Higher-Order Separation Logic for Distributed Systems and Security

Simon Oddershede Gregersen

Ph.D. Dissertation

Department of Computer Science
Aarhus University
Denmark

Higher-Order Separation Logic for Distributed Systems and Security

A dissertation
presented to the Faculty of Natural Sciences
of Aarhus University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

by
Simon Oddershede Gregersen
31st January, 2023

Abstract

Rigorous reasoning about implementations of software systems requires a detailed mathematical model of the behavior of the programming language. However, real-world programming languages are rich in features, and their mathematical models are complex and unfeasible to reason about directly. We need powerful mathematical machinery to alleviate this complexity and make it viable to reason formally about programs implemented in real-world programming languages.

This Ph.D. dissertation is a collection of five papers that develop and apply higher-order separation logics to tame the complexity of mathematical models for rich programming languages in the context of distributed systems, a type system for information-flow control, and contextual equivalence of probabilistic programs.

The first part of the dissertation develops a higher-order distributed separation logic that allows us to reason modularly about network-connected distributed applications that run on multiple machines while communicating over an unreliable network. We argue through several case studies that the logic is a solid foundation that makes verifying a range of distributed systems and protocols feasible. We also show how distributed systems can be verified and specified by establishing simulation relations with abstract models.

The second part of the dissertation is concerned with an expressive static information-flow control type system that guarantees that a well-typed program's public behavior is independent of its secret inputs. We develop a semantic model of the type system in a higher-order separation logic that allows us to prove that the type system is sound. Using the model, we also show how to compositionally integrate syntactically ill-typed—but semantically secure—components with well-typed programs.

The third and final part of the dissertation develops a relational higher-order separation logic for reasoning about contextual equivalence of probabilistic programs implemented in an expressive programming language with higher-order local state and polymorphism. We develop a proof method for relating asynchronous probabilistic samplings in a program logic and demonstrate the approach's usefulness with several case studies.

Resumé

Stringent matematisk ræsonnering om computerprogrammer kræver en detaljeret matematisk model af programmeringssprogets opførsel. Realistiske programmeringsprog har mange funktionaliteter, og deres modeller er komplicerede og tilnærmelsesvist umulige at ræsonnere direkte om. Vi har brug for kraftfulde matematiske værktøjer til at håndtere kompleksiteten og for at gøre det muligt at ræsonnere formelt om programmer implementeret i realistiske programmeringssprog.

Denne Ph.D.-afhandling er en samling af fem forskningsartikler, der udvikler og anvender højereordens separationslogik til at håndtere matematiske modeller af realistiske programmeringsprog i kontekst af distribuerede systemer, et typesystem, der garanterer informationssikkerhed, og kontekstuel ækvivalens af probabilistiske programmer.

Første del af afhandlingen udvikler en højereordens distribueret separationslogik, der tillader os at ræsonnere modulært om netværksforbundne distribuerede applikationer, der bliver afviklet på flere maskiner, og som kommunikerer over et upålideligt netværk. Gennem flere casestudier argumenterer vi for, at logikken er et solidt fundament, der gør det muligt at verificere en række distribuerede systemer og protokoller. Vi viser også, hvordan distribuerede systemer kan verificeres og specificeres ved hjælp af simuleringsrelationer og abstrakte modeller.

Anden del af afhandlingen beskæftiger sig med et statisk typesystem, der garanterer, at den offentligt observerbare opførsel af programmer, der typetjekker, er uafhængig af deres hemmelige input. Vi konstruerer en semantisk model af typesystemet i en højereordens separationslogik, hvilket tillader os at vise, at typesystemet er sundt. Ved brug af modellen viser vi også, hvordan man kompositionelt kan integrere programmer, der typetjekker, med programmer, der ikke typetjekker, men som er semantisk sikre.

Den tredje og sidste del af afhandlingen udvikler en relationel højereordens separationslogik til at ræsonnere omkring kontekstuel ækvivalens af probabilistiske programmer implementeret i et udtryksfuldt programmeringssprog med blandt andet lokal hukommelse og typepolymorfi. Vi udvikler en bevisteknik til at relatere asynkrone probabilistiske prøvetagelser i en programlogik, og vi demonstrerer brugbarheden af vores tilgang gennem flere casestudier.

Acknowledgments

My deepest gratitude is due to Lars Birkedal, my primary advisor, for all the guidance, support, understanding, insights, and camaraderie provided throughout the years. I am sincerely grateful that you believed in me and took me on as a student. Your warm-hearted but no-nonsense approach to the advisory role is truly inspiring. Lars has always had my best interest in mind.

To my co-advisor, Amin Timany, I also express my deepest gratitude for all the wisdom bestowed on me, all the technical discussions, and for always finding the time to answer my questions. Without Lars and Amin, this dissertation would never have happened.

A special thank you to Anders Møller, who took me on as a student programmer, and whose guidance I was initially under when getting admitted to graduate school. You showed me great compassion while I found my academic footing, and I thoroughly enjoyed developing and teaching the course on programming languages with you.

Thanks to all former and present members of the Logic and Semantics group and the Programming Languages group for a great environment and friendship through all these years. Many cups of coffee have been had while discussing both serious and not-so-serious matters.

To my co-authors, both at the department and outside of the department, thank you for all the great ideas and hard efforts that went into our collaborations.

To Sabine, I am eternally grateful for taking me to Scotland, the many meaningful conversations about academic hardships, and the discussions that inspired parts of this dissertation. I am excited to see what is next to come.

Simon Oddershede Gregersen
Aarhus, 31st January, 2023

Contents

Part I

Overview

1 Introduction

Formal reasoning about software systems requires precise mathematical specifications of how they behave. While crucial, it is not enough to *only* verify the high-level algorithm or protocol in an abstract model of computation—we *also* need to show that the program implementing an algorithm does so correctly and does not introduce errors or vulnerabilities not visible at the abstract algorithmic level. Many software system errors and security breaches stem from subtle problems introduced by the implementation, and it is hence necessary to consider detailed models of the actual program execution. However, semantic models of execution for rich and realistic programming languages introduce an abundance of complexity that is difficult to handle and contain.

This Ph.D. dissertation is a collection of five papers that develop and apply *higher-order separation logics* to tame the complexity of detailed operational models of program execution in the context of *distributed systems*, *a type system for information-flow control*, and *contextual equivalence of probabilistic programs*. This chapter explains the background and motivation for considering each of the problems addressed in the included papers, and we give a concise overview of the key concepts and ideas that will make the technical details considered in these works more accessible. We will discuss related work throughout the introduction as needed, but detailed comparisons are left to the individual chapters.

1.1 Background

The correctness of computer programs has been the subject of much scientific interest since the dawn of computer science. From the endeavors of Turing [Tur49], Floyd [Flo67], and Hoare [Hoa69] to Owicki-Gries [OG76], O'Hearn [ORY01], and Reynolds [Rey02], many efforts have been devoted to not only writing correct programs but also *proving* with mathematical rigor that the programs are indeed correct.

To mathematically prove properties of systems, one must have a precise mathematical specification of what the system actually *is*. *Structural operational semantics* [Plo04] gives formal meaning to programming languages through a sequence of syntax-oriented and inductively defined computational steps of a hypothetical computer. Operational semantics are simple first-order state-transition systems that are easy to define and manipulate even for feature-rich languages as evidenced by, *e.g.*, formalizations of real-world languages like C [Ler09]. This fact stands in stark contrast to other approaches, such as *denotational semantics* [SS71; Sco70], that attach mathematical meanings to its terms more directly by mapping syntactical terms of the language to—often very sophisticated—mathematical objects. Operational semantics has the benefit of being very easy to define and understand, but reasoning directly about programs quickly becomes unfeasible as the number of cases to consider increases dramatically when phenomena, such as non-determinism, are introduced in the semantics to model, *e.g.*,

concurrency and distribution. To mitigate this complexity, one can develop a *program logic*, a mathematical tool with much more high-level reasoning principles, to reason about programs and programming languages with a detailed and challenging operational model.

Not only do we want mathematical machinery to reason about complicated systems, but the machinery we develop must allow us to reason *modularly* for our tools to scale to large and complex real-world systems. Just as large software systems are programmed by composing many different software components (using features such as objects, interfaces, higher-order functions, and module systems), modular verification allows us to specify and verify individual software modules *in isolation*—specifications of the individual components are then combined to specify and verify larger systems. To realize this vision, program specifications and proofs must concentrate on the resources that a program component acts upon instead of, *e.g.*, the entire state of the system. The underlying hypothesis explored and developed in this dissertation is that *higher-order separation logic* is a crucial ingredient and fundamental building block to achieving this goal.

With the rise of higher-order separation logic [Din+13; Din+10; Jun+16; Jun+18b; Jun+15; Kre+17; LN13; Nan+14; OHe07; RDG14; SB14; TDB13], recent years have seen considerable progress on modular reasoning using program logics for programming languages with operational models and increasingly sophisticated features such as local and higher-order state, concurrency, weak memory, effect handlers, polymorphism, *etc.* [Ahm+10; App11; Car+22; Dan+22; JTD21; Jun+18a; Kai+17a; MP22; MJ21; Spi+21; TB19; Tim+18; TVD14; Tur+13; VP21], to name just a few. Much of this work is concerned with *partial correctness* or *safety*, *i.e.*, that "bad things do not happen," such as the program crashing. But there are many more properties that one might be interested in showing—such as *liveness* ("good things will happen"), *security*, and *refinement*—and many more aspects to consider—such as *distribution* and *randomization*—for which existing approaches do not apply. This dissertation not only tests the hypothesis that higher-order separation logic is essential to modular program verification, but it pushes the frontiers of what properties we can prove and what systems we can verify.

In 2010, Matthew Parkinson's position paper *The Next 700 Separation Logics* [Par10] observed that, at the time, each new application or primitive seemed to require a new separation logic. He hypothesized that "by finding the right core logic" we would be able to "concentrate on the difficult problems." *Iris* [Jun+16; Jun+18b; Jun+15; Kre+17] is a proposal for such a core logic and framework for building higher-order separation logics. Iris lets users introduce their own notion of ownership based on the idea of *fictional separation* [JB12] and with built-in support for advanced semantic features such as *step-indexing* [AM01], *impredicative invariants* [SB14], and *higher-order ghost state* [Jun+16], the framework allows users to derive new reasoning principles *inside* the core logic. This fact avoids having to reprove many variations of the same semantic arguments. The framework acts as a unifying foundation into which various specifications can be encoded and exploited via the abstraction facilities offered by the logic. Iris also has extensive support for interactive machine-checked proofs in the Coq proof assistant [Kre+18; KTB17] which allows you to get the highest level of assurance that all details and results are indeed correct.

The Iris separation logic framework is the semantic bedrock underlying this dissertation. The framework comes with a program logic for shared-memory concurrent programs but—as will be evident by the end of this dissertation—this program logic does *not* suffice for all the kinds of properties and systems that we would want. On top of the Iris foundations, however, we will build new program logics, new notions of ownership, and new logical abstractions to reason modularly about *distributed systems* and *security*.

1.2 Distributed separation logic

Separation logic [ORY01; Rey02] is a resource-oriented logic that supports *local reasoning* about shared mutable state. Propositions denote not only facts but *ownership* of resources that can be manipulated through the basic connectives of bunched implications [OP99], namely *separating conjunction* ($P * Q$) and its adjoint sibling *magic wand* ($P \mathrel{-\!\!*} Q$), in that

$$P * Q \vdash R \text{ iff } P \vdash Q \mathrel{-\!\!*} R$$

The quintessential idea of separation logic is that we can give a Hoare-logic style local specification $\{P\}\ e\ \{Q\}$ to a program e involving only the resources (*e.g.*, parts of the heap) that are used by e. By using the *frame rule*, one can extend a local specification with arbitrary predicates that are not modified or mutated by e:

$$\frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}}$$

Concurrent separation logic [OHe07] further develops the idea of separation to support *thread-local* reasoning in the presence of concurrency. The specification for each program component continues to concentrate on only the resources used by the component, not mentioning the states of other threads or resources—in a way, the "frame" includes the execution of other threads which can be interleaved throughout the execution. This manifests in the rule for disjoint concurrency:

$$\frac{\{P_1\}\ e_1\ \{Q_1\} \qquad \{P_2\}\ e_2\ \{Q_2\}}{\{P_1 * P_2\}\ e_1 \mid\mid e_2\ \{Q_1 * Q_2\}}$$

where $e_1 \mid\mid e_2$ denotes disjoint concurrency execution of expressions e_1 and e_2. The original formulation of concurrent separation logic adds the ability to share resources between threads through a form of shared resource invariants [OHe07]. *Higher-order* concurrent separation logic [SB14; SBP13] further generalizes the logical machinery with *impredicative invariants* to facilitate even more modular reasoning principles. In either case, the fundamental reasoning principle is to govern shared resources by invariants through an *invariant assertion* $\boxed{P}$ that can be shared freely in that $\boxed{P} \vdash \boxed{P} * \boxed{P}$. Concurrently running threads may then *rely* on the proposition P to hold at all times as long as they *guarantee* not to violate it themselves.

Higher-order *distributed* separation logic, as developed in this dissertation throughout Chapter 3, Chapter 4, and Chapter 5 in the shape of the *Aneris* program logic, further develops the idea of separation to support both *node-local* and thread-local reasoning in distributed systems where communication among nodes is unreliable. Concretely, we consider a communication model with datagram-like sockets and socket-based communication with guarantees at the User Datagram Protocol (UDP) level. The quintessential principle of separation for distributed systems manifests in the rule for starting distributed nodes:

$$\frac{\{P_1 * \mathsf{FreePorts}(ip_1, \mathfrak{P})\}\ e_1\ \{\mathsf{True}\} \qquad \{P_2 * \mathsf{FreePorts}(ip_2, \mathfrak{P})\}\ e_2\ \{\mathsf{True}\}}{\{P_1 * P_2 * \mathsf{FreeIp}(ip_1) * \mathsf{FreeIp}(ip_2)\}\ (e_1; ip_1) \mid\mid\mid (e_2; ip_2)\ \{\mathsf{True}\}}$$

where $(e_1; ip_1) \mid\mid\mid (e_2; ip_2)$ denotes execution of expressions e_1 and e_2 on nodes with addresses ip_1 and ip_2.[1] Besides a "frame" that includes the execution of other threads *and*

[1] This presentation is not entirely correct in details but serve to illustrate the core principles and intuitions. For instance, in the concrete language studied in this dissertation only a distinguished primordial node may spawn nodes and expressions include a node identifier as further detailed in Chapter 3.

nodes, the rule also requires that two nodes may never have the same address and that freshly spawned nodes may assume that all communication ports are available. For modular reasoning, ownership of free ports is splittable, *i.e.*,

$$\mathsf{FreePorts}(ip, P \uplus \{p\}) \dashv\vdash \mathsf{FreePorts}(ip, P) * \mathsf{FreePort}(ip, p)$$

Note that the identity of communication endpoints in distributed systems is essential: to communicate with a particular party, we must unambiguously know where it is located; hence no endpoints may use the same identifier. This fact also surfaces in the rule for binding addresses to sockets by requiring and consuming ownership of the address being bound:

$$\{z \hookrightarrow - * \mathsf{FreePort}(a)\}\ \texttt{socketbind}\ z\ a\ \{v.v = 0 * z \hookrightarrow a\}$$

The assertion $z \hookrightarrow a$ expresses that the socket z is bound to the address a.

Distributed communication

Where concurrently running threads communicate through shared memory, distributed nodes communicate via messages that may be arbitrarily delayed, dropped, duplicated, or reordered during transmission.[2] To reason about communication, Aneris introduces the notion of *socket protocols* embodied by the separation logic assertion $a \Rrightarrow \Phi$ and the notion of *socket history* through the resource $a \rightsquigarrow (R, T)$. The protocol assertion $a \Rrightarrow \Phi$ states that messages sent to the address a are governed by a protocol $\Phi : \mathit{Message} \to \mathit{iProp}$ where *iProp* is the type of propositions in the logic. This means that nodes receiving messages on an address a may *rely* on Φ to hold for any received message; conversely, nodes sending messages to the address a have to *guarantee* that the messages satisfy the protocol. The history assertion $a \rightsquigarrow (R, T)$ tracks the sets R and T of received and sent messages on socket address a.

Formally, to send a message m to a destination address d through a socket z bound to an address a, we are required to establish that m satisfies the protocol Φ:

$$\begin{aligned}&\{z \hookrightarrow a * a \rightsquigarrow (R, T) * d \Rrightarrow \Phi * \Phi(m)\}\\ &\quad \texttt{sendto}\ z\ m\ d\\ &\{v.\ v = |m| * z \hookrightarrow a * a \rightsquigarrow (R, T \cup \{(m, d)\})\}\end{aligned}$$

In the postcondition, we get back the ownership of the socket handle z as well as an updated message history that reflects that the message has been sent.

Retransmitting already sent messages, on the other hand, is free—intuitively, the receiving party should already be able to handle message duplication happening during transmission so resending messages will always be safe:

$$\begin{aligned}&\{z \hookrightarrow a * a \rightsquigarrow (R, T) * (m, d) \in T\}\\ &\quad \texttt{sendto}\ z\ m\ d\\ &\{v.\ v = |m| * z \hookrightarrow a * a \rightsquigarrow (R, T)\}\end{aligned}$$

[2] The Aneris logic as presented in Chapter 3 assumes duplicate protection but this assumption has since been lifted and the technical development used in Chapter 4 and Chapter 5 and described in this introduction makes no such assumption. For this reason, some logical connectives and rules vary.

