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.

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

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 \[3,8,12\] 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, \textit{i.e.}, threads, functions, \textit{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, \textit{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, \textit{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 reads the counter, checks that the value read before is smaller than the value read after. Proving that the main program is safe and does not crash relies on the knowledge that the value of the counter is monotonically increasing.

In separation logic the assumptions that a module makes about other modules is usually represented using invariants

\[
\text{let } c = \text{mkCounter} () \text{in} \\
\text{let } \text{incloop} () = \text{while}(\text{true})\{\text{increment } c\} \text{in} \\
\text{let } \text{checkmonotone} () = \\
\text{let } x = \text{read } c \text{in} \\
\text{let } y = \text{read } c \text{in} \\
\text{if } y < x \text{ then } () \text{ else } () \\
\text{in} \\
\text{fork } \{\text{incloop} ()\}; \\
\text{while}(\text{true})\{\text{checkmonotone} ()\}
\]

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference ’17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-xxxx-xxxx-xxxx-x/YY/MM . . . $15.00
https://doi.org/10.1145/mnnnnn.mnnnnnn
and ghost state. Invariants, or invariant enforcing mechanisms, e.g., concorrids in Nanverki et al. [12], are used to express protocols on shared resources. Most modern separation logics [3, 8, 12] 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 monotonocity relation. In this paper we answer this question by giving a general construction that given any preorder relation \( R \) constructs a PCM \( \text{Monotone}(R) \) such that the extension order of \( \text{Monotone}(R) \) 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.

Structure of the Paper. In Section 2 we present a basic separation logic that we use throughout the paper. In that section 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. 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.

\[
e ::= () | n | \mathbf{true} | \mathbf{false} | x | \text{rec} \ f \ x = e \ | e \ | e, e | \pi_1 \ e | \\
\text{inj}_1 \ e | \text{match} \ e \ \text{with} \ \text{inj}_1 \ x \Rightarrow e | \text{inj}_2 \ x \Rightarrow e \ | e \end{equation}
\]

\[
v ::= () | n | \mathbf{true} | \mathbf{false} | x | \text{rec} \ f \ x = e \ | (v, v) | \cdots \]

\[
K ::= [] | K \ e | (K, e) | (v, K) | \pi_1 K | \text{inj}_1 K | \\
\text{match} e \ \text{with} \ \text{inj}_1 x \Rightarrow e | \text{inj}_2 x \Rightarrow e \ | e \end{equation}
\]

Here \( n \in \mathbb{Z} \) is a number, \( l \) is a memory location, and \( \text{rec} \ f \ x = e \) is a recursive function with name \( f \), argument \( x \), and body \( e \). We write \( \text{Loc} \) 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.

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. [7, 8].
2.2 Higher-Order Concurrent Separation Logic

The propositions of separation logic are as follows:\(^2\)

\[ P ::= \text{True} \mid \text{False} \mid P \rightarrow P \] (higher-order logic)

\[ P \land P \mid P \lor P \mid \forall x. \ P \mid \exists x. \ P \]

\[ P \land P \mid P \rightarrow P \mid [\phi] \mid \text{basic separation logic} \]

\[ \ell \vdash v \mid \{P\} \ e \{x. \ P\} \] (program logic)

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

**Separation Logic.** The separating conjunction, *, is where the name separation logic is derived from. Intuitively, \(P \land Q\) holds for a resource if it can be split into two disjoint parts such that one satisfies \(P\) and one satisfies \(Q\). The connective \(\rightarrow\), pronounced wand, or magic wand, is the separating implication. Intuitively, \(P \rightarrow Q\) 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:

\[
\begin{align*}
\text{SEP-COMM} & : \quad P \land Q \vdash Q \land P \\
\text{SEP-ASSOC} & : \quad P \land (Q \land R) \vdash (P \land Q) \land R \\
\text{WAND-INTRO} & : \quad P \land Q \vdash R \\
\text{WAND-ELIM} & : \quad P \land Q \vdash R \\
\text{FUSE} & : \quad P \lor Q \vdash R
\end{align*}
\]

where \(\vdash\) is the logical entailment relation and \(\leftrightarrow\) is the logical equivalence relation.

The proposition \([\phi]\) asserts that the pure proposition \(\phi\) holds. The proposition \(\phi\) in \([\phi]\) 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.

