{Witness}\;\gamma_{\mathrm{st}}\;i\;(s',\mathcal{S}')}$$

(b) Properties of witnesses

Token-Exclusive-RR

$$\frac{\mathsf{TokenR}\;\gamma_{\mathrm{tok}}\;k\;*\;\mathsf{TokenR}\;\gamma_{\mathrm{tok}}\;k'}{\widehat{k}\neq\widehat{k'}}$$

Token-Exclusive-RW

$$\frac{\mathsf{TokenR}\;\gamma_{\mathrm{tok}}\;k\;*\;\mathsf{TokenW}\;\gamma_{\mathrm{tok}}\;k'\;(v',\mathcal{V}')}{\widehat{k}\neq\widehat{k'}}$$

Token-Exclusive-WW

$$\frac{\mathsf{TokenW}\;\gamma_{\mathrm{tok}}\;k\;(v,\mathcal{V})\;*\;\mathsf{TokenW}\;\gamma_{\mathrm{tok}}\;k'\;(v',\mathcal{V}')}{\widehat{k}\neq\widehat{k'}}$$

TokenR-Agree

$$\frac{\boxed{\bullet\,m}^{\gamma_{\mathrm{tok}}} \;*\; \mathsf{TokenR}\;\gamma_{\mathrm{tok}}\;k}{m(k)=()}$$

TokenW-Agree

$$\frac{\boxed{\bullet\,m}^{\gamma_{\mathrm{tok}}} \;*\; \mathsf{TokenW}\;\gamma_{\mathrm{tok}}\;k\;(v,\mathcal{V})}{m(k)=(v,\mathcal{V})}$$

Token-Update-RW

$$\frac{\boxed{\bullet\,m}^{\gamma_{\mathrm{tok}}} \;*\; \mathsf{TokenR}\;\gamma_{\mathrm{tok}}\;(k-C)}{\Rightarrow\boxed{\bullet\,m[k-C\mapsto\bot][k\mapsto(v,\mathcal{V})]}^{\gamma_{\mathrm{tok}}} \;*\; \mathsf{TokenW}\;\gamma_{\mathrm{tok}}\;k\;(v,\mathcal{V})}$$

Token-Update-WR

$$\frac{\boxed{\bullet\,m}^{\gamma_{\mathrm{tok}}} \;*\; \mathsf{TokenW}\;\gamma_{\mathrm{tok}}\;k\;(v,\mathcal{V})}{\Rightarrow\boxed{\bullet\,m[k\mapsto()]}^{\gamma_{\mathrm{tok}}} \;*\; \mathsf{TokenR}\;\gamma_{\mathrm{tok}}\;k}$$

(c) Properties of tokens

Figure 6.7: Axiomatic description of the ghost state of the queue

### 6.3.2 Internal invariant

Along with the exclusive representation predicate $\mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})]$, we provide the user with a persistent assertion $\mathsf{QueueInv}\ q\ \gamma$ defined in Figure 6.6e. It contains the internal invariant governing the queue $q$, whose public state is exposed via the ghost variable $\gamma$. In addition to the public state, there are two more ghost variables, named $\gamma_{\mathrm{st}}$ and $\gamma_{\mathrm{tok}}$, which are hidden to the user of the queue but needed internally. Thus they are existentially quantified in this persistent assertion. We will explain the purpose and meaning of these ghost variables in a moment. For now, we look at the internal invariant, $\mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\mathrm{st}}\ \gamma_{\mathrm{tok}}$.

This invariant owns most of the physical cells of the queue: `tail`, `head`, `statuses`, and some parts of the array `items`. Recall that points-to assertions for atomic cells are objective and can be placed inside an invariant. The block-points-to assertion $\texttt{statuses} \rightsquigarrow^*_{\mathrm{at}} [(s_0, \mathcal{S}_0), ..., (s_{C-1}, \mathcal{S}_{C-1})]$ is a shorthand for the following iterated conjunction:

$$\texttt{statuses}\ \mathsf{has\ length}\ C\ *\ \mathop{\Large *}_{0 \le i < C} \texttt{statuses}[i] \rightsquigarrow_{\mathrm{at}} \langle s_i, \mathcal{S}_i \rangle$$

Also, since we encode references as arrays of length one, we write $\texttt{tail} \rightsquigarrow_{\mathrm{at}} \langle t, \mathcal{T} \rangle$ as a shorthand for $\texttt{tail}[0] \rightsquigarrow_{\mathrm{at}} \langle t, \mathcal{T} \rangle$.

Apart from this physical state, the invariant also stores ghost state. It owns the authority on all three ghost variables, $\gamma$, $\gamma_{\mathrm{st}}$ and $\gamma_{\mathrm{tok}}$. The authority of $\gamma$ is simple: it ties internal values to the public state of the queue, as explained earlier. We now explain the other two pieces of ghost state.

### 6.3.3 Monotonicity of statuses

The purpose of the ghost variable $\gamma_{\mathrm{st}}$ is to reflect the fact that statuses are monotonic. More precisely, they are *strictly* monotonic: every write to a status cell necessarily increases its value. As a consequence, as long as the value of a status cell has not increased, we know that no write happened to it and, in particular, that the view that it stores has not increased either. In other words, the value-view pair stored in a status cell is monotonic, with respect to the lexicographic order where the order on views is reversed:

$$(s_1, \mathcal{S}_1) \sqsubseteq (s_2, \mathcal{S}_2) \iff s_1 < s_2 \vee (s_1 = s_2 \wedge \mathcal{S}_1 \sqsupseteq \mathcal{S}_2)$$

This stronger monotonicity property will be used in proofs, and specifying it is thus an additional requirement of working with a weak memory model.

To reflect monotonicity of the status of offset $i$, we use two assertions, $\boxed{\{i \mapsto \bullet\ (s, \mathcal{S})\}}^{\gamma_{\mathrm{st}}}$ and $\mathsf{Witness}\ \gamma_{\mathrm{st}}\ i\ (s, \mathcal{S})$, connected via a ghost variable $\gamma_{\mathrm{st}}$. Relevant definitions appear in Figure 6.6c. The first assertion, owned by the invariant of the queue is connected by the invariant to the value-view pair stored in the status cell. It is exclusive: for any offset $i$, two assertions of the form $\boxed{\{i \mapsto \bullet -\}}^{\gamma_{\mathrm{st}}}$ cannot hold simultaneously. The second assertion $\mathsf{Witness}\ \gamma_{\mathrm{st}}\ i\ (s, \mathcal{S})$ means that the value-view pair stored in the status cell is at least $(s, \mathcal{S})$. Importantly, a witness assertion is persistent: once it has been established, it remains true forever and can be duplicated at will.

We thus have the properties summarized in Figure 6.7b. Rule Witness-Persistent is the persistence just mentioned. Rule Witness-Order asserts that a witness gives a lower bound on what the status cell currently stores. Rule Witness-Update asserts that we can update a status cell to any larger (or equal) content, and obtain a witness for that content.

We achieve these properties by constructing an adequate camera for the values taken by the ghost variable $\gamma_{\mathrm{st}}$. Again, we use standard Iris constructs. The camera StatusWithView is such that its carrier set is $\mathbb{Z} \times \textsc{View}$, the set of status-view pairs, and its inclusion order[8] coincides with the desired order $\sqsubseteq$. It is derived from the join-semilattice structure whose definition appears in Figure 6.6b. Indeed, a general recipe for deriving a camera structure with a given inclusion order, if that order admits binary joins, consists of taking the join operation as the composition of the camera (Timany and Birkedal, 2021).

### 6.3.4 Available and occupied slots

In Figure 6.6e, the last two lines of the invariant describe the state of each slot. For clarity, we introduce two abbreviations: the assertion $\mathsf{Available}\ k\ (s, \mathcal{S})$ represents slot $\widehat{k}$ being available for a future item of rank $k + C$; the assertion $\mathsf{Occupied}\ k\ (s, \mathcal{S})\ (v, \mathcal{V})$ represents slot $\widehat{k}$ being occupied by the item of rank $k$, whose value is $v$ with the associated view $\mathcal{V}$. In these two abbreviations, the status field of the slot has value $s$ and stores view $\mathcal{S}$. These abbreviations are also where we keep the ownership of the *non-atomic* cell $\texttt{items}[\widehat{k}]$, via a points-to assertion.

Recall that, in Cosmo, unlike an atomic points-to assertion, a non-atomic points-to assertion is *subjective*: its truth depends on the view of the subject thread. As a consequence, it cannot be placed in an invariant as is. In order to share this assertion, we must explicitly indicate at which view it holds. This is the purpose of the @ connective.