When trying to receive a message, three scenarios are logically possible as encoded in the postcondition of the rule below:

$$\{z \hookrightarrow a * a \rightsquigarrow (R, T) * a \Mapsto \Phi\}$$

`receivefrom` z

$$\left\{ v.\ z \hookrightarrow a\ * \begin{pmatrix} (v = \mathsf{None} * a \rightsquigarrow (R, T)) \lor \\ (\exists m, s.\ v = \mathsf{Some}\,(m, s) * a \rightsquigarrow (R, T) * (m, s) \in T) \lor \\ (\exists m, s.\ v = \mathsf{Some}\,(m, s) * a \rightsquigarrow (R \uplus \{(m, s)\}, T) * \Phi(m)) \end{pmatrix} \right\}$$

Either (1) no message is available, (2) a message is received from address s, but it is a duplicate of a previously received message, or (3) a new message is received, and it satisfies the protocol.[3]

As the possibility of message duplication and omission failure is inevitable for communication over an unreliable network, it will often be natural to specify communication protocols that reflect *knowledge* rather than actual ownership. Logically, this means that socket protocols will be *persistent*, which implies that the resources being transferred are duplicable, *i.e.*,

$$\forall m.\ \Phi(m) \dashv\vdash \Phi(m) * \Phi(m)$$

When this is the case, one can derive a unified *persistent socket protocol* connective, written $a \rightsquigarrow^{\Phi}_{\square} (R, T)$, that allows us to ignore message duplication altogether in the rules of the logic:

$$\{z \hookrightarrow a * a \rightsquigarrow^{\Phi}_{\square} (R, T)\}$$

`receivefrom` z

$$\left\{ v.\ z \hookrightarrow a\ * \begin{pmatrix} (v = \mathsf{None} * a \rightsquigarrow^{\Phi}_{\square} (R, T)) \lor \\ (\exists m, s.\ v = \mathsf{Some}\,(m, s) * a \rightsquigarrow^{\Phi}_{\square} (R \cup \{(m, s)\}, T) * \Phi(m)) \end{pmatrix} \right\}$$

Whereas socket histories are primordial, socket protocols can be decided and allocated dynamically during proofs. The permission to allocate a socket protocol for an address a is represented by the assertion $a \Mapsto -$. This assertion is exclusive, *i.e.*, $a \Mapsto - * a \Mapsto - \vdash \mathsf{False}$. Protocols can be allocated at any time using the following rule:

$$\frac{\{P * a \Mapsto \Phi\}\, e\, \{Q\}}{\{P * a \Mapsto -\}\, e\, \{Q\}}$$

This could be, *e.g.*, dynamically based on run-time information or primordially before nodes have been started if it is necessary or convenient that the protocol is statically known *a priori*. Once a protocol has been allocated, the resource becomes duplicable, *i.e.*,

$$a \Mapsto \Phi \dashv\vdash a \Mapsto \Phi * a \Mapsto \Phi$$

and hence freely sharable among nodes.

In Chapters 4 and 5, we will argue that the Aneris program logic and the higher-order distributed separation logic principles are solid foundations for modular verification of a large variety of distributed systems and protocols. More extensive realistic case studies include at the time of writing a distributed key-value store with causal consistency guarantees (Chapter 4), the Paxos consensus protocol (Chapter 5), op-based CRDTs [Nie+22], and session-type-inspired abstractions for reliable communication [Gon+22].

[3] The full development also supports *blocking* message receival in the style of UNIX sockets where the `receivefrom` operation blocks until a message is available. This is reflected in the logic with a similar rule where the case where no message is available is absent.

1.3 Simulation

Higher-order distributed separation logic makes it feasible to specify and verify complex distributed systems. Nevertheless, even using high-level separation logic abstractions, complex systems are naturally complex to reason about when all details must be accounted for, one way or another. More abstract formal verification systems such as SPIN [Hol97] and TLA$^+$ [Lam92] have been widely used to design, model, and verify complex concurrent and distributed systems. In these tool suites, systems are typically modeled as state transition systems, and the tools can usually semi-automatically check both safety and liveness properties. As such, they offer much more abstract and high-level means for specifying and verifying *algorithms* and *protocols*. However, they offer no guarantees about the *implementations* of systems nor the relationship between an implementation and its abstract specification. A natural question is whether it is possible to exploit or connect an abstract specification to verify an actual implementation. This methodology would allow us to split the verification of distributed systems into two independent parts: first, we model the protocol abstractly and verify in an idealized setting that it solves the problem at hand. Then we show that the implementation correctly implements the model while dealing with implementation issues and details.

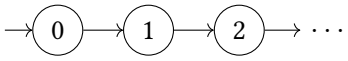
The Trillium separation logic framework developed in Chapter 5 answers this question positively by developing a program logic that allows us show that an implementation in a rich programming language correctly implements an abstract state-transition model.

What does it mean for a program to implement a model?

Consider the example program inc below that makes use of a reference that it increments concurrently in two infinite loops using an atomic *compare-and-set* (CAS) operation:

```
let l = ref 0 in
let rec inc () =
  let n = !l in
  cas(l, n, n + 1);
  inc ()
in inc () || inc ()
```

The reference takes successively the values 0, 1, 2, ... without skipping any numbers. We can express this property using a state-transition system model $\mathcal{M}_{inc}$:



and require that inc simulates $\mathcal{M}_{inc}$: for every program step of inc according to the operational semantics there must be a corresponding model transition in $\mathcal{M}_{inc}$ and the reached states correspond by agreeing on the contents of the reference. Such simulation properties do, however, not follow from most Iris-based program logics.

To demonstrate that a program logic makes the expected statements about program executions, one often proves an *adequacy theorem*. The adequacy theorem of program logics, such as Aneris and many other Iris-based logics, allows us to extract safety properties but also properties that can be derived from *invariant assertions* using a strengthened soundness theorem. The invariant $\boxed{P}$ guarantees that proposition P is guaranteed to hold at all times. For example, a specification

$$\boxed{\exists n.\, \ell \mapsto n * \mathsf{Even}(n)} \vdash \{P\}\, e\, \{Q\}$$

will allow us to conclude that e is safe and that the location ℓ only contains even numbers at all times. This approach allows us to express and "extract" extrinsic properties about *what* is achieved, but not generally intrinsic properties about *how* it is achieved, which is essential for proving simulations.

A strategy for carrying out relational reasoning in a unary program logic is to embed a model (or "specification") in the unary logic as two resources $\mathsf{Model}_\bullet(\delta)$ and $\mathsf{Model}_\circ(\delta)$, where δ is a state of the model, such that

$$\mathsf{Model}_\bullet(\delta) * \mathsf{Model}_\circ(\delta') \vdash \delta = \delta'$$

and guarantee that only valid transitions according to the transition relation of the model are permitted. This approach is used in Chapter 7, by Krebbers et al. [KTB17], and many others to prove contextual refinements; for proving simulation properties it does not suffice. For the example at hand, such a specification would have the shape

$$\boxed{\exists n.\, \ell \mapsto n * \mathsf{Model}_\bullet(n) * 0 \to^* n} \vdash \{\mathsf{Model}_\circ(n) * P\}\ inc\ \{Q\}$$

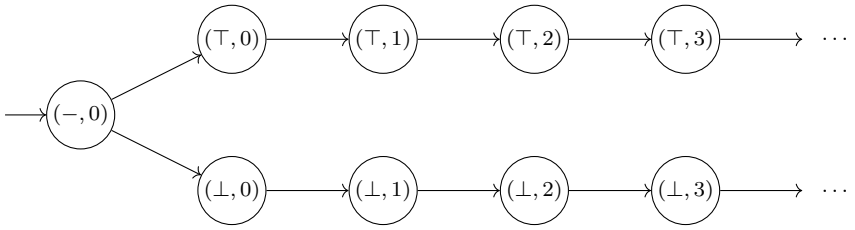
where $\to^*$ denotes the reflexive transitive closure of the transition relation of the model. Sadly, this approach does *not* allow us to extract a simulation relation through the adequacy theorem. This is immediately realized by observing that if inc had incremented the counter in steps of two, it would also satisfy the specification—an implementation that obviously does not simulate the model as it skips transitions.

Using the adequacy theorem of a program logic aimed at safety properties, we *can* prove that at every step k of the program execution, there exists a state δ_k of the model that is reachable from the initial state such that the mapping relation holds, *i.e.*, the contents of location ℓ agrees with the model state. This is precisely the kind of result needed to prove contextual refinement as done by, *e.g.*, Krebbers et al. [KTB17]—contextual refinement is about the "what," not the "how." What is missing is proof that the sequence $\delta_1, \ldots, \delta_n$ of model state witnesses denote a valid sequence of transitions in the model. In essence, the approach only allows us to talk about the *reachability* of states but *not* how the states are reached.

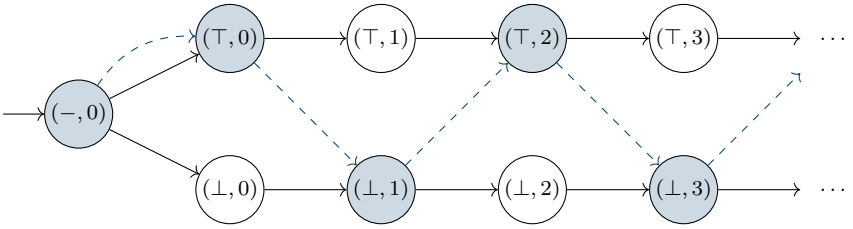
To further convince ourselves why this is important and why it does not suffice to just talk about the reachability of states when implementing a model, consider the following example:

```
1  let l = ref (true, 0) in
2  let rec inc () =
3    let (_, n) = !l in
4    let m = n + 1 in
5    l := (isEven m, m);
6    inc ()
7  in inc ()
```

The program continuously increments a reference in a loop but together with a Boolean that indicates the parity. Surprisingly, when only talking about the reachability of states, the program *does* refine the following model if we map the location to states of the model:



Notice how the model immediately decides on a Boolean value whereas the program switches back and forth as the parity of the counter changes—the program and the model are not progressing in the same way. The "refinement" is witnessed by the following sequence of model states indicated in blue:



However, the program is *not* a faithful implementation of the model. Every state in the sequence is reachable from the initial state, and the mapping to the program holds all along, but the sequence does *not* correspond to a valid sequence of transitions in the model. The technical achievement of Chapter 5 is the construction of a program logic, Trillium, that provides precisely this kind of guarantee. To do so, we have to establish a closer connection between the program and the model than achieved by only embedding the model in an invariant.

The formal notion of correspondence between programs and models that we consider is *simulation*. This approach becomes sensible when observing that the small-step operational semantics of a programming language defines a state-transition system (STS).

Definition 1.3.1 (Simulation)**.** Let $(\mathcal{M}_1, \to_1)$ and $(\mathcal{M}_2, \to_2)$ be two STSs. A relation $R \subseteq \mathcal{M}_1 \times \mathcal{M}_2$ is a *simulation* if and only if for all $(\delta_1, \delta_2) \in \mathcal{M}_1 \times \mathcal{M}_2$, if $\delta_1 \to_1 \delta_1'$ then there exists a δ_2' such that $\delta_2 \to_2 \delta_2'$ and $(\delta_1', \delta_2') \in R$.

Let a *trace* of $(\mathcal{M}, \to)$ be a sequence $\delta_0\delta_1\delta_2\ldots$ of model states $\delta_i \in \mathcal{M}$ such that $\delta_i \to \delta_{i+1}$. Any STS $\mathcal{M}$ induces an STS, written $\mathsf{Tr}(\mathcal{M})$, where its states are traces and transitions

$$\delta_0\delta_1\cdots\delta_n \to_{\mathsf{Tr}(\mathcal{M})} \delta_0\delta_1\cdots\delta_n\delta_{n+1}$$

if and only if $\delta_n \to_{\mathcal{M}} \delta_{n+1}$. This allows us to define the notion of *history-sensitive simulation*.

Definition 1.3.2 (History-sensitive simulation)**.** Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be STSs. A *history-sensitive simulation* is a simulation of $\mathsf{Tr}(\mathcal{M}_1)$ and $\mathsf{Tr}(\mathcal{M}_2)$.

The Trillium program logic developed in Chapter 5 allows us to prove, given proof of a Hoare Triple $\{P\}\ e\ \{Q\}^{\mathcal{M}}$, that some user-chosen relation R contains a history-sensitive simulation relating the program e and the model $\mathcal{M}$. In Section 5.4, we pick a relation such that history-sensitive simulation reduces to "just" simulation and show how two implementations of the distributed protocols *two-phase commit* and *single-decree Paxos* simulate their TLA$^+$ models. Finally, in Section 5.6, we sketch how one can use Trillium to prove history-sensitive simulations that are *fairness* and *termination* preserving and, as a consequence, prove *fair termination* for concurrent programs by proving termination of an abstract model. This dissertation only briefly touches upon proving such liveness properties using Trillium, and we refer to Stefanesco [Ste21] for more details on these aspects.

1.4 Termination-insensitive noninterference

Many important program properties are not only relational but phrased as a relation between two *programs*. It is prevalent to say, *e.g.*, that an implementation of a fine-grained concurrent queue is correct if it is contextually equivalent to a coarse-grained "obviously" correct queue implementation, that a program is secure if it is indistinguishable from another trivially secure implementation; or that a program is information-flow secure if two instantiations of the program but with different secret inputs always give the same observable outputs. This stands in contrast to *unary* properties, such as safety ("the program does not crash") and liveness ("the program eventually terminates"), that are phrased using a single program. Many unary properties are, unsurprisingly, targeted by unary program logics or unary models, whereas relational properties are often established using relational logics or *binary logical relations*.

Work on logical relations has made much progress in building step-indexed logical relations [Ahm04; Ahm+10; AM01; App01; Bir+11] that can handle many of the program and type system features of modern languages. Recent works give a "logical" account of step-indexed logical relations [DAB09] by interpreting types as relations expressed in a suitably powerful logic, such as Iris, and typing judgments as logical entailment between these relations. This approach allows us to define and reason about logical-relations models at a much higher level of abstraction. Relational separation logics and relational models have been developed for a range of contexts and properties, *e.g.*, contextual refinement [FKB21b; KTB17; TB19; Tim+18], simulation [Cha+19; Gäh+22; Tim+21], and security [FKB21a; Geo+21; GTB22] to name a few.

As suggested in the previous section, an approach to proving relational properties is to encode the relational model in a unary logic with resources that track the specification program. One tracks the specification program (the "right-hand side") in an invariant assertion which guarantees that the specification resource only progresses in valid ways (in this case according to the operational semantics):

$$\boxed{\exists e''.\, \mathsf{spec}_\bullet(e'') * e' \to^* e''} \vdash \left\{\mathsf{spec}_\circ(e')\right\} e \left\{v.\exists v'.\, \mathsf{spec}_\circ(v') * R(v, v')\right\}$$

By requiring in the postcondition of the unary Hoare triple that the specification program has reached a value, we may conclude properties of the shape "*if* e terminates *then* so does e'." This is precisely what is needed for contextual refinement: Intuitively, e contextually refines e' if whenever e terminates with some value v, then e' must also terminate with some value v', and v and v' should be suitably related. In symbols:

$$e \to^* v \implies e' \to^* v' \land v \approx v'$$

This is crucially different from the idea of *termination-insensitive* noninterference that we consider in Chapter 6 where two programs are considered equivalent if, assuming that *both* e and e' terminate, then their resulting values should be suitably related:

$$e \to^* v \land e' \to^* v' \implies v \approx v'$$

This is a subtle, yet pivotal commutation of operators that leaves the approach ineffective.

Static information-flow control enforcement is often specified as a special-purpose static type system (*e.g.*, [Aba+99; AM16b; HR98; LC15; Mye99; Sim03b]) or via an encoding into an existing type system (*e.g.*, [AR17; GTA19; LZ06; PS03; Rus15; RCH08; Vas+18]). A prevalent idea is to label syntactic types t with a *security label* ι drawn from a lattice that classifies the security level of the typed term, such as "public" or "secret". The lattice order $\sqsubseteq$ determines

the security policy such that if $\iota_1 \sqsubseteq \iota_2$ then ι_1 is interpreted as being "less sensitive" information than ι_2. The typing judgment $\Gamma \vdash_{pc} e : t^{\iota}$ guarantees that secret information does not influence public (or "less sensitive") information. The *program counter label* pc classifies the context in which the expression is getting evaluated in—if the execution branches on secret information just before evaluating e, it is important that no publicly observable side-effects happen in e. If that was the case, it would constitute an *implicit leak* as one might learn information about the secret based on the publicly observable side-effects—even if the program is not explicitly leaking anything. To counter this, the pc label is raised accordingly whenever execution branches on sensitive information and side-effects are restricted in sensitive contexts relative to the pc label.

The security property satisfied by this kind of information-flow control type system is often (a variation of) *noninterference*. Exactly which notion is the right one for a system at hand is a balance between permissiveness (which programs can you write) and your attacker model (which attackers do you care about and what can they observe). In Chapter 6 we consider *termination-insensitive noninterference* of sequential programs written in a rich language where the observable behaviour is the return value of the program. Termination-insensitive noninterference requires that if you have a publicly ($\bot$) observable program e that depends on some secret ($\top$) variable x, *i.e.*,

$$x : \mathbb{B}^{\top} \vdash_{\bot} e : \mathbb{B}^{\bot}$$

and given two secrets v_1 and v_2 such that $\vdash_{\bot} v_1 : \mathbb{B}^{\top}$ and $\vdash_{\bot} v_2 : \mathbb{B}^{\top}$, then whether you execute e with v_1 or v_2 will yield the same results in the following sense:

$$(\emptyset, e[v_1/x]) \to^* (\sigma_1, v_1') \land (\emptyset, e[v_2/x]) \to^* (\sigma_2, v_2') \implies v_1' = v_2'.$$

Due to its termination-insensitive nature, however, existing approaches for constructing such "logical" logical relations do not allow us to establish this property.

The key technical novelties of Chapter 6 that allows us to model the type system using a "logical" logical relation and prove termination-insensitive noninterference is (1) a new *modal weakest precondition* theory and (2) a binary logical relations model that incorporates a unary logical relation inside of it.

The modal weakest precondition predicate

$$\mathsf{mwp}^{\mathsf{M}}\, e\, \{\Phi\}$$

is a novel language-agnostic program logic construct that is parameterized by a modal operator M. Intuitively, it says that if the program e reduces to a value v in n steps, then $\Phi(v, n)$ holds under modality M. Different instantiations of the theory automatically inherit a set of basic structural rules, such as the "bind" rule for weakest preconditions, that hold irrespective of the particular modality or language instantiation. Most importantly, the theory is flexible enough so that we can use a modal weakest precondition predicate *as the modality* of another modal weakest precondition predicate. This allows defining both a unary and a binary predicate for reasoning about computations using a single unified theory and the different instantiations interact suitably to allow for termination-insensitive reasoning in a model with both binary and unary predicates.

Logical-relations models of static information-flow control type systems with termination-insensitive noninterference guarantees seem to require the use of both a unary and a binary

model in the presence of effects such as state. This has been realized by others as well [RG20]. Too see why, let us consider a typing rule for typing conditional expressions showcased below.

$$\frac{\Gamma \vdash_{pc} e : \texttt{bool}^{\iota} \qquad \forall i \in \{1,2\} \,.\, \Gamma \vdash_{pc \sqcup \iota} e_i : \tau \qquad \tau \searrow \iota}{\Gamma \vdash_{pc} \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau}$$

When branching on the expression $e : \texttt{bool}^{\iota}$, the pc label is raised with ι when type-checking the branches e_i—this reflects that the computation now depends on something of label ι as discussed above. Moreover, the label of τ is required to be *at least* as high as ι: the *protected-at* relation $\tau \searrow \iota$ is defined as $t^{\iota'} \searrow \iota \triangleq \iota \sqsubseteq \iota'$. This implies that if ι is $\top$, then the label of τ—the label of the whole conditional and hence its branches—has to be $\top$ as well. Practically speaking, the fact that something has the label $\top$ means that the two programs that we are relating in the proof of noninterference (or in the model that captures it) need not be related at all: we only care about *publicly* observable outputs, so the secret outputs do not matter. What we *do* care about, however, is that the two branches e_i, independently, do not produce publicly observable side-effects such as writing to references that have a public label. If they did, a continuation would be able to observe secret-dependent public values, and the program would not be noninterfering. The fact that the type system satisfies this property is often called a *confinement lemma* in proofs of noninterference.