**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 separate logic formulæ. 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, \(\ell \vdash v\), asserts that the location \(\ell\) in memory (heap) has value \(v\). This proposition asserts exclusive ownership over location \(\ell\):

\[ \ell \vdash v * \ell \vdash \ell' + \text{False} \]

Points-to propositions is 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 \text{HOARE-ALLOC} and \text{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:

\[
\begin{align*}
\text{HOARE-ALLOC} & : \quad \{\text{True}\} \ \text{ref} \ v \{x. \ \exists \ell. \ [x = \ell] * \ell \vdash v\} \\
\text{HOARE-LOAD} & : \quad \{\ell \vdash v\} ! \ell \{x. \ [x = v] * \ell \vdash v\} \\
\text{HOARE-REC} & : \quad \{P \land \{\text{rec} \ f x = e \} \ v \{x. \ Q\}\} \\
& \quad e[v/x][\text{rec} \ f x = e/f] \\
& \quad \{x. \ Q\} \\
\text{HOARE-BIND} & : \quad \{P\} \ K[e] \{x. \ Q\} \\
\end{align*}
\]

The rule \text{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 \textit{ephemeral} while we call the latter kind of proposition \textit{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 \text{POINTS-TO-EXCLUSIVE}.

\(^2\)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.

\(^3\)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].
for points-to propositions and the following rule for pure propositions:

\[
\text{pure-duplicable} \quad \vdash \phi \Rightarrow \phi \Rightarrow \phi
\]

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 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):

- \(\text{isCounter} : \text{Loc} \to \text{Names} \to \text{iProp}\)
- \(\text{CounterAtLeast} : \text{Names} \to \mathbb{N} \to \text{iProp}\)

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

\[
\begin{align*}
\text{isCounter}(\ell, \gamma) &\Rightarrow \text{isCounter}(\ell, \gamma) \\
\text{CounterAtLeast}(\gamma, n) &\Rightarrow \text{CounterAtLeast}(\gamma, n)
\end{align*}
\]

The specifications for the counter are as follows:

- \(\{\text{True}\}\)
- \(\{\text{x. \exists t, y. } [x = t] \Rightarrow \text{isCounter}(\ell, \gamma) \Rightarrow \text{CounterAtLeast}(\gamma, 0)\}\)
- \(\{\text{isCounter}(\ell, \gamma)\} \text{ increment } \ell \{x. \ [x = (\)]\}\)
- \(\{\text{isCounter}(\ell, \gamma) \Rightarrow \text{CounterAtLeast}(\gamma, n)\}\)
- \(\{\text{x. \exists m. } [x = m \land n \leq m] \Rightarrow \text{CounterAtLeast}(\gamma, m)\}\)

The function \(\text{mkCounter}\) 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:

\[
\begin{align*}
&\{\text{isCounter}(\ell, \gamma)\} \text{ incrloop } \ell \{x. \text{ True}\} \quad \text{(incrloop-spec)} \\
&\{\text{isCounter}(\ell, \gamma) \Rightarrow \text{CounterAtLeast}(\gamma, 0)\} \\
&\{x. \text{ True}\} \\
&\quad \text{(checkmonotone-spec)}
\end{align*}
\]

To see that spec (incrloop-spec) holds, note that \(\text{isCounter}(\ell, \gamma)\) is an invariant for the loop in this function. That is, since \(\text{isCounter}(\ell, \gamma)\) is persistent, we can duplicate it and give a copy as 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) above, we use \(\text{isCounter}(\ell, \gamma)\) and \(\text{CounterAtLeast}(\gamma, 0)\) as the precondition of the (counter-read-spec) specs. Notice that we retain \(\text{isCounter}(\ell, \gamma)\) and \(\text{CounterAtLeast}(\gamma, 0)\) as they are persistent. Moreover, we obtain \(\text{CounterAtLeast}(\gamma, x)\) for the read value, \(x\). We use the specs (counter-read-spec) with \(\text{isCounter}(\ell, \gamma)\) and \(\text{CounterAtLeast}(\gamma, x)\) as the precondition, this time for reading \(y\). As a result, we obtain \(\text{CounterAtLeast}(\gamma, y)\) together with a proof that \(x \leq y\). 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

\[
\text{isCounter}(\ell, \gamma) \Rightarrow \text{CounterAtLeast}(\gamma, 0)
\]