At which view can we own a non-atomic memory cell? At a view which contains the latest write event to that cell. Fortunately, in our case, any thread—enqueuer or dequeuer—which writes to the non-atomic cell $\texttt{items}[\widehat{k}]$ then writes to the atomic cell $\texttt{statuses}[\widehat{k}]$. Thus it adds its knowledge, including its own write to the item field, to the view $\mathcal{S}$ stored by the status field.

With all this said, a first attempt at representing the buffer might look as follows:

$$
\begin{aligned}
\mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\mathrm{st}}\ \gamma_{\mathrm{tok}} &\overset{?}{=} * \left\{ \begin{array}{l} \vdots \\ \displaystyle\ast_{h-C \le k < t}\ \mathsf{Available}\ k\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}}) \\ \displaystyle\ast_{t \le k < h}\ \mathsf{Occupied}\ k\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}})\ (v_k, \mathcal{V}_k) \end{array} \right. \\
\mathsf{Available}\ k\ (s, \mathcal{S}) &\overset{?}{=} s = 2(k + C)\ *\ (\texttt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} -)\ @\ \mathcal{S} \\
\mathsf{Occupied}\ k\ (s, \mathcal{S})\ (v, \mathcal{V}) &\overset{?}{=} s = 2k + 1\ \ *\ (\texttt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} v\ )\ @\ \mathcal{S}\ *\ \mathcal{V} \sqsubseteq \mathcal{S}
\end{aligned}
$$

That is, we describe the $C$ slots by ranging from $h - C$ to $h$. The indices from $h - C$ to $t$ correspond to available slots, while indices from $t$ to $h$ correspond to slots occupied by the items of the queue. In both cases, we own the item field at the view $\mathcal{S}$ which is stored in the corresponding status field. The item field of an available slot stores an arbitrary value, while for an occupied slot it stores the item $v$.

An occupied slot should also carry the view $\mathcal{V}$ which the queue is supposed to transfer from the enqueuer to the dequeuer alongside item $v$. This again relies on the view $\mathcal{S}$: the enqueuer adds $\mathcal{V}$ to $\mathcal{S}$ when updating the status, and the dequeuer adds $\mathcal{S}$ into its own view when reading the status; so, to retrieve $\mathcal{V}$, it is enough to state the inclusion $\mathcal{V} \sqsubseteq \mathcal{S}$.

[8] For two elements $x$ and $y$ of a camera $M$, $y$ is included in $x$ if there exists $c$ such that $x = y \cdot c$ (§2.3.2).

The tentative invariant stated above, however, is not correct: while an invariant has to hold at any point of the execution, the assertion above is temporarily invalidated when a thread enqueues or dequeues. Specifically, the thread breaks the assertion when it increments `head` or `tail`, thus committing to enqueuing or dequeuing, until it updates the status of the corresponding slot. It is thus necessary to represent slots which are in a temporary state. In the actual invariant shown in Figure 6.6e, slots from $h - C$ to $t$ are either available or in a temporary state where they appear as occupied ($s_{\widehat{k}} = 2k + 1$), until a dequeuer finishes emptying them; slots from $t$ to $h$ are either occupied or in a temporary state where they appear as available ($s_{\widehat{k}} = 2k$), until an enqueuer finishes filling them.

When an enqueuer or dequeuer moves a slot into a temporary state, it takes ownership of its item field, so that it can write to it. Hence the invariant does not have the corresponding points-to assertion. The thread must give it back when updating the status.

### 6.3.5 Slot tokens

This time frame—when a slot is in a temporary state—is also when the last piece of ghost state, stored in the ghost variable $\gamma_{\text{tok}}$, intervenes. Other threads can make the queue progress between the moment when an enqueuer is attributed rank $k$, and the moment when it returns the updated slot to the invariant. An enqueuer needs the assurance that the queue has not gotten too far and attributed the slot on which it was working to a dequeuer, or to another enqueuer in a subsequent cycle.

To this effect, we start by stating how advances of the `head` and `tail` are limited with respect to one another; indeed, we prove these inequalities as part of the invariant:

$$0 \leq t \leq h \leq t + C$$

We also maintain in existence one token for each rank from $h - C$ to $h$. These tokens are exclusive assertions, and there cannot exist two tokens whose ranks are congruent modulo $C$. Hence the token of rank $k$ is enough to grant unique write access to slot $\widehat{k}$. We use it as follows.

1. When an enqueuer is attributed rank $k$, it borrows a newly created token of the same rank.
2. When returning the updated slot to the invariant, the enqueuer also returns the token; from that moment the token is thus kept in the assertion $\mathsf{Occupied}\ k\ (s, \mathcal{S})\ (v, \mathcal{V})$.
3. When a dequeuer is attributed rank $k$, it claims that assertion and borrows the token.
4. When returning the updated slot to the invariant, the dequeuer also returns the token; from that moment the token is thus kept in the assertion $\mathsf{Available}\ k\ (s, \mathcal{S})$.

In step 1, the token is created while destructing the token of rank $k - C$ taken from the assertion $\mathsf{Available}\ (k - C)\ -$, which represents the available cell that the thread claims for enqueuing.

To be able to distinguish between the two temporary states (enqueuing and dequeuing), we give the token a flavor: from steps 1 to 2 it is a *write token*; from steps 3 to 4 it is a *read token.* At any moment, there are read tokens from rank $h - C$ to $t$, and write tokens from $t$ to $h$.

We have a last requirement: when an enqueuer is attributed rank $k$, the new item is added to the public state immediately—the CAS operation on `head` is the commit point of enqueuing—even though the enqueuer has not actually written the item yet. When it finally returns the

updated slot, the enqueuer has lost track of the public state, which may have continued to progress in the meantime. At that moment, it thus needs a reminder that the item it just wrote is indeed the one it was expected to write. We implement this by adding the value $v$—and view $\mathcal{V}$—of the item as a payload to the write token.

The read token of rank $k$ is denoted by $\mathsf{TokenR}\ \gamma_{\mathrm{tok}}\ k$, while the write token of rank $k$ with payload $v$ and $\mathcal{V}$ is denoted by $\mathsf{TokenW}\ \gamma_{\mathrm{tok}}\ k\ (v, \mathcal{V})$. Their authoritative counterpart, owned by the invariant, is an assertion of the form $\boxed{\bullet\, m}^{\gamma_{\mathrm{tok}}}$ where $m$ is a finite map. Its domain is the range $[h - C, h)$ of ranks which have a token, and its images are the payload (considering that read tokens bear a payload of `()`). In the invariant, the value of the map is connected to that of the public state.

The assertions are defined in Figure 6.6d and satisfy the properties in Figure 6.7c. The first three properties say that tokens are exclusive. The next two say that tokens agree with the authoritative counterpart, hence with the public state. Rule Token-Update-RW corresponds to step 1 in the list above, where we create a write token of rank $k$ by destructing a read token of rank $k - C$. Likewise, rule Token-Update-WR corresponds to step 3, where we turn a write token into a read token of the same rank.

In addition to these rules, the finite map described in the internal invariant is such that, whenever we own a read token (respectively a write token), the rank of this token necessarily lies in the range $[h-C, t)$ (respectively $[t, h)$), where $t$ and $h$ are the values of `tail` and `head` which are existentially quantified in the invariant. Thanks to that property, at step 4 (respectively 2), when a dequeuer (respectively an enqueuer) returns the token, it knows that the rank it has been operating on is still in the right range—in other words, that the queue has not advanced too far while the thread was working.

There are more properties that are invariants of the queue, and thus could be stated and verified. However, they are not needed to prove that the code satisfies its specification. For example, the fact that `tail` and `head` are strictly monotonic, and the fact that statuses are non-negative, are not explicitly used.

