Reasoning about Monotonicity in Separation Logic

Amin Timany
timany@cs.au.dk
Aarhus University
Aarhus, Denmark

Lars Birkedal
birkedal@cs.au.dk
Aarhus University
Aarhus, Denmark

Abstract

Reasoning about monotonicity is of key importance in concurrent separation logics. For instance, one needs to reason about monotonicity to show that the value of a concurrent counter with an increment operation only grows over time. Modern concurrent separation logics, such as VST, FCSL, and Iris, are based on resource models defined using partial commutative monoids. For any partial commutative monoid, there is a canonical ordering relation, the so-called extension order, and in a sense the logics are designed to reason about monotonicity wrt. the extension ordering.

Thus a natural question is: given an arbitrary preorder, can we construct a partial commutative monoid, where the extension order captures the given preorder.

In this paper, we answer this question in the affirmative and show that there is a canonical construction, which given any preorder produces a partial commutative monoid for which the extension order, restricted to the elements of the preorder, is exactly the given preorder. We prove that our construction is a free construction in the category-theoretic sense.

We demonstrate, using examples, that the general construction is useful. We have formalized the construction and its properties in Coq. Moreover, we have integrated it in the Iris program logic framework and used that to formalize our examples.

CCS Concepts: • **Theory of computation** → **Program specifications**; **Program verification**; *Invariants*; *Hoare logic*; **Separation logic**; **Programming logic**.

Keywords: monotonicity, separation logic, partial commutative monoids, program verification

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439931

ACM Reference Format:
Amin Timany and Lars Birkedal. 2021. Reasoning about Monotonicity in Separation Logic. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3437992.3439931

1 Introduction

Shared memory concurrent programs are notoriously hard to reason about due to intricate interactions between threads. The concurrent separation logic methodology used in current state-of-the-art mechanizations such as Iris, VST, and FCSL [3, 10, 14] has proven to be successful in taming the complexity of shared memory concurrency reasoning. Arguably, the main reason for this success is *modular reasoning*. By modular reasoning we here mean that program modules, *i.e.*, threads, functions, *etc.*, are verified in isolation. A correctness proof of a compound program is then obtained by composing proofs of correctness of its individual modules. The methodology supports modular reasoning because each program module is verified under certain assumptions about the other modules. In many cases the assumptions on other modules are phrased as some kind of monotonicity with respect to some relation, *e.g.*, a module may assume that all other modules only change shared state in some monotone way. In other words, each program module gets to know that the computations performed on the shared memory by other modules always amounts to progress in the algorithm that is being carried out by the program, *i.e.*, no other thread introduces a regression. As an oversimplified and contrived, yet, still illustrative example consider the program in Figure 1. This program allocates a counter and forks a thread that repeatedly increments it. The main program then repeatedly checks that the counter's value does not decrease. It makes two consecutive reads and crashes if the most recent value is smaller than the value read before. Proving that the main program is safe and does not crash relies on the knowledge that the value of the counter is monotonically increasing. We do not include the exact code of the counter implementation in this paper; it is entirely standard and uses an atomic increment operation.

In separation logic the assumptions that a module makes about other modules are usually represented using *invariants* and *ghost state*. Invariants, or invariant enforcing mechanisms, *e.g.*, concurroids in Nanevski et al. [14], are used to

```
let c = mkCounter () in
let incrloop () = while(true){increment c} in
let checkmonotone () =
  let x = read c in
  let y = read c in
  if y < x then () () else ()
in
fork {incrloop ()};
while(true){checkmonotone ()}
```

Figure 1. A simple monotonic counter: this program should not crash, *i.e.*, the first branch of the conditional which applies the unit value, (), to itself is never executed.

express protocols on shared resources. Most modern separation logics [3, 10, 14] use partial commutative monoids (PCM's) to represent resources that model their ghost state. Each PCM is naturally equipped with a so-called extension order. As we will see, in cases where the monotonicity relation that we wish to reason about arises as the extension order of a PCM, *e.g.*, the monotone counter example above, reasoning about that relation is straightforward. The question then is whether there is a systematic way of finding "the right PCM" for reasoning with respect to a particular monotonicity relation. In this paper we answer this question by giving a general construction that given any preorder relation R constructs a PCM $Monotone(R)$ such that the extension order of $Monotone(R)$ suitably embeds the preorder R. We furthermore show that our construction of the monotone PCM is canonical, in the sense that it arises as a *free* construction in the category-theoretic sense.

We demonstrate the usefulness of our construction by giving three interesting examples. The first example is for references: we show that an arbitrary preorder relation (on any set) can be used to track the monotonic evolution of the contents of a reference. The second example shows how monotonicity with respect to a simple relation defined by a transition system can allow us to conclude that certain paths in a program's execution are unreachable. The third example concerns (aspects of) a more serious verification challenge. In order to verify a causally consistent replicated database one needs to reason about observations that each replica makes relative to the rest of the system. It is crucial to be able to conclude that if a replica observes an event, it must also have observed all events that it depends on. In our third example we consider a program that captures the essence of such a scenario and use the monotone construction to reason about observation of events.

All the results presented in this paper have been formalized in the Coq proof assistant; partly on top of the Iris program logic and partly on top of a category theory library for Coq. These formalizations are available at: https://github.com/amintimany/monotone.

Structure of the Paper. In Section 2 we present a basic separation logic that we use throughout the paper. We also specify the monotone counter example without giving the definition of some propositions that are based on ghost resources. We use these specs to prove the correctness of the program in Figure 1. In Section 3 we present PCM's and how they are used as ghost state in our separation logic. Afterwards, we see a few interesting examples of PCM's and introduce invariants in our separation logic. We use invariants and the PCM's we introduce in this section to define the propositions that we used in Section 2 for specifying the monotone counter and show that the monotone counter indeed satisfies the specs that we had ascribed it. In Section 4 we present our construction of the monotone PCM and prove the interesting properties that make it useful. We present three interesting and motivating example use cases of the monotone PCM in Section 5. Section 6 presents the category-theoretic arguments why our construction is canonical. In Section 7 we remark on some technical aspects of our Coq formalization of the results presented in this paper. We discuss related work in Section 8, and finish the paper with some concluding remarks in Section 9.

2 Separation Logic

In this section we present a separation logic and see a simple application of it to the code in Figure 1. The system that we consider here is very close to Iris and can be thought of as a simplified version of it. In particular, we gloss over the step-indexing features of Iris including the fact that in Iris resource algebras are more general than partial commutative monoids; they can be thought of as step-indexed partial commutative monoids.[1] We use this system to present our ideas, however, our ideas should be applicable to any PCM-based separation logic.

2.1 Programming Language

We use a simple call-by-value *untyped* lambda calculus with a unit value, which we write as (), sums (disjoint unions), products, Booleans, concurrency (a fork command), and higher-order references. We do not present this language formally in all details and use a standard ML-like syntax for writing programs. The following is an excerpt of expressions, values, and evaluation contexts.

$$\begin{aligned} e ::=\ & () \mid n \mid \mathtt{true} \mid \mathtt{false} \mid \ell \mid \mathtt{rec}\, f\, x = e \mid e\ e \mid (e, e) \mid \pi_1\, e \mid \\ & \mathtt{inj}_1\, e \mid \mathtt{match}\, e\, \mathtt{with}\, \mathtt{inj}_1\, x \Rightarrow e \mid \mathtt{inj}_2\, x \Rightarrow e\, \mathtt{end} \mid \\ & \mathtt{ref}\, e \mid !\, e \mid e \leftarrow e \mid \cdots \\ v ::=\ & () \mid n \mid \mathtt{true} \mid \mathtt{false} \mid \ell \mid \mathtt{rec}\, f\, x = e \mid (v, v) \mid \cdots \end{aligned}$$

[1] In Iris resources algebras are essentially step-indexed PCM's so as to allow encoding of impredicative invariants as special resources. Interested readers can find more details in Jung et al. [9, 10].

$$\begin{aligned}K ::= & [\,] \mid K\,e \mid (K, e) \mid (v, K) \mid \pi_1\,K \mid \texttt{inj}_1\,K \mid \\ & \texttt{match}\,e\,\texttt{with}\,\texttt{inj}_1\,x \Rightarrow e \mid \texttt{inj}_2\,x \Rightarrow e\,\texttt{end} \mid \\ & \texttt{ref}\,K \mid !K \mid K \leftarrow e \mid v \leftarrow K \mid \cdots\end{aligned}$$

Here $n \in \mathbb{Z}$ is a number, ℓ is a memory location, and $\texttt{rec}\,f\,x = e$ is a recursive function with name f, argument x, and body e. We write *Loc* for the set of all memory locations. Notice how the evaluation contexts reflect the call-by-value nature of the language, *e.g.*, in an application first the term in the function position is evaluated to a value and only then the argument is evaluated.

2.2 Higher-Order Concurrent Separation Logic

The propositions of separation logic are as follows:[2]

$$\begin{aligned}P ::= & \mathsf{True} \mid \mathsf{False} \mid P \to P \mid && \text{(higher-order logic)}\\ & P \wedge P \mid P \vee P \mid \forall x.\, P \mid \exists x.\, P \mid && \\ & P * P \mid P \mathrel{-\!\!*} P \mid \lceil \phi \rceil \mid && \text{(basic separation logic)}\\ & \ell \mapsto v \mid \{P\}\,e\,\{x.\, P\} && \text{(program logic)}\end{aligned}$$

Following Iris jargon, we refer to the universe of these propositions as *iProp*. Propositions includes ordinary (dependent) higher-order logic.