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

### 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 \(\text{mkCounter}\), \(\text{read}\), and \(\text{increment}\), 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 \((M, \equiv_M, \cdot_M, \epsilon_M, \gamma_M)\) 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, \( \sqrt{M} \) is the validity predicate, and the following conditions hold:

\[
a \cdot M b \equiv_M b \cdot M a \quad \text{(commutativity)}
\]

\[
a \cdot M (b \cdot M c) \equiv_M (a \cdot M b) \cdot M c \quad \text{(associativity)}
\]

\[
a \cdot M \epsilon M \equiv_M a \quad \text{(unit element)}
\]

\[
a \equiv_M a' \land b \equiv_M b' \Rightarrow \sqrt{M} \equiv_M a' \cdot M b' \quad \text{(respect equiv)}
\]

\[
\sqrt{M} \epsilon M \quad \text{(unit validity)}
\]

\[
a \equiv_M b \land \sqrt{M} a \Rightarrow \sqrt{M} b \quad \text{(validity respects)}
\]

\[
\sqrt{M} (a \cdot M b) \Rightarrow \sqrt{M} a \quad \text{(validity-op)}
\]

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 \( \equiv_M, \cdot_M, \epsilon_M \), and \( \sqrt{M} \) 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.

\[
P ::= \ldots | \frac{\equiv}{P} | \Rightarrow P | \ldots \quad \text{(resources)}
\]

The proposition \( \equiv \) 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, \( \gamma \in Names \), to distinguish these different instances. If necessary, we write \( \equiv^{M} \) 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 \( \text{SEP-COMM} \)) and associative (rule \( \text{SEP-ASSOC} \)). Moreover, in some cases separating conjunction of two propositions is contradictory while those propositions are not contradictory on their own, e.g., in rule \( \text{POINTS-TO-EXCLUSIVE} \). This is captured by the partiality of PCM’s. The following facts about the ownership proposition reflect these concepts:

\[
\sqrt{M} a \equiv \frac{\equiv}{\equiv} \quad \text{OWN-OP}
\]

\[
\equiv^{M} a \equiv \frac{\equiv}{\equiv} + \frac{\equiv}{\equiv} \quad \text{OWN-VALID}
\]

\[
\Rightarrow \equiv^{M} a \quad \text{[\( \sqrt{M} a \)]}
\]

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 \( \text{POINTS-TO-EXCLUSIVE} \).

As the rules \( \text{OWN-OP} \) and \( \text{OWN-VALID} \) suggest, all resources of any instance \( \gamma \) owned in the system combined are 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 update\( s[8] \). We say that there is a frame preserving update from \( a \) to \( a' \), written \( a \dashv\vdash_M b \), if

\[
\forall a_f. \sqrt{a \cdot a_f} \Rightarrow \sqrt{(a \cdot a_f)} \quad \text{(frame-preserving update)}
\]

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 \( \Rightarrow \) enables allocation and updating of resources. We write \( P \equiv Q \) as a shorthand for \( P \Rightarrow \Rightarrow Q \).

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

\[\begin{align*}
\text{OWN-ALLOC} & : \quad \sqrt{M} a \quad a \Rightarrow_M b \quad a \equiv_M b \\
\text{OWN-UPDATE} & : \quad \Rightarrow \equiv \quad a \Rightarrow_M b \\
\text{OWN-RESPECTS} & : \quad \equiv \Rightarrow \equiv
\end{align*}\]

\[\text{HOARE-UPD} \quad \{P\} e \{y, \gamma\} \quad \{P\} e \{y, Q\} \]

\[\text{UPD-HOARE} \quad \{P\} e \{y, Q\} \quad \{P\} e \{y, Q\} \]

The rule \( \text{OWN-ALLOC} \) states that any valid element of any PCM \( M \) can be allocated. Note that allocating an instance of a resource creates a fresh name \( \gamma \). The rule \( \text{OWN-UPDATE} \) 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 \( \text{HOARE-UPD} \) and \( \text{UPD-HOARE} \) 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:

\[a \preceq_M b \equiv \exists c. b \equiv a \cdot c \quad \text{(extension order)}\]

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 \( \equiv^M \) allows us to conclude that the collection \( c \) of all the resources in the system under the name \( \gamma \) 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 \( : P \rightarrow (A \times A) \), can we construct a PCM \( \text{Monotone}(R) \) and a function \( f : A \rightarrow \text{Monotone}(R) \) such that \( f(x) \preceq_M \text{Monotone}(R) \) \( f(y) \) 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 \( \mathbb{N}_{\text{max}} \). 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 and hence, ownership of the elements of \( \mathbb{N}_{\text{max}} \) is persistent.