### 6.3.6 Logical atomicity

The specification that we wish to prove is a logically atomic Hoare triple. The definition of such triples for Iris is given by Jung et al. (2015, §7) and further refined by Jung (2019). It turns out that this definition can be ported as is using the connectives of Cosmo. As we will see in §6.3.7 and §6.4.3, the logically atomic triples so defined can be proved and are sufficient for interesting clients. We do not attempt to replicate in this dissertation the full definition. An approximate definition that suffices to capture the essence of logical atomicity, and to understand our proof, is:

$$\langle x.\, P\rangle\ e\ \langle \Phi\rangle \quad \triangleq \quad \forall \Psi, \left[{}^{\top}\!\Rrightarrow^{\emptyset} \exists x.\ P\ *\ \left(\forall v.\, \Phi\ v\ \mathrel{-\!\!*}\ {}^{\emptyset}\!\Rrightarrow^{\top} \Psi\ v\right)\right]\ \mathrel{-\!\!*}\ \mathsf{wp}\ e\ \{\Psi\}$$

In this formula, the variable $P$ is a Cosmo assertion (of type vProp); the variables $\Phi$ and $\Psi$ are predicates on values (of type $\mathrm{Val} \to$ vProp); $P$ and $\Phi$ may refer to the name $x$. The assertion $\mathsf{wp}\ e\ \{\Psi\}$ is the weakest precondition for program $e$ and postcondition $\Psi$ (recall that, in Iris, Hoare triples are syntactic sugar for weakest preconditions).

The purpose of a logically atomic triple is to give a specification to a non-atomic program *e as if it were atomic.* In practice, we require that the proof of $e$ accesses the precondition and turns it into the postcondition in *one atomic step* only, which we call the commit point of this

logically atomic program. That is, if $e$ satisfies the triple $\langle x.\ P\rangle\ e\ \langle\Phi\rangle$, then it can perform several steps of computation but, as soon as it accesses the resource $P$, it must return the resource $\Phi$ in the same step of computation.[9] Once it has done so, $e$ can perform further computation steps but $P$ is not available anymore. As explained in §6.1.2, thanks to this constraint, the client of this specification can open invariants around $e$ as if $e$ were atomic.

To capture this atomicity requirement, we ask the proof of the logically atomic triple for $e$ to be valid for any postcondition $\Psi$ chosen by the client. Given that $\Psi$ is arbitrary, the only means of establishing this postcondition is to use the premise ${}^{\top}\!\Rightarrow^{\emptyset}\exists x.\ P\ *\ (\forall v.\ \Phi\ v\ \mathrel{-\!\!*}\ {}^{\emptyset}\!\Rightarrow^{\top}\Psi\ v)$, which is known as an *atomic update*. When desired, this atomic update gives access to the precondition $P$ for some value of $x$, and, in exchange for the postcondition $\Phi$ of the logically atomic triple, it returns the resource $\Psi$, which can then be used to finish the proof. Crucially, the *masks* $\emptyset$ and $\top$ annotating the *fancy updates* ${}^{\top}\!\Rightarrow^{\emptyset}$ and ${}^{\emptyset}\!\Rightarrow^{\top}$ require that the atomic update be used during one atomic step only, as required.

Using the invariant rules of Iris (§2.3.2), it is easy to show that atomic updates can be used to open and close invariants. Rule LAInv follows as a corollary, rule LAHoare is immediate (Figure 6.2).

### 6.3.7 Proof of `try_enqueue`

We now outline the proof that `try_enqueue` satisfies its specification from Figure 6.4. The proof for `try_dequeue` is similar; those for `enqueue` and `dequeue` are deduced from the previous two by an obvious induction; and the proof of `make` is simply a matter of initializing the ghost state. The interested reader may find these proofs, conducted in the Coq proof assistant, in our repository (Mével et al., 2021).

Recalling here the specification in Figure 6.4, and unfolding the definition of $\mathsf{QueueInv}\ q\ \gamma$, we ought to prove the following assertion:

$$\boxed{\mathsf{QueueInvInner}\ q\ \gamma\ \gamma_{\mathrm{st}}\ \gamma_{\mathrm{tok}}}$$

$$\left\langle \mathcal{T}, \mathcal{H}, n, (v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1}).\ \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \ \mathcal{H}\ \ \ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})]\ \ \ *\ \uparrow\mathcal{V} \right\rangle$$

$$\texttt{try\_enqueue}\ q\ v$$

$$\left\langle \lambda b.\ \vee \begin{bmatrix} \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \ \mathcal{H}\ \ \ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})] & * & b = \texttt{false} \\ \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ (\mathcal{H} \sqcup \mathcal{V})\ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1}), (v, \mathcal{V})] & * \uparrow\mathcal{H} * & b = \texttt{true} \end{bmatrix} \right\rangle$$

After unfolding the logically atomic triple, we must prove $\mathsf{wp}\ (\texttt{try\_enqueue}\ q\ v)\ \{\Psi\}$ for any $\Psi$, when in the proof context we have the internal invariant of the queue (with ghost variables $\gamma, \gamma_{\mathrm{st}}, \gamma_{\mathrm{tok}}$) as well as the atomic update whose precondition and postcondition are that of the triple above. We then step through the program using usual weakest-precondition calculus.

The first interesting step is the atomic read of `head`. The ownership of that reference is shared in the invariant of the queue. Hence, to access it, we must open the invariant; then we get the points-to assertion, we can step through the read operation, return the points-to assertion

[9] The full definition of logically atomic triples allows to access the precondition atomically before the commit point, hence without turning it into the postcondition. This is called *aborting*; it is not needed in our proof, and out of the scope of this dissertation.

and close the invariant again. After we have done so, and thus forgotten all the quantities which are existentially quantified inside that invariant, we learn little about the value that has just been read, excepted that it is a non-negative integer, say $k$.

The second interesting step is the atomic read at index $\widehat{k}$ of the array `statuses`. Again the invariant owns this cell, so we open it around the read instruction. This read yields some value $s^1$ and, since it is atomic, it also augments our current (thread) view with the view $\mathcal{S}^1$ which, at this moment, is stored in this cell. In other words, we gain the (persistent) assertion $\uparrow \mathcal{S}^1$. We can remember more information before closing the invariant: indeed, from the authority of $\gamma_{\mathrm{st}}$ found in the invariant, we derive a witness for the strict monotonicity of the status that we just read: $\mathsf{Witness}\ \gamma_{\mathrm{st}}\ \widehat{k}\ (s^1, \mathcal{S}^1)$.

Next, the program tests whether $s^1 = 2k$. If the test fails, then the program returns `false`. In this case, we have to provide as postcondition of the logically atomic triple the untouched representation predicate that is in its precondition $(\mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(v_0, \mathcal{V}_0), ..., (v_{n-1}, \mathcal{V}_{n-1})])$. We do this by committing the atomic update in a trivial way, then conclude the proof.

If $s^1 = 2k$, the program proceeds to performing `CAS head` $k\ (k+1)$. To access `head`, we open the invariant again. If that operation fails, the program also returns `false` and, after closing the invariant without having updated ghost state, we conclude as before.

If the CAS succeeds, then a number of things happen logically. First, if $h$ and $t$ are the values of `head` and `tail` at the moment of the CAS, then $h = k$. Second, we deduce that the buffer is not full, i.e. $h < t + C$. Indeed, the invariant directly gives us $h \leq t + C$; if we had $h = t + C$, then in particular, $t \leq k - C < h$, so the invariant would own the following for slot $\widehat{k - C}$:

$$\mathsf{Occupied}\ (k - C)\ (s_{\widehat{k-C}}, \mathcal{S}_{\widehat{k-C}})\ (v_{k-C}, \mathcal{V}_{k-C})\ \lor\ s_{\widehat{k-C}} = 2(k - C)$$

Because $\widehat{k - C} = \widehat{k}$, this implies:

$$s_{\widehat{k}} = 2(k - C) + 1\ \lor\ s_{\widehat{k}} = 2(k - C)$$

In either case, we get $s_{\widehat{k}} < 2k = s^1$, which contradicts the monotonicity of the status of that slot. We derive the contradiction by combining the assertion $\mathsf{Witness}\ \gamma_{\mathrm{st}}\ \widehat{k}\ (s^1, \mathcal{S}^1)$ that we had since we read the status, and the authority $\boxed{\bullet\, (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}})}^{\gamma_{\mathrm{st}}}$ that is found in the invariant.

Third, we thus know that $h - C \leq k - C < t$, so that the invariant gives us:

$$\mathsf{Available}\ (k - C)\ (s_{\widehat{k-C}}, \mathcal{S}_{\widehat{k-C}})\ \lor\ s_{\widehat{k-C}} = 2(k - C) + 1$$

Again the second disjunct is absurd because the status is monotonic. Hence the slot we are claiming is available indeed. From this we get $s_{\widehat{k}} = 2k = s^1$, the points-to assertion $(\mathtt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} -) \mathbin{@} \mathcal{S}_{\widehat{k}}$ and the read token of rank $k - C$. The sets of read tokens and write tokens depend on the value of `tail` and `head`, and we have just incremented the latter, to $k + 1$, so we destruct the read token of rank $k - C$ and create a write token of rank $k$ instead, giving it as payload the item $(v, \mathcal{V})$ that we are trying to enqueue.