To reflect this intuition in a logical-relations model we make use of both a binary and a unary model where the binary model captures that two programs "behave the same" and the unary model captures that a program does not have public side-effects. The main point of interaction between the two models appears in the interpretation of a labeled type t^{ι}:

$$[\![t^{\iota}]\!](v, v') \triangleq \begin{cases} [\![t]\!](v, v') & \text{if } \iota = \bot \\ [\![t]\!](v) * [\![t]\!](v') & \text{if } \iota = \top \end{cases}$$

If the label is publicly observable, the two values v and v' are required to inhabit the binary relation which guarantees that they behave the same. On the other hand, if the label is not publicly observable, we only require that v and v' inhabit the unary relation which guarantees that they do not produce any public side-effects.

In Chapter 6, we build a "logical" logical-relations model for proving termination-insensitive noninterference for a rich language that also includes features such as higher-order state, recursive types, and impredicative polymorphism. Here we also address how to hide properly the details of step-indexing and recursive Kripke worlds (as needed to model such a rich language) when using a unary and a binary "logical" logical-relations model in combination.

1.5 Asynchronous probabilistic couplings

In network-connected distributed applications, communication happens across a network that can generally *not* be trusted, as some machines in the network might act on behalf of an adversary with malicious intent. Cryptographic protocols constitute one of the essential classes of distributed algorithms for ensuring security properties such as data confidentiality, integrity, and authentication in Byzantine settings where parties cannot be trusted. For a concrete example, consider the Transport Layer Security (TLS) protocol that makes web (HTTPS) connections more secure.

Higher-order distributed separation logic, as developed in this dissertation, makes compositional reasoning about effects such as state, concurrency, and unreliable distributed communication viable. However, it relies on the fundamental assumption that all components are verified and act according to the specification. Moreover, for cryptographic protocols, *randomization* is a crucial effect needed to implement secure protocols, and cryptographic security is many times phrased as an *observational equivalence* of two probabilistic programs—one implementing the actual system and one implementing an "ideal" and trivially secure system. In Chapter 7 we take the first steps towards modular reasoning principles for such protocols by developing a relational higher-order separation logic for reasoning about the contextual equivalence of probabilistic sequential programs.

Probabilistic couplings [Lin02; Tho00; Vil08] is a mathematical tool for reasoning about pairs of probabilistic processes, and the method has been applied in a variety of settings to relate probabilistic programs, *e.g.*, in pRHL [Bar+15], Polaris [TH19], and HO-RPL [Agu+21]. In Chapter 7, we will establish *step-wise* probabilistic couplings between the distribution of individual execution steps through a relational separation logic. In this section, we give a high-level intuition for how this mechanism semantically works.

To account for the non-terminating behavior of programs, we will work with probability sub-distributions over discrete countable sets to define our operational semantics.

Definition 1.5.1 (Sub-distribution)**.** A (discrete) *sub-distribution* over a countable set A is a function $\mu : A \to [0,1]$ such that $\sum_{a \in A} \mu(a) \leq 1$. We write $\mathcal{D}(A)$ for the set of all sub-distributions over A.

For example, the function $\mu_{\mathsf{coin}}(b \in \mathbb{B}) \triangleq \frac{1}{2}$ denotes the distribution of a fair coin flip—a uniform distribution on Booleans that gives equal probability to both heads and tails. The *support* of a sub-distribution $\mu \in \mathcal{D}(A)$ is the set of elements $\mathrm{supp}(\mu) = \{a \in A \mid \mu(a) > 0\}$ and $\mathcal{D}$ can be given monadic structure.

Lemma 1.5.2 (Probability Monad)**.** *Let $\mu \in \mathcal{D}(A)$, $a \in A$, and $f : A \to \mathcal{D}(B)$. Then*

1. $\mathrm{bind}(f, \mu)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$
2. $\mathrm{ret}(a)(a') \triangleq \begin{cases} 1 & \text{if } a = a' \\ 0 & \text{otherwise} \end{cases}$

gives sub-distributions monadic structure. We write $\mu \gg\!= f$ for $\mathrm{bind}(f, \mu)$.

Probabilistic couplings are useful to prove relations between probability distributions. Intuitively, couplings correlate outputs of two processes by coordinating corresponding samples through a joint distribution with marginals that are equal to the processes being related.

Definition 1.5.3 (Coupling)**.** Let $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$. A sub-distribution $\mu \in \mathcal{D}(A \times B)$ is a *coupling* of μ_1 and μ_2 if

1. $\forall a.\ \sum_{b \in B} \mu(a, b) = \mu_1(a)$
2. $\forall b.\ \sum_{a \in A} \mu(a, b) = \mu_2(b)$

Given relation $R : A \times B$ we say μ is an R-coupling if furthermore $\mathrm{supp}(\mu) \subseteq R$. We write $\mu_1 \sim \mu_2 : R$ if there exists an R-coupling of μ_1 and μ_2.

For example, the distribution $\mu_{\mathsf{coins}} \in \mathcal{D}(\mathbb{B} \times \mathbb{B})$ where

$$\mu_{\mathsf{coins}}(b_1, b_2) \triangleq \begin{cases} \frac{1}{2} & \text{if } b_1 = b_2 \\ 0 & \text{otherwise} \end{cases}$$

is a witness of the coupling $\mu_{coin} \sim \mu_{coin} : (=)$ as can be easily verified. Once a coupling has been established, we can often extract a concrete relation from it between the probability distributions. In particular, for $(=)$-couplings, we have that if $\mu_1 \sim \mu_2 : (=)$, then $\mu_1 = \mu_2$.

One of the key strengths of the coupling approach is that they can be derived piece by piece as couplings can be constructed and composed along the monadic structure of the sub-distribution monad, which will allow us to do compositional proofs.

Lemma 1.5.4 (Composition of couplings)**.** *Let* $R : A \times B$, $S : A' \times B'$, $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$, $f_1 : A \to \mathcal{D}(A')$, *and* $f_2 : B \to \mathcal{D}(B')$.

1. *If* $(a, b) \in R$ *then* $\mathrm{ret}(a) \sim \mathrm{ret}(b) : R$.
2. *If* $\forall (a, b) \in R.\ f_1(a) \sim f_2(b) : S$ *and* $\mu_1 \sim \mu_2 : R$ *then* $\mu_1 \gg\!= f_1 \sim \mu_2 \gg\!= f_2 : S$

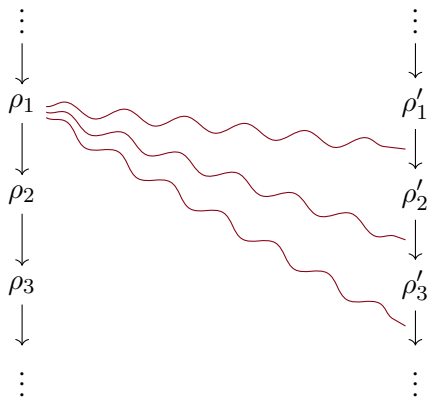
For example, if we were to construct a coupling for a system that uses a fair coin flip, say, between $\mu_{\mathsf{coin}} \gg\!= f_1$ and $\mu_{\mathsf{coin}} \gg\!= f_2$ for some f_1 and f_2, the second condition will allow us to continue reasoning about $f_1(b)$ and $f_2(b)$ since $\mu_1 \sim \mu_2 : (=)$, *i.e.*, as if the two coin flips had had the same outcome. This is a powerful method that we will incorporate into our logic and it integrates well with existing reasoning principles from non-probabilistic relational logics.

In Chapter 7, we develop a higher-order probabilistic relational separation logic named Clutch. The main component of Clutch is a *logical refinement judgment* $\vDash e_1 \precsim e_2 : \tau$. Intuitively, this judgment will imply that the expression e_1 contextually refines the expression e_2 at type τ, which means that, for all well-typed program contexts $\mathcal{C}$ expecting something of type τ, then the termination probability of $\mathcal{C}[\,e_1\,]$ is bounded by the termination probability of $\mathcal{C}[\,e_2\,]$. Semantically, this is established through a logical-relations model built on top of a new program logic that constructs probabilistic couplings between the distribution of *individual* computation steps of the two programs. In the logic's soundness theorem, we construct a coupling between the full program executions by composing the individual couplings along the monadic structure of the sub-distribution monad using Lemma 1.5.4.

The refinement judgment satisfies a range of structural and computational rules. For example, the following rule allows us to "symbolically execute" the right-hand side program for pure execution steps, *i.e.*, steps that do not involve state.

$$\frac{e_1' \overset{\text{pure}}{\rightsquigarrow} e_2' \qquad \vDash e_1 \precsim K[\,e_2'\,] : \tau}{\vDash e_1 \precsim K[\,e_1'\,] : \tau}$$

where K denotes an evaluation context. The semantic justification of this rule constructs a trivial coupling between the distribution $\mathrm{ret}(e_1, \sigma_1)$ and the one-step program execution distribution $\mathrm{step}(e_1', \sigma_1')$ where σ_1 and σ_1' are the states of the programs. The couplings that are constructed from successive applications of this rule are illustrated by the diagram below.



The diagram illustrates the evolution of the symbolic configuration of the two programs as considered during proofs. The red squiggly line represents a coupling. When applying the rule for pure reductions on the right-hand side, we fix the left-hand side but progress the right-hand side independently using a trivial coupling witnessed by the product distribution.

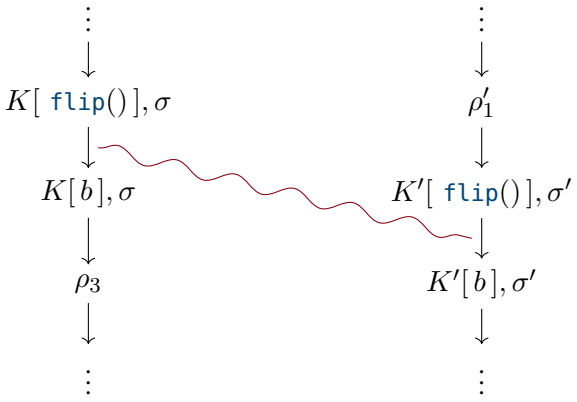
The language considered in Chapter 7 has a single probabilistic primitive `flip` that reduces uniformly at random to either `true` or `false`, just like μ_{coin}. To relate two `flip`s across two programs, we construct a coupling similar to μ_{coins} as embodied by the following rule

$$\frac{\forall b.\ \vDash K[\,b\,] \precsim K'[\,b\,] : \tau}{\vDash K[\,\texttt{flip}()\,] \precsim K'[\,\texttt{flip}()\,] : \tau}$$

which allows us to continue reasoning as if the two sampling statements had the same outcome. More concretely, the semantic justification of this rule constructs a coupling

$$\mathrm{step}(\texttt{flip}(), \sigma) \sim \mathrm{step}(\texttt{flip}(), \sigma') : R$$

where $R(\rho, \rho') \triangleq \exists b \in \mathbb{B}.\ \rho = (b, \sigma) \land \rho' = (b, \sigma')$ with a witness similar to μ_{coins}. An application of this rule is illustrated diagrammatically below.



Notice how applying the coupling rule allows the proof to progress as if the outcomes of the samplings are the same b on both sides.

Coupling rules that relate two sampling statements are useful and powerful as witnessed by its many application in pRHL-like logics [Agu+21; Bar+15; Bar+18; Bar+16a; Bar+16b; Bar+12].

However, this kind of coupling rule requires aligning or "synchronizing" the sampling statements of the two programs: both programs have to be executing the sample statements we want to couple for their next step. Nevertheless, it is not always possible with the rules from existing logics. For example, consider the following program that *eagerly* performs a probabilistic coin flip and returns the result in a thunk:

$$\texttt{let}\, b = \texttt{flip}()\, \texttt{in}\, \lambda\_.\, b$$

An indistinguishable—but *lazy*—version of the program does the probabilistic coin flip only when the thunk is invoked for the first time, and then stores the result in a reference that is read from in future invocations:

$$\begin{array}{l}\texttt{let}\, r = \texttt{ref}(\mathsf{None})\, \texttt{in}\\ \lambda\_.\, \texttt{match}\, !\, r\, \texttt{with}\\ \quad \mathsf{Some}\,(b) \Rightarrow b\\ \quad \mid \mathsf{None} \quad \Rightarrow \texttt{let}\, b = \texttt{flip}()\, \texttt{in}\\ \qquad\qquad\quad\; r \leftarrow \mathsf{Some}\,(b);\\ \qquad\qquad\quad\; b\\ \quad \texttt{end}\end{array}$$

The $\texttt{flip}()$ expression in the first program is evaluated immediately, but the $\texttt{flip}()$ expression in the second program only gets evaluated when the thunk is invoked. To relate the two thunks, one is forced first to symbolically evaluate the eager sampling, but this makes it impossible to construct a coupling with the lazy sampling on the right.

Clutch supports *asynchronous probabilistic couplings* to resolve this issue by introducing a novel kind of ghost state called *presampling tapes*. Presampling tapes let us reason about sampling statements as if they were executed ahead of time and stored their results for later use. Presampling tapes surface in the program's syntax and state but can be entirely erased through refinement, as shown and discussed in Chapter 7.

Operationally, a tape is a finite sequence of Booleans, representing future outcomes of specific `flip` commands. Each tape is labeled with an identifier ι, and a program's state is extended with a finite map from labels to tapes. A fresh tape can be dynamically (and deterministically) allocated using a `tape` language primitive:

$$\texttt{tape}, \sigma \to^1 \iota, \sigma[\iota \mapsto \epsilon] \qquad\qquad \text{if } \iota = \mathrm{fresh}(\sigma)$$

which extends the mapping with an empty tape and returns its fresh label ι. The `flip` primitive can then be annotated with a tape label ι. If the corresponding tape is empty, $\texttt{flip}(\iota)$ reduces uniformly:

$$\texttt{flip}(\iota), \sigma \to^{1/2} b, \sigma \qquad\qquad \text{if } \sigma(\iota) = \epsilon \text{ and } b \in \{\texttt{true}, \texttt{false}\}$$

but if the tape is *not* empty, $\texttt{flip}(\iota)$ reduces deterministically by taking off the first element of the tape and returning it:

$$\texttt{flip}(\iota), \sigma[\iota \mapsto b \cdot \vec{b}] \to^1 b, \sigma[\iota \mapsto \vec{b}]$$

However, *no* primitives in the language operationally add values to the tapes. The key technical constructions *of the logic*, however, allows us to add coupled presampled values to the tapes.

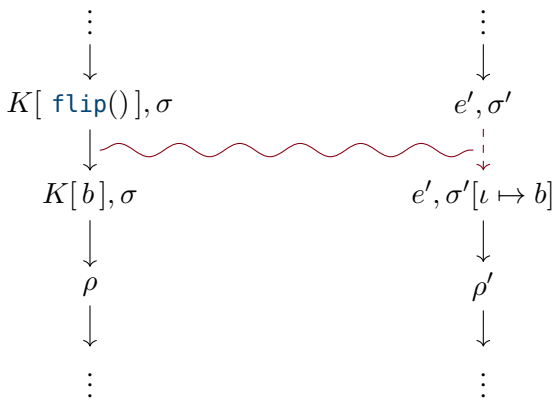
The Clutch logic comes with a $\iota \hookrightarrow \vec{b}$ assertion that denotes *ownership* of the tape ι and its contents $\vec{b}$, analogously to how the traditional points-to-connective $\ell \mapsto v$ of separation logic denotes ownership of the location ℓ and its contents on the heap. By owning a tape ι we can couple samplings onto ι with a physical $\texttt{flip}()$ on the opposite side as enabled by rules like

$$\frac{\iota \hookrightarrow \vec{b} \qquad \forall b.\, \iota \hookrightarrow (\vec{b} \cdot b) \twoheadrightarrow \vDash K[\,b\,] \precsim e : \tau}{\vDash K[\,\texttt{flip}()\,] \precsim e : \tau}$$

The rule gives back the ownership of $\iota \hookrightarrow (\vec{b} \cdot b)$ and requires us to show $\vDash K[\,b\,] \precsim e : \tau$. When we, in the future, encounter a sampling statement with label ι on the right-hand side we simply read off the presampled b from the tape.

$$\frac{\iota \hookrightarrow b \cdot \vec{b} \qquad \iota \hookrightarrow \vec{b} \twoheadrightarrow \vDash e_1 \precsim K[\,b\,] : \tau}{\vDash e_1 \precsim K[\,\texttt{flip}(\iota)\,] : \tau}$$

An application of the asynchronous coupling rule is illustrated diagrammatically below.



Notice how the coupling is established between a physical sampling on the left and a *ghost sampling* (indicated with a dashed red arrow) on the right for an arbitrary configuration (e', σ'). The justification of this asynchronous coupling rule is a coupling

$$\mathrm{step}(\texttt{flip}(), \sigma) \sim \mathrm{step}_\iota(\sigma') : R$$

where $\mathrm{step}_\iota(\sigma)$ is the distribution that with equal probability adds either $\texttt{true}$ or $\texttt{false}$ to the ι tape in σ and $R(\rho, \sigma'') \triangleq \exists b \in \mathbb{B}.\, \rho = (b, \sigma) \land \sigma'' = \sigma'[\iota \mapsto b]$. This allows us to continue reasoning as if the program sampled the same value as was presampled onto the tape. The key to the soundness theorem is that these tape samplings do not affect the final result of program execution—they merely act as a logical proof device without any operational effect. Chapter 7 provides more details on how this is proved, and we demonstrate the approach's usefulness with several case studies.

2 Academic Contributions

This dissertation contains text and material from the publications listed below.