All elements of the \( \mathbb{N}_{\text{max}} \) are valid. As a result, there is a frame preserving update from any element to any other element. In other words, owning \( \frac{\mathbb{N}_{\text{max}}}{m} \), we can update our resources to own \( \frac{\mathbb{N}_{\text{max}}}{m} \) 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 \( \mathbb{N}_{\text{max}} \) coincides with the \( \leq \) order on natural numbers.

**Authoritative PCM.** The authoritative PCM [9] over a PCM \( M \), \( \text{Aut}(M) \), is a PCM that allows us to give an instance-wide 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:

\[
\text{Auth}(M) \triangleq \{ \circ a | a \in M \} \cup \{ \bullet a | a \in M \} \cup \{ \bot \text{-Auth}(M) \}
\]

\[
a \equiv_{\text{Auth}(M)} b \triangleq \begin{cases}
a = a' 
& \text{if } a = \circ a' \text{ and } a' = b' \\
= \bullet a' 
& \text{if } a = \bullet a' \text{ and } a' = b' \\
\equiv_{M} a' \land \equiv_{M} b' 
& \text{if } a = \circ (a', b') \text{ and } b = \bullet (b', a') \\
\text{True} 
& \text{if } a = \bot \text{-Auth}(M) \text{ and } b = \bot \text{-Auth}(M) \\
\text{False} 
& \text{otherwise}
\end{cases}
\]

\[
a \circ \text{-Auth}(M) b \triangleq \begin{cases}
\circ (a', b') 
& \text{if } a = \circ a' \text{ and } b = \circ b' \\
\bullet (a', b') 
& \text{if } a = \bullet a' \text{ and } b = \circ b' \\
\circ (b', a') 
& \text{if } a = \circ a' \text{ and } b = \bullet b' \\
\bullet (a_1', a_2') 
& \text{if } a = \circ (a_1', a_2') \text{ and } b = b' \\
\circ (b_1', a_2') 
& \text{if } a = \bullet a' \text{ and } b = \bullet b' \\
\text{False} 
& \text{otherwise}
\end{cases}
\]

\[
\varepsilon_{\text{Auth}(M)} \triangleq \circ E_{M}
\]

\[
\text{Auth}(M) a \triangleq \begin{cases}
\sqrt{M^a} 
& \text{if } a = \circ a' \\
\text{True} 
& \text{if } a = \bullet a' \\
\sqrt{M}a_1 \land a_2 \leq_M a_1 
& \text{if } a = \circ (a_1', a_2') \\
\text{False} 
& \text{otherwise}
\end{cases}
\]

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 \( \bot_{\text{Auth}(M)} \) (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.

\[
\text{AUTH-FULL-EXCLUSIVE} \\
\circ a \neq \bullet b' 
\]

The operation on the fragments, on the other hand, is defined pointwise. Hence, we have that \( \circ a \neq \text{Auth}(M) \) is persistent whenever \( \circ a \neq \text{Auth}(M) \) is persistent. Importantly, if an element \( \circ (a, b) \) is valid, then \( b \leq a \) holds. This fact, together with the rules rules \text{OWN-OP} and \text{OWN-VALID}, allows us to derive the following:

\[
\text{AUTH-INCLUDED} \\
\circ a : \text{Auth}(M)^V \neq \circ b : \text{Auth}(M)^V 
\]

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

**Authoritative PCM over \( \mathbb{N}_{\text{max}} \).** As expected, the PCM \( \text{Auth}(\mathbb{N}_{\text{max}}) \) 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 \text{AUTH-INCLUDED} we have that if we own \( \frac{\mathbb{N}_{\text{max}}}{m} \) and \( \frac{\mathbb{N}_{\text{max}}}{m} \), we can conclude \( m \leq n \). The more interesting aspect of the \( \text{Auth}(\mathbb{N}_{\text{max}}) \) is the following frame-preserving update \text{AUTH-NAT-MAX-FP-UPD} and its consequence \text{AUTH-NAT-MAX-OWN-UPD}:

\[
\text{AUTH-NAT-MAX-FP-UPD} \\
\circ n \sim_M \circ (m, k) \quad \frac{\circ n \neq \circ (m, k)}{\circ n \neq \circ (m, k)}
\]