This is also when the *strict* monotonicity of the status comes into play: because $s_k = s^1$, it gives us $\mathcal{S}_k \sqsubseteq \mathcal{S}^1$. But we have $\uparrow \mathcal{S}^1$ in our proof context, so we obtain the points-to assertion as a subjective assertion: $\mathtt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} -$.

We commit the atomic update now. Indeed the successful CAS is the commit point of `try_enqueue`. We know that the program will return `true`, so we must provide the corresponding

postcondition of the logically atomic triple, where our item $(v, \mathcal{V})$ has been appended to the public state of the queue. Thus we take a ghost step to update this public state. By committing, we finally obtain the assertion $\Psi\ \texttt{true}$ that will serve at the end of the proof, since `true` is the return value of the operation. Along the way, we also collect the persistent assertion $\uparrow \mathcal{V}$ from the precondition of the logically atomic triple.

Finally, we keep on our side the non-atomic points-to assertion $\texttt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} -$ and the write token, we reconstruct the invariant updated for the new value of `head`, and we close it.

The next step of the program writes the value $v$ to the non-atomic item field, which is easy since we have the points-to assertion at hand. This assertion then becomes $\texttt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} v$. We turn it back to an objective assertion, which gives us a view $\mathcal{W}$ and two assertions $\uparrow \mathcal{W}$ and $(\texttt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} v) \mathbin{@} \mathcal{W}$.

The last step of the program is to update the (atomic) status of the slot. Once more we open the invariant. If we again note $h$ and $t$ the current values of `head` and `tail` (potentially different from the last time we opened the invariant), then owning a write token for rank $k$ teaches us that $t \leq k < h$. The invariant then gives us for slot $\widehat{k}$:

$$\mathsf{Occupied}\ k\ (s_{\widehat{k}}, \mathcal{S}_{\widehat{k}})\ (v_k, \mathcal{V}_k)\ \lor\ s_{\widehat{k}} = 2k$$

The left disjunct would own our write token, but we already have it and it is exclusive; hence we are in the right disjunct, $s_{\widehat{k}} = 2k = s^1$. We perform the atomic write with value $s^2 \triangleq 2k + 1$ (strict monotonicity is respected), and since we have both $\uparrow \mathcal{V}$ and $\uparrow \mathcal{W}$ in context, we can push the view $\mathcal{S}^2 = \mathcal{V} \sqcup \mathcal{W}$ to this atomic cell while writing. We then switch to the left disjunct, by constituting the assertion:

$$\mathsf{Occupied}\ k\ \left(s^2, \mathcal{S}^2\right)\ (v, \mathcal{V})\ \dashv\vdash\ * \begin{cases} s^2 = 2k + 1 \\ (\texttt{items}[\widehat{k}] \rightsquigarrow_{\mathrm{na}} v) \mathbin{@} \mathcal{S}^2 \\ \mathsf{TokenW}\ \gamma_{\mathrm{tok}}\ k\ (v, \mathcal{V}) \\ \mathcal{V} \sqsubseteq \mathcal{S}^2 \end{cases}$$

Hence we return the non-atomic points-to assertion and the write token to the invariant before closing it.

## 6.4 A simple pipeline

We now demonstrate the use of our specification of a concurrent queue by a simple client application, that chains two treatments on a sequence of data, where each treatment is applied in a separate thread. Thus the sequence of intermediate values is transferred from a producer to a consumer using a concurrent queue.

### 6.4.1 Implementation of the pipeline

The code of the application is presented in Figure 6.8. It provides a single function, `pipeline`, which takes as arguments two functions $g$ and $f$, a sequence `xs`, and returns a sequence obtained by applying $g \circ f$ to each item of the input sequence. The functions $g$ and $f$ need not be pure: they can have side effects and rely on some state.

For simplicity, input and output sequences are encoded as (non-atomic) arrays, whose length can be obtained via a primitive operation `length` $-$. However the implementation can be

```
let pipeline g f xs =
  let n = length xs in
  let ys = make () in
  fork (pipef n xs ys f);
  let zs = array_na[n] () in
  pipeg n ys zs g;
  zs

let pipef n xs ys f =
  for k from 0 to n − 1 do
    let x = xs[k]_na in
    let y = f x in
    enqueue ys y
  done

let pipeg n ys zs g =
  for k from 0 to n − 1 do
    let y = dequeue ys in
    let z = g y in
    zs[i]_na ← z
  done
```

Figure 6.8: Implementation of a pipeline

$$\left\{ \mathtt{xs} \rightsquigarrow^{*}_{\mathrm{na}} [x_0, ..., x_{n-1}] \ * \ \ast_{0 \le k < n} \ \mathsf{wp}\ f\ x_k\ \{\lambda y.\ \mathsf{wp}\ g\ y\ \{\lambda z.\ R\ k\ z\}\} \right\}$$

$$\mathtt{pipeline}\ g\ f\ \mathtt{xs}$$

$$\left\{ \lambda \mathtt{zs}.\ \exists z_0, ..., z_{n-1}.\ \mathtt{zs} \rightsquigarrow^{*}_{\mathrm{na}} [z_0, ..., z_{n-1}] \ * \ \ast_{0 \le k < n} \ R\ k\ z_k \right\}$$

Figure 6.9: A specification for the pipeline

modified to consume and produce lists of unknown length, or even infinite streams, provided an encoding of such data structures; we would then use a sentinel value in the queue to signal the end of stream.

The code is straightforward: we create a concurrent queue `ys`, then we fork a thread. The queue is shared between the main thread and the forked thread, while `xs` is transmitted to the forked thread. The forked thread reads items from `xs` in turn, applies $f$ to them and enqueues the results. The main thread creates a new (non-atomic) array `zs` to store the output; then, it dequeues $n$ items, where $n$ is the number of items in the input sequence, applies $g$ to them, and adds the results to `zs`. Finally, it returns `zs`.

### 6.4.2 Specification of the pipeline

A possible specification for this pipeline is shown in Figure 6.9. It is higher-order, and expressed using the weakest-precondition predicate.

In postcondition we obtain one assertion $R\ k\ z_k$ for each item of the stream, related to its position $k$ (in particular, $R$ may relate the output value $z_k$ to the input value $x_k$).

In the precondition, essentially, we want to state that for some predicates $P$ and $Q$, we have the assertions $P\ k\ x_k$ for all items of the input stream, and functions $f$ and $g$ satisfy Hoare triples of the form:

$$\{P\ k\ x\}\ f\ x\ \{\lambda y.\ Q\ k\ y\}$$
$$\{Q\ k\ y\}\ g\ y\ \{\lambda z.\ R\ k\ z\}$$

so that, by the chaining rule, the composition satisfies:

$$\{P\ k\ x\}\ g\ (f\ x)\ \{\lambda z.\ R\ k\ z\}$$

$$\begin{aligned}
\gamma &: \textsc{Auth}(\textsc{Ex}(\textsc{View} \times \textsc{View} \times \textsc{List}(\textsc{Val} \times \textsc{View}))) \\
\gamma_\mathrm{f},\ \gamma_\mathrm{g} &: \textsc{Auth}(\textsc{Ex}(\mathbb{N}))
\end{aligned}$$

$$\mathsf{PipeInv}\ g\ f\ R \triangleq \exists q, \gamma, \gamma_\mathrm{f}, \gamma_\mathrm{g}.\ \mathsf{QueueInv}\ q\ \gamma\ *\ \boxed{\mathsf{PipeInvInner}\ g\ f\ R\ \gamma\ \gamma_\mathrm{f}\ \gamma_\mathrm{g}}$$

$$\mathsf{PipeInvInner}\ g\ f\ R\ \gamma\ \gamma_\mathrm{f}\ \gamma_\mathrm{g} \triangleq \left\{ \begin{array}{l} \exists n_\mathrm{f}, n_\mathrm{g}, \mathcal{T}, \mathcal{H}, (y_{n_\mathrm{g}}, \mathcal{V}_{n_\mathrm{g}}), \ldots, (y_{n_\mathrm{f}-1}, \mathcal{V}_{n_\mathrm{f}-1}). \\ \quad * \left\{ \begin{array}{l} n_\mathrm{g} \le n_\mathrm{f}\ *\ \boxed{\bullet n_\mathrm{f}}^{\gamma_\mathrm{f}}\ *\ \boxed{\bullet n_\mathrm{g}}^{\gamma_\mathrm{g}} \\ \mathsf{IsQueue}\ \gamma\ \mathcal{T}\ \mathcal{H}\ [(y_{n_\mathrm{g}}, \mathcal{V}_{n_\mathrm{g}}), \ldots, (y_{n_\mathrm{f}-1}, \mathcal{V}_{n_\mathrm{f}-1})] \\ \quad \ast_{n_\mathrm{g} \le k < n_\mathrm{f}}\ (\mathsf{wp}\ g\ y_k\ \{\lambda z.\ R\ k\ z\}) \mathbin{@} \mathcal{V}_k \end{array} \right. \end{array} \right.$$