Refereed publications

[Kro+20a] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. "Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Proceedings*. 2020, pp. 336–365. DOI: 10.1007/978-3-030-44914-8_13

[Gon+21a] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. "Distributed causal memory: Modular specification and verification in higher-order distributed separation logic". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434323

[Gre+21a] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. "Mechanized logical relations for termination-insensitive noninterference". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434291

Manuscripts

[Tim+21] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. "Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic". In: *CoRR* abs/2109.07863 (2021). arXiv: 2109.07863

[Gre+23] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. "Asynchronous Probabilistic Couplings in Higher-Order Separation Logic". In: *CoRR* abs/2301.10061 (2023). arXiv: 2301.10061

During the time span of my Ph.D. studies I published a peer-reviewed paper listed below that is not included as the topic is tangential to the focus of the dissertation.

Refereed publications not included in this thesis

[GTA19] Simon Oddershede Gregersen, Søren Eller Thomsen, and Aslan Askarov. "A Dependently Typed Library for Static Information-Flow Control in Idris". In: *Principles of Security and Trust - 8th International Conference, POST 2019, Proceedings*. 2019, pp. 51–75. DOI: 10.1007/978-3-030-17138-4_3

The following list details the origin of the text of the individual chapters and sections.

- Chapter 3 reproduces Krogh-Jespersen et al. [Kro+20a] with the additional inclusion of Section 3.5 and extensions to Sections 3.6 and 3.7 using material from the technical appendix [Kro+20b] accompanying the paper.
- Chapter 4 reproduces Gondelman et al. [Gon+21a] with the addition of Figure 4.4 from the technical appendix [Gon+21b] accompanying the paper. Appendix A and B are reproduced verbatim from the technical appendix.
- Chapter 5 is a heavily revised version of the Timany et al. [Tim+21] manuscript.
- Chapter 6 reproduces Gregersen et al. [Gre+21a] with extensions to Section 6.1 using material from the technical appendix [Gre+21b] accompanying the paper.
- Chapter 7 reproduces Gregersen et al. [Gre+23]. Appendix C to F are reproduced verbatim from this work.

2.1 Collaborations and contributions

The work presented in this dissertation is the result of collaborative projects. The individual chapters of the dissertation are presented in (almost) chronological order, with the extent of my[1] contributions increasing throughout. Below I explain the nature of my contributions to each chapter in terms of both scientific work and writing.

The initial development of the Aneris logic, as described in Chapter 3, was led by Morten Krogh-Jespersen, and I joined the project in its later stages. I contributed the case study presented in Section 3.5 and revised multiple aspects of the paper and its presentation, eventually leading to its publication.

In collaboration with Léon Gondelman, I subsequently lifted the duplicate protection assumption from the initial Aneris development, developed new logical machinery for a new and refined logic, and improved multiple aspects of the technical development. This work culminated in a substantial verification effort presented in Chapter 4, which was led by Léon and Amin Timany. I partook in technical discussions, independently developed the client-side examples presented in Section 4.4 and Appendix A, and contributed to the writing phase.

The Trillium program logic presented in Chapter 5 is a joint achievement, where my main technical contribution is the TLA$^+$ axis presented in Section 5.4. In particular, I independently implemented and carried out the refinement proofs for the two-phase commit protocol and single-decree Paxos, including proving the correctness of the transcribed TLA$^+$ models in the Coq proof assistant. In addition, I was the lead on the majority of the writing phase.

I spear-headed all aspects of the work presented in Chapters 6 and 7.

[1] Only in this section I will use the first person singular to refer specifically to my own contributions in contrast to those of my collaborators. Following common scientific practice, I otherwise use the first person plural to refer to work done by the author, whether done in collaboration with others or not.

2.2 Coq mechanization

All of the work presented in this dissertation is *foundational* [App01] in that all results, including the operational semantics, program logics, underlying mathematics, and all the examples, are formalized in the Coq proof assistant [Coq22]. This gives the highest assurance that all details and results are correct. In addition, all the developments are open source and available online through the links below.

Chapter 3	`https://iris-project.org/artifacts/2020-esop-aneris.tar.gz`
Chapters 4 and 5	`https://github.com/logsem/aneris`
Chapter 6	`https://github.com/logsem/iris-tini`
Chapter 7	`https://github.com/logsem/clutch`

Part II

Publications

3 Aneris

Abstract

Building network-connected programs and distributed systems is a powerful way to provide scalability and availability in a digital, always-connected era. However, with great power comes great complexity. Reasoning about distributed systems is well-known to be difficult.

In this work we present Aneris, a novel framework based on separation logic supporting modular, node-local reasoning about concurrent and distributed systems. The logic is higher-order, concurrent, with higher-order store and network sockets, and is fully mechanized in the Coq proof assistant. We use our framework to verify an implementation of a load balancer that uses multi-threading to distribute load amongst multiple servers and an implementation of the *two-phase-commit* protocol with a replicated logging service as a client. The two examples certify that Aneris is well-suited for both horizontal and vertical modular reasoning.

Reasoning about distributed systems is notoriously difficult due to their sheer complexity. This is largely the reason why previous work has traditionally focused on verification of protocols of core network components. In particular, in the context of model checking where safety and liveness assertions [Pnu77] are considered, tools such as SPIN [Hol97], TLA+ [Lam92], and Mace [Kil+07] have been developed. More recently, significant contributions have been made in the field of formal proofs of *implementations* of challenging protocols, such as two-phase-commit, lease-based key-value stores, Paxos, and Raft [Haw+15; LBC16; Rah+15; SWT18; Wil+15]. All of these developments define domain specific languages (DSLs) specialized for distributed systems verification. Protocols and modules proven correct can be compiled to an executable, often relying on some trusted code-base.

Formal reasoning about distributed systems has often been carried out by giving an abstract model in the form of a *state transition system* or *flow-chart* in the tradition of Floyd [Flo67] and Lamport [Lam77; Lam78]. A state is normally taken to be a view of the global state and events are observable changes to this state. State transition systems are quite versatile and have been used in other verification applications. However, reasoning based on state transition systems often suffer from a lack of modularity due to their very global perspective. As a consequence, separate nodes or components cannot be verified in isolation and the system has to be verified as a whole.

IronFleet [Haw+15] is the first system that supports node-local reasoning for verifying the implementation of programs that run on different nodes. In IronFleet, a distributed system is modeled by a transition system. This transition system is shown to be refined by the composition of a number of transition systems, each pertaining to one of the nodes in the system.

Each node in the distributed system is shown to be correct and a refinement of its corresponding transition system. Nevertheless, IronFleet does not allow you to reason compositionally; a correctness proof for a distributed system cannot be used to show the correctness of a larger system.

Higher-order concurrent separation logics [Din+13; Din+10; Jun+16; Jun+15; Kre+17; LN13; Nan+14; OHe07; RDG14; SNB15; SB14; TDB13] simplify reasoning about higher-order imperative concurrent programs by offering facilities for specifying and proving correctness of programs in a modular way. Indeed, their support for modular reasoning (a.k.a. compositional reasoning) is the key reason for their success. Disel [SWT18] is a separation logic that does support compositional reasoning about distributed systems, allowing correctness proofs of distributed systems to be used for verifying larger systems. However, Disel struggles with node-local reasoning in that it cannot hide node-local usage of mutable state. That is, the use of internal state in nodes must be exposed in the high-level protocol of the system and changes to the internal state are only possible upon sending and receiving messages over the network.

Finally, both Disel and IronFleet restrict nodes to run only sequential programs and no node-level concurrency is supported.

In this work we present Aneris, a framework for implementing and reasoning about functional correctness of distributed systems. Aneris is based on concurrent separation logic and supports modular reasoning with respect to both nodes (node-local reasoning) and threads within nodes (thread-local reasoning). The Aneris framework consists of a programming language, AnerisLang, for writing realistic, real-world distributed systems and a higher-order concurrent separation logic for reasoning about these systems. AnerisLang is a concurrent ML-like programming language with higher-order functions, local state, threads, and network primitives. The operational semantics of the language, naturally, involves multiple hosts (each with their own heap and multiple threads) running in a network. The Aneris logic is build on top of the Iris framework [Jun+16; Jun+15; Kre+17] and supports machine-verified formal proofs in the Coq proof assistant about distributed systems written in AnerisLang.

Networking. There are several ways of adding network primitives to a programming language. One approach is *message-passing* using first-class communication channels á la the π-calculus or using an implementation of the actor model as done in high-level languages like Erlang, Elixir, Go, and Scala. However, any such implementation is an abstraction built on top of network sockets where all data has to be serialized, data packets may be dropped, and packet reception may not follow the transmission order. Network sockets are a quintessential part of building efficient, real-world distributed systems and all major operating systems provide an application programming interface (API) to them. Likewise, AnerisLang provides support for datagram-like sockets by directly exposing a simple API with the core methods necessary for socket-based communication using the User Datagram Protocol (UDP) with duplicate protection. This allows for a wide range of real-world systems and protocols to be implemented (and verified) using the Aneris framework.

Modular reasoning in Aneris. In general, there are two different ways to support modular reasoning about distributed systems corresponding to how components can be composed. Aneris enables simultaneously both:

- *Vertical composition*: when reasoning about programs within each node, one is able to compose proofs of different components to prove correctness of the whole program. For

instance, the specification of a verified data structure, e.g. a concurrent queue, should suffice for verifying programs written against that data structure, independently of its implementation.

- *Horizontal composition*: at each node, a verified thread is composable with other verified threads. Similarly, a verified node is composable with other verified nodes which potentially engage in different protocols. This naturally aids implementing and verifying large-scale distributed systems.

Node-local variants of the standard rules of CSLs like, for example, the bind rule and the frame rule (as explained in Section 3.1) enable vertical reasoning. Section 3.6 showcases vertical reasoning in Aneris using a replicated distributed logging service that is implemented and verified using a separate implementation and specification of the two-phase commit protocol.

Horizontal reasoning in Aneris is achieved through the Thread-par-rule and the Node-par-rule (further explained in Section 3.1) which intuitively says that to verify a distributed system, it suffices to verify each thread and each node in isolation. This is analogous to how CSLs allow us to reason about multi-threaded programs by considering individual threads in isolation; in Aneris we extend this methodology to include both threads and nodes. Where most variants of concurrent separation logic use some form of an invariant mechanism to reason about shared-memory concurrency, we abstract the communication between nodes over the network through *socket protocols* that restrict what can be sent and received on a socket and allow us to share ownership of logical resources among nodes. Section 3.4 showcases horizontal reasoning in Aneris using an implementation and a correctness proof for a simple addition service that uses a load balancer to distribute the workload among several addition servers. Each node is verified in isolation and composed to form the final distributed system.

Contributions. In summary, we make the following contributions:

- We present AnerisLang, a formalized higher-order functional programming language for writing distributed systems. The language features higher-order store, node-local concurrency, and network sockets, allowing for dynamic creation and binding of sockets to addresses with serialization and deserialization primitives for encoding and parsing messages.
- We define the Aneris logic, the first higher-order concurrent separation logic with support for network sockets and with support for node-local and thread-local reasoning.
- We introduce a simple and novel approach to specifying network protocols; a mechanism that supports separation-logic-style modular specifications of distributed systems.
- We conduct two case studies that showcase how our framework aids the implementation and verification of real-world distributed systems using compositional reasoning:
  - A replicated logging service that is implemented and verified using a separate implementation and specification of the two-phase commit protocol, demonstrating vertical compositional reasoning.
  - A load balancer that distributes work on multiple servers by means of node-local multi-threading. We use this to verify a simple addition service that uses the load

balancer to distribute its requests over multiple servers, demonstrating horizontal compositional reasoning.

We have formalized all of the theory and examples on top of Iris in the Coq proof assistant using the MoSeL framework [KTB17].

Outline. We start by describing the core concepts of the Aneris framework (Section 3.1). We then describe the AnerisLang programming language (Section 3.2) before presenting the Aneris logic proof rules and stating our adequacy theorem, *i.e.*, soundness of Aneris (Section 3.3). Subsequently, we use the logic to verify a load balancer (Section 3.4), a bag service (Section 3.5), and a two-phase-commit implementation with a replicated logging client (Section 3.6). Finally, we discuss related work in (Section 3.8 and conclude (Section 3.9).

3.1 The core concepts of Aneris

In this section we present our methodology to modular verification of distributed systems. We begin by recalling the ideas of thread-local reasoning and protocols from concurrent separation logic and explain how we lift those ideas to *node-local* reasoning. Finally, we illustrate the Aneris methodology for specifying, implementing, and verifying distributed systems by developing a simple addition service and a lock server. The distributed systems are composed of individually verified concurrently running nodes communicating asynchronously by exchanging messages that can be reordered or dropped.

3.1.1 Local and thread-local reasoning

The most important feature of (concurrent) separation logic is, arguably, how it enables scalable modular reasoning about pointer-manipulating programs. Separation logic is a resource logic, in the sense that propositions denote not only facts about the state, but *ownership* of resources. Originally, separation logic [Rey02] was introduced for modular reasoning about the heap—i.e. the notion of resource was fixed to be logical pieces of the heap. The essential idea is that we can give a local specification $\{P\}\ e\ \{v.Q\}$ to a program e involving only the *footprint* of e. Hence, while verifying e, we need not consider the possibility that another piece of code in the program might interfere with e; the program e can be verified without concern for the environment in which e may occur. Local specifications can then be lifted to more global specifications by framing and binding:

$$\frac{\{P\}\ e\ \{v.Q\}}{\{P * R\}\ e\ \{v.Q * R\}} \qquad \frac{\{P\}\ e\ \{v.Q\} \qquad \forall v.\{Q\}\ K[v]\ \{w.R\}}{\{P\}\ K[e]\ \{w.R\}}$$

where K denotes an evaluation context. The symbol $*$ denotes separating conjunction. Intuitively, $P * Q$ holds for a given resource (in this case a heap) if it can be divided into two disjoint resources such that P holds for one and Q holds for the other. Thus, the frame rule essentially says that executing e for which we know $\{P\}\ e\ \{x.Q\}$ cannot possibly affect parts of the heap that are *separate* from its footprint. Another related separation logic connective is $-\!\!*$, the separating implication. Proposition $P\ -\!\!*\ Q$ describes a resource that, combined with a disjoint resource satisfying P, results in a resource satisfying Q.

Since its introduction, separation logic has been extended to resources beyond heaps and with more sophisticated mechanisms for modular control of interference. Concurrent separation logics (CSLs) [OHe07] allow reasoning about concurrent programs and a preeminent feature of these program logics is again the support for modular reasoning, in this case with respect to concurrency through *thread-local* reasoning. When reasoning about a concurrent program we consider threads one at a time and need not reason about interleavings of threads explicitly. In a way, our frame here includes, in addition to the shared fragments of the heap and other resources, the execution of other threads which can be interleaved throughout the execution of the thread being verified. This can be seen from the following disjoint concurrency rule:

THREAD-PAR
$$\frac{\{P_1\}\ e_1\ \{v.Q_1\} \qquad \{P_2\}\ e_2\ \{v.Q_2\}}{\{P_1 * P_2\}\ e_1\ ||\ e_2\ \{v.\exists v_1, v_2.v = (v_1, v_2) * Q_1[v_1/v] * Q_2[v_2/v]\}}$$

where $e_1\ ||\ e_2$ denotes parallel composition of expressions e_1 and e_2.[1]

Inevitably, at some point threads typically have to communicate with one another through some kind of shared state, an unavoidable form of interference. The original CSL used a simple form of resource invariant in which ownership of a shared resource can be transferred between threads.

A notable program logic in the family of concurrent separation logics is Iris that is specifically designed for reasoning about programs written in concurrent higher-order imperative programming languages. Iris has already proven to be versatile for reasoning about a number of sophisticated properties of programming languages [Jun+18a; Kai+17b; Tim+18]. In order to support modular reasoning about concurrent programs Iris features (1) *impredicative invariants* for expressing protocols on shared state among multiple threads and (2) allows for encoding of *higher-order ghost state* using a form of partial commutative monoids for reasoning about resources. We will give examples of these features and explain them in more detail as needed.

3.1.2 Node-local reasoning

Programs written in AnerisLang are higher-order imperative concurrent programs that run on multiple nodes in a distributed system. When reasoning about distributed systems in Aneris, alongside heap-local and thread-local reasoning, we also reason *node-locally*. When proving correctness of AnerisLang programs we reason about each node of the system in isolation, akin to how we in CSLs reason about each thread in isolation.

By virtue of building on Iris, reasoning in Aneris is naturally modular with respect to separation logic frames and with respect to threads. What Aneris adds on top of this is support for *node-local* reasoning about programs. This is expressed by the following rule:

NODE-PAR
$$\frac{\begin{array}{l}\{P_1 * \mathsf{IsNode}(n_1) * \mathsf{FreePorts}(ip_1, \mathfrak{P})\}\ \langle n_1;\ e_1\rangle\ \{\mathsf{True}\} \\ \{P_2 * \mathsf{IsNode}(n_2) * \mathsf{FreePorts}(ip_2, \mathfrak{P})\}\ \langle n_2;\ e_2\rangle\ \{\mathsf{True}\}\end{array}}{\{P_1 * P_2 * \mathsf{FreeIp}(ip_1) * \mathsf{FreeIp}(ip_2)\}\ \langle \mathfrak{S};\ (n_1; ip_1; e_1)\ |||\ (n_2; ip_2; e_2)\rangle\ \{\mathsf{True}\}}$$

[1] In a language with fork-based concurrency, the parallel composition operator is an easily defined construct and the rule is derivable from a more general fork-rule.

where $|||$ denotes parallel composition of two nodes with identifier n_1 and n_2 running expressions e_1 and e_2 with IP addresses ip_1 and ip_2.[2] The set $\mathfrak{P} = \{p \mid 0 \leq p \leq 65535\}$ denotes a finite set of ports.

Note that only a distinguished system node $\mathfrak{S}$ can start new nodes (as elaborated on in Section 3.2). In Aneris, the execution of the distributed system starts with the execution of $\mathfrak{S}$ as the only node in the system. In order to start a new node associated with ip address ip one provides the resource $\mathsf{FreeIp}(ip)$ which indicates that ip is not used by other nodes. The node can then rely on the fact that when it starts, all ports on ip are available. The resource $\mathsf{IsNode}(n)$ indicates that the node n is a node in the system and keeps track of abstract state related to our modeling of node n's heap and allocated sockets. To facilitate modular reasoning, free ports can be split: if $A \cap B = \emptyset$ then

$$\mathsf{FreePorts}(ip, A) * \mathsf{FreePorts}(ip, B) \dashv\vdash \mathsf{FreePorts}(ip, A \cup B)$$

where $\dashv\vdash$ denotes logical equivalence of Aneris propositions (of type *iProp*). We will use $\mathsf{FreePort}(a)$ as shorthand for $\mathsf{FreePorts}(ip, \{p\})$ where $a = (ip, p)$.