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:

\[
P := \cdots | \Box \text{(invariants)}
\]

The proposition \( \Box \) asserts that \( P \) holds at all times, i.e., \( P \) is an invariant.\(^4\)

\(^4\)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.
INV-ALLOC
\[ P \vdash \exists P \]

HOARE-ATOMIC
\[ (P \cdot R) \cdot (y. Q \cdot R) \quad R \quad e \text{ is physically atomic} \]
\[ \{P\} e \cdot (y. Q) \]

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 \( P \) only asserts the knowledge that \( P \) should hold invariantly. Hence, invariants are persistent, i.e., duplicable:

INV-DUPLICABLE
\[ P \vdash \exists P \cdot P \]

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:

\[ \text{isCounter}(\ell, \gamma) \triangleq \exists n. \ell \rightarrow n \cdot \mathcal{O} n : \mathcal{A}(N_{\max}) \]
\[ \text{CounterAtLeast}(y, n) \triangleq \forall n : \mathcal{A}(N_{\max}) \]

Both of these predicates are persistent. In the following we briefly discuss the proofs that the specs for the counter operations hold. We do not include the exact code of the counter implementation; it is entirely standard and uses an atomic increment operation.

Proof of \( \text{mkCounter-spec} \). Creating the counter simply consists of allocating a reference \( \ell \) with value 0 which gives us \( \ell \rightarrow 0 \). We use the rule \( \text{OWN-ALLOC} \) to allocate a fresh instance \( y \) of the ghost state \([\mathcal{O} y, \mathcal{O} y']\), which is a valid element of \( \mathcal{A}(N_{\max}) \). Hence, we obtain \( [\mathcal{O} y, \mathcal{O} y'] \) and \( [\mathcal{O} y', \mathcal{O} y] \). We use the latter to establish \( \text{CounterAtLeast}(y, n) \), and the former together with \( \ell \rightarrow 0 \) and the rule \( \text{INV-ALLOC} \) to establish \( \text{isCounter}(\ell, y) \).

Proof of \( \text{counter-increment-spec} \). At the atomic point of incrementing the counter we use the \( \text{HOARE-ATOMIC} \) to access the invariant. Thus we get that there is a number \( n \) for which we have \( \ell \rightarrow n \) and \( \mathcal{O} n' \). The atomic increment operation updates \( \ell \rightarrow n + 1 \). In order to reestablish the invariant we use the rule \( \text{AUTH-NAT-MAX-OWN-UPD} \) to obtain \( \mathcal{O} n' + \mathcal{O} n' \) and \( \mathcal{O} n' + \mathcal{O} n' \). We use the former to reestablish the invariant and simply ignore the latter.\(^5\)

Proof of \( \text{counter-read-spec} \). The precondition gives us \( \text{CounterAtLeast}(y, n) \). When we perform the atomic read operation we access the invariant using the \( \text{HOARE-ATOMIC} \). At this point we get that there is a number \( k \) for which we have \( \ell \rightarrow k \) and \( \mathcal{O} k' \). We use the rule \( \text{AUTH-INCLUDED} \) together with \( \mathcal{O} k' \) and \( \mathcal{O} k' \) to establish that \( n \leq k \). We then use the rule \( \text{AUTH-NAT-MAX-OWN-UPD} \) with \( \mathcal{O} k' \) to get \( \mathcal{O} k' \) and \( \mathcal{O} k' \) without actually increasing the value of the ghost state. We then reestablish the invariant and use the freshly created \( \mathcal{O} k' \) 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 \( N_{\max} \) PCM whose extension order corresponds to the \( \leq \) relation.

4 The Monotone PCM

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

\[ \text{principal}_R(x) \leq \text{Monotone}(R) \]
\[ \text{principal}_R(x) \equiv R(x, y) \]

(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 \( (\mathcal{A}, R, \wedge, \bot) \) where \( R \subseteq \mathcal{A} \times \mathcal{A} \) is a partial order relation, \( \wedge \) 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 \( \wedge \) as the PCM operation and \( \bot \) as the unit element. Indeed, we can easily check that \( \text{monotone-order} \) holds by taking the \( \text{principal}_R \) function to be the identity function:

\[ (\exists z. y = x \wedge z) \equiv R(x, y) \]