$$\mathsf{PipeF}\ g\ f\ R\ \gamma\ \gamma_\mathrm{f}\ n_\mathrm{f}\ \mathtt{xs}\ [x_0, \ldots, x_{n-1}] \triangleq * \left\{ \begin{array}{l} n_\mathrm{f} \le n\ *\ \boxed{\bullet n_\mathrm{f}}^{\gamma_\mathrm{f}} \\ \mathtt{xs} \rightsquigarrow^*_\mathrm{na} [x_0, \ldots, x_{n-1}] \\ \quad \ast_{n_\mathrm{f} \le k < n}\ \mathsf{wp}\ f\ x_k\ \{\lambda y.\ \mathsf{wp}\ g\ y\ \{\lambda z.\ R\ k\ z\}\} \end{array} \right.$$

$$\mathsf{PipeG}\ g\ f\ R\ \gamma\ \gamma_\mathrm{g}\ n_\mathrm{g}\ \mathtt{zs}\ [z_0, \ldots, z_{n-1}] \triangleq * \left\{ \begin{array}{l} n_\mathrm{g} \le n\ *\ \boxed{\bullet n_\mathrm{g}}^{\gamma_\mathrm{g}} \\ \mathtt{zs} \rightsquigarrow^*_\mathrm{na} [z_0, \ldots, z_{n-1}] \\ \quad \ast_{0 \le k < n_\mathrm{g}}\ R\ k\ z_k \end{array} \right.$$

Figure 6.10: Internal invariants of the pipeline

By using weakest preconditions instead of Hoare triples, we avoid mentioning the predicate $P$; by taking advantage of the higher-order nature of Iris, we can nest the triples so as to conceal the intermediate predicate $Q$.

### 6.4.3 Proof of the specification for the pipeline

We now prove the following result.

**Theorem 5** *The code shown in Figure 6.8 satisfies the specification appearing in Figure 6.9.*

The proof relies on the assertions presented in Figure 6.10. The assertion $\mathsf{PipeInv}\ g\ f\ R$, which is persistent, join together the internal invariant of the queue with that of the pipeline. The two assertions $\mathsf{PipeF}\ g\ f\ R\ \gamma\ \gamma_\mathrm{f}\ n_\mathrm{f}\ \mathtt{xs}\ [x_0, \ldots, x_{n-1}]$ and $\mathsf{PipeG}\ g\ f\ R\ \gamma\ \gamma_\mathrm{g}\ n_\mathrm{g}\ \mathtt{zs}\ [z_0, \ldots, z_{n-1}]$ are owned by the threads which compute $f$ and $g$, respectively, and describe their loop invariants.

We again associate the queue to a ghost variable $\gamma$. In addition, we use two ghost variables $\gamma_\mathrm{f}$ and $\gamma_\mathrm{g}$ whose values are the current positions $n_\mathrm{f}$ and $n_\mathrm{g}$ of the loops computing $f$ and $g$, respectively. This ghost state allows both threads, while in their respecting loops, to agree with the shared invariant on these values. At any time, we have $0 \le n_\mathrm{g} \le n_\mathrm{f} \le n$.

- Indices in the range $[n_\mathrm{f}, n)$ have not been processed by $f$ yet. Hence, for these indices, we still have the weakest-precondition assertions $\mathsf{wp}\ f\ x_k\ \{-\}$ initially provided to the pipeline. These assertions can be regarded as a permission to run $f$ once on the corresponding items. The thread computing $f$ owns these assertions, and it also owns the array $\mathtt{xs}$ containing the input items.

- Indices in the range $[n_\mathrm{g}, n_\mathrm{f})$ have been processed by $f$ but are yet to be processed by $g$. Hence, we have consumed their initial weakest-precondition assertions, and have obtained weakest-precondition assertions $\mathsf{wp}\ g\ y_k\ \{-\}$ as a result. These assertions are stored in the shared invariant. The invariant also owns the queue `ys`, whose contents are the intermediate items for exactly this range of indices.

- Indices in the range $[0, n_\mathrm{g})$ have been processed by both $f$ and $g$. Hence, we have consumed their intermediate weakest-precondition assertions, and have obtained postconditions $R\ k\ z_k$ as a result. These are owned by the thread computing $g$, along with the array `zs` which contains the output items.

The key point is that the assertion $\mathsf{wp}\ g\ y_k\ \{-\}$ has been given by the thread computing $f$ to the invariant when enqueuing the corresponding item, and will be taken by the thread computing $g$ when dequeuing that item. This assertion is thus exchanged between two threads with differing views of the shared memory, and transits via a neutral ground: an objective invariant. We make the assertion objective by specifying at which view it holds: namely, the view $\mathcal{V}_k$ which the enqueuer had and which the dequeuer will acquire. Therefore, we rely crucially on the enqueuer-to-dequeuer synchronization guaranteed by the queue.

With these invariant assertions correctly stated, the proof is rather straightforward. When creating the pipeline, we have $0 = n_\mathrm{g} = n_\mathrm{f}$ and assertions PipeInvInner and PipeG hold trivially (the queue is empty and there are no output items computed yet); the assertion PipeF is constituted exactly from the preconditions of the pipeline (Figure 6.9), and is given by the main thread to the child thread that will compute $f$. When the pipeline has completed its work, we have $n_\mathrm{g} = n_\mathrm{f} = n$ and the assertion PipeG provides exactly the postcondition of the pipeline.

Thanks to the logically atomic triples in the specification of the queue, when enqueuing (respectively, dequeuing), we can open the invariant of the pipeline and move assertions to it (respectively, from it) as already explained.

# Chapter 7

# Related Work

## 7.1 Program verification in weak memory models

There is a wide variety of work on weak memory models and on approaches to program verification with respect to weak memory models. We restrict our attention to program verification based on extensions of Concurrent Separation Logic, because this is the most closely related work and because we believe that the abstraction and compositionality of Separation Logic are features that will be absolutely essential in the long run. Vafeiadis (2017) offers a good survey of some of the papers that we cite now.

The first instance of Separation Logic for weak memory appears to be RSL (Vafeiadis and Narayan, 2013). It is based on an axiomatic semantics of a fragment of the C11 memory model. It supports non-atomic accesses, where it enforces the absence of data races; release/acquire accesses, with reasoning rules that allow ownership transfers from writer to reader; and relaxed accesses, without any ownership transfer. The logic involves a permission $Rel(\ell, Q)$ to perform a release write at location $\ell$ of a value $v$ while relinquishing the assertion $Q\ v$. Symmetrically, there is a permission $Acq(\ell, Q)$ to perform an acquire read at location $\ell$ of a value $v$ and obtain the assertion $Q\ v$. The release and acquire permissions are created, and the predicate $Q$ is fixed, when the location is allocated. RSL can verify simple concurrent data structures, such as a spin lock. However, because it lacks invariants and ghost state, its expressive power is limited.

FSL (Doko and Vafeiadis, 2016) extends RSL with support for release/acquire fences. A release write can be replaced with a release fence followed with a relaxed write; symmetrically, an acquire read can be replaced with a relaxed read followed with an acquire fence. Two new assertions $\Delta P$ and $\nabla P$ witness that $P$ has been released by the last release fence or will be acquired by the next acquire fence, respectively. In Cosmo, both of these assertions would be replaced by $P \mathbin{@} \mathcal{V}$, for a well-chosen view $\mathcal{V}$. Multicore OCaml has no fences; instead, atomic read and write instructions are used to transmit views. FSL++ (Doko and Vafeiadis, 2017) extends FSL with shared read permissions for non-atomic accesses, support for CAS, and ghost state. As an application, the authors of FSL++ prove the correctness of the Rust library "ARC" (atomic reference counter).