Separation Logic. The separating conjunction, $*$, is where the name separation logic is derived from. Intuitively, $P * Q$ holds for a resource if it can be split into two *disjoint* parts such that one satisfies P and one satisfies Q. The connective $\mathrel{-\!\!*}$, pronounced wand, or magic wand, is the separating implication. Intuitively, $P \mathrel{-\!\!*} Q$ holds for a resource a if for any resource b, disjoint from a, that satisfies P the combination of a and b satisfies Q. As we will see below partial commutative monoids are used to make the notions of "disjoint" and "combination" formal. The following are some of the proof rules for reasoning about separating conjunction and magic wands:

SEP-COMM
$$P * Q \dashv\vdash Q * P$$

SEP-ASSOC
$$P * (Q * R) \dashv\vdash (P * Q) * R$$

WAND-INTRO
$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

WAND-ELIM
$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

where $\vdash$ is the logical entailment relation and $\dashv\vdash$ is the logical equivalence relation.

The proposition $\lceil \phi \rceil$ asserts that the pure proposition ϕ holds. The proposition ϕ in $\lceil \phi \rceil$ is a proposition from the meta logic, *e.g.*, in the Coq formalization of separation logic this would be a Coq proposition. Whether a pure proposition holds or not does not depend on resources.

[2]Higher-order logic, or impredicative invariants are not strictly speaking necessary for our development. Nonetheless, we include them so that we can present our proofs in the style of Iris.

Separation Logic Inference Rules. So far we have used the entailment relation explicitly in writing proof rules. However, in addition to this style of proof rules, we sometimes write proof rules where antecedents and the consequent are all separation logic formulae. Such an inference rule should be understood as follows: the separating conjunction of the antecedents entails the consequent. Moreover, we sometimes present inference rules without antecedents, *i.e.* a single proposition P. This is to be understood as $\vdash P$.

Program Logic. The points-to proposition, $\ell \mapsto v$, asserts that the location ℓ in memory (heap) has value v. This proposition asserts exclusive ownership over location ℓ:

POINTS-TO-EXCLUSIVE
$$\ell \mapsto v * \ell \mapsto v' \vdash \mathsf{False}$$

Points-to propositions are not a primitive propositions of the system and are defined in terms of ghost resources, however, we do not present how points-to propositions are defined in this paper.

The proposition $\{P\}\,e\,\{x.\, Q\}$ is a Hoare triple where the result of computation can appear in the postcondition. This is represented using the binder x in the postcondition of the triple. Intuitively, if the Hoare triple $\{P\}\,e\,\{x.\, Q\}$ holds, then whenever the precondition P holds the program e is safe to execute, *i.e.*, it does not get stuck, and whenever it terminates with a value v the postcondition $Q[v/x]$ holds. The proof rules for reasoning about Hoare triples correspond very closely to the steps in the operational semantics of the programming language, *e.g.*, rules HOARE-ALLOC and HOARE-LOAD below, and some rules that allow manipulation of ghost state and invariants. Some of the interesting proof rules for Hoare triples are as follows:

HOARE-ALLOC
$$\{\mathsf{True}\}\,\texttt{ref}\,v\,\{x.\, \exists \ell.\, \lceil x = \ell \rceil * \ell \mapsto v\}$$

HOARE-LOAD
$$\{\ell \mapsto v\}\,!\,\ell\,\{x.\, \lceil x = v \rceil * \ell \mapsto v\}$$

HOARE-REC
$$\frac{\begin{array}{l}\{P * \{P\}\,(\texttt{rec}\,f\,x = e)\,v\,\{x.\, Q\}\}\\ \quad e[v/x][\texttt{rec}\,f\,x = e/f]\\ \{x.\, Q\}\end{array}}{\{P\}\,(\texttt{rec}\,f\,x = e)\,v\,\{x.\, Q\}}$$

HOARE-BIND
$$\frac{\{P\}\,e\,\{y.\, R\} \qquad \forall v.\, \{R[v/y]\}\,e\,\{x.\, Q\}}{\{P\}\,K[e]\,\{x.\, Q\}}$$

The rule HOARE-BIND states that in order to prove correctness of the program e under the evaluation context K, it suffices to prove that for any value v that satisfies the postcondition of e, $K[v]$ is correct.

Persistent Propositions. In our separation logic certain propositions assert ownership over exclusive resources while some other propositions only assert knowledge. We say the former kind of proposition is *ephemeral* while we call the latter kind of proposition *persistent*, *i.e.*, duplicable.[3] The quintessential examples of ephemeral and persistent propositions are points-to propositions and pure propositions, respectively. This is evidenced by the rule POINTS-TO-EXCLUSIVE for points-to propositions and the following rule for pure propositions:

$$\frac{\text{PURE-DUPLICABLE}}{\lceil \phi \rceil \dashv\vdash \lceil \phi \rceil * \lceil \phi \rceil}$$

As we will discuss later on, invariants and ownership of certain ghost resources are also persistent.

2.3 Monotone Counter: High-Level Specs

In this section we give a high level specification of the monotone counter and use it to verify the code in Figure 1. The specifications that we give here are very simple specifications that only allow us to reason about monotonicity of the counter. See Birkedal and Bizjak [4] for different ways of giving (much stronger) specifications to concurrent counters. In order to express the specifications for the monotone counter, we assume that we have two predicates (which will define later on in Section 3):

$$\begin{aligned} \mathsf{isCounter} &: \mathit{Loc} \to \mathit{Names} \to \mathit{iProp} \\ \mathsf{CounterAtLeast} &: \mathit{Names} \to \mathbb{N} \to \mathit{iProp} \end{aligned}$$

The predicate isCounter takes a location and a ghost name, while the predicate CounterAtLeast takes a ghost name and a natural number. Intuitively, the predicate $\mathsf{isCounter}(\ell, \gamma)$ states that ℓ is a counter and its value is tracked by the ghost state named γ. The predicate $\mathsf{CounterAtLeast}(\gamma, n)$ indicates that the value of the counter being tracked by ghost state γ is at least n. Both of these predicates are persistent:

$$\frac{\mathsf{isCounter}(\ell, \gamma)}{\mathsf{isCounter}(\ell, \gamma) * \mathsf{isCounter}(\ell, \gamma)}$$

$$\frac{\mathsf{CounterAtLeast}(\gamma, n)}{\mathsf{CounterAtLeast}(\gamma, n) * \mathsf{CounterAtLeast}(\gamma, n)}$$

The specifications for the counter are as follows:

$$\{\mathsf{True}\}\ \ \mathit{mkCounter}\ ()\ \ \{x.\ \exists \ell, \gamma.\ \lceil x = \ell \rceil * \mathsf{isCounter}(\ell, \gamma) * \mathsf{CounterAtLeast}(\gamma, 0)\} \quad \text{(mkCounter-spec)}$$

$$\{\mathsf{isCounter}(\ell, \gamma)\}\ \mathit{increment}\ \ell\ \{x.\ \lceil x = () \rceil\} \quad \text{(counter-increment-spec)}$$

$$\{\mathsf{isCounter}(\ell, \gamma) * \mathsf{CounterAtLeast}(\gamma, n)\}\ \ \mathit{read}\ \ell\ \ \{x.\ \exists m.\ \lceil x = m \land n \le m \rceil * \mathsf{CounterAtLeast}(\gamma, m)\} \quad \text{(counter-read-spec)}$$

The function *mkCounter* returns a new counter whose value is at least 0. Incrementing the counter does not have any observable effect. Reading the counter, on the other hand, gives us two important pieces of information: (1) the value of the counter is at least as big as the returned value, and (2) the value returned is greater than any previously observed value that we pick.

Given these specs we can prove that the program in Figure 1 does not crash. We first show the following two Hoare-triples:

$$\{\mathsf{isCounter}(\ell, \gamma)\}\ \mathit{incrloop}\ \ell\ \{x.\ \mathsf{True}\} \quad \text{(incrloop-spec)}$$

$$\{\mathsf{isCounter}(\ell, \gamma) * \mathsf{CounterAtLeast}(\gamma, 0)\}\ \ \mathit{checkmonotone}\ \ell\ \ \{x.\ \mathsf{True}\} \quad \text{(checkmonotone-spec)}$$

To see that spec (incrloop-spec) holds, note that the proposition $\mathsf{isCounter}(\ell, \gamma)$ is an invariant for the loop in this function. That is, since $\mathsf{isCounter}(\ell, \gamma)$ is persistent, we can duplicate it and use one copy to establish the precondition for (counter-increment-spec). We use the other copy to establish the loop invariant at the end of execution of the *increment* function. For proving (checkmonotone-spec), we use $\mathsf{isCounter}(\ell, \gamma)$ and $\mathsf{CounterAtLeast}(\gamma, 0)$ as the precondition of the (counter-read-spec) specs. Notice that we retain $\mathsf{isCounter}(\ell, \gamma)$ and $\mathsf{CounterAtLeast}(\gamma, 0)$ as they are persistent. Moreover, we obtain $\mathsf{CounterAtLeast}(\gamma, x)$ for the read value, x. We use the specs (counter-read-spec) together with $\mathsf{isCounter}(\ell, \gamma)$ and $\mathsf{CounterAtLeast}(\gamma, x)$ as the precondition, this time for reading y. As a result, we obtain $\mathsf{CounterAtLeast}(\gamma, y)$ together with a proof that $x \le y$. The latter suffices to show that the subsequent conditional will necessarily take the else branch and hence does not crash.