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, \( \text{Monotone}(R) \), is defined as the PCM \( (\mathcal{P}_{\text{fin}}(\mathcal{A}), \equiv_{\text{Monotone}(R)}, \cup, \emptyset, \vee_{\text{Monotone}(R)} ) \) where:

\[ A \equiv_{\text{Monotone}(R)} B \iff \forall x \in \mathcal{A}. \text{Below}(x, A) \equiv \text{Below}(x, B) \]

(monotone-equiv)

\[ \text{Below}(x, A) \triangleq (\exists y. y \in A \wedge R(x, y)) \]

\[ \vee_{\text{Monotone}(R)} A \equiv \text{True} \]

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

\[ \text{principal}_R(x) \triangleq \{ x \} \]

It is easy to check that the relation \( \equiv_{\text{Monotone}(R)} \) 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 \( \text{Below-union} \) below.
Lemma 4.2. The predicate Below satisfies the following properties:

\[ \text{Below}(x, A \cup B) \iff \text{Below}(x, A) \lor \text{Below}(x, B) \quad \text{(Below-union)} \]

\[ \text{Below}(x, \text{principal}_R(y)) \iff R(x, y) \quad \text{(Below-principal)} \]

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.

\((\Rightarrow)\). We know that there is a set \( A \subseteq \mathcal{A} \) such that \( \text{principal}_R(y) \equiv \text{principal}_R(x) \cup A \). We need to show that \( R(x, y) \) holds. By Below-principal above it suffices to show that \( \text{Below}(x, \text{principal}_R(y)) \). However, since \( \text{principal}_R(y) \equiv \text{principal}_R(x) \cup A \) holds, this is equivalent to showing \( \text{Below}(x, \text{principal}_R(x) \cup A) \) which holds trivially by Below-union and Below-principal.

\((\Leftarrow)\). We know that \( R(x, y) \) and we need to show that \( \text{principal}_R(y) \leq \text{principal}_R(x) \). We prove \( \text{principal}_R(y) \equiv \text{principal}_R(x) \cup \text{principal}_R(y) \), or equivalently, by Below-union, \( \text{Below}(z, \text{principal}_R(y)) \iff \text{Below}(z, \text{principal}_R(x)) \lor \text{Below}(z, \text{principal}_R(y)) \). The forward direction holds trivially. For backward direction we only need to show that \( \text{Below}(z, \text{principal}_R(x)) \) implies \( \text{Below}(z, \text{principal}_R(y)) \). 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 \( \text{Auth}(\text{Monotone}(R)) \) elements.

\[ \text{AUTH-MONOTONE-ORDER} \]

\[ \bullet \text{principal}_R(y)^\dagger \equiv \text{principal}_R(x)^\dagger \lor [R(x, y)] \]

Similarly to \( \mathbb{N}_{\text{max}} \), 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 \( \text{Auth}(\text{Monotone}(R)) \).

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

\[ \text{AUTH-MONOTONE-FP-UPD} \]

\[ R(x, y) \]

\[ \bullet \text{principal}_R(x) \sim_{M} \bullet (\text{principal}_R(y), \text{principal}_R(z)) \]

\[ \text{AUTH-MONOTONE-OWN-UPD} \]

\[ R(x, y) \]

\[ \bullet \text{principal}_R(x)^\dagger \lor [\bullet \text{principal}_R(y)^\dagger \equiv \text{principal}_R(z)^\dagger] \]

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 \( V \) from values to values in \( \mathcal{A} \), 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 \( \ell \rightarrow_y v \), and a proposition AtLeast \( (y, v) \) which indicates that the monotone reference whose state is tracked by the ghost state \( y \) 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:

\[ \text{AtLeast}(y, v) \iff \exists x. [V(x) = x] \cdot [\text{principal}_R(x)^\dagger] \]

\[ \ell \rightarrow_y v \iff \ell \rightarrow v \cdot \text{Exact}(y, v) \]

\[ \text{Exact}(y, v) \iff \exists x. [V(x) = x] \cdot [\text{principal}_R(x)^\dagger] \]