GPS (Turon et al., 2014) supports a fragment of C11 that includes non-atomic accesses and release/acquire accesses. Like the papers cited above, it is based on an axiomatic presentation of the C11 memory model. It introduces ghost state and a notion of per-location protocol that governs a single atomic memory location. At the cost of rather complex reasoning rules, it

offers good expressive power. The case studies described in the paper include a spin lock, a bounded ticket lock, a Michael-Scott queue, and a circular FIFO queue. In comparison, Cosmo does not need per-location protocols. Because atomic memory locations in Multicore OCaml have sequentially consistent behavior, our "atomic points-to" predicate is objective. Therefore, in Cosmo, an invariant can refer to one or more atomic memory locations if desired. This has been illustrated in §5.3 and §5.6. On these examples, Cosmo is significantly simpler to use than GPS: the proof of the ticket lock in Cosmo is about 170 lines of Coq code[1] (specifications and proofs combined) whereas the corresponding proof in iGPS (a reconstruction of GPS, covered later) is about 700 lines[2].

Sieczkowski et al. (2015), already cited earlier (§1), present iCAP-TSO, a variant of Concurrent Separation Logic that is sound with respect to the TSO memory model. The logic includes a high-level fragment whose reasoning rules are those of Concurrent Separation Logic, where the pre- and postcondition are interpreted relative to the current thread's point of view. Informally, an assertion holds from the point of view of a thread if it holds of the global state updated with the thread's store buffer and no other thread has pending writes that could affect this assertion. Thus, the subjective points-to assertion $x \mapsto v$ guarantees that a read instruction will return the value $v$. The logic is proved sound with respect to an operational semantics where store buffers are explicit. This work and ours seem quite close in spirit, but differ due to the choice of a different memory model. In particular, Sieczkowski et al. have no notion of view. In order to reason about the behavior of store buffers, they propose a few ad hoc logical constructs, such as an "until" modality $P \,\mathcal{U}_t\, Q$, which means that $P$ holds until an update from thread $t$ is flushed to main memory, at which point $Q$ holds. By contrast with Cosmo, iCAP-TSO's high-level logic cannot reason about transfers of ownership. This kind of reasoning must be carried out in a lower-level logic. It is possible to verify a data structure (e.g., a lock) in the low-level logic and to establish a specification expressed in the high-level logic.

Kaiser et al. (2017) propose the first instantiation of Iris in a weak-memory setting. This involves abandoning the axiomatic memory models used by the papers cited above and switching to an operational semantics. The paper proposes such a semantics for a fragment of C11 that includes two kinds of memory locations, namely non-atomic locations and release/acquire locations. Like Dolan et al.'s semantics of Multicore OCaml (2018), this semantics involves timestamps and histories. It also includes a "race detector" which ensures that data races on non-atomic memory locations lead to undefined behavior. In comparison, Dolan et al.'s semantics does not need a race detector: every Multicore OCaml program has a well-defined set of permitted behaviors. Several aspects of our work are modeled after Kaiser et al.'s paper. Indeed, in a first step, they instantiate Iris, yielding a low-level "base logic". Then, in a second step, they define a higher-level logic, whose reasoning rules are easier to use, and whose assertions are implicitly parameterized with the thread's view. We follow this approach. Kaiser et al. construct not one, but two high-level logics, iGPS and iRSL, which are inspired by GPS and RSL, and benefit from the power of Iris. iGPS introduces a new feature, namely single-writer protocols. Cosmo follows a different route: as explained above, it does not need per-location protocols. Furthermore, it puts emphasis on explicit user-level reasoning with abstract views, via assertions like $P \mathbin{@} \mathcal{V}$ and $\uparrow \mathcal{V}$.

iRC11 (Dang et al., 2020) extends iGPS with additional features of the C11 memory model,

[1] `https://gitlab.inria.fr/cambium/cosmo/-/blob/master/theories/examples/ticketlock.v`
[2] `https://gitlab.mpi-sws.org/FP/igps/-/blob/master/theories/examples/ticket_lock.v`

namely relaxed accesses and release/acquire fences. It is based on ORC11, an operational presentation of the Repaired C11 memory model (Lahav et al., 2017). One of its key features is support for cancellable invariants, an abstraction whose implementation in Iris in an SC setting has been well-understood for some time,[3] but whose implementation in a weak-memory setting is significantly more challenging. In particular, the tokens that represent a fraction of the ownership of an invariant are not objective, and therefore cannot appear in an invariant; they must be transmitted from one thread to another via a synchronization operation. Dang et al.'s implementation of cancellable invariants involves explicit reasoning about views, yet this is not apparent in the cancellable invariant API, a remarkable achievement. A user of iRC11 need not know about views.

On top of iRC11, Dang et al. reconstruct Lifetime Logic and the model of the Rust type system previously built by Jung et al. in an SC setting (2018a). Furthermore, they prove the soundness of Rust's ARC library.

## 7.2 Verification of fine-grained concurrent data structures

Putting weak-memory concerns aside, the verification of fine-grained concurrent data structures is a well-studied problem with a particularly rich literature. Several approaches are tried, targeting various verification frameworks, various data structures in different contexts.

The notion of *linearizability* is traditionally regarded as central for specifying such libraries. Dongol and Derrick (2015) give a survey of the different techniques used to prove linearizability of concurrent libraries. Of particular interest in the context of separation logic is the technique of *logical atomicity*. In a sequentially consistent context, logically atomic specifications have recently been proved to imply linearizability (Birkedal et al., 2021). Logical atomicity has been developed through several iterations over the last decade (da Rocha Pinto et al., 2014; Jung et al., 2015; Jacobs and Piessens, 2011; Svendsen et al., 2013; Jung et al., 2020). In this dissertation, we adapt a modern version of Iris's logically atomic triples (Jung, 2019) to the setting of Cosmo.

Another popular approach for proving the correctness of concurrent libraries is the use of refinement with respect to a simpler implementation. This is the track chosen by ReLoC (Frumin et al., 2018), which has recently been combined with logical atomicity (Frumin et al., 2020). Interestingly, ReloC has been recently used for proving the correctness of several concurrent queue implementations (Vindum and Birkedal, 2021; Vindum et al., 2021), one of which is very close to ours. However, these proofs do not handle relaxed memory behaviors, so that they do not provide a solution to the problem of specifying the lack of happens-before relationship between some data structure accesses, which we discussed in §6.1.3. Because it lacks some happens-before relationships, our queue implementation *is not* a refinement of a naive implementation which would use a lock to protect a sequential implementation, so a refinement-based approach would not be useful for proving our library correct. The refinement approach has also been used to prove correct some data structures used in a concurrent garbage collector (Zakowski et al., 2018).

In a weakly consistent setting, new problems arise. As discussed in §6.1.3, even the definition of linearizability needs special care. Smith et al. (2019) propose new definitions of linearizability

[3]For example, in RustBelt (Jung et al., 2018a), non-atomic persistent borrows are a form of cancellable invariant.

for the case of weak memory models. In contrast, other authors develop new kinds of specifications which allow for weak behaviors of the library itself (Krishna et al., 2020; Emmi and Enea, 2019; Raad et al., 2019). Notably, Raad et al. propose a compositional framework in which the specification of a library describes the events associated to the library and their allowed orderings. In other words, each library interface describes its own axiomatic memory model, and there is no built-in model: models such as RC11 and TSO are implemented as the specification of a library of memory access primitives. Dang et al. (2022) use this approach to verify a finer specification for a concurrent queue similar to ours. This formalism is heavier than Cosmo—as it involves manipulating event graphs—and might be more expressive, although that point should be investigated. We found that our method of combining logically atomic triples with views is expressive and allows for concise specifications at the same time. Previous works include the generalization of various methods to weak consistency: Lê et al. (2013b,a) use manual methods directly tied to the axiomatic memory model to prove the correctness of a queue and of a work-stealing algorithm, while Lahav and Vafeiadis (2015) adapt the Owicki-Gries methodology to the release-acquire fragment of the C11 memory model and apply it to verify a read-copy-update library. Although various earlier works develop the idea of a separation logic for programs with a relaxed memory semantics, as shown in §7.1, few of these papers address the problem of the full functional correctness of a data structure. In particular, the specification proposed for a circular buffer in GPS (Turon et al., 2014) is a weak specification in the style of the persistent specification given at the end of §6.1.2: in contrast to ours, it does not specify in which order the elements leave the queue.

# Chapter 8

# Conclusion and future work

In this dissertation, we have laid the groundwork for verifying Multicore OCaml programs in a suitable variant of Concurrent Separation Logic. We have instantiated Iris (Jung et al., 2018b) for Multicore OCaml, yielding a low-level logic, BaseCosmo, which exposes all of the details of the Multicore OCaml memory model (Dolan et al., 2018), including timestamps, histories, and views. BaseCosmo can in principle verify arbitrary Multicore OCaml programs, including programs that involve data races on non-atomic memory locations.