Finally, observe that the node-local postconditions are simply True, in contrast to the arbitrary thread-local postconditions in the Thread-par-rule that carry over to the main thread. In the concurrent setting, shared memory provides reliable communication and synchronization between the child threads and the main thread; in the rule for parallel thread composition, the main thread will wait for the two child processes to finish. In the distributed setting, there are no such guarantees and nodes are separate entities that cannot synchronize with the distinguished system node.

Socket protocols. Similar to how classical CSLs introduce the concept of resource invariants for expressing protocols on shared state among multiple threads, we introduce the simple and novel concept of *socket protocols* for expressing protocols among multiple nodes. With each socket address—a pair of an IP address and a port—a protocol is associated, which restricts what can be communicated on that socket.

A socket protocol is a predicate $\Phi : \mathit{Message} \to \mathit{iProp}$ on incoming messages received on a particular socket. One can think of this as a form of rely-guarantee reasoning since the socket protocol will be used to restrict the distributed environment's interference with a node on a particular socket. In Aneris we write $a \Rrightarrow \Phi$ to mean that socket address a is governed by the protocol Φ. In particular, if $a \Rrightarrow \Phi$ and $a \Rrightarrow \Psi$ then Φ and Ψ are equivalent.[3] Moreover, the proposition is duplicable: $a \Rrightarrow \Phi \dashv\vdash a \Rrightarrow \Phi * a \Rrightarrow \Phi$.

Conceptually, a socket is an abstract representation of a handle for a local endpoint of some channel. We further restrict channels to use the User Datagram Protocol (UDP) which is *asynchronous*, *connectionless*, and *stateless*. In accordance with UDP, Aneris provides no guarantee of delivery or ordering although we assume duplicate protection. We assume duplicate protection to simplify our examples, as otherwise the code of all of our examples would have to be adapted to cope with duplication of messages. One can think of sockets in Aneris as open-ended multi-party communication channels without synchronization.

It is noteworthy that inter-process communication can happen in two ways. Thread-concurrent programs can communicate both through the shared heap and by sending messages

[2] In the same way as the parallel composition rule is derived from a more general fork-based rule, this composition rule is also an instance of a more general rule for spawning nodes shown in Section 3.2.

[3] The predicate equivalence is under a later modality in order to avoid self-referential paradoxes. We omit it for the sake of presentation as this is an orthogonal issue.

through sockets. For memory-separated programs running on different nodes all communication is by message-passing over an unreliable network.

In the logic, we consider both *static* and *dynamic* socket addresses. This distinction is entirely abstract and at the level of the logic. Static addresses come with primordial protocols, agreed upon before starting the distributed system, whereas dynamic addresses do not. Protocols on static addresses are primarily intended for addresses pointing to nodes that offer a service.

To distinguish between static and dynamic addresses, we use a resource $\mathsf{Fixed}(A)$ which denotes that the addresses in A are static and should have a fixed interpretation. This proposition expresses knowledge without asserting ownership of resources and is duplicable:

$$\mathsf{Fixed}(A) \dashv\vdash \mathsf{Fixed}(A) * \mathsf{Fixed}(A).$$

Corresponding to the two kinds of addresses we have two different rules, SocketBind-static and SocketBind-dynamic, for binding an address to a socket as seen below. Both rules consume an instance of $\mathsf{Fixed}(A)$ and $\mathsf{FreePort}(a)$ as well as a resource $z \hookrightarrow_n \bot$. The latter keeps track of the address associated with the socket handle z on node n and ensures that the socket is bound only once as further explained in Section 3.3. Notice that the protocol Φ in SocketBind-dynamic can be freely chosen.

SocketBind-static
$$\{\mathsf{Fixed}(A) * a \in A * \mathsf{FreePort}(a) * z \hookrightarrow_n \bot\}$$
$$\langle n; \texttt{socketbind}\ z\ a\rangle$$
$$\{x.\, x = 0 * z \hookrightarrow_n a\}$$

SocketBind-dynamic
$$\{\mathsf{Fixed}(A) * a \notin A * \mathsf{FreePort}(a) * z \hookrightarrow_n \bot\}$$
$$\langle n; \texttt{socketbind}\ z\ a\rangle$$
$$\{x.\, x = 0 * z \hookrightarrow_n a * a \Rrightarrow \Phi\}$$

In the remainder of the chapter we will use the following shorthands in order to simplify the presentation of our specifications.

$$\mathsf{Static}(a, A, \Phi) \triangleq \mathsf{Fixed}(A) * a \in A * \mathsf{FreePort}(a) * a \Rrightarrow \Phi$$
$$\mathsf{Dynamic}(a, A) \triangleq \mathsf{Fixed}(A) * a \notin A * \mathsf{FreePort}(a)$$

3.1.3 Example: An addition service

To illustrate node-local reasoning, socket protocols, and the Aneris methodology for specifying, implementing, and verifying distributed systems we develop a simple addition service that offers to add numbers for clients.

Figure 3.1 depicts an implementation of a server and a client written in AnerisLang. Notice that the programs look as if they were written in a realistic functional language with sockets like OCaml. Messages are strings to make programming with sockets easier (similar to `send_substring` in the Unix module in OCaml).

The server is parameterized over an address on which it will listen for requests. The server allocates a new socket and binds the address to the socket. Then the server starts listening for

an incoming message on the socket, calling a handler function on the message, if any. The handler function will deserialize the message, perform the addition, serialize the result, and return it to the sender before recursively listening for new messages.

The client is parameterized over two numbers to compute on, a server address, and a client address. The client allocates a new socket, binds the address to the socket, and serializes the two numbers. In the end, it sends the serialized message to the server address and waits for a response, deserializing the result of the addition on arrival.

```
let rec listen skt handler =
  match receivefrom skt with
  | Some m => handler (fst m) (snd m)
  | None => listen skt handler
  end

let server a =
  let skt = socket () in
  socketbind skt a;
  let rec handler msg from =
    let (x, y) = deserialize msg in
    let res = serialize (x + y) in
    sendto skt res from
  in listen skt handler
```

```
let client x y srv a =
  let skt = socket () in
  socketbind skt a;
  let m = serialize (x, y) in
  sendto skt m srv;
  listen skt (fun m _ -> deserialize m)
```

Figure 3.1: An implementation of an addition service and a client written in AnerisLang.

In order to give the server code a specification we will fix a primordial socket protocol that will govern the address given to the server. The protocol will spell out how the server relies on the socket. We will use $m.\mathsf{orig}$ and $m.\mathsf{body}$ for projections of the sender and the message body, respectively, from the message m. We define Φ_{add} as follows:

$$\Phi_{add}(m) \triangleq \exists \Psi, x, y.\, m.\mathsf{orig} \Mapsto \Psi * m.\mathsf{body} = \mathit{serialize}(x, y) * \\ \forall m', m'.\mathsf{body} = \mathit{serialize}(x + y) \mathrel{-\!\!*} \Psi(m')$$

Intuitively, the protocol demands that the sender of a message m is governed by some protocol Ψ and that the message body $m.\mathsf{body}$ must be the serialization of two numbers x and y. Moreover, the sender's protocol must be satisfied if the serialization of $x + y$ is sent as a response.

Using Φ_{add} as the socket protocol, we can give `server` the specification

$$\{\mathsf{Static}(a, A, \Phi_{add}) * \mathsf{IsNode}(n)\}\ \langle n;\ \texttt{server}\ a\rangle\ \{\mathsf{False}\}.$$

The postcondition is allowed to be False as the program does not terminate. The triple guarantees safety which, among others, means that *if* the server responds to communication on address a it does so according to Φ_{add}.

Similarly, using Φ_{add} as a primordial protocol for the server address, we can also give `client` a specification

$$\{\mathit{srv} \Mapsto \Phi_{add} * \mathit{srv} \in A * \mathsf{Dynamic}(a, A) * \mathsf{IsNode}(m)\} \\ \langle m;\ \texttt{client}\ x\ y\ \mathit{srv}\ a\rangle \\ \{v.v = x + y\}$$

```
let lockserver a =
  let lock = ref NONE in
  let skt = socket () in
  socketbind skt a;
  listen skt (fun handler msg from ->
    (if msg = "LOCK" then
       match !lock with
       | None => lock := Some ();
                 sendto skt "YES" from
       | Some _ => sendto skt "NO" from
       end
     else lock := NONE;
          sendto skt "RELEASED" from);
    listen skt handler)
```

Figure 3.2: A lock server in AnerisLang.

that showcases how the client is able to conclude that the response from the server is the sum of the numbers it sent to it. In the proof, when binding a to the socket using SOCKETBIND-DYNAMIC, we introduce the proposition $a \Rightarrow \Phi_{client}$ where

$$\Phi_{client}(m) \triangleq m.\mathsf{body} = serialize(x + y)$$

and use it to instantiate Ψ when satisfying Φ_{add}. Using the two specifications and the NODE-PAR-rule it is straightforward to specify and verify a distributed system composed of, e.g., a server and multiple clients.

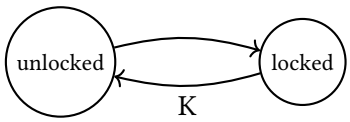
3.1.4 Example: A lock server

Mutual exclusion in distributed systems is often a necessity and there are many different approaches for providing it. The simplest solution is a centralized algorithm with a single node acting as the coordinator. We will develop this example to showcase a more interesting protocol that relies on ownership transfer of spatial resources between nodes to ensure correctness. The code for a centralized lock server implementation is shown in Figure 3.2.

The lock server declares a node-local variable `lock` to keep track of whether the lock is taken or not. It allocates a socket, binds the input address to the socket and continuously listens for incoming messages. When a `"LOCK"` message arrives and the lock is available, the lock gets taken and the server responds `"YES"`. If the lock was already taken, the server will respond `"NO"`. Finally, if the message was not `"LOCK"`, the lock is released and the server responds with `"RELEASED"`.

Our specification of the lock server will be inspired by how a lock can be specified in concurrent separation logic. Thus we first recall how such a specification usually looks like.

Conceptually, a lock can either be unlocked or locked, as described by a two-state labeled transition system.



In concurrent separation logic, the lock specification does not describe this transition system directly, but instead focuses on the resources needed for the transitions to take place. In

the case of the lock, the resource is simply a non-duplicable resource K, which is needed in order to call the lock's `release` method. Intuitively, this resource corresponds to the key of the lock.

A typical concurrent separation logic specification for a spin lock module looks roughly like the following:

$$
\begin{aligned}
&\exists\, \mathrm{isLock}\,. \\
\wedge\quad & \forall v, K.\ \mathrm{isLock}(v, K) \dashv\vdash \mathrm{isLock}(v, K) * \mathrm{isLock}(v, K) \\
\wedge\quad & \forall v, K.\ \mathrm{isLock}(v, K) \vdash K * K \Rightarrow \mathsf{False} \\
\wedge\quad & \{\mathsf{True}\}\ \texttt{newLock}\ ()\ \{v.\ \exists K.\ \mathrm{isLock}(v, K)\} \\
\wedge\quad & \forall v.\ \{\mathrm{isLock}(v, K)\}\ \texttt{acquire}\ v\ \{v.K\} \\
\wedge\quad & \forall v.\ \{\mathrm{isLock}(v, K) * K\}\ \texttt{release}\ v\ \{\mathsf{True}\}
\end{aligned}
$$

The intuitive reading of such a specification is:

- Calling `newLock` will lead to the duplicable knowledge of the return value v being a lock.
- Knowing that a value is a lock, a thread can try to acquire the lock and when it eventually succeeds it will get the key K.
- Only a thread holding this key is allowed to call release.

Sharing of the lock among several threads is achieved by the isLock predicate being duplicable. Mutual exclusion is ensured by the last bullet point together with the requirement of K being non-duplicable whenever we have $\mathrm{isLock}(v, K)$. For a leisurely introduction to such specifications, the reader may consult Birkedal and Bizjak [BB17].

Let us now return to the distributed lock synchronization. To give clients the possibility of interacting with the lock server as they would with such a concurrent lock module, the specification for the lock server will look like follows.

$$\{K * \mathsf{Static}(a, A, \Phi_{lock})\}\ \langle n;\ \texttt{lockserver}\ a\rangle\ \{\mathsf{False}\}.$$

This specification simply states that a lock server should have a primordial protocol Φ_{lock} and that it needs the key resource to begin with. To allow for the desired interaction with the server, we define the socket protocol Φ_{lock} as follows:

$$
\begin{aligned}
acq(m, \Psi) &\triangleq (m.\mathsf{body} = \texttt{"LOCK"}) * \\
&\qquad \forall m'.\ (m'.\mathsf{body} = \texttt{"NO"}) \vee (m'.\mathsf{body} = \texttt{"YES"} * K) \mathrel{-\!\!*} \Psi(m') \\
rel(m, \Psi) &\triangleq (m.\mathsf{body} = \texttt{"RELEASE"}) * K * \\
&\qquad \forall m'.\ (m'.\mathsf{body} = \texttt{"RELEASED"}) \mathrel{-\!\!*} \Psi(m') \\
\Phi_{lock}(m) &\triangleq \exists \Psi.\ m.\mathsf{orig} \mapsto \Psi * (acq(m, \Psi) \vee rel(m, \Psi))
\end{aligned}
$$

The protocol Φ_{lock} demands that a client of the lock has to be bound to some protocol Ψ and that the server can receive two types of messages fulfilling either $acq(m, \Psi)$ or $rel(m, \Psi)$. These correspond to the module's two methods `acquire` and `release` respectively. In the case of a `"LOCK"` message, the server will answer either `"NO"` or `"YES"` along with the key resource. In either case, the answer should suffice for fulfilling the client protocol Ψ.

Receiving a `"RELEASE"` request is similar, but the important part is that we require a client to send the key resource K along with the message, which ensures that only the current holder can release the lock.

One difference between the distributed and the concurrent specification is that we allow for the distributed lock to directly deny access. The client can use a simple loop, asking for the lock until it is acquired, if it wishes to wait until the lock can be acquired.

There are several interesting observations one can make about the lock server example:

1. The lock server can allocate, read, and write node-local references but these are hidden in the specification.
2. There are no channel descriptors or assertions on the socket in the code.
3. The lock server provides mutual exclusion by requiring clients to satisfy a sufficient protocol.

3.2 *AnerisLang*

AnerisLang is an untyped functional language with higher-order functions, fork-based concurrency, higher-order mutable references, and primitives for communicating over network sockets. The syntax is as follows:

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid b \mid i \mid s \mid \ell \mid z \mid \texttt{rec}\ f\ x = e \mid \ldots \\
e \in \mathit{Expr} &::= v \mid x \mid \texttt{rec}\ f\ x = e \mid e_1\ e_2 \mid \texttt{ref}(e) \mid\ \texttt{!}\ e \mid e_1 \texttt{ := } e_2 \mid \texttt{CAS}\ e_1\ e_2\ e_3 \\
&\quad \mid \texttt{find}\ e_1\ e_2\ e_3 \mid \texttt{substring}\ e_1\ e_2\ e_3 \mid \texttt{i2s}\ e \mid \texttt{s2i}\ e \\
&\quad \mid \texttt{fork}\ \{e\} \mid \texttt{start}\ \{n; ip; e\} \mid \texttt{makeaddress}\ e_1\ e_2 \\
&\quad \mid \texttt{socket}\ e \mid \texttt{socketbind}\ e_1\ e_2 \mid \texttt{sendto}\ e_1\ e_2\ e_3 \mid \texttt{receivefrom}\ e \mid \ldots
\end{aligned}
$$

We omit the usual operations on pairs, sums, booleans $b \in \mathbb{B}$, and integers $i \in \mathbb{Z}$ which are all standard. We introduce the following syntactic sugar: lambda abstractions $\lambda x.\ e$ defined as $\texttt{rec}\ \_\ x = e$, let-bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ defined as $(\lambda x.\ e_2)(e_1)$, and sequencing $e_1; e_2$ defined as $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$.

We have the usual operations on locations $\ell \in \mathit{Loc}$ in the heap: $\texttt{ref}(v)$ for allocating a new reference, $\texttt{!}\ \ell$ for dereferencing, and $\ell \texttt{ := } v$ for assignment. $\texttt{CAS}\ \ell\ v_1\ v_2$ is an atomic compare-and-set operation used to achieve synchronization between threads on a specific memory location ℓ. Operationally, it tests whether ℓ has value v_1 and if so, updates the location to v_2, returning a boolean indicating whether the swap succeeded or not.

The operation `find` finds the index of a particular substring in a string $s \in \mathit{String}$ and `substring` splits a string at given indices, producing the corresponding substring. `i2s` and `s2i` convert between integers and strings. These operations are mainly used for serialization and deserialization purposes.

The expression $\texttt{fork}\ \{e\}$ forks off a new (node-local) thread and $\texttt{start}\ \{n; ip; e\}$ will spawn a new node $n \in \mathit{Node}$ with ip address $ip \in \mathit{Ip}$ running the program e. Note that it is only at the bootstrapping phase of a distributed system that a special system-node $\mathfrak{S}$ will be able to spawn nodes.

We use $z \in \mathit{Handle}$ to range over socket handles created by the `socket` operation. The operation `makeaddress` constructs an address $(ip, p) \in \mathit{Address} = \mathit{Ip} \times \mathit{Port}$ given an ip address

$$\frac{\mathcal{S}(n)(z) = \bot \qquad p \notin \mathcal{P}(ip) \qquad \mathcal{S}' = \mathcal{S}[n \mapsto \mathcal{S}(n)[z \mapsto (ip, p)]] \qquad \mathcal{P}' = \mathcal{P}[ip \mapsto \mathcal{P}(ip) \cup \{p\}]}{\langle n;\ \texttt{socketbind}\ z\ (ip, p)\rangle, (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M}) \rightsquigarrow \langle n;\ 0\rangle, (\mathcal{H}, \mathcal{S}', \mathcal{P}', \mathcal{M})}$$

$$\frac{\mathcal{S}(n)(z) = orig \qquad i \notin \mathrm{dom}(\mathcal{M}) \qquad \mathcal{M}' = \mathcal{M}[i \mapsto (orig, to, msg, \textsc{Sent})]}{\langle n;\ \texttt{sendto}\ z\ msg\ to\rangle, (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M}) \rightsquigarrow \langle n;\ |msg|\rangle, (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M}')}$$

$$\frac{\mathcal{S}(n)(z) = to \qquad \mathcal{M}(i) = (orig, to, msg, \textsc{Sent}) \qquad \mathcal{M}' = \mathcal{M}[i \mapsto (orig, to, msg, \textsc{Received})]}{\langle n;\ \texttt{receivefrom}\ z\rangle, (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M}) \rightsquigarrow \langle n;\ \texttt{Some}\ (msg, orig)\rangle, (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M}')}$$

$$\frac{\mathcal{S}(n)(z) = to}{\langle n;\ \texttt{receivefrom}\ z\rangle, (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M}) \rightsquigarrow \langle n;\ \texttt{None}\rangle, (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M})}$$

Figure 3.3: An excerpt of the rules for network-aware head reduction.