The monotone reference construction satisfies the following rules:

\[ \text{MON-REF-ALLOC} \]

\[ [\text{V}(v) = x] \]

\[ \ell \rightarrow v \]

\[ \Rightarrow \exists y. \ell \rightarrow_y v \]

\[ \text{MON-REF-SNAPSHOT} \]

\[ \ell \rightarrow_y v \]

\[ \Rightarrow \ell \rightarrow_y v \cdot \text{AtLeast}(y, v) \]

\[ \text{MON-REF-RECALL} \]

\[ \ell \rightarrow_y v \]

\[ \text{AtLeast}(y, w) \]

\[ \ell \rightarrow_y [\text{V}(w), \text{V}(v)] \]

\[ \text{MON-REF-CANCEL} \]

\[ \ell \rightarrow_y v \]

\[ \ell \rightarrow v \cdot [\text{V}(v), \text{V}(w)] \]

\[ \Rightarrow \ell \rightarrow w \]

\[ \text{MON-REF-LOAD} \]

\[ \ell \rightarrow_y v \]

\[ \ell \rightarrow x \cdot [x = v] \cdot \ell \rightarrow_y v \]

\[ \text{MON-REF-STORE} \]

\[ \ell \rightarrow_y v \]

\[ \ell \rightarrow w \cdot [x. \ell \rightarrow_y w] \]

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 \( \bullet \text{principal}_R(x), \) principal \( (y) \) to obtain a name \( y \) and establish \( \text{Exact}(y, v) \) and \( \text{AtLeast}(y, v) \). 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 \( \ell \) 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:

\[ \text{let } x = \text{ref } 0 \text{ in} \]

\[ \text{let } y = \text{ref } 0 \text{ in} \]
The intuitive idea is that fractional points-to propositions 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 $\ell_x$ and $\ell_y$, by letting the existentially quantified $s$ be the state $(0, 0)$:

$$\text{inv}_{\text{ST}} \equiv \exists s. \ell_x \mapsto_{i/j} \text{val}_{\ell_x}(s) \land \ell_y \mapsto_{i/j} \text{val}_{\ell_y}(s) \land \text{ExactST}(\gamma, y, s)$$

Given a state $s$, the $\text{val}_{\ell_x}(s)$ returns the value of $\ell_x$ 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 $\text{val}_{\ell_y}(s)$ returns the value of $\ell_y$ in state $s$.

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

- $\{ \ell_x \mapsto_{i/j} 0 \ast \text{inv}_{\text{ST}} \}$
- $\{ \ell_y \mapsto_{i/j} 0 \ast \text{inv}_{\text{ST}} \}$

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 $\text{ObsST}(\gamma, (1, 0))$ and $\text{ObsST}(\gamma, (0, 1))$. But combining these two propositions with the fact (obtained from the invariant) that there is some state $s$ for which we have $\text{ExactST}(\gamma, s)$, we would then get $\text{Reach}(1, 0), s)$ and $\text{Reach}(0, 1), s)$ by rule $\text{STATE-REACHABLE}$. 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: $\text{Observed}_s$ and $\text{Observed}_e$ 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 Observed, to express, through resources, what events
have been observed, and threads would use ownership of
Observed, 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 ev0 and
ev1 in Events for which we know ev0.time < ev1.time, i.e.,
the logical time of ev0 is smaller than that of ev1, or, ev1
causally depends on ev0. Crucially, ev0 is the only event that
ev1 depends on, and ev0 itself has no dependencies. We verify
the following the program, i.e., we show it does not crash:

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

Since our programming language does not support network-
ing we simply fork functions that simulate receiving
events. Notice that the order at which the events are
received depends on scheduling of threads and hence is
non-deterministic. The function simulate_receieve 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 ev1 and then asserts that
the event ev0 is also observed, i.e., it crashes if this is not the
case.

We define the predicates Observed, and Observed, as follows:

\[
\text{Observed}_\bullet(y, E) \triangleq \exists_e \text{Observed}_\circ(y, E) \cap \text{principal}_{CCS E}^y
\]

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

\[
\text{CCS}(E, F) \triangleq E \subseteq F \land
\forall e, e' \in F. \text{ev.time} \leq \text{ev'.time} \land e' \in E \Rightarrow e \in E
\]

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

\[
isDB(y, dbp) \triangleq \exists_e, E. dbp \mapsto e \ast \text{DBcontents}(e, E) \ast \left[E \subseteq \{ev_0, ev_1\}\right] \ast \text{Observed}_\bullet(y, \text{ObservedSubset}(E))
\]

Here, the predicate DBcontents(e, E) asserts that the contents of the database e 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. Further-
more, we instantiate resources so that next to initializing the
invariant we also obtain Observed, as adding events can grow the set of observed
events in the database, ObservedSubset(E).