To complete the proof it suffices to show that both the forked thread and the subsequent while loop do not crash. The former is what we established in previous paragraph. The latter follows from the fact that

$$\mathsf{isCounter}(\ell, \gamma) * \mathsf{CounterAtLeast}(\gamma, 0)$$

is an invariant for the loop. This follows in an argument similar to the proof of (incrloop-spec) above.

[3]In practice Iris defines persistent propositions by endowing resource algebras which a special operation that removes all ephemeral parts of the resource, and a persistently modality defined in terms of this operation. In this paper, we conflate duplicability and persistence as the difference is orthogonal to what we are presenting in this paper. For a theoretical discussion on the difference between the two concepts see Bizjak and Birkedal [5].

3 Resources and Partial Commutative Monoids

In this section we discuss partial commutative monoid (PCM) based resources in our separation logic. We will thereafter see some useful examples of partial commutative monoids and discuss invariants. We will then use these concepts to prove correctness of the specifications that we gave to *mkCounter*, *read*, and *increment*, in section 2.

3.1 Partial Commutative Monoids

Following Iris's practice we represent PCM's as ordinary commutative monoids with a validity predicate.

Definition 3.1 (PCM)**.** A PCM is an algebraic structure $(M, \equiv_M, \cdot_M, \varepsilon_M, \checkmark_M)$ where M is the carrier set, $\equiv_M$ is the an equivalence relation on M, $\cdot_M : M \times M \to M$ is a binary operation, $\checkmark M$ is the validity predicate, and the following conditions hold:

$$a \cdot_M b \equiv_M b \cdot_M a \qquad \text{(commutativity)}$$

$$a \cdot_M (b \cdot_M c) \equiv_M (a \cdot_M b) \cdot_M c \qquad \text{(associativity)}$$

$$a \cdot_M \varepsilon_M \equiv_M a \qquad \text{(unit element)}$$

$$a \equiv_M a' \wedge b \equiv_M b' \Rightarrow a \cdot_M b \equiv_M a' \cdot_M b' \qquad \text{(respect equiv)}$$

$$\checkmark_M \varepsilon_M \qquad \text{(unit validity)}$$

$$a \equiv_M b \wedge \checkmark_M a \Rightarrow \checkmark_M b \qquad \text{(validity respects)}$$

$$\checkmark_M (a \cdot_M b) \Rightarrow \checkmark_M a \qquad \text{(validity-op)}$$

Each PCM M comes equipped with a notion of equality $\equiv_M$. This is useful for defining PCM's without having to use quotients. We will discuss this in more details later on. We drop the subscript M in $\equiv_M$, $\cdot_M$, ε_M, and $\checkmark_M$ whenever it is clear from the context what M is.

Resource Ownership. There are two forms of propositions in our separation logic related to resources.

$$P ::= \cdots \mid \boxed{a}^{\gamma} \mid \Rrightarrow P \mid \cdots \qquad \text{(resources)}$$

The proposition $\boxed{a}^{\gamma}$ asserts that we *own* the resource a. Here, a is an element of some PCM. There can be multiple instances of the same PCM used as different resources. We use names, $\gamma \in \mathit{Names}$, to distinguish these different instances. If necessary, we write $\boxed{a : M}^{\gamma}$ to clarify that the owned element a belongs to the PCM M. The main idea in embedding ownership of elements of a PCM in the logic is that the PCM operation models the separating conjunction. Note how the separating conjunction is commutative (rule SEP-COMM) and associative (rule SEP-ASSOC). Moreover, in some cases separating conjunction of two propositions is contradictory while those propositions are not contradictory on their own, *e.g.*, in rule POINTS-TO-EXCLUSIVE. This is captured by the partiality of PCM's. The following facts about the ownership proposition reflect these concepts:

OWN-OP
$$\boxed{a \cdot_M b}^{\gamma} \dashv\vdash \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$$

OWN-VALID
$$\boxed{a : M}^{\gamma} \vdash \lceil \checkmark_M a \rceil$$

These rules allow us to split and combine resources and to exclude ownership of certain resources, *i.e.*, those that are invalid. For instance, these two rules are used to derive the rule POINTS-TO-EXCLUSIVE.

As the rules OWN-OP and OWN-VALID suggest, the combination of all resources owned in the system under an instance γ is always valid. This is indeed an important property for the soundness of our system in the presence of these rules. In order to maintain this property, updating of ghost resources is restricted to *frame preserving updates* [10]. We say that there is a frame preserving update from a to a b, written $a \rightsquigarrow_M b$, if

$$\forall a_f.\ \checkmark(a \cdot a_f) \Rightarrow \checkmark(b \cdot a_f) \qquad \text{(frame-preserving update)}$$

The definition (frame-preserving update) above states that ownership of a can be updated to ownership of b if for any other resource (frame) a_f that is compatible with a, a_f is also compatible with b. In such a case, updating from a to b may never invalidates the frame.

Update Modality. The update modality $\Rrightarrow$ enables allocation and updating of resources. We write $P \Rrightarrow\!\!* Q$ as a shorthand for $P -\!\!* \Rrightarrow Q$.

The relevant proof rules for the update modality are the following:

OWN-ALLOC
$$\frac{\checkmark_M a}{\vdash \Rrightarrow \exists \gamma.\ \boxed{a}^{\gamma}}$$

OWN-UPDATE
$$\frac{a \rightsquigarrow_M b}{\boxed{a}^{\gamma} \vdash \Rrightarrow \boxed{b}^{\gamma}}$$

OWN-RESPECTS
$$\frac{a \equiv_M b}{\boxed{a}^{\gamma} \dashv\vdash \boxed{b}^{\gamma}}$$

HOARE-UPD
$$\frac{\{P\}\, e\, \{y.\ \Rrightarrow Q\}}{\{P\}\, e\, \{y.\ Q\}}$$

UPD-HOARE
$$\frac{\Rrightarrow \{P\}\, e\, \{y.\ Q\}}{\{P\}\, e\, \{y.\ Q\}}$$

The rule OWN-ALLOC states that any valid element of any PCM M can be allocated. Note that allocating an instance of a resource creates a fresh name γ. The rule OWN-UPDATE allows us to update a $a \in M$ to an element b if there is a frame-preserving update from a to b in M. The rules HOARE-UPD and UPD-HOARE allow us to allocate and update resources throughout the proofs of correctness of programs.

3.2 The Order of a Partial Commutative Monoid

Each PCM M induces an *extension order* $\preceq_M$ defined by:

$$a \preceq_M b \triangleq \exists c.\ b \equiv a \cdot c \qquad \text{(extension order)}$$

It is easy to see that the extension order $\preceq_M$ is always a preorder relation, *i.e.*, it is reflexive and transitive. When clear from the context we might drop the subscript M in $\preceq_M$. The extension order plays a crucial role in our PCM-based separation logic. Indeed, in a sense, the whole logic is *monotonic* with respect to this preorder. Ownership of $\boxed{a : M}^{\gamma}$ allows us to conclude that the collection c of all the resources in the system under the name γ is in the extension order relation with a, *i.e.*, $a \preceq_M c$. In fact, the key to many proofs in PCM-based separation logics it to find the right (combination of) PCM's so that the extension-order reasoning provides the necessary information for the proofs; for an example, see the proof of the monotone counter below.

We can now formally state the main question that this paper answers as follows: given a preorder relation $R : \mathcal{P}(\mathcal{A} \times \mathcal{A})$, can we construct a PCM $Monotone(R)$ and a function $f : \mathcal{A} \to Monotone(R)$ such that $f(x) \preceq_{Monotone(R)} f(y)$ if and only if $R(x, y)$? We answer this question in Section 4.

3.3 Some Useful PCM's

A very simple example of a PCM is the PCM of natural numbers together with 0 as the unit element and maximum as the operation, which we write as $\mathbb{N}_{\max}$. For this PCM we take the equivalence relation to be the equality relation on natural numbers. This PCM is a total PCM, *i.e.*, the validity predicate holds for all elements. It is easy to check that all the axioms of a PCM defined earlier are satisfied by this construction, *e.g.*, max is commutative and associative, *etc.* Note that as max is an idempotent operation, ownership of the elements of $\mathbb{N}_{\max}$ is persistent.

All elements of the $\mathbb{N}_{\max}$ are valid. As a result, there is a frame preserving update from any element to any other element. In other words, owning $\boxed{n}^{\gamma}$, we can update our resources to own $\boxed{m}^{\gamma}$ for any arbitrary m. Hence, this PCM is not very useful for tracking the state of a program. As we will shortly discuss, though, this PCM is very useful when combined with the Authoritative PCM, which we will now describe.