$ip \in \mathit{Ip}$ and a port $p \in \mathit{Port}$, and the network primitives `socketbind`, `sendto`, and `receivefrom` correspond to the similar BSD-socket API methods.

Operational semantics. We define the operational semantics of AnerisLang in three stages.

We first define a node-local, thread-local, head-step reduction $(e, h) \rightarrow_{\mathsf{h}} (e', h')$ for $e, e' \in \mathit{Expr}$ and $h, h' \in \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}$ that handles all pure and heap-related node-local reductions. All rules of the relation are standard.

Next, the node-local head step reduction induces a network-aware head step reduction $(\langle n;\ e\rangle, \Sigma) \rightsquigarrow (\langle n;\ e'\rangle, \Sigma')$.

$$\frac{(e, h) \rightarrow_{\mathsf{h}} (e', h')}{\langle n;\ e\rangle, (\mathcal{H}[n \mapsto h], \mathcal{S}, \mathcal{P}, \mathcal{M}) \rightsquigarrow \langle n;\ e'\rangle, (\mathcal{H}[n \mapsto h'], \mathcal{S}, \mathcal{P}, \mathcal{M})}$$

Here $n \in \mathit{Node}$ denotes a node identifier and $\Sigma, \Sigma' \in \mathit{GState}$ the global state. Elements of *GState* are tuples $(\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M})$ tracking heaps $\mathcal{H} \in \mathit{Node} \xrightarrow{\text{fin}} \mathit{Heap}$ and sockets $\mathcal{S} \in \mathit{Node} \xrightarrow{\text{fin}} \mathit{Handle} \xrightarrow{\text{fin}} \mathit{Option}\ \mathit{Address}$ for all nodes, ports in use $\mathcal{P} \in \mathit{Ip} \xrightarrow{\text{fin}} \wp^{\text{fin}}(\mathit{Port})$, and messages sent $\mathcal{M} \in \mathit{Id} \xrightarrow{\text{fin}} \mathit{Message}$ where $\mathit{Message} = \mathit{Address} \times \mathit{Address} \times \mathit{String} \times \mathit{Flag}$. The induced network-aware reduction is furthermore extended with rules for the network primitives as seen in Figure 3.3. The `socket` operation allocates a new unbound socket using a fresh handle z for a node n and `socketbind` binds a socket address a to an unbound socket z if the address and port p is not already in use. Hereafter, the port is no longer available in $\mathcal{P}'(ip)$. For bound sockets, `sendto` sends a message msg to a destination address to from the sender's address $orig$ found in the bound socket. The message is assigned a unique identifier and tagged with a status flag Sent indicating that the message has been sent and not received. The operation returns the number of characters sent.

To model possibly dropped or delayed messages we introduce two rules for receiving messages using the `receivefrom` operation that on a bound socket either returns a previously unreceived message or nothing. If a message is received the status flag of the message is updated to Received

Third and finally, using standard *call-by-value right-to-left evaluation contexts* $K \in \mathit{Ectx}$ we lift the node-local head reduction to a *distributed systems* reduction $\twoheadrightarrow$ shown below. We write $\twoheadrightarrow^*$ for its reflexive-transitive closure. The distributed systems relation reduces by reducing a thread on some node, by forking off a thread on some node, or by starting a new node.

$$\frac{(\langle n;\, e\rangle, \Sigma) \rightsquigarrow (\langle n;\, e'\rangle, \Sigma')}{(\vec{T}_1 +\!\!+ [\langle n;\, K[e]\rangle] +\!\!+ \vec{T}_2, \Sigma) \twoheadrightarrow (\vec{T}_1 +\!\!+ [\langle n;\, K[e']\rangle] +\!\!+ \vec{T}_2; \Sigma')}$$

$$(\vec{T}_1 +\!\!+ [\langle n;\, K[\texttt{fork}\ \{e\}]\rangle] +\!\!+ \vec{T}_2, \Sigma) \twoheadrightarrow (\vec{T}_1 +\!\!+ [\langle n;\, K[()]\rangle] +\!\!+ \vec{T}_2 +\!\!+ [\langle n;\, e\rangle], \Sigma)$$

$$\frac{\begin{array}{c}\Sigma = (\mathcal{H}, \mathcal{S}, \mathcal{P}, \mathcal{M}) \qquad \Sigma' = (\mathcal{H}[n \mapsto \emptyset], \mathcal{S}[n \mapsto \emptyset], \mathcal{P}, \mathcal{M}) \\ n \neq \mathfrak{S} \qquad n \notin \mathsf{dom}(\mathcal{H}) \qquad n \notin \mathsf{dom}(\mathcal{S}) \qquad \mathit{ip} \in \mathsf{dom}(\mathcal{P})\end{array}}{(\vec{T}_1 +\!\!+ [\langle \mathfrak{S};\, K[\texttt{start}\ \{n; \mathit{ip}; e\}]\rangle] +\!\!+ \vec{T}_2, \Sigma) \twoheadrightarrow (\vec{T}_1 +\!\!+ [\langle \mathfrak{S};\, K[()]\rangle] +\!\!+ \vec{T}_2 +\!\!+ [\langle n;\, e\rangle], \Sigma')}$$

3.3 The Aneris logic

As a consequence of building on the Iris framework, the Aneris logic features all the usual connectives and rules of higher-order separation logic, some of which are shown in the grammar below.[4] The full expressiveness of the logic can be exploited when giving specifications to programs or stating protocols.

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \\ & \forall x.\, P \mid \exists x.\, P \mid P * Q \mid P \mathrel{-\!\!*} Q \mid t = u \mid \\ & \ell \mapsto_n v \mid \boxed{P} \mid \boxed{a}^{\gamma} \mid \{P\}\, \langle n;\, e\rangle\, \{x.\, Q\} \mid \ldots \end{aligned}$$

Note that in Aneris the usual points-to connective about the heap, $\ell \mapsto_n v$, is indexed by a node identifier $n \in \mathit{Node}$, asserting ownership of the singleton heap mapping ℓ to v on node n.

The logic features (impredicative) invariants $\boxed{P}$ and user-definable ghost state via the proposition $\boxed{a}^{\gamma}$, which asserts ownership of a piece of ghost state a at ghost location γ. The logical support for user-defined invariants and ghost state allows one to relate (ghost and physical) resources to each other; this is vital for our specifications as will become evident in Section 3.4 and Section 3.6. We refer to Jung et al. [Jun+18b] for a more thorough treatment of user-defined ghost state.

To reason about AnerisLang programs, the logic features Hoare triples.[5] The intuitive reading of the Hoare triple $\{P\}\, \langle n;\, e\rangle\, \{x.\, Q\}$ is that if the program e on node n is run in a distributed system s satisfying P, then the computation does not get stuck and, moreover, if it terminates with a value v and in a system s', then s' satisfies $Q[v/x]$. In other words, a Hoare triple implies safety and states that all spatial resources that are used by e are contained in the precondition P.

In contrast to spatial propositions that express *ownership*, e.g., $\ell \mapsto_n v$, propositions like $\boxed{P}$ and $\{P\}\, \langle n;\, e\rangle\, \{x.\, Q\}$ express *knowledge* of properties that, once true, hold true forever.

[4] To avoid the issue of reentrancy, invariants are annotated with a *namespace* and Hoare triples with a *mask*. We omit both for the sake of presentation as they are orthogonal issues.

[5] In both Iris and Aneris the notion of a Hoare triple is defined in terms of a *weakest precondition* but this will not be important for the remainder of this chapter.

We call this class of propositions *persistent*. Persistent propositions P can be freely duplicated: $P \dashv\vdash P * P$.

3.3.1 The program logic

The Aneris proof rules include the usual rules of concurrent separation logic for Hoare triples, allowing formal reasoning about node-local pure computations, manipulations of the the heap, and forking of threads. Expressions e are annotated with a node identifier n, but the rules are otherwise standard.

To reason about individual nodes in a distributed system in isolation, Aneris introduces the following rule:

$$\textsc{Start}\quad \frac{\{P * \mathsf{IsNode}(n) * \mathsf{FreePorts}(ip, \mathfrak{P})\}\ \langle n;\ e\rangle\ \{\mathsf{True}\}}{\{P * \mathsf{FreeIp}(ip)\}\ \langle \mathfrak{S};\ \texttt{start}\ \{n; ip; e\}\rangle\ \{x.\ x = ()\}}$$

where $\mathfrak{P} = \{p \mid 0 \leq p \leq 65535\}$. This rule is the key rule allowing node-local reasoning; the rule expresses exactly that to reason about a distributed system it suffices to reason about each node in isolation.

As described in Section 3.2, only the distinguished system node $\mathfrak{S}$ can start new nodes—this is also reflected in the Start-rule. In order to start a new node associated with IP address ip, the resource $\mathsf{FreeIp}(ip)$ is provided. This indicates that ip is not used by other nodes. When reasoning about the node n, the proof can rely on all ports on ip being available. The resource $\mathsf{IsNode}(n)$ indicates that the node n is a valid node in the system and keeps track of abstract state related to the modeling of node n's heap and sockets. $\mathsf{IsNode}(n)$ is persistent and hence duplicable.

Network communication. To reason about network communication in a distributed system, the logic includes a series of rules for reasoning about socket manipulation: allocation of sockets, binding of addresses to sockets, sending via sockets, and receiving from sockets.

To allocate a socket it suffices to prove that the node n is valid by providing the $\mathsf{IsNode}(n)$ resource. In return, an unbound socket resource $z \hookrightarrow_n \bot$ is provided.

$$\textsc{Socket}\quad \{\mathsf{IsNode}(n)\}\ \langle n;\ \texttt{socket}\ ()\rangle\ \{z.\ z \hookrightarrow_n \bot\}$$

The socket resource $z \hookrightarrow_n o$ keeps track of the address associated with the socket handle z on node n and takes part in ensuring that the socket is bound only once. It behaves similarly to the points-to connective for the heap, e.g., $z \hookrightarrow_n o * z \hookrightarrow_n o' \vdash \mathsf{False}$.

As briefly touched upon in Section 3.1, the logic offers two different rules for binding an address to a socket depending on whether or not the address has a (at the level of the logic) primordial, agreed upon protocol. To distinguish between such static and dynamic addresses, we use a persistent resource $\mathsf{Fixed}(A)$ to keep track of the set of addresses that have a fixed socket protocol.

To reason about a static address binding to a socket z it suffices to show that the address a being bound has a fixed interpretation (by being in the "fixed" set), that the port of the address

is free, and that the socket is not bound.

$$\begin{array}{l}\textsc{Socketbind-static}\\ \{\mathsf{Fixed}(A) * a \in A * \mathsf{FreePort}(a) * z \hookrightarrow_n \bot\}\\ \quad \langle n;\ \texttt{socketbind}\ z\ a\rangle\\ \{x.\ x = 0 * z \hookrightarrow_n a\}\end{array}$$

In accordance with the BSD-socket API, the bind operation returns the integer 0 and the socket resource gets updated, reflecting the fact that the binding took place.

The rule for dynamic address binding is similar but the address a should not have a fixed interpretation. Moreover, the user of the logic is free to pick the socket protocol Φ to govern address a.

$$\begin{array}{l}\textsc{Socketbind-dynamic}\\ \{\mathsf{Fixed}(A) * a \notin A * \mathsf{FreePort}(a) * z \hookrightarrow_n \bot\}\\ \quad \langle n;\ \texttt{socketbind}\ z\ a\rangle\\ \{x.\ x = 0 * z \hookrightarrow_n a * a \Rrightarrow \Phi\}\end{array}$$

To reason about sending a message on a socket z it suffices to show that z is bound, that the destination of the message is governed by a protocol Φ, and that the message satisfies the protocol.

$$\begin{array}{l}\textsc{Sendto}\\ \{z \hookrightarrow_n orig * to \Rrightarrow \Phi * \Phi((orig, to, msg, \textsc{Sent}))\}\\ \quad \langle n;\ \texttt{sendto}\ z\ msg\ to\rangle\\ \{x.\ x = |msg| * z \hookrightarrow_n orig\}\end{array}$$

Finally, to reason about receiving a message on a socket z the socket must be bound to an address governed by a protocol Φ.

$$\begin{array}{l}\textsc{Receivefrom}\\ \{z \hookrightarrow_n to * to \Rrightarrow \Phi\}\\ \quad \langle n;\ \texttt{receivefrom}\ z\rangle\\ \{x.\ z \hookrightarrow_n to\ * (x = \mathsf{None} \vee (\exists m.\ x = \mathsf{Some}\ (m.\mathsf{body}, m.\mathsf{orig}) * \Phi(m) * \mathsf{R}(m)))\}\end{array}$$

When trying to receive a message on a socket, either a message will be received or no message is available. This is reflected directly in the logic: if no message was received, no resources are obtained. If a message m is received, the resources prescribed by $\Phi(m)$ are transferred together with an unmodifiable certificate $\mathsf{R}(m)$ accounting logically for the fact that message m was received. This certificate can in the logic be used to talk about messages that have actually been received in contrast to arbitrary messages. In our specification of the two-phase commit protocol presented in Section 3.6, the notion of a vote denotes not just a message with the right content but only one that has been sent by a participant and received by the coordinator.

3.3.2 *Adequacy for Aneris*

We now state a formal adequacy theorem, which expresses that Aneris guarantees both safety, and, that all protocols are adhered to.

To state our theorem we introduce a notion of *initial state coherence*: A set of addresses $A \subseteq \mathit{Address} = \mathit{Ip} \times \mathit{Port}$ and a map $\mathcal{P} : \mathit{Ip} \xrightarrow{\text{fin}} \wp^{\text{fin}}(\mathit{Port})$ are said to satisfy initial state coherence if the following hold: (1) if $(i, p) \in A$ then $i \in \mathrm{dom}(\mathcal{P})$, and (2) if $i \in \mathrm{dom}(\mathcal{P})$ then $\mathcal{P}(i) = \emptyset$.

Theorem 3.3.1 (Adequacy)**.** *Let φ be a first-order predicate over values, i.e., a meta logic predicate (as opposed to Iris predicates), let $\mathcal{P}$ be a map $\mathit{Ip} \xrightarrow{\text{fin}} \wp^{\text{fin}}(\mathit{Port})$, and $A \subseteq \mathit{Address}$ such that A and $\mathcal{P}$ satisfy initial state coherence. Given a primordial socket protocol Φ_a for each $a \in A$, suppose that the Hoare triple*

$$\left\{ \mathsf{Fixed}(A) * \mathop{\Large *}_{a \in A} a \Mapsto \Phi_a * \mathop{\Large *}_{i \in \mathrm{dom}(\mathcal{P})} \mathsf{FreeIp}(i) \right\} \langle n_1;\, e \rangle \, \{v.\varphi(v)\}$$

is derivable in Aneris.

If we have

$$(\langle n_1;\, e \rangle, (\emptyset, \emptyset, \mathcal{P}, \emptyset)) \twoheadrightarrow^* ([\langle n_1;\, e_1 \rangle, \langle n_2;\, e_2 \rangle, \ldots \langle n_m;\, e_m \rangle], \Sigma)$$

then the following properties hold:

1. *If e_1 is a value, then $\varphi(e_1)$ holds at the meta-level.*
2. *Each e_i that is not a value can make a node-local, thread-local reduction step.*

Given predefined socket protocols for all primordial protocols and the necessary free IP addresses, this theorem provides the normal adequacy guarantees of Iris-like logics, namely *safety*, i.e., that nodes and threads on nodes cannot get stuck and that the postcondition holds for the resulting value. Notice, however, that this theorem also implies that all nodes adhere to the agreed upon protocols; otherwise, a node not adhering to a protocol would be able to cause another node to get stuck, which the adequacy theorem explicitly guarantees against.

3.4 Case study: Load balancer

AnerisLang supports concurrent execution of threads on nodes through the `fork` $\{e\}$ primitive. We will illustrate the benefits of node-local concurrency by presenting an example of server-side load balancing.

Implementation. In the case of server-side load balancing, the work distribution is implemented by a program listening on a socket that clients send their requests to. The program forwards the requests to an available server, waits for the response from the server, and sends the answer back to the client. In order to handle requests from several clients simultaneously, the load balancer can employ concurrency by forking off a new thread for every available server in the system that is capable of handling such requests. Each of these threads will then listen for and forward requests. The architecture of such a system with two servers and n clients is illustrated in Figure 3.4.

An implementation of a load balancer is shown in Figure 3.5. The load balancer is parameterized over an IP address, a port, and a list of servers. It creates a socket (corresponding to z_0 in Figure 3.4), binds the address, and folds a function over the list of servers. This function

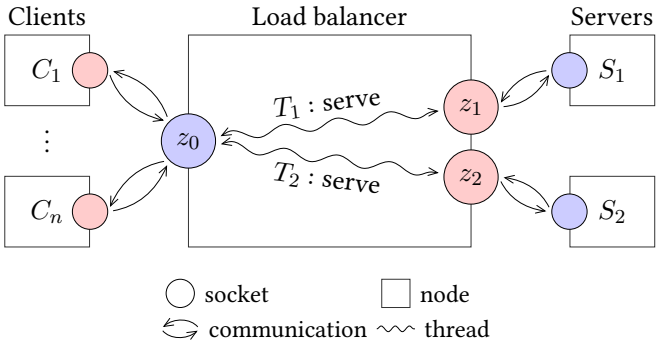


Figure 3.4: The architecture of a distributed system with a load balancer and two servers.

```
let serve main ip port srv =
  let skt = socket () in
  let a = makeaddress ip port in
  socketbind skt a;
  let rec loop () =
    match receivefrom main with
    | Some m =>
      sendto skt (fst m) srv;
      listen skt (fun _ msg from ->
        sendto main msg from; loop ())
    | None => loop ()
    end in
  loop ()
```

```
let load_balancer ip port servers =
  let skt = socket () in
  let a = makeaddress ip port in
  socketbind skt a;
  List.fold (fun server acc ->
    fork { serve skt ip acc server };
    acc + 1) 1100 servers
```

Figure 3.5: An implementation of a load balancer in AnerisLang.

forks off a new thread (corresponding to T_1 and T_2 in Figure 3.4) for each server that runs the `serve` function with the newly-created socket, the given IP address, a fresh port number, and a server as arguments.