However, BaseCosmo is too low-level to be pleasant and convenient. Thus, we have built a higher-level logic, Cosmo, aiming to provide simpler reasoning principles. In order to achieve this result, we have removed the ability of reasoning about programs that race on non-atomic memory locations—we believe it is reasonable to assume that most (correct) practical programs are exempt of such races. This has allowed us to offer the illusion that a non-atomic memory location stores a single value, to remove all mention of timestamps and histories, and to present views to the user as an abstract type, equipped with the structure of a bounded semilattice. We believe that the manner in which Cosmo allows reasoning about weak memory is original and relatively simple and natural. A key mechanism is the ability to split an arbitrary assertion $P$ into an objective fragment $P@\mathcal{V}$ and a subjective fragment $\uparrow\mathcal{V}$ and to later reassemble these fragments. The two fragments are transmitted from one thread to another by different means: whereas sharing an objective assertion requires no runtime synchronization and is typically achieved via an Iris invariant, transmitting a view is typically achieved by relying on the "release/acquire" behavior of atomic writes and reads.

Cosmo contains Concurrent Separation Logic as a fragment, that allows reasoning about coarse-grained concurrent programs in the same way as in Concurrent Separation Logic.

We have illustrated the applicability of the high-level logic Cosmo to the study of several concurrent data structures, including a relaxed multiple-reader multiple-write queue. In these case studies, we found the view mechanism to be a pleasant and concise formalism to specify and reason about thread synchronization. In order to write satisfactory specifications for such lock-free data structures, we have used a notion of logical atomicity. Although it is straightforward to port the existing definition of that notion (Jung, 2019) to a weak memory setting, logical atomicity in such a setting does not imply "linearizability" in the traditional sense: logical atomicity does imply the existence of a linear history, but synchronization between successive events in that history is not guaranteed.

As said earlier, our high-level logic Cosmo does not support reasoning about data races on

non-atomic memory locations. Yet, there are some legitimate programs which exploit such races correctly. For instance, idempotent work-stealing (Michael et al., 2009) aims at reducing synchronizations between cooperating threads in a load-balancing algorithm, under the assumption that the tasks can be repeated without compromising program correctness. Writing concurrent programs which are willingly racy is a preserve of memory model experts, and the correctness of such programs is especially subtle. Therefore, there is a call for a program logic that allows checking them. Although it is possible to reason about these racy programs in the low-level logic BaseCosmo, an opportunity for future research is to discover what rules can be proposed to reason about these programs without abandoning the simplicity of Cosmo. Drawing inspiration from iGPS (Kaiser et al., 2017), one direction might be to have racy non-atomic locations be governed by per-location protocols, involving some notion of state monotonicity.

Another future work may consist in proposing suitable specifications for Multicore OCaml's `Domain` API, which offers a set of primitive synchronization operations. On this basis, one may verify concurrent data structures that have been developed for Multicore OCaml, for instance a multiword compare-and-swap,[1] a suite of lock-free work-stealing queues, bags, and hash tables, and an implementation of Turon's reagents (2012).

On an even more practical ground, much engineering work may be done to make BaseCosmo applicable to *actual* Multicore OCaml code. At the moment, the code that BaseCosmo reasons about belongs to a toy language whose syntax is deeply embedded within Coq. This minimalist language focuses on memory-concerned operations and lacks most of the features of the actual Multicore OCaml language, including types, modules, objects, algebraic data types, exceptions and algebraic effects. A realistic verification tool should

1. support a satisfying subset of Multicore OCaml, including the aforementioned features,

2. be connected to actual Multicore OCaml code

3. and offer an interface to proving specifications that minimizes human effort.

Regarding item 1, many programming features do not interact with the memory model and should be straightforward to support, if cumbersome. However, exceptions, algebraic effects and their handlers[2] have a non-local impact on the control flow and thus cannot be dealt with seamlessly. Specialized program logics have been designed to reason about languages with exceptions and algebraic effects (de Vilhena and Pottier, 2021). Work remains to be done to build a unified framework that would support them in addition to weak-memory concurrency. Item 2 could involve implementing an automatic translation tool from deep-embedded code to Multicore OCaml source files, or the converse. For now, Multicore OCaml itself has no notion of formal specification—neither does it have, in fact, a formal semantics. However, the Gospel project (Charguéraud et al., 2019) aims at defining a specification language for OCaml; that specification language is translated to Separation Logic. Provided that Gospel be extended to Concurrent Separation Logic, it would then be natural that our verification framework be capable of reading these specifications.

[1] This uses RDCSS, which has been verified in Iris by Jung et al. (2020).

[2] Algebraic effects and handlers are another feature added to OCaml by Multicore OCaml. Although this feature is conceptually independent from concurrency, it constitutes a convenient tool for implementing cooperative scheduling, hence it has been deemed useful to ship both features together.

Regarding item 3: in the vein of the Iris Proof Mode (Krebbers et al., 2017), the proof mode we have built for Cosmo is easy to use, but verbose: it follows the syntax of the implementation closely and requires writing one tactic invocation per reduction step of the subject program. Such a proof script is legible, but lengthy and sensitive to small code changes. A user might hope for a more automated interface which would tackle the mundane parts and let her focus on the interesting parts, those where human intelligence is the most required: designing ghost state and invariants, and choosing when to open and close them. Several proof efforts have been conducted to automate proofs of programs in Iris (Sammler et al., 2021; Mulder et al., 2022; Malecha et al., 2022). Designing a good automation infrastructure for Cosmo is thus a possible future work.

Lastly, on the theoretical ground, one may build a semantic model of Multicore OCaml's type system in Cosmo—that is, interpreting types of the source language as predicates of the logic. In addition to proving type soundness, this would help specifying typed functions which could be called by untyped (and unsafe) code, as long as the latter would conform to the binary interface for the input and output types.

# Bibliography


Sarita V Adve and Hans-J Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010. doi: 10.1145/1787234.1787255.

David Aspinall and Jaroslav Ševčík. Formalising Java's data race free guarantee. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 22–37. Springer, 2007. doi: 10.1007/978-3-540-74591-4_4.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Principles of Programming Languages (POPL)*, pages 55–66, January 2011. URL `https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf`.

John Bender and Jens Palsberg. A formalization of Java's concurrent access modes. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):142:1–142:28, 2019. URL `https://johnbender.us/assets/oopsla-2019.pdf`.

Lars Birkedal and Aleš Bizjak. Lecture notes on Iris: Higher-order concurrent separation logic. Lectures notes, December 2018. URL `https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf`.

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. Theorems for free from separation logic specifications. 5(ICFP), 2021. doi: 10.1145/3473586. URL `https://cs.au.dk/~birke/papers/free-theorems-sep-logic.pdf`.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*, pages 259–270, January 2005. URL `http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/permissions_paper.pdf`.

John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, June 2003. URL `http://www.cs.uwm.edu/~boyland/papers/permissions.pdf`.

Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016. URL `http://siglog.hosting.acm.org/wp-content/uploads/2016/07/siglog_news_9.pdf`.

Stephen D. Brookes. A semantics for concurrent separation logic. In *International Conference on Concurrency Theory (CONCUR)*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, August 2004. URL `http://dx.doi.org/10.1007/978-3-540-28644-8_2`.

Alexandre Buisse, Lars Birkedal, and Kristian Støvring. A step-indexed Kripke model of separation logic for storable locks. *Electronic Notes in Theoretical Computer Science*, 276:121–143, September 2011. URL `https://cs.au.dk/~birke/papers/locks.pdf`.

Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL - providing OCaml with a formal specification language. In *Formal Methods (FM)*, volume 11800 of *Lecture Notes in Computer Science*, pages 484–501. Springer, October 2019. URL `https://hal.inria.fr/hal-02157484v2`.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2014. doi: 10.1007/978-3-662-44202-9_9.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4(POPL):34:1–34:29, 2020. URL `https://hal.inria.fr/hal-02351793/`.

Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. Compass: Strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 792–808, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523451. URL `https://plv.mpi-sws.org/compass/paper.pdf`.

Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021. URL `http://cambium.inria.fr/~fpottier/publis/de-vilhena-pottier-sleh.pdf`.

Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 9583 of *Lecture Notes in Computer Science*, pages 413–430. Springer, January 2016. URL `https://plv.mpi-sws.org/fsl/base/paper.pdf`.

Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 448–475. Springer, April 2017. URL `https://plv.mpi-sws.org/fsl/ARC/paper.pdf`.