Note that the extension order in $\mathbb{N}_{\max}$ coincides with the $\leq$ order on natural numbers.

Authoritative PCM. The authoritative PCM [11] over a PCM M, $\textsc{Auth}(M)$, is a PCM that allows us to give an instance-wide upper bound on the ownership of elements of M. Intuitively, elements of the PCM are of the form $\circ\, a$, meaning that we own a, or $\bullet\, a$ meaning that a is the upper bound on all resources owned in the ownership instance. We call elements of the form $\circ\, a$ *fragment parts* and the elements of the form $\bullet\, a$ *full parts.*

The authoritative PCM is defined as follows:

$$\textsc{Auth}(M) \triangleq \{\circ\, a | a \in M\} \cup \{\bullet\, a | a \in M\} \cup \{\bullet\!\circ(a, b) | a, b \in M\} \cup \left\{\bot_{\textsc{Auth}(M)}\right\}$$

$$a \equiv_{\textsc{Auth}(M)} b \triangleq \begin{cases} a' \equiv_M b' & \text{if } a = \circ\, a' \text{ and } b = \circ\, b' \\ a' \equiv_M b' & \text{if } a = \bullet\, a' \text{ and } b = \bullet\, b' \\ a'_1 \equiv_M b'_1 \wedge a'_2 \equiv_M b'_2 & \text{if } a = \bullet\!\circ(a'_1, a'_2) \text{ and } \\ & \quad b = \bullet\!\circ(b'_1, b'_2) \\ \text{True} & \text{if } a = \bot_{\textsc{Auth}(M)} \text{ and } \\ & \quad b = \bot_{\textsc{Auth}(M)} \\ \text{False} & \text{otherwise} \end{cases}$$

$$a \cdot_{\textsc{Auth}(M)} b \triangleq \begin{cases} \circ\,(a' \cdot_M b') & \text{if } a = \circ\, a' \text{ and } b = \circ\, b' \\ \bullet\!\circ(a', b') & \text{if } a = \bullet\, a' \text{ and } b = \circ\, b' \\ \bullet\!\circ(b', a') & \text{if } a = \circ\, a' \text{ and } b = \bullet\, b' \\ \bullet\!\circ(a_1, a'_2 \cdot_M b') & \text{if } a = \bullet\!\circ(a'_1, a'_2) \text{ and } \\ & \quad b = \circ\, b' \\ \bullet\!\circ(b_1, a' \cdot_M b'_2) & \text{if } a = \circ\, a' \text{ and } \\ & \quad b = \bullet\!\circ(b'_1, b'_2) \\ \bot_{\textsc{Auth}(M)} & \text{otherwise} \end{cases}$$

$$\varepsilon_{\textsc{Auth}(M)} \triangleq \circ\, \varepsilon_M$$

$$\checkmark_{\textsc{Auth}(M)}\, a \triangleq \begin{cases} \checkmark_M a & \text{if } a = \circ\, a' \\ \checkmark_M a & \text{if } a = \bullet\, a' \\ \checkmark_M a_1 \wedge a_2 \preceq_M a_1 & \text{if } a = \bullet\!\circ(a'_1, a'_2) \\ \text{False} & \text{otherwise} \end{cases}$$

Note how this definition, apart from the full parts and fragments, also includes elements that are a combination of both. The PCM operation is defined in such a way that the operation is $\bot_{\textsc{Auth}(M)}$ (which is an invalid element) as soon as more than one operand includes a full part. Therefore, there can always be at most a unique full part owned in any instance.

AUTH-FULL-EXCLUSIVE
$$\boxed{\bullet\, a}^{\gamma} * \boxed{\bullet\, b}^{\gamma} \vdash \text{False}$$

The operation on the fragments, on the other hand, is defined pointwise. Hence, we have that $\boxed{\circ\, a}^{\gamma}$ is persistent whenever $\boxed{a}^{\gamma}$ is persistent. Importantly, if an element $\bullet\!\circ(a, b)$ is valid, then $b \leq a$ holds. This fact, together with the rules own-op and own-valid, allows us to derive the following:

AUTH-INCLUDED
$$\boxed{\bullet\, a : \textsc{Auth}(M)}^{\gamma} * \boxed{\circ\, b : \textsc{Auth}(M)}^{\gamma} \vdash \ulcorner b \preceq_M a \urcorner$$

The frame preserving updates of the authoritative PCM $\textsc{Auth}(M)$ depend on M. Therefore, we will not discuss it in general and only look at specific examples.

Authoritative PCM over $\mathbb{N}_{\max}$. As expected, the PCM $\textsc{Auth}(\mathbb{N}_{\max})$ inherits most of its properties from M and the authoritative PCM. For instance, the authoritative part is exclusive while the fragment part is persistent. Moreover, by auth-included we have that if we own $\boxed{\bullet\, n}^{\gamma}$ and $\boxed{\circ\, m}^{\gamma}$, we can conclude $m \leq n$. The more interesting aspect of the $\textsc{Auth}(\mathbb{N}_{\max})$ is the following frame-preserving update auth-nat-max-fp-upd and its consequence auth-nat-max-own-upd:

AUTH-NAT-MAX-FP-UPD
$$\frac{n \leq m \qquad k \leq m}{\bullet\, n \rightsquigarrow_M \bullet\!\circ(m, k)}$$

AUTH-NAT-MAX-OWN-UPD
$$\frac{n \leq m \qquad k \leq m}{\boxed{\bullet\, n}^{\gamma} \vdash \Rrightarrow \boxed{\bullet\, m}^{\gamma} * \boxed{\circ\, k}^{\gamma}}$$

Note that the number tracked in the full part $\bullet\, n$ can never decrease through a frame-preserving update because the element $\circ\, n$ is a possible frame for $\bullet\, n$.

3.4 Invariants

Before we give more details about the monotone counter, we present invariants, which we will use to verify the monotone counter:

$$P ::= \cdots \mid \boxed{P} \qquad\qquad \text{(invariants)}$$

The proposition $\boxed{P}$ asserts that P holds at all times, *i.e.*, P is an invariant.[4]

$$\frac{\text{INV-ALLOC}}{P \vdash \Rrightarrow \boxed{P}}$$

$$\text{HOARE-ATOMIC}\quad \frac{\{P * R\}\, e\, \{y.\ Q * R\} \qquad \boxed{R} \qquad e \text{ is physically atomic}}{\{P\}\, e\, \{y.\ Q\}}$$

The rule INV-ALLOC states that if P holds, we can assert it as an invariant, *i.e.*, we can assert that from now on P should always hold. An invariant proposition $\boxed{P}$ only asserts the knowledge that P should hold invariantly. Hence, invariants are persistent, *i.e.*, duplicable:

$$\text{INV-DUPLICABLE}\qquad \boxed{P} \dashv\vdash \boxed{P} * \boxed{P}$$

3.5 Verifying the Monotone Counter Impl.

We now define the predicates isCounter and CounterAtLeast and briefly discuss how they are used to derive the specs: mkCounter-spec, counter-increment-spec, counter-read-spec. The predicates are defined as follows:

$$\mathsf{isCounter}(\ell, \gamma) \triangleq \boxed{\exists n.\ \ell \mapsto n * \boxed{\bullet\, n : \textsc{Auth}(\mathbb{N}_{\max})}^{\gamma}}$$

$$\mathsf{CounterAtLeast}(\gamma, n) \triangleq \boxed{\circ\, n : \textsc{Auth}(\mathbb{N}_{\max})}^{\gamma}$$

Both of these predicates are persistent. In the following we briefly discuss the proofs that the specs for the counter operations hold.

Proof of mkCounter-spec. Creating the counter simply consists of allocating a reference ℓ with value 0 which gives us $\ell \mapsto 0$. We use the rule OWN-ALLOC to allocate a fresh instance γ of the ghost state $\boxed{\bullet\!\circ (0, 0)}^{\gamma}$, which is a valid element of $\textsc{Auth}(\mathbb{N}_{\max})$. Hence, we obtain $\boxed{\bullet\, 0}^{\gamma}$ and $\boxed{\circ\, 0}^{\gamma}$. We use the latter to establish $\mathsf{CounterAtLeast}(\gamma, n)$, and the former together with $\ell \mapsto 0$ and the rule INV-ALLOC to establish $\mathsf{isCounter}(\ell, \gamma)$.

Proof of counter-increment-spec. At the atomic point of incrementing the counter we use the HOARE-ATOMIC to access the invariant. Thus we get that there is a number n for which we have $\ell \mapsto n$ and $\boxed{\bullet\, n}^{\gamma}$. The atomic increment operation updates $\ell \mapsto n + 1$. In order to reestablish the invariant we use the rule AUTH-NAT-MAX-OWN-UPD to obtain $\boxed{\bullet\, n + 1}^{\gamma}$ and $\boxed{\circ\, n + 1}^{\gamma}$. We use the former to reestablish the invariant and simply ignore the latter.[5]

Proof of counter-read-spec. The precondition gives us $\mathsf{CounterAtLeast}(\gamma, n)$. When we perform the atomic read operation we access the invariant using the HOARE-ATOMIC. At this point we get that there is a number k for which we have $\ell \mapsto k$ and $\boxed{\bullet\, k}^{\gamma}$. We use the rule AUTH-INCLUDED together with $\boxed{\bullet\, k}^{\gamma}$ and $\mathsf{CounterAtLeast}(\gamma, n)$ to establish that $n \leq k$. We then use the rule AUTH-NAT-MAX-OWN-UPD with $\boxed{\bullet\, k}^{\gamma}$ to get $\boxed{\bullet\, k}^{\gamma}$ and $\boxed{\circ\, k}^{\gamma}$. without actually increasing the value of the ghost state. We then reestablish the invariant and use the freshly created $\boxed{\circ\, k}^{\gamma}$ to establish the postcondition.