The `serve` function creates a new socket (corresponding to z_1 and z_2 in Figure 3.4), binds the given address to the socket, and continuously tries to receive a client request on the main socket (z_0) given as input. If a request is received, it forwards the request to its server and waits for an answer. The answer is passed on to the client via the main socket. In this way, the entire load balancing process is transparent to the client, whose view will be the same as if it was communicating with just a single server handling all requests itself as the load balancer is simply relaying requests and responses.

Specification and protocols. To provide a general, reusable specification of the load balancer, we will parameterize its socket protocol by two predicates P_{in} and P_{out} that are both predicates on a message m and a meta-language value v. The two predicates are application specific and used to give logical accounts of the client requests and the server responses, respectively. Furthermore, we parameterize the protocol by a predicate P_{val} on a meta-language value that will allows us to maintain ghost state between the request and response as will become evident in following.

In our specification, the sockets where the load balancer and the servers receive requests (the blue sockets in Figure 3.4) will all be governed by the same socket protocol Φ_{rel} such that the load balancer may seamlessly relay requests and responses between the main socket and the servers, without invalidating any socket protocols. We define the generic relay socket

protocol Φ_{rel} as follows:

$$\Phi_{rel}(P_{val}, P_{in}, P_{out})(m) \triangleq \exists \Psi, v.\, m.\mathsf{orig} \Mapsto \Psi * P_{in}(m, v) * P_{val}(v) * \\ (\forall m'.\, P_{val}(v) * P_{out}(m', v) \mathrel{-\!\!*} \Psi(m'))$$

When verifying a request, this protocol demands that the sender (corresponding to the red sockets in Figure 3.4) is governed by some protocol Ψ, that the request fulfills the P_{in} and P_{val} predicates, and that Ψ is satisfied given a response that maintains P_{val} and satisfies P_{out}.

When verifying the load balancer receiving a request m from a client, we obtain the resources $P_{in}(m, v)$ and $P_{val}(v)$ for some v according to Φ_{rel}. This suffices for passing the request along to a server. However, to forward the server's response to the client we must know that the server behaves faithfully and gave us the response to the right request value v. Φ_{rel} does not give us this immediately as the v is existentially quantified. Hence we define a ghost resource $\mathsf{LB}(\pi, s, v)$ that provides fractional ownership for $\pi \in (0, 1]$, which satisfies $\mathsf{LB}(1, s, v) \dashv\vdash \mathsf{LB}(\frac{1}{2}, s, v) * \mathsf{LB}(\frac{1}{2}, s, v)$, and for which v can only get updated if $\pi = 1$ and in particular $\mathsf{LB}(\pi, s, v) * \mathsf{LB}(\pi, s, v') \implies v = v'$ for any π. Using this resource, the server with address s will have $P_{LB}(s)$ as its instantiation of P_{val} where

$$P_{LB}(s)(v) \triangleq \mathsf{LB}(\tfrac{1}{2}, s, v).$$

When verifying the load balancer, we will update this resource to the request value v when receiving a request (as we have the full fraction) and transfer $\mathsf{LB}(\frac{1}{2}, s, v)$ to the server with address s handling the request and, according to Φ_{rel}, it will be required to send it back along with the result. Since the server logically only gets half ownership, the value cannot be changed. Together with the fact that v is also an argument to P_{in} and P_{out}, this ensures that the server fulfills P_{out} for the same value as it received P_{in} for. The socket protocol for the `serve` function's socket (z_1 and z_2 in Figure 3.4) that communicates with a server with address s can now be stated as follows.

$$\Phi_{serve}(s, P_{out})(m) \triangleq \exists v.\, \mathsf{LB}(\tfrac{1}{2}, s, v) * P_{out}(m, v)$$

Since all calls to the `serve` function need access to the main socket in order to receive requests, we will keep the socket resource required in an invariant I_{LB} which is shared among all the threads:

$$I_{LB}(n, z, a) \triangleq \boxed{z \hookrightarrow_n a}$$

The specification for the `serve` function becomes:

$$\left\{ \begin{array}{l} I_{LB}(n, main, a_{main}) * \mathsf{Dynamic}((ip, p), A) * \mathsf{IsNode}(n) * \mathsf{LB}(1, s, v) * \\ a_{main} \Mapsto \Phi_{rel}(\lambda\_.\mathsf{True}, P_{in}, P_{out}) * s \Mapsto \Phi_{rel}(P_{LB}(s), P_{in}, P_{out}) \end{array} \right\} \\ \quad \langle n;\ \texttt{serve}\ main\ ip\ p\ s \rangle \\ \{\mathsf{False}\}$$

The specification requires the address a_{main} of the socket $main$ to be governed by Φ_{rel} with a trivial instantiation of P_{val} and the address s of the server to be governed by Φ_{rel} with P_{val} instantiated by P_{LB}. The specification moreover expects resources for a dynamic setup, the invariant that owns the resource needed to verify use of the $main$ socket, and a full instance of the $\mathsf{LB}(1, s, v)$ resource for some arbitrary v.

With this specification in place the complete specification of our load balancer is immediate (note that it is parameterized by P_{in} and P_{out}):

$$\left\{\begin{array}{l}\mathsf{Static}((ip,p),A,\phi_{rel}(\lambda\_.\mathsf{True},P_{in},P_{out}))*\mathsf{IsNode}(n)* \\ \left(\ast_{p'\in ports}\ \mathsf{Dynamic}((ip,p'),A)\right)* \\ \left(\ast_{s\in srvs}\ \exists v.\ \mathsf{LB}(1,s,v)*s \Rrightarrow \phi_{rel}(P_{LB}(s),P_{in},P_{out})\right)\end{array}\right\}$$
$$\langle n;\ \texttt{load\_balancer}\ ip\ p\ srvs\rangle$$
$$\{\mathsf{True}\}$$

where $ports = [1100, \cdots, 1100 + |srvs|]$. In addition to the protocol setup for each server as just described, for each port $p' \in ports$ which will become the endpoint for a corresponding server, we need the resources for a dynamic setup, and we need the resource for a static setup on the main input address (ip, p).

In the accompanying Coq development we provide an implementation of the addition service from Section 3.1.3, both in the single server case and in a load balanced case. For this particular proof we let the meta-language value v be a pair of integers corresponding to the expected arguments. In order to instantiate the load balancer specification we choose

$$P_{in}^{add}(m,(v_1,v_2)) \triangleq m.\mathsf{body} = serialize(v_1,v_2)$$
$$P_{out}^{add}(m,(v_1,v_2)) \triangleq m.\mathsf{body} = serialize(v_1+v_2)$$

with $serialize$ being the same serialization function from Section 3.1.3. We build and verify two distributed systems, (1) one consisting of two clients and an addition server and (2) one including two clients, a load balancer and three addition servers. We prove both of these systems safe and the proofs utilize the specifications we have given for the individual components. Notice that $\Phi_{rel}(\lambda\_.\mathsf{True}, P_{in}^{add}, P_{out}^{add})$ and Φ_{add} from Section 3.1.3 are the same. This is why we can use the same client specification in both system proofs. Hence, we have demonstrated Aneris' ability and support for horizontal composition of the same modules in different systems.

3.5 Case study: Bag service

While the load balancer demonstrates the use of node-local concurrency, its implementation does not involve shared memory concurrency, i.e., synchronization among the node-local threads. In order to handle multiple client requests simultaneously servers may employ concurrency by forking multiple threads. However, such servers may still have data structures or resources that are not safe to use in a concurrent setting. It is therefore often necessary to deploy synchronization mechanisms to ensure correctness. Figure 3.6 shows the architecture of a concurrent bag service that exploits multiple threads in order to handle several client requests at the same time while working on a shared bag data structure.

Figure 3.7 shows a thread-safe implementation of a bag module that uses a linked list as its internal representation of the bag and a lock in order to guarantee that only one thread at a time operates on the linked list. A weak, but still useful specification is the following: Given a predicate Ω, the bag contains elements x for which $\Omega(x)$ holds. When inserting an element

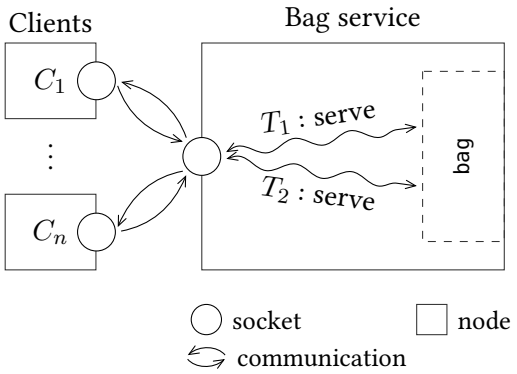


Figure 3.6: The architecture of a concurrent bag service with threads working on a shared data structure governed by a lock.

```
let newbag () =
  let l = ref None in
  let lock = newLock () in (l, lock)

let insert b e =
  let (l, lock) = b in
  acquire lock;
  l := Some (e, !l);
  release lock
```

```
let remove b =
  let (l, lock) = b in
  acquire lock;
  let res =
    match !l with
    | Some (h, tl) => l := tl; Some h
    | None => None
    end in
  release lock;
  res
```

Figure 3.7: A thread-safe bag implemented using a linked list and a lock.

we give away the resources, and when removing an element we give back an element plus the knowledge that it satisfies the predicate. This looks as follows:

$$
\begin{aligned}
&\exists\, \mathrm{isBag}\,. \\
&\quad\wedge\quad \forall n, v, \Omega.\ \mathrm{isBag}(n, v, \Omega) \dashv\vdash \mathrm{isBag}(n, v, \Omega) * \mathrm{isBag}(n, v, \Omega) \\
&\quad\wedge\quad \forall n, \Omega.\ \{\mathsf{IsNode}(n)\}\ \texttt{newbag}\ ()\ \{v.\ \mathrm{isBag}(n, v, \Omega)\} \\
&\quad\wedge\quad \forall v, e.\ \{\mathrm{isBag}(n, v, \Omega) * \Omega(e)\}\ \texttt{insert}\ v\ e\ \{\mathsf{True}\} \\
&\quad\wedge\quad \forall n, v, \Omega.\ \{\mathrm{isBag}(n, v, \Omega)\}\ \texttt{remove}\ b\ \{v.\ v = \mathsf{None} \vee \exists x.\ v = \mathsf{Some}\, x \wedge \Omega(x)\}
\end{aligned}
$$

Note how the isBag predicate is duplicable and therefore sharable among multiple threads. The isBag predicate is defined as follows:

$$
\begin{aligned}
P_{bag} &\triangleq \exists u.\ \ell \mapsto_n u * \mathrm{bagList}(\Omega, u) \\
\mathrm{isBag}(n, v, \Omega) &\triangleq \exists \ell, l.\ v = (\ell, l) * \mathrm{isLock}(n, l, P_{bag})
\end{aligned}
$$

where bagList is defined by recursion as the unique predicate satisfying

$$
\mathrm{bagList}(\Omega, u) \triangleq u = \mathsf{None} \vee \exists x, r.\ u = \mathsf{Some}\,(x, r) * \Omega(x) * \mathrm{bagList}(\Omega, r).
$$

Note that the isLock predicate is parameterized by a user-defined resource P_{bag} that follows the key resource: when the lock is acquired, the resources described by P_{bag} are given and the resources have to be given back when the lock is released.

```
let rec serve skt bag =
  match receivefrom skt with
  | Some (m, from) =>
    if m = "" then
      match Bag.remove bag with
      | Some v => sendto skt v from
      | None  => sendto skt "" from
      end
    else
      Bag.insert bag m; sendto skt "" from
  | None => ()
  end in serve skt bag
```

```
let bag_service a =
  let skt = socket () in
  let bag = newbag () in
  socketbind skt a;
  fork { serve skt bag };
  fork { serve skt bag }

let bag_client arg a server =
  let skt = socket () in
  socketbind skt a;
  sendto skt arg server;
  listen (fun _ m _ -> m)
```

Figure 3.8: An implementation of a concurrent bag service in AnerisLang. The bag service forks multiple threads for concurrently processing requests.

Figure 3.8 shows an AnerisLang implementation of a concurrent bag service. The main function creates a socket and a bag and forks two threads each executing the the `serve` function. This function listens for incoming messages. If the input message is an empty string it tries to remove an element from the bag and, if any, it sends the element back to the client, otherwise an empty string. If the input message is nonempty it inserts the message into the bag and acknowledges with an empty string. A bag client simply sends an argument to a server and returns the response.

In order to provide a specification for the bag service we define the socket protocol Φ_{bag} that will govern the socket on which the service listens for requests. Similar to the thread-safe bag implementation, the socket protocol will also be parameterized by a predicate Ω.

$$\mathit{insert}(\Omega, \Psi, m) \triangleq m.\mathsf{body} \neq \texttt{""} * \Omega(m.\mathsf{body}) * \forall m'.\, m'.\mathsf{body} = \texttt{""} \mathrel{-\!\!*} \Psi(m')$$
$$\mathit{remove}(\Omega, \Psi, m) \triangleq m.\mathsf{body} = \texttt{""} * \forall m'.\, (m'.\mathsf{body} = \texttt{""} \lor \Omega(m'.\mathsf{body}) \mathrel{-\!\!*} \Psi(m'))$$
$$\Phi_{bag}(\Omega)(m) \triangleq \exists \Psi.\, m.\mathsf{orig} \mapsto \Psi * (\mathit{insert}(\Omega, \Psi, m) \lor \mathit{remove}(\Omega, \Psi, m))$$

The protocol Φ_{bag} demands that the client should be bound to some protocol Ψ and that the server can receive two types of messages fulfilling either $\mathit{insert}(\Omega, \Psi, m)$ or $\mathit{remove}(\Omega, \Psi, m)$), corresponding to either inserting an element into the bag or removing one. To insert an element, the resources described by $\Omega(m.\mathsf{body})$ has to be provided and it should suffice for the client to receive an empty string as a response. When asking to retrieve an element, either the answer is the empty string or the message will satisfy $\Omega(m.\mathsf{body})$.

Using the socket protocol we can specify and verify the bag service as follows.

$$\{\mathsf{Static}(a, A, \Phi_{bag}(\Omega)) * \mathsf{IsNode}(n)\}$$
$$\langle n;\, \texttt{bag\_service}\ a\rangle$$
$$\{\mathsf{False}\}$$

The client code can either add or remove an element from the bag service, and the specification is straightforward given a server address srv governed by $\Phi_{bag}(\Omega)$.

$$\left\{\begin{array}{l} srv \mapsto \Phi_{bag}(\Omega) * \mathsf{Dynamic}(a, A) * \mathsf{IsNode}(n) * \\ arg = \texttt{""} \lor (arg \neq \texttt{""} \land \Omega(arg)) \end{array}\right\}$$
$$\langle n;\, \texttt{bag\_client}\ arg\ a\ srv\rangle$$
$$\{v.v = \texttt{""} \lor \Omega(v)\}$$

3.6 Case study: Two-phase commit

A typical problem in distributed systems is that of consensus and distributed commit; an operation should be performed by all participants in a system or none at all. The *two-phase commit* protocol (TPC) by [Gra78] is a classic solution to this problem. We study this protocol in Aneris as (1) it is widely used in the real-world, (2) it is a complex network protocol and thus serves as a decent benchmark for reasoning in Aneris, and (3) to show how an implementation can be given a specification that is usable for a client that abstractly relies on some consensus protocol.

The two-phase commit protocol consists of the following two phases, each involving two steps:

1. a) The coordinator sends out a vote request to each participant.
   b) A participant that receives a vote request replies with a vote for either commit or abort.
2. a) The coordinator collects all votes and determines a result. If all participants voted commit, the coordinator sends a global commit to all. Otherwise, the coordinator sends a global abort to all.
   b) All participants that voted for a commit wait for the final verdict from the coordinator. If the participant receives a global commit it locally commits the transaction, otherwise the transaction is locally aborted. All participants must acknowledge.

To provide general, reusable implementations and specifications of the coordinator and participants implementing TPC, we do not define how requests, votes, nor decisions look like. We leave it to a user of the module to provide decidable predicates matching the application specific needs and to define the logical, local pre- and postconditions, P and Q, of participants for the operation in question.

Our specifications use fractional ghost resources to keep track of coordinator and participant state w.r.t. the coordinator and participant transition systems indicated in the protocol description above. Similar to our previous case studies, we exploit partial ownership to limit when transitions can be made. When verifying a participant, we keep track of their state and the coordinator's state and require all participants' view of the coordinator state to be in agreement through an invariant.

In short, our specification of TPC

- ensures the participants and coordinator act according to the protocol, *i.e.*,
  - the coordinator decides based on all the participant votes,
  - participants act according to the global decision,
  - if the decision was to commit, we obtain the resources described by Q for all participants,
  - if the decision was to abort, we still have the resources described by P for all participants,
- does not require the coordinator to be primordial, so the coordinator could change from round to round.

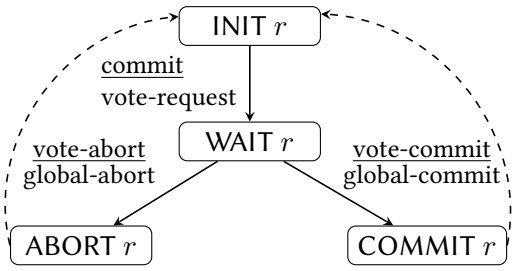


Figure 3.9: The coordinator state transition system.

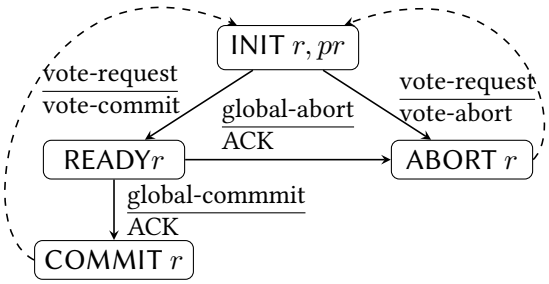


Figure 3.10: The participant state transition system.

Implementation. The abstract protocol steps are shown as transition systems in Figures 3.9 and 3.10 and an implementation of a TPC module that satisfies the conceptual description is shown in Figure 3.11. Our abstract model differs slightly from the traditional diagram as we reuse the same code and sockets for communication between coordinators and participants. Every state is therefore tagged with a unique round number and dashed arrows are local transitions allowing reuse of the state transition systems by incrementing round numbers. To allow each participant to locally transition to the INIT state upon round completion and still communicating commit or abort, the INIT state is tagged with the previous result pr (initially, COMMIT suffices).

The `tpc_coordinate` module expects an initial request message to be provided, along with a bound socket, a list of participants, and a function to make a decision when all votes have been received. Internally, it uses two local references; one to collect all the votes and one to count the number of acknowledgments.