Stephan Dolan, Anil Madhavapeddy, and KC Sivaramakrishnan. Multicore OCaml. `https://github.com/ocaml-multicore/ocaml-multicore/wiki`, 2020.

Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *Programming Language Design and Implementation (PLDI)*, pages 242–255, June 2018. URL `http://kcsrk.info/papers/pldi18-memory.pdf`.

Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Computing Surveys (CSUR)*, 48(2):1–43, 2015. doi: 10.1145/2796550.

Michael Emmi and Constantin Enea. Weak-consistency specification via visibility relaxation. *Proceedings of the ACM on Programming Languages*, 3(POPL):60:1–60:28, 2019. URL `https://doi.org/10.1145/3290373`.

Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In *Logic in Computer Science (LICS)*, pages 442–451, July 2018. URL `https://iris-project.org/pdfs/2018-lics-reloc-final.pdf`.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. ReLoC Reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity. *arXiv preprint arXiv:2006.13635*, 2020.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, November 2007. URL `http://dx.doi.org/10.1007/978-3-540-76637-7_3`.

Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, April 2008. URL `http://www.cs.princeton.edu/~appel/papers/concurrent.pdf`.

Iris developers and contributors. Iris examples, 2021. URL `https://gitlab.mpi-sws.org/iris/examples`.

Iris developers and contributors. The Iris 3.6 documentation, January 2022. URL `https://plv.mpi-sws.org/iris/appendix-3.6.pdf`.

Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–282, 2011. doi: 10.1145/1926385.1926417.

Ralf Jung. Logical atomicity in Iris: the good, the bad, and the ugly. Iris Workshop, October 2019. URL `https://people.mpi-sws.org/~jung/iris/talk-iris2019.pdf`.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages (POPL)*, pages 637–650, January 2015. URL `http://plv.mpi-sws.org/iris/paper.pdf`.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018a. URL `https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf`.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018b. URL `https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf`.

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: Prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):45:1–45:32, January 2020. URL `https://plv.mpi-sws.org/prophecies/paper.pdf`.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 17:1–17:29, June 2017. URL `https://people.mpi-sws.org/~dreyer/papers/iris-weak/paper.pdf`.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Principles of Programming Languages (POPL)*, pages 175–189, January 2017. URL `https://sf.snu.ac.kr/publications/promising.pdf`.

Robert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*, January 2017. URL `http://cs.au.dk/~birke/papers/ipm-conf.pdf`.

Siddharth Krishna, Michael Emmi, Constantin Enea, and Dejan Jovanovic. Verifying visibility-based weak consistency. In *European Symposium on Programming (ESOP)*, volume 12075 of *Lecture Notes in Computer Science*, pages 280–307. Springer, April 2020. URL `https://raw.githubusercontent.com/michael-emmi/research-papers/master/conf-esop-KrishnaEEJ20.pdf`.

Ori Lahav and Viktor Vafeiadis. Owicki-Gries reasoning for weak memory models. In *International Colloquium on Automata, Languages, and Programming*, pages 311–323. Springer, 2015. doi: 10.1007/978-3-662-47666-6_25.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Programming Language Design and Implementation (PLDI)*, pages 618–632, June 2017. URL `https://plv.mpi-sws.org/scfix/paper.pdf`.

Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. URL `https://www.microsoft.com/en-us/research/uploads/prod/2016/12/How-to-Make-a-Multiprocessor-Computer-That-Correctly-Executes-Multiprocess-Programs.pdf`.

Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and efficient bounded FIFO queues. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 144–151. IEEE, 2013a. doi: 10.1109/SBAC-PAD.2013.8.

Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN symposium*

*on Principles and practice of parallel programming*, pages 69–80, 2013b. doi: 10.1145/2442516.2442524.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system: documentation and user's manual, September 2019. URL `http://caml.inria.fr/pub/docs/manual-ocaml/`.

Andreas Lochbihler. Java and the Java memory model – a unified, machine-checked formalisation. In *European Symposium on Programming (ESOP)*, volume 7211 of *Lecture Notes in Computer Science*, pages 497–517. Springer, March 2012. URL `https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/people/andreloc/lochbihler12esop.pdf`.

Gregory Malecha, Gordon Stewart, František Farka, Jasper Haag, and Yoichi Hirai. Developing with formal methods at BedRock Systems, Inc. *IEEE Security & Privacy*, 20(3):33–42, 2022. doi: 10.1109/MSEC.2022.3158196.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, pages 378–391, January 2005. URL `http://rsim.cs.uiuc.edu/Pubs/popl05.pdf`.

Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 45–54, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583976. doi: 10.1145/1504176.1504186. URL `https://doi.org/10.1145/1504176.1504186`.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 809–824, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523432. URL `https://ikemulder.nl/media/papers/pldi22-diaframe.pdf`.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. Coq proofs for Cosmo and examples. `https://gitlab.inria.fr/cambium/cosmo`, 2021.

Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 2007. URL `http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/concurrency.pdf`.

Peter W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019. URL `https://doi.org/10.1145/3211968`.

Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, may 1976. ISSN 0001-0782. doi: 10.1145/360051.360224. URL `https://doi.org/10.1145/360051.360224`.

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In *Principles of Programming Languages (POPL)*, pages 297–302, 2007. URL https://doi.org/10.1145/1190216.1190261.

William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

Azalea Raad, Marko Doko, Lovro Rozic, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *Proceedings of the ACM on Programming Languages*, 3(POPL): 68:1–68:31, 2019. URL https://spiral.imperial.ac.uk/bitstream/10044/1/75940/4/Libraries.pdf.

Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, 2013. URL https://doi.org/10.1007/978-3-642-32027-9.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002. URL http://www.cs.cmu.edu/~jcr/seplogic.pdf.

Erik Rigtorp. MPMCQueue, March 2021. https://github.com/rigtorp/MPMCQueue.

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 158–174, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454036. URL https://plv.mpi-sws.org/refinedc/paper.pdf.

Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A separation logic for fictional sequential consistency. In *European Symposium on Programming (ESOP)*, volume 9032 of *Lecture Notes in Computer Science*, pages 736–761. Springer, April 2015. URL https://www.cl.cam.ac.uk/~jp622/a_separation_logic_for_fictional_sequential_consistency.pdf.

Graeme Smith, Kirsten Winter, and Robert J Colvin. Linearizability on hardware weak memory models. *Formal Aspects of Computing*, pages 1–32, 2019. doi: 10.1007/s00165-019-00499-8.

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. Modular reasoning about separation of concurrent data structures. In *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 169–188. Springer, March 2013. URL http://cs.au.dk/~birke/papers/hocap-conf.pdf.

Amin Timany and Lars Birkedal. Reasoning about monotonicity in separation logic. In *Certified Programs and Proofs (CPP)*, pages 91–104, January 2021. URL https://iris-project.org/pdfs/2021-CPP-monotone-final.pdf.

Aaron Turon. Reagents: expressing and composing fine-grained concurrency. In *Programming Language Design and Implementation (PLDI)*, pages 157–168, June 2012. URL https://aturon.github.io/academic/reagents.pdf.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 691–707, October 2014. URL `http://plv.mpi-sws.org/gps/paper.pdf`.

Viktor Vafeiadis. Program verification under weak memory consistency using separation logic. In *Computer Aided Verification (CAV)*, volume 10426 of *Lecture Notes in Computer Science*, pages 30–46. Springer, July 2017. URL `https://people.mpi-sws.org/~viktor/papers/cav2017-invited.pdf`.

Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 867–884, October 2013. URL `https://people.mpi-sws.org/~viktor/papers/oopsla2013-rsl.pdf`.

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220, 2015. URL `https://plv.mpi-sws.org/c11comp/popl15.pdf`.

Simon Friis Vindum and Lars Birkedal. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*, pages 76–90, January 2021. URL `https://cs.au.dk/~birke/papers/2021-ms-queue-final.pdf`.

Simon Friis Vindum, Dan Frumin, and Lars Birkedal. Mechanized verification of a fine-grained concurrent queue from Facebook's Folly library. Submitted for publication, March 2021. URL `https://cs.au.dk/~birke/papers/mpmc-queue.pdf`.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. URL `http://www.cs.rice.edu/CS/PLT/Publications/Scheme/ic94-wf.ps.gz`.

Yannick Zakowski, David Cachera, Delphine Demange, and David Pichardie. Verified compilation of linearizable data structures: mechanizing rely guarantee for semantic refinement. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1881–1890, 2018. doi: 10.1145/3167132.3167333.