Discussion. Observe the high-level picture of the specification and verification of the monotone counter: we needed to reason about monotonicity with respect to the $\leq$ relation on natural numbers and for this purpose we picked the $\mathbb{N}_{\max}$ PCM whose extension order corresponds to the $\leq$ relation.

4 The Monotone PCM

In this section we present our general construction of the PCM $\mathit{Monotone}(R)$ for a given preorder $R \subseteq \mathcal{A} \times \mathcal{A}$ with an injection function $\mathit{principal}_R : \mathcal{A} \to \mathit{Monotone}(R)$ such that:

$$\mathit{principal}_R(x) \preceq_{\mathit{Monotone}(R)} \mathit{principal}_R(y) \Leftrightarrow R(x, y) \qquad \text{(monotone-order)}$$

The first observation we make is that if the preorder relation R is join-semilattice with a bottom (least) element then such a construction becomes trivial. A join-semilattice with a bottom element is a structure $(\mathcal{A}, R, \Cup, \bot)$ where $R \subseteq \mathcal{A} \times \mathcal{A}$ is a partial order relation, $\Cup$ is the join operation (least upper bound with respect to R), and $\bot$ is the least element of $\mathcal{A}$ with respect to R. Any join-semilattice is a PCM with the trivial equality relation as its equivalence relation where every element is valid: take $\Cup$ as the PCM operation and $\bot$ as the unit element. Indeed, we can easily check that monotone-order holds by taking the $\mathit{principal}_R$ function to be the identity function:

$$(\exists z.\ y = x \Cup z) \Leftrightarrow R(x, y)$$

Our general construction is based on this observation in that it essentially constructs a join-semilattice with a bottom element out of R.

Definition 4.1 (Monotone PCM)**.** Given a preorder $R \subseteq \mathcal{A} \times \mathcal{A}$, the monotone PCM, $\mathit{Monotone}(R)$, is defined as the PCM $(\mathcal{P}_{\mathrm{fin}}(\mathcal{A}), \equiv_{\mathit{Monotone}(R)}, \cup, \emptyset, \checkmark_{\mathit{Monotone}(R)})$ where

$$A \equiv_{\mathit{Monotone}(R)} B \triangleq \forall x \in \mathcal{A}.\ \mathit{Below}(x, A) \Leftrightarrow \mathit{Below}(x, B) \qquad \text{(monotone-equiv)}$$

[4] In Iris, Invariants have a names which we omit in this paper as they only clutter the presentation. These names are used for ensuring that invariants are not accessed multiple times in a nested fashion which is generally unsound. For the same reason, in Iris the update modality, weakest preconditions, and Hoare triples are also indexed with masks (sets of invariant names) to track which invariants are available to access.

[5] We can do this because our separation logic is *affine* (as opposed to linear). However, we stress that our development here does not rely on the logic being affine; a small variation of the proof could have been used in case the logic had been linear.

$$Below(x, A) \triangleq (\exists y.\ y \in A \wedge R(x, y))$$
$$\checkmark_{Monotone(R)} A \triangleq \mathsf{True}$$

Here, $\mathcal{P}_{\text{fin}}(\mathcal{A})$ is the set of all finite subsets of $\mathcal{A}$. We define the $principal_R$ function as follows:

$$principal_R(x) \triangleq \{x\}$$

It is easy to check that the relation $\equiv_{Monotone(R)}$ is an equivalence relation, and that the definition above satisfies all the axioms of PCM in Definition 3.1. The only non-trivial axiom is respect equiv which, however, immediately follows from *Below*-union below.

Lemma 4.2. *The predicate Below satisfies the following properties:*

$$Below(x, A \cup B) \Leftrightarrow Below(x, A) \vee Below(x, B) \qquad (Below\text{-union})$$

$$Below(x, principal_R(y)) \Leftrightarrow R(x, y) \qquad (Below\text{-principal})$$

The monotone PCM as defined above satisfies monotone-order.

Theorem 4.3. *The monotone PCM reflects the order of the preorder relation R, i.e., monotone-order holds.*

Proof. We prove the two directions separately.

(⇒). We know that there is a set $A \subseteq \mathcal{A}$ for which $principal_R(y) \equiv principal_R(x) \cup A$. We need to show that $R(x, y)$ holds. By *Below*-principal above it suffices to show that $Below(x, principal_R(y))$. However, since $principal_R(y) \equiv principal_R(x) \cup A$ holds, this is equivalent to showing that $Below(x, principal_R(x) \cup A)$. This, however, holds trivially by *Below*-union and *Below*-principal.

(⇐). We know that $R(x, y)$ and we need to show that $principal_R(x) \preceq principal_R(y)$. We prove that $principal_R(y) \equiv principal_R(x) \cup principal_R(y)$ holds, or equivalently, by *Below*-union, $Below(z, principal_R(y)) \Leftrightarrow Below(z, principal_R(x)) \vee Below(z, principal_R(y))$. The forward direction holds trivially. For backward direction we only need to show that $Below(z, principal_R(x))$ implies $Below(z, principal_R(y))$. This follows by *Below*-principal and transitivity of R, because we know $R(x, y)$. □

Corollary 4.4. *The following inference rule holds for ownership of* Auth*(Monotone(R)) elements.*

AUTH-MONOTONE-ORDER
$$\boxed{\bullet\ principal_R(y)}^{\gamma} * \boxed{\circ\ principal_R(x)}^{\gamma} \vdash \ulcorner R(x, y) \urcorner$$

Similarly to $\mathbb{N}_{\max}$, $Monotone(R)$ is a total PCM and hence there is a frame-preserving update from any element to any element in this PCM. The following theorem gives a useful frame-preserving update for Auth$(Monotone(R))$.

Lemma 4.5. *The following frame-preserving update and update rule hold for the monotone PCM:*

AUTH-MONOTONE-FP-UPD
$$\frac{R(x, y) \qquad R(z, y)}{\bullet\ principal_R(x) \rightsquigarrow_M \bullet\!\circ (principal_R(y), principal_R(z))}$$

AUTH-MONOTONE-OWN-UPD
$$\frac{R(x, y) \qquad R(z, y)}{\boxed{\bullet\ principal_R(x)}^{\gamma} \vdash \rrightarrow \boxed{\bullet\ principal_R(y)}^{\gamma} * \boxed{\circ\ principal_R(z)}^{\gamma}}$$

5 Example Use Cases

In this section we present three illustrative examples of how the monotone PCM can be used in reasoning about programs.

5.1 General Monotonic References

Given a set $\mathcal{A}$ and a preorder relation $R \subseteq \mathcal{A} \times \mathcal{A}$, and a partial function $\mathcal{V}$ from values to $\mathcal{A}$, we construct a mechanism for tracking the value of a reference and updating it monotonically according to R. In particular, this construction provides two propositions, a monotone points-to proposition $\ell \overset{mon}{\mapsto}_{\gamma} v$, and a proposition $AtLeast(\gamma, v)$ which indicates that the monotone reference whose state is tracked by the ghost state γ has a value that is at least v. Notice that the *AtLeast* predicate uses the value of the reference and not an element in $\mathcal{A}$ that corresponds to it. This construction is designed in such a way that any points-to predicate with a suitable value can be turned into a monotone reference and back.