We ascribe the following specifications to the is_observed function:

\[
\text{isDB}(y, dbp) \ast \text{Observed}_\circ(y, E)
\]

\[
\text{is_observed dbp ev}
\]

\[
x. \exists b \in \{\text{true}, \text{false}\}, F. \left[x = b \ast [E \subseteq F] \ast \text{Observed}_\circ(y, F) \ast \left(\left([b = \text{true} \land e \in F]\right) \lor \left([b = \text{false} \land e \notin F]\right)\right)\right] \ast \text{is_observed-spec})
\]

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

\[
\text{isDB}(y, dbp) \ast \text{Observed}_\circ(y, E)
\]

\[
\text{wait_for dbp ev}_1
\]

\[
x. \exists F. \text{Observed}_\circ(y, F) \ast ev_1 \in F
\]

That is, after wait_for dbp ev1 we know Observed, as some set F for which ev1 \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 ev0 \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, as obtained from the post-
condition of call to is_observed dbp ev0. This time we obtain a set
Observed, (y, G) for some set G such that F \subseteq G. Moreover,
we are guaranteed that if this call returns false, ev0 \notin G
which is a contradiction. Hence, the crashing else branch
is not reachable.

## 6 Canonicity of Construction

In this section we show that the construction of the mono-
tone 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
Definition 6.1. Let \((M, \equiv_M, \cdot_M, \epsilon_M, \vee_M)\) and \((M', \equiv_{M'}, \cdot_{M'}, \epsilon_{M'}, \vee_{M'})\) be two PCM’s. A PCM-morphism from \(M\) to \(M'\) is a function \(f : M \to M'\) such that the following hold:

\begin{align*}
\forall a \equiv_M b & \Rightarrow f(a) \equiv_{M'} f(b) & \text{(respect-equiv)} \\
\forall f(a \cdot_M b) & \equiv_{M'} f(a) \cdot_{M'} f(b) & \text{(respc-op)} \\
\forall \epsilon_M & \equiv_{M'} \epsilon_{M'} & \text{(respc-unit)} \\
\forall f(M) & \equiv_{M'} f(M) & \text{(respec-validity)} \\
\end{align*}

Perhaps the first thought that comes to mind is that there should be an adjunction between the functors \(\text{Monotone} : \text{PreOrder} \to \text{PCM}\) and \(\text{Extension} : \text{PCM} \to \text{PreOrder}\). 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 \(\text{Monotone} \circ \text{Extension}\) to the identity functor \(\mathbb{1}_{\text{PCM}}\). This natural transformation should produce PCM morphisms of the form \(\text{Monotone}(\leq_M) \to M\) 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 \(\mathbb{JSL}\) 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 \(\text{PCM}\) of PCM’s because the inclusion functor \(\mathbb{JSL} \hookrightarrow \text{PCM}\) 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 orders, i.e., if they hold for \(n\) they should hold for any \(m \leq n\). In our

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 \([8,9]\) proof mode \([10,11]\). Moreover, all the category-theoretic results presented in Figure 2 are formalized on top of the category theory library of Timany and Jacobs \([14]\). 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.,...
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 de-
termines 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 mono-
tone 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 that is
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
elements presented in this paper are almost exactly as ex-
plained 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 pre-
sentated 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 state-
ments 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 \( \text{dbp} \) is accessed and updated concurrently. These
concurrent accesses should somehow be synchronized. In
our implementation we opted to use the atomic CAS opera-
tion in a CAS-loop for synchronization purposes; a standard
technique in fine-grained concurrent separation logic. Al-
ternatively, one could have used a lock for synchronization.
Since the CAS operation in HeapLang only works on refer-
ences to base values or other references, the implementation
of the empty\_DB function in fact returns a reference to a list
and \( \text{dbp} \) 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 for-
malization we define the monotone construction as a con-
struction 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 construc-
tion. 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 construc-
tion of quotients is the only place where we use classical
axioms in our formalization.

We define the monotone construction using lists to rep-
resent 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 rela-
tion. Ahman et al. [2] essentially provide a variant MST of
F*’s state monad 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 F* 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 pre-
order. 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.

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 \textit{Monotone}(\mathcal{R}) to reason about monotonicity, with respect to an arbitrary preorder relation \mathcal{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

References