We define the following predicates to reason about the value stored in the monotone reference:

$$AtLeast(\gamma, v) \triangleq \exists x.\ \ulcorner \mathcal{V}(v) = x \urcorner * \boxed{\circ\ principal_R(x)}^{\gamma}$$
$$\ell \overset{mon}{\mapsto}_{\gamma} v \triangleq \ell \mapsto v * Exact(\gamma, v)$$
$$Exact(\gamma, v) \triangleq \exists x.\ \ulcorner \mathcal{V}(v) = x \urcorner * \boxed{\bullet\ principal_R(x)}^{\gamma}$$

The monotone reference construction satisfies the following rules:

MON-REF-ALLOC
$$\frac{\ulcorner \mathcal{V}(v) = x \urcorner \qquad \ell \mapsto v}{\rrightarrow \exists \gamma.\ \ell \overset{mon}{\mapsto}_{\gamma} v}$$

MON-REF-SNAPSHOT
$$\frac{\ell \overset{mon}{\mapsto}_{\gamma} v}{\rrightarrow \ell \overset{mon}{\mapsto}_{\gamma} v * AtLeast(\gamma, v)}$$

MON-REF-RECALL
$$\frac{\ell \overset{mon}{\mapsto}_{\gamma} v \qquad AtLeast(\gamma, w)}{\ulcorner R(\mathcal{V}(w), \mathcal{V}(v)) \urcorner}$$

MON-REF-CANCEL
$$\frac{\ell \overset{mon}{\mapsto}_{\gamma} v}{\ell \mapsto v * \forall w.\ \ulcorner R(\mathcal{V}(v), \mathcal{V}(w)) \urcorner * \ell \mapsto w \Rrightarrow\!\!* \ \ell \overset{mon}{\mapsto}_{\gamma} w}$$

MON-REF-LOAD
$$\left\{\ell \overset{mon}{\mapsto}_{\gamma} v\right\} \, !\, \ell \left\{x.\ \ulcorner x = v \urcorner * \ell \overset{mon}{\mapsto}_{\gamma} v\right\}$$

MON-REF-STORE
$$\left\{\ell \overset{mon}{\mapsto}_{\gamma} v * \ulcorner R(\mathcal{V}(v), \mathcal{V}(w)) \urcorner\right\} \ell \leftarrow w \left\{x.\ \ell \overset{mon}{\mapsto}_{\gamma} w\right\}$$

The arguments for why these rules are sound are very close to the verification of the monotone counter we presented earlier. For instance, to validate MON-REF-ALLOC we simply allocate $\bullet\circ(principal_R(x), principal_R(x))$ to obtain a name γ and establish $Exact(\gamma, v)$ and $AtLeast(\gamma, v)$. Note how the rule MON-REF-CANCEL cancels the monotone reference but gives us a way back: if at some point the value w of ℓ is greater than v (according to R), then we can reestablish the monotone reference.

5.2 Excluding Unreachable Execution Paths

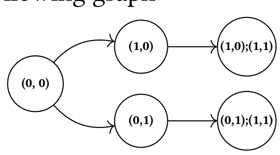
Consider the following program, *excl_path*:

$$\begin{array}{l}\texttt{let}\; x = \texttt{ref}\; 0 \;\texttt{in} \\ \texttt{let}\; y = \texttt{ref}\; 0 \;\texttt{in} \\ (x \leftarrow 1; !\, y) \mid\mid (y \leftarrow 1; !\, x)\end{array}$$

Here, the parallel composition operation $e_1 \mid\mid e_2$ runs e_1 and e_2 concurrently and returns (v_1, v_2) where v_i is the resulting value of e_i. This program may return $(0, 1)$, $(1, 0)$, or $(1, 1)$, but *never* $(0, 0)$. We prove this by showing that the following specs hold:

$$\{\mathsf{True}\}\; excl\_path\; \{x.\ \lceil x \in \{(1, 0), (0, 1), (1, 1)\}\rceil\} \qquad \text{(excl-path-spec)}$$

To prove this property we show that the state of the values stored in x and y evolves monotonically according to a preorder relation *Reach*. We define *Reach* as the reachability relation of the following graph:



The main idea of the proof is to use the graph above to track the order of writes. If the left thread returns 0, then it must have read y when the write to x had already happened but the write to y had not, *i.e.*, the program must be in state $(1, 0)$ at the time of reading y. Similarly, if the right thread returns 0, it must be the case that it has read x when program was in the state $(0, 1)$. However, it is impossible for the execution to have taken such a path as it does not exists in the graph above.

We define two predicates for tracking the state of the graph:

$$ExactST(\gamma, s) \triangleq \boxed{\bullet\, principal(s)}^{\gamma}$$
$$ObsST(\gamma, s) \triangleq \boxed{\circ\, principal(s)}^{\gamma}$$

Using the rule AUTH-INCLUDED we can prove the following rule:

STATE-REACHABLE
$$ExactST(\gamma, s) * ObsST(\gamma, s') \vdash \lceil Reach(s', s)\rceil$$

Before presenting more details, we briefly present fractional points-to propositions. Fractional permissions are a well-known technique frequently used in separation logic. Given a points-to proposition, we can split it into fractions which can be later combined to obtain back the whole. In fact, the points-to proposition $\ell \mapsto v$ is a fractional points-to proposition where the fraction is 1, *i.e.*, $\ell \mapsto v = \ell \mapsto_1 v$.

POINTS-TO-SPLIT
$$\frac{\ell \mapsto_q v}{\ell \mapsto_{q/2} v * \ell \mapsto_{q/2} v}$$

POINTS-TO-AGREE
$$\frac{\ell \mapsto_q v \qquad \ell \mapsto_{q'} w}{\lceil v = w \rceil}$$

POINTS-TO-MERGE
$$\frac{\ell \mapsto_{q/2} v \qquad \ell \mapsto_{q/2} v}{\ell \mapsto_q v}$$

The intuitive idea is that fractional points-to propositions can be used to share ownership of a location in a safe way: having a fraction allows one to read the location but not to write to that location. In other words, the HOARE-LOAD rule also works with a fractional (with any fraction) points-to.

To prove (excl-path-spec) above we establish the following invariant, after the two references have been allocated as ℓ_x and ℓ_y, by letting the existentially quantified s be the state $(0, 0)$:

$$inv_{ST} \triangleq \boxed{\exists s.\ \ell_x \mapsto_{1/2} val_x(s) * \ell_y \mapsto_{1/2} val_y(s) * ExactST(\gamma, s)}$$

Given a state s, the $val_x(s)$ returns the value of ℓ_x in that state (i.e., if $s = (0, 0)$, then it returns 0, if $s = (1, 0)$ then it returns 1, if $s = (0, 1); (1, 1)$, then it returns 1, etc.). Likewise $val_y(s)$ returns the value of ℓ_y in state s.

After this, we prove that each thread satisfies the following specs:

$$\begin{array}{l}\{\ell_x \mapsto_{1/2} 0 * inv_{ST}\} \\ \quad \ell_x \leftarrow 1; !\, \ell_y \\ \left\{z.\ \begin{array}{l}\left(\begin{array}{l}\lceil z = 0\rceil * \\ ObsST(\gamma, (1, 0))\end{array}\right) \vee \\ \left(\begin{array}{l}\lceil z = 1\rceil * \\ \left(\begin{array}{l}ObsST(\gamma, (1, 0); (1, 1)) \vee \\ ObsST(\gamma, (0, 1); (1, 1))\end{array}\right)\end{array}\right)\end{array}\right\}\end{array}
\qquad
\begin{array}{l}\{\ell_y \mapsto_{1/2} 0 * inv_{ST}\} \\ \quad \ell_y \leftarrow 1; !\, \ell_x \\ \left\{z.\ \begin{array}{l}\left(\begin{array}{l}\lceil z = 0\rceil * \\ ObsST(\gamma, (0, 1))\end{array}\right) \vee \\ \left(\begin{array}{l}\lceil z = 1\rceil * \\ \left(\begin{array}{l}ObsST(\gamma, (1, 0); (1, 1)) \vee \\ ObsST(\gamma, (0, 1); (1, 1))\end{array}\right)\end{array}\right)\end{array}\right\}\end{array}$$

Each thread combines the fractional points-to proposition it is given together with its counterpart in the invariant to perform its write. The fractional permissions in the invariant suffice for reading. If the program were to return $(0, 0)$ then it must be the case that both threads return 0 and, according to the specs above, that is only possible if we have both $ObsST(\gamma, (1, 0))$ and $ObsST(\gamma, (0, 1))$. But combining these two propositions with the fact (obtained from the invariant) that there is some state s for which we have $ExactST(\gamma, s)$, we would then get $Reach((1, 0), s)$ and $Reach((0, 1), s)$ by rule STATE-REACHABLE. However, by the definition of the graph there is no state s that satisfies these criteria. Hence, the postcondition of excl-path-spec holds.

5.3 The Causal Closure Relation

This example is inspired by causally consistent distributed key-value stores, *e.g.*, Ahamad et al. [1]. In such a system a database is replicated and replicas use messages over the network to keep one another up to date. However, network messages are not guaranteed to arrive in the exact order of the events that caused those messages to be sent. Therefore, these algorithms [1] use some mechanism, *e.g.*, logical vector clocks, to reflect the causal order between events. A replica only observes (registers) an event if it has already observed all the events that it causally depends on.

Since on each replica multiple threads interact with the database, no single thread can have exclusive authority over observed events — in an actual implementation a separate thread (or threads) receives messages over the network and applies them. In order to verify programs in such a setting we introduce two predicates: $\mathit{Observed}_\bullet$ and $\mathit{Observed}_\circ$ to track the authoritative and the fragment part of the information on the set of observed events. The idea is that an invariant would use $\mathit{Observed}_\bullet$ to express, through resources, what events have been observed, and threads would use ownership of $\mathit{Observed}_\circ$ to express a lower bound on the set of observed events. However, it is not sufficient to have that the set of observed events tracked by the fragment is a subset of the set of all observed events. Indeed, as we will see through a simple example, one needs to know that the fragment is a *causally-closed subset* of the set of all observed events. That is, if we have an event in the fragment, then any other event in the set of all observed events that this event depends on should also be in the fragment.

In order to demonstrate the ideas above we fix a set of events *Events*. We consider two particular events ev_0 and ev_1 in *Events* for which we know $ev_0.\mathtt{time} < ev_1.\mathtt{time}$, *i.e.*, the logical time of ev_0 is smaller than that of ev_1, or, ev_1 causally depends on ev_0. Crucially, ev_0 is the only event that ev_1 depends on, and ev_0 itself has no dependencies. We verify the following program, *i.e.*, we show it does not crash:

```
let db = empty_DB () in
let dbp = ref db in
let wait_for = λx. while(¬ (is_observed dbp x)){} in
fork {simulate_receive dbp ev0} ;
fork {simulate_receive dbp ev1} ;
wait_for ev1;
if is_observed dbp ev0 then () else () ()
```

Since our programming language does not support networking, we simply *fork* functions that simulate receiving events. Notice that the order at which the events are received depends on scheduling of threads and thus is non-deterministic. The function *simulate_receive* simply adds events to the database *dbp* which is simply a collection of all received events, whether they are observed or not. For simplicity we have used a reference, *dbp*, to an algebraic list, *db*, to model the database. The function *is_observed* checks if the given event is observed, *i.e.*, that the event itself *as well as all its dependencies* are in the database. This program waits for observation of the event ev_1 and then asserts that the event ev_0 is also observed, *i.e.*, it crashes if this is not the case.

We define the predicates $\mathit{Observed}_\bullet$ and $\mathit{Observed}_\circ$ as follows:

$$\mathit{Observed}_\bullet(\gamma, E) \triangleq \boxed{\bullet\, \mathit{principal}_{\mathbf{CCS}} E}^{\gamma}$$
$$\mathit{Observed}_\circ(\gamma, E) \triangleq \boxed{\circ\, \mathit{principal}_{\mathbf{CCS}} E}^{\gamma}$$

where the causally-closed-subset relation, **CCS** is defined as follows:

$$\mathbf{CCS}(E, F) \triangleq E \subseteq F \wedge \forall ev, ev' \in F.\ ev.\mathit{time} \leq ev'.\mathit{time} \wedge ev' \in E \Rightarrow ev \in E$$

It is easy to see that this relation is both reflexive and transitive, and hence a preorder. We use the following invariant to verify the program above:

$$\mathit{isDB}(\gamma, \mathit{dbp}) \triangleq \boxed{\begin{array}{l}\exists v, E.\ \mathit{dbp} \mapsto v * \mathit{DBcontents}(v, E) * \\ \quad \lceil E \subseteq \{ev_0, ev_1\}\rceil * \\ \quad \mathit{Observed}_\bullet(\gamma, \mathit{ObservedSubset}(E))\end{array}}$$

Here, the predicate $\mathit{DBcontents}(v, E)$ asserts that the contents of the database v is exactly the set of events E. Initially, we establish the invariant above by picking E to be the empty set and v to be the newly created empty database. Furthermore, we instantiate resources so that next to initializing the invariant we also obtain $\mathit{Observed}_\circ(\gamma, \emptyset)$.

Given this invariant we can verify adding ev_0 and ev_1 in the two forks — we might have to update the resource $\mathit{Observed}_\bullet$ as adding events can grow the set of observed events in the database, $\mathit{ObservedSubset}(E)$.

We ascribe the following specifications to the *is_observed* function:

$$\begin{array}{l}\{\mathit{isDB}(\gamma, \mathit{dbp}) * \mathit{Observed}_\circ(\gamma, E)\} \\ \quad \mathit{is\_observed}\ \mathit{dbp}\ \mathit{ev} \\ \left\{\begin{array}{l} x.\ \exists b \in \{\mathtt{true}, \mathtt{false}\}, F.\ \lceil x = b\rceil * \lceil E \subseteq F\rceil * \\ \quad \mathit{Observed}_\circ(\gamma, F) * \\ \quad ((\lceil b = \mathtt{true} \wedge ev \in F\rceil) \vee (\lceil b = \mathtt{false} \wedge ev \notin F\rceil)) \end{array}\right\}\end{array} \qquad (\mathit{is\_observed}\text{-spec})$$

Given *is_observed*-spec above we can verify the call to the *wait_for* function with the following specs:

$$\begin{array}{l}\{\mathit{isDB}(\gamma, \mathit{dbp}) * \mathit{Observed}_\circ(\gamma, \emptyset)\} \\ \quad \mathit{wait\_for}\ \mathit{dbp}\ ev_1 \\ \{x.\ \exists F.\ \mathit{Observed}_\circ(\gamma, F) * ev_1 \in F\}\end{array}$$

That is, after *wait_for dbp* ev_1 we know $\mathit{Observed}_\circ(\gamma, F)$ for some set F for which $ev_1 \in F$. On the other hand, since F must be a *causally-closed subset of* the set of events in the invariant we also know that $ev_0 \in F$. At this point, we can verify the last line of the code. We appeal to the *is_observed-spec* above and use $Observed_\circ(\gamma, F)$ obtained from the postcondition of *wait_for dbp* ev_1 to satisfy the precondition of the call to *is_observed dbp* ev_0. This time we obtain a set $Observed_\circ(\gamma, G)$ for some set G such that $F \subseteq G$. Moreover, we are guaranteed that if this call returns `false`, $ev_0 \notin G$ which is a contradiction. Hence, the crashing `else` branch is not reachable.

The approach presented here for reasoning about causal closure is exactly how Gondelman et al. [7] reason about casual closure in Iris using the monotone resource algebra.

6 Canonicity of Construction

In this section we show that the construction of the monotone PCM that we presented is canonical in the sense that it arises as a free functor in the category-theoretic sense, *i.e.*, it is the left adjoint to a forgetful functor.

Before we continue, let us make precise what we mean by the category **PCM** of PCM's. The objects of this category are PCM's and its morphisms are PCM-morphisms defined as follows:

Definition 6.1. Let $(M, \equiv_M, \cdot_M, \varepsilon_M, \checkmark_M)$ and $(M', \equiv_{M'}, \cdot_{M'}, \varepsilon_{M'}, \checkmark_{M'})$ be two PCM's. A PCM-morphism from M to M' is a function $f : M \to M'$ such that the following hold:

$$a \equiv_M b \Rightarrow f(a) \equiv_{M'} f(b) \qquad \text{(respect-equiv)}$$

$$f(a \cdot_M b) \equiv_{M'} f(a) \cdot_{M'} f(b) \qquad \text{(respec-op)}$$

$$f(\varepsilon_M) \equiv_{M'} \varepsilon_{M'} \qquad \text{(respec-unit)}$$

$$\checkmark_M a \Rightarrow \checkmark_{M'} f(a) \qquad \text{(respec-validity)}$$

Perhaps the first thought that comes to mind is that there should be an adjunction between the functors *Monotone* : **PreOrder** $\to$ **PCM** and *Extension* : **PCM** $\to$ **PreOrder**. However, this is not the case. Both the monotone construction and the extension order (as the forgetful functor) form functors between the categories. However, they do not constitute an adjunction. To see why, consider the co-unit of the adjunction, which should be a natural transformation from $Monotone \circ Extension$ to the identity functor $\mathbb{1}_{\mathbf{PCM}}$. This natural transformation should produce PCM morphisms of the form $Monotone(\preceq_M) \to M$ for any PCM M. In other words, given a finite set of elements of a PCM M, the natural transformation should produce an element of M. Intuitively, such an element should be the join (least upper bound) of the elements of the given finite set with respect to the extension order of the M. But such an element may simply not exist.

Instead we consider a more refined situation: First recall that the monotone construction is inspired by join-semilattices with a bottom element. Hence, we consider the category **JSL** whose objects are join-semilattices with a bottom element and whose morphisms are monotone functions that preserve both the join operation and the bottom element. This category can be seen as a full subcategory of the category **PCM** of PCM's because the inclusion functor **JSL** $\hookrightarrow$ **PCM** is fully-faithful. And indeed, the monotone construction that we have presented constructs join-semilattices with a bottom element, with a small caveat: the notion of join is only well-defined for partial order relations, not for arbitrary preorders. The problem is that in a preorder relation a set may have multiple least upper bounds. However, when applied to a partial order, the monotone construction that we have presented does in fact construct a join-semilattice with a bottom element. Moreover, we can verify that the monotone construction is indeed the left adjoint to the forgetful functor that maps join-semilattices with bottom elements to their underlying partial orders. Furthermore, we note the well-known fact that the category of partial orders and monotone functions, **Poset**, is a reflective subcategory of the category of preorders and monotone functions, **PreOrder**. The diagram in Fig. 2 summarizes these results.

$$\begin{tikzcd}
\mathbf{Poset} \arrow[rr, bend left, "Monotone"] \arrow[rr, phantom, "\perp"] \arrow[d, hook, bend left, "\text{fully-faithful inclusion}"] & & \mathbf{JSL} \arrow[ll, bend left, "Forgetful"] \arrow[d, hook, "\text{fully-faithful inclusion}"] \\
\mathbf{PreOrder} \arrow[u, bend left, "\text{anitsymmetric closure}"] \arrow[u, phantom, "\dashv"] & & \mathbf{PCM} \arrow[ll, "Extension"]
\end{tikzcd}$$

Figure 2. The category-theoretic results establishing that the monotone construction forms a free functor.

The unit of the adjunction between the two categories **Poset** and **JSL**, of type $\forall (\mathcal{A}, R) : \mathbf{Poset}.\ \mathcal{A} \to Monotone(R)$, sends an element of the given poset to the singleton set containing that element (as an element of the monotone construction). The co-unit of the adjunction, of type $\forall (\mathcal{A}, R, \Cup, \bot) : \mathbf{JSL}.\ Monotone(R) \to \mathcal{A}$, sends a finite set of elements of $\mathcal{A}$ to the join of this set.

7 Coq Formalization

All the results presented in this paper are machine checked by the Coq proof assistant. In particular, the monotone construction and all the examples presented in Section 5 are formalized on top of the Iris program logic and the Iris [10, 11] proof mode [12, 13]. Moreover, all the category-theoretic results presented in Figure 2 are formalized on top of the category theory library of Timany and Jacobs [16]. This category theory library provides all the basic concepts that we need: categories, functors, adjunctions, *etc.*. Below, we discuss some technical aspects of these formalizations.

Axioms in Coq Formalizations. The formalization on top of Iris does not make use of any Coq axioms. The formalization of our categorical constructions, however, do make use of axioms, as the category theory library that we use for our formalization does. In particular, the category theory library makes extensive use of function extensionality and proof irrelevance throughout the entire library. It also uses the axiom of choice and propositional extensionality in order to construct quotient types which we use as explained below.

7.1 Formalization on top of Iris

Iris uses resource algebras for modeling resources. Resource algebras are very close to PCM's with one main difference: Resource algebras are step-indexed PCM's in the sense that the equivalence relations and the validity predicates of resource algebras are indexed by natural numbers. Moreover, these relations are expected to be downwards closed, *i.e.*, if they hold for n they should hold for any $m \le n$. In our formalization of the monotone construction on top of Iris the equivalence relation and the validity relations are constant with respect to the step-index and hence trivially downwards closed. Another minor difference is that resource algebras in Iris are equipped with a *core*, a partial function which determines the duplicable part of each element of the resource algebra. The notion of persistence in Iris is defined using the *core* function. In practice, in any total PCM, *e.g.* the monotone construction, the *core* function can be taken to be the identity function, *i.e.*, the duplicable part of any element is the element itself. This way, ownership of all the elements are persistent just as we discussed in our construction of the monotone PCM and our examples. In the Coq formalization of the monotone construction we use lists to represent finite sets of elements.

All the examples presented in Section 5 are implemented on top of *HeapLang*, the programming language shipped with Iris. *HeapLang* is an untyped lambda-calculus with first-class concurrency primitives *i.e.*, a fork command, an atomic compare-and-set (CAS) operation, and an atomic fetch-and-add (FAA) operation (on integer references).

The formalization of the code and verification of all the examples presented in this paper are almost exactly as explained above. For instance, each of the rules for general monotonic references is formalized as a Coq lemma, with a statement that very closely resembles what we have presented in the paper. The only slight deviation is due to the fact that the partial function $\mathcal{V}$ in Coq is represented as a function to an optional type and hence some of the statements are cluttered with side conditions in order to express what is presented in the paper.

One detail that we did not explicate in the causal closure example is what we hid in the definition of the *empty_DB* function. As we discussed in the example, for simplicity we use algebraic lists to represent the "database". However, the database *dbp* is accessed and updated concurrently. These concurrent accesses should somehow be synchronized. In our implementation we opted to use the atomic CAS operation in a CAS-loop for synchronization purposes; a standard technique in fine-grained concurrent separation logic. Alternatively, one could have used a lock for synchronization. Since the CAS operation in *HeapLang* only works on references to base values or other references, the implementation of the *empty_DB* function in fact returns a reference to a list and *dbp* is then a reference to a reference to a list on which we can perform the CAS operation.

7.2 Formalization of Canonicity

In our Coq formalization we construct all the categories, functors and adjunctions presented in Figure 2. In this formalization we define the monotone construction as a construction that from a partially ordered set constructs a join-semilattice with a bottom element, just as in the figure. The category theory library that we work with does not support setoids, *i.e.*, user-defined equivalence relations, for objects and morphisms. It uses the definition of equality from the standard library of Coq. Hence, we use the classical theory of quotients used by this library for the monotone construction. The category theory library uses the axiom of choice (the axiom `ConstructiveIndefiniteDescription_on` in Coq's standard library) together with functional and propositional extensionality axioms to construct quotients; this construction of quotients is the only place where we use classical axioms in our formalization.

We define the monotone construction using lists to represent finite sets of elements. We show that the relation monotone-equiv is an equivalence relation which allows us to use the quotient construction provided by the category theory library. We also use this quotient construction for constructing the antisymmetric closure of preorders.

8 Related Work

The only work directly related to our work is the work of Ahman et al. [2], which also supports a form of reasoning about monotonicity with respect to a general preorder relation. Ahman et al. [2] essentially provide a variant of $\mathbf{F}^{\star}$'s state monad called MST where the put operation of the monad is restricted so that the state may only evolve according to the given preorder relation. This allows the MST monad also to provide a *witness* operation, which gives a witness that some property p holds, as long as p is stable under the preorder of MST. The MST monad also has a *recall* operation that proves that the current state satisfies a property that was witnessed in the past. The MST construction is useful for reasoning about programs written in a monadic style in a higher-order dependently typed language such as $\mathbf{F}^{\star}$ which can express the MST monad. In contrast, our construction is useful for reasoning in a (concurrent) separation logic about programs written in whatever untyped programming language the separation logic is developed for, *e.g.*, Iris's *HeapLang*. A weakness of the approach of Ahman et al. [2] is that it restricts the evolution of the state to the given preorder. For instance, in their section "Discussion: Temporarily Escaping the Preorder" Ahman et al. [2] discuss that in order to escape the preorder they need to change the type of the state and preorder relation. In contrast, our construction is more flexible. For instance, our general monotonic references can be temporarily canceled and reestablished using the rule MON-REF-CANCEL.

FCSL [14] uses so-called concurroids, in addition to PCM's, to encode protocols on shared resources. A concurroid is equipped with a state transition system (STS) and the protocol is then only allowed to evolve according to that STS. Hence, concurroids can be used to enforce monotonicity with respecct to an STS. This is, for instance, how Delbianco et al. [6] reason about monotonicity of the history of events in a system. Our construction of the monotone resource algebra suggests that it might be possible to simplify FCSL's concurroids so that they are expressed entirely in terms of PCM's.

Iris includes a so-called STS resource algebra, which encodes an STS whose states are associated with a collection of tokens [11]. The STS resource algebra is defined in such a way that owning certain tokens allows one to conclude that certain states have not yet been reached. We have very recently observed that, given a preorder R, one may define an instance of the STS resource algebra, with R as step relation of the STS and *without any tokens*, which can be used to encode most of the functionalities of the resource algebra $\text{AUTH}(\mathit{Monotone}(R))$ presented in this paper. The STS resource algebra formalized in the Coq formalization of Iris is almost never used in practise because it is too difficult to use in practice in Coq formalizations. This difficulty is mostly due to the fact that in virtually all operations on this resource algebra one needs to reason about tokens and how they are exchanged between different components of the resources algebra. In contrast, our monotone resource algebra construction has proved to be very useable in practise, also in connection with the authoritative resource algebra.

Pilkiewicz and Pottier [15] and Gordon et al. [8] restrict individual references with an arbitrary preorder in the types of their respective programming languages. Our work differs from these works in that we do not tie preorders to individual references. We embed the preorder relation into the ghost state which can freely be tied to the state of the program, directly or indirectly.

9 Conclusion

We presented a general construction $\mathit{Monotone}(R)$ to reason about monotonicity, with respect to an arbitrary preorder relation R, in concurrent separation logics. We established its utility by presenting three illustrative examples and showed its canonicity by establishing that our construction is a free functor in the category-theoretic sense.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions.

Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO) during parts of this project. This research was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

References

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241

[2] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.* 2, POPL (2018), 65:1–65:30. https://doi.org/10.1145/3158153

[3] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press, USA.

[4] Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. (2017). http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf

[5] Ales Bizjak and Lars Birkedal. 2018. On Models of Higher-Order Separation Logic. In *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018 (Electronic Notes in Theoretical Computer Science, Vol. 341)*, Sam Staton (Ed.). Elsevier, 57–78. https://doi.org/10.1016/j.entcs.2018.03.016

[6] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time (Artifact). *Dagstuhl Artifacts Series* 3, 2 (2017), 4:1–4:4. https://doi.org/10.4230/DARTS.3.2.4

[7] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic. *Proc. ACM Program. Lang.* POPL (2021). To appear.

[8] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2013. Rely-guarantee references for refinement types over aliased mutable data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 73–84. https://doi.org/10.1145/2491956.2462160

[9] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. https://doi.org/10.1145/2951913.2951943

[10] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[11] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[12] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[13] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. http://dl.acm.org/citation.cfm?id=3009855

[14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[15] Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, Stephanie Weirich and Derek Dreyer (Eds.). ACM, 73–86. https://doi.org/10.1145/1929553.1929565

[16] Amin Timany and Bart Jacobs. 2016. Category Theory in Coq 8.5. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal (LIPIcs, Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:18. https://doi.org/10.4230/LIPIcs.FSCD.2016.30