The `tpc_participant` module expects a socket and two handlers—one to decide on a vote and one to finalize the decision made by the coordinator. When invoked, the module listens for incoming requests, decides on a vote and waits for a global decision from the coordinator. Since each node can employ concurrency, the blocking wait for the decision does not prevent the client from doing concurrent work, in particular engaging in other rounds of TPC with other coordinators. Notice as well that there are no round numbers in the implementation; the round numbers are only in the abstract model to strengthen the specification.

Specification and protocols. In order to specify and prove the TPC protocol correct, we will use the following resources, having a coordinator c and participants $p \in ps$:

- Parts(ps): Accounts for the set ps of participants for a concrete TPC round. The resource

```
let tpc_coordinate m skt ps dec =
  let count = List.length ps in
  let msgs = ref [] in
  let ack = ref 0 in
  List.iter (fun n -> sendto skt m n) ps;
  listen skt (fun handler m from ->
    msgs := m :: !msgs;
    if List.length !msgs = count
    then () else listen skt handler);
  let res = dec !msgs in
  List.iter (fun n -> sendto skt res n) ps;
  listen skt (fun handler m from ->
    ack := !ack + 1;
    if !ack = count then res
    else listen skt handler)
```

```
let rec tpc_participant skt vote fin =
  listen skt (fun _ m from ->
    let act = vote m in
    sendto skt act from;
    listen skt (fun _ m from ->
      fin m;
      sendto skt "ACK" from;
      tpc_participant skt vote fin))
```

Figure 3.11: An implementation of the two-phase commit protocol in AnerisLang.

is duplicable and unmodifiable.

- $\mathsf{Coord}(p, r, s_c)$: Accounts for participant p's view of the coordinators current state s_C (*cf.* Figure 3.9) in round r. The coordinator c owns an assertion regarding its own state $\mathsf{Coord}(c, r, s_C)$. We require that all parties agree which round and state the coordinator is in. Technically, this is stated in an invariant, I_{TPC}.
- $\mathsf{Part}(\pi, p, r, s_P)$: Accounts with fraction π for participant p's current state s_P (*cf.* Figure 3.10) in round r.

We leave it to a user of the module to provide decidable predicates $isReq$, $isVote$, $isAbort$ and $isGlobal$ of type $(\mathit{String} \times \mathbb{N}) \to \mathit{iProp}$. The user is free to pick $P : (\mathit{Address} \times \mathit{String}) \to \mathit{iProp}$ and $Q : (\mathit{Address} \times \mathbb{N}) \to iProp$, the local pre- and postcondition for each participant. The socket protocol for the coordinator is shown below.

$$\begin{aligned}
\Phi_{vote}(m) \triangleq\ & \exists p, r, ps.\ m.\mathsf{orig} = p * \mathsf{Parts}(\{p\} \cup ps)\ * isVote(m.\mathsf{body}, r) * \\
& \mathsf{Coord}(p, r, \mathsf{WAIT}) * (isAbort(m.\mathsf{body}, r) * \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{ABORT})\ \vee \\
& \neg isAbort(m.\mathsf{body}, r) * \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{READY})) \\
\Phi_{ack}(m) \triangleq\ & \exists p, r, ps, m', cs, pr.\ m.\mathsf{orig} = p * \mathsf{Parts}(\{p\} \cup ps)\ * \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{INIT}\ pr) * \\
& (\mathsf{Coord}(p, r, \mathsf{COMMIT}) * pr = \mathsf{COMMIT} * Q(p, r)\ \vee \\
& \mathsf{Coord}(p, r, \mathsf{ABORT}) * pr = \mathsf{READY} * P(p, m')) \\
\Phi_{coord}(m) \triangleq\ & \Phi_{vote}(m) \vee \Phi_{ack}(m)
\end{aligned}$$

For a participant p to send a vote to the coordinator c, it has to show that it is indeed a participant $\mathsf{Parts}(\{p\} \cup ps)$, that the message is a vote for that round, that it knows the coordinator is in the WAIT state, $\mathsf{Coord}(p, r, \mathsf{WAIT})$, and that the logical state of p matches p's actual vote. For the participant to send an acknowledgment, it has to prove it transitioned to the INIT pr where pr should match the global decision made by the coordinator. If the decision was to commit, the participant provides the updated resources for Q, otherwise it returns the resources described by P.

The socket protocol for the participants is as follows:

$$\Phi_{req}(p)(m) \triangleq \exists r, ps, s_P.\ \mathsf{Parts}(\{p\} \cup ps) * P(m.\mathsf{body}, p) *$$

$$
\begin{aligned}
&\quad\; \mathit{isReq}(m.\mathsf{body}, r+1) * m.\mathsf{orig} \Mapsto \Phi_{\mathit{coord}} * \\
&\quad\; \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{INIT}\ s_P) * \mathsf{Coord}(p, r+1, \mathsf{WAIT}) \\
\Phi_{\mathit{glob}}(p)(m) \triangleq\; &\exists r, ps, ga, ms, s_C, s_P.\ \mathsf{Parts}(\{m'.\mathsf{orig} \mid m' \in ms\}) * \\
&\quad\; \mathit{isGlobal}(m.\mathsf{body}, r) * m.\mathsf{orig} \Mapsto \Phi_{\mathit{coord}} * \\
&\quad\; \mathsf{Part}(\tfrac{3}{4}, p, r, s_P) * \mathsf{Coord}(p, r, s_C) * \\
&\quad\; ga = \{m' \mid m \in ms \wedge \mathit{isAbort}(m', r)\} * \\
&\quad\; \left(\mathop{\Large *}_{m' \in ms} \mathit{isVote}(ms.\mathsf{body}, r)\ * \mathsf{R}(m') \right) * \\
&\quad\; (ga = \emptyset \wedge \neg \mathit{isAbort}(m.\mathsf{body}, r) \wedge s_C = \mathsf{COMMIT})\ \vee \\
&\quad\; (ga \neq \emptyset \wedge \mathit{isAbort}(m.\mathsf{body}, r) \wedge s_C = \mathsf{ABORT}) \\
\Phi_{\mathit{part}}(p)(m) \triangleq\; &\Phi_{\mathit{req}}(p)(m) \vee \Phi_{\mathit{glob}}(p)(m)
\end{aligned}
$$

In order to send a request for a round $r+1$ of TPC to a participant p, a coordinator has to show p is indeed a participant of this instance and provide the resource described by P. The request should also be valid (through the isReq predicate) and the coordinator should be bound to the coordinator protocol Φ_{coord}. Furthermore, the coordinator has to show it is in the WAIT state and give up $\mathsf{Part}(\frac{3}{4}, p, r, \mathsf{INIT}\ s_P)$ in order to allow the participant to make a transition.

The coordinator can broadcast a global decision when having received a message from all the participants (where ms is the set of messages received) and the decision is a valid global decision. All the messages has to have been received and be valid votes. The coordinator also has to be honest: if any participant replied with an abort message ($ga \neq \emptyset$), the global message and the final state of the coordinator has to be ABORT.

Notice that for each message to a participant, the coordinator will provide the assertion $m.\mathsf{orig} \Mapsto \Phi_{\mathit{coord}}$. This means the coordinator do not have to be primordial since the participant does not need to have prior knowledge of the coordinator. The coordinator could change from round to round.

With the TPC protocols in place, we can finally give a specification to the two TPC modules. The `tpc_participant` specification is straightforward:

$$
\begin{array}{l}
\left\{ \begin{array}{l} I_{TPC} * \mathit{isReqSpec}(\mathit{req}) * \mathit{isFinSpec}(\mathit{fin}) * \mathsf{Parts}(ps) * \\ z \hookrightarrow_n p * p \Mapsto \Phi_{\mathit{part}}(p) * \mathsf{Part}(\frac{1}{4}, p, r, \mathsf{INIT}\ s_P) \end{array} \right\} \\
\quad \langle n;\ \texttt{tpc\_participant}\ z\ \mathit{req}\ \mathit{fin} \rangle \\
\{\mathsf{True}\}
\end{array}
$$

where req and fin are appropriate handlers for requests and finalization. The specification requires ownership of a bound socket bound by the participant protocol $\Phi_{\mathit{part}}(p)$ and fractional ownership of its own state, initialized to be INIT. Furthermore, the handlers req and fin should satisfy simple specifications that we elide to the Coq development.

The specification for `tpc_coordinate` is more involved:

$$\left\{\begin{array}{l} I_{TPC} * isDecSpec(dec) * isReq(m, r+1) * \mathsf{Parts}(ps) * \mathsf{IsNode}(n) * \\ z \hookrightarrow_n c * a \Rrightarrow \Phi_{coord} * \mathsf{Coord}(c, r, \mathsf{INIT}\ s_C) * \\ \left(\ast_{p \in ps}\ p \Rrightarrow \Phi_{part}(p) * \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{INIT}\ s_P) * \mathsf{Coord}(p, r, \mathsf{INIT}\ s_C) * P(p, m)\right) \end{array}\right\}$$

$$\langle n;\ \texttt{tpc\_coordinate}\ m\ z\ ps\ dec\rangle$$

$$\left\{\begin{array}{l} \langle n;\ v\rangle.\ \exists s_C, s_P.\ isGlobal(v, r+1) * \mathsf{Coord}(c, r+1, s_C) * z \hookrightarrow_n c \\ \qquad \left(\ast_{p \in ps} \mathsf{Coord}(p, r+1, s_C) * \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{INIT}\ s_P)\right) * \\ \qquad \left(isAbort(v, r+1) * s_C = \mathsf{ABORT} * s_P = \mathsf{ABORT} * \ast_{p \in ps} \exists m.\ P(p, m)\right) \vee \\ \qquad \left(\neg isAbort(v, r+1) * s_C = \mathsf{COMMIT}\ * s_P = \mathsf{COMMIT} * \ast_{p \in ps} Q(p, r+1)\right) \end{array}\right\}$$

To invoke `tpc_coordinate`, one has to provide a valid request m, a socket z already bound to some address guarded by the Φ_{coord} protocol, a list of participants p, and a decision handler dec. For each participant p, the address should be governed $\Phi_{part}(p)$ and the resources describing the participant's view of its own and the coordinator's state should be passed along. Finally, the resources described by $P(p, m)$ must also be provided.

The postcondition here is the most exciting part: it is exactly what one would expect. Either all participants along with the coordinator agreed to commit in which case we obtain $Q(p, r)$ for each participant p or they all agreed to abort, in which case we get back $P(p, m)$ for each participant p.

3.7 Case study: Replicated logging

In a distributed replicated logging system, a log is stored on several databases distributed across several nodes where the system ensures consistency among the logs through a consensus protocol. We have verified such a system implemented on top of the TPC coordinator and participant modules to showcase vertical composition of complex protocols in Aneris as illustrated in Figure 3.12. The blue parts of the diagram constitute node-local instantiations of the TPC modules invoked by the nodes to handle the consensus process. As noted by Sergey et al. [SWT18], clients of core consensus protocols have not received much focus from other major verification efforts [Haw+15; Rah+15; Wil+15].

Our specification of a replicated logging system draws on the generality of the TPC specification. In this case, we use fractional ghost state to keep track of two related pieces of information. The first keeps a logical account of the log l already stored in the database at a node at address a, $\mathsf{LOG}(\pi, a, l)$. The second one keeps track of what the log should be updated to, if the pending round of consensus succeeds. This is a pair of the existing log l and the (pending) change s proposed in this round, $\mathsf{PEND}(\pi, a, (l, s))$. We exploit fractional resource ownership by letting the coordinator, logically, keep half of the pending log resources at all times. Together with suitable local pre- and postconditions for the databases, this prevents the

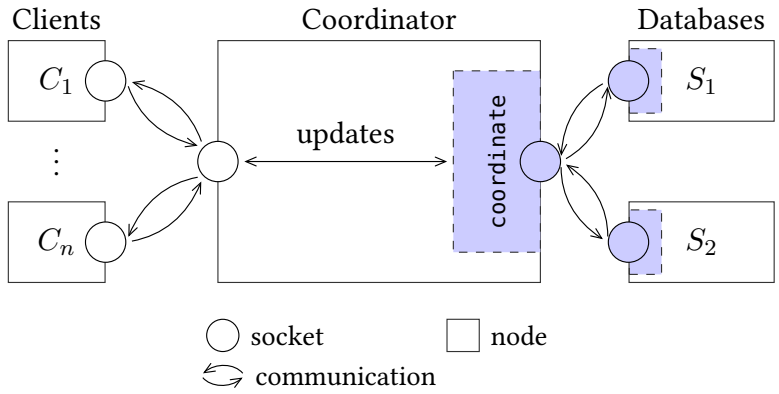


Figure 3.12: The architecture of a replicated logging system implemented using the TPC modules (the blue parts of the diagram) with a coordinator and two databases (S_1 and S_2) each storing a copy of the log.

databases from doing arbitrary changes to the log. Concretely, we instantiate P and Q of the TPC module as follows:

$$P_{rep}(p)(m) \triangleq \exists l, s.\, (m = \texttt{"REQUEST\_"} +\!\!+ s) * \mathsf{LOG}(\tfrac{1}{2}, p, l) * \mathsf{PEND}(\tfrac{1}{2}, p, (l, s))$$
$$Q_{rep}(p)(n) \triangleq \exists l, s.\, \mathsf{LOG}(\tfrac{1}{2}, p, l +\!\!+ s) * \mathsf{PEND}(\tfrac{1}{2}, p, (l, s))$$

where $+\!\!+$ denotes string concatenation. Note how the request message specifies the proposed change (since the string that we would like to add to the log is appended to the requests message) and how we ensure consistency by making sure the two ghost assertions hold for the same log. Even though l and s are existentially quantified, we know the logs cannot be inconsistent since the coordinator retains partial knowledge of the log. Due to the guarantees given by TPC specification, this implies that if the global decision was to commit a change this change will have happened locally on all databases, *cf.* $\mathsf{LOG}(\frac{1}{2}, p, l +\!\!+ s)$ in Q_{rep}, and if the decision was to abort, then the log remains unchanged on all databases, *cf.* $\mathsf{LOG}(\frac{1}{2}, p, l)$ in P_{rep}.

Implementation and specification. An implementation of a replicated logging system is shown in Figure 3.13. `logger` creates a socket `skt`, binds the address `a` to it, and initiates a TPC round for all databases in `dbs`. The decision handler `dec` is called by the TPC coordinator module when all votes have been received.

From the perspective of the database, `db`, an internal reference `log` keeps the log.[6] Upon an incoming request, the message is parsed and the proposed change is stored in the `wait` reference. If the global decision by `logger` is to commit, the string stored in `wait` will be appended to the log. To give a logical account of the local state of each database we introduce the fractional ghost resources $\mathsf{LOG}(\pi, p, l)$ and $\mathsf{PEND}(\pi, p, (l, s))$ that keep track of the log l and the proposed change s for each participant p. With the resources in place, the specification

[6] Ideally, this would be stable storage, however, for the sake of the example a reference suffices.

```
let logger log a m dbs =
  let skt = socket () in
  let dec = fun msgs ->
    let r = List.fold
      (fun a m -> a && m = "COMMIT") true msgs in
    if r then "COMMIT" else "ABORT" in
  socketbind skt a;
  tpc_coordinate ("REQUEST_" ^ m) skt dbs dec
```

```
let db addr =
  let skt = socket() in
  let wait = ref "" in
  let log = ref "" in
  let req = (fun m -> wait := m; "COMMIT") in
  let fin = fun m ->
    if m = "COMMIT"
    then log := !log ^ !wait else () in
  socketbind skt addr;
  tpc_participant skt req fin
```

Figure 3.13: An implementation in AnerisLang of a replicated logging system that uses the two-phase commit modules . ^ denotes string concatenation in AnerisLang.

of db is straightforward and follows from the specification of the TPC participant module:

$$\left\{\begin{array}{l} I_{TPC} * \mathsf{Dynamic}(a, \Phi_{part}(a)) * \mathsf{IsNode}(n) * \\ \mathsf{Part}(\frac{1}{4}, a, r, \mathsf{INIT}\ s_P) * \mathsf{LOG}(\frac{1}{2}, a, \texttt{""}) \end{array}\right\}$$
$$\langle n;\ \mathsf{db}\ a\rangle$$
$$\{\mathsf{True}\}$$

$$\left\{\begin{array}{l} I_{TPC} * \mathsf{Parts}(dbs) * \mathsf{FreePort}(a) * isReq(m) * \mathsf{Part}(\frac{3}{4}, p, r, \mathsf{INIT}\ s_P) * \\ \ast_{p \in dbs}\ \exists s_P, p \Rightarrow \Phi_{part}(p) * \mathsf{Coord}(p, r, \mathsf{INIT}\ s_P) * P_{rep}(p, m) \end{array}\right\}$$
$$\langle n;\ \texttt{logger}\ log\ a\ m\ dbs\rangle$$
$$\left\{\begin{array}{l} \langle n;\ v\rangle.\ \exists m, r.\ \ast_{p \in dbs}\ \exists s_P.\ \mathsf{Coord}(p, r, \mathsf{INIT}\ s_P) * \mathsf{Part}(\frac{3}{4}, p, r, \mathsf{INIT}\ s_P) * \\ \qquad \left(v = \texttt{"COMMIT"} * \ast_{p \in dbs}\ Q_{rep}(p, r)\right) \vee \left(v = \texttt{"ABORT"} * \ast_{p \in dbs}\ P_{rep}(p, m)\right) \end{array}\right\}$$

Verification of our replicated logging client using two-phased-commit follows directly in a modular, node-local fashion by applying the specification of `tpc_coordinate`. Due to the TPC specification, this implies that if the global decision was to commit a change this change will have happened locally on all databases, *cf.* $\mathsf{LOG}(\frac{1}{2}, p, l@s)$ in Q_{rep}, and if the decision was to abort, then the log remains unchanged on all databases, *cf.* $\mathsf{LOG}(\frac{1}{2}, p, l)$ in P_{rep}.

3.8 Related work

Verification of distributed systems has received a fair amount of attention. In order to give a better overview, we have divided related work into four categories.

Model-checking of distributed protocols. Previous work on verification of distributed systems has mainly focused on verification of protocols or core network components through model-checking. Frameworks for showing safety and liveness properties, such as SPIN [Hol97], and TLA+ [Lam92], have had great success. A benefit of using model-checking frameworks is that they allow to state both safety and liveness assertions as LTL assertions [Pnu77]. Mace [Kil+07]

provides a suite for building and model-checking distributed systems with asynchronous protocols, including liveness conditions. Chapar [LBC16] allows for model-checking of programs that use causally consistent distributed key-value stores. Neither of these languages provide higher-order functions or thread-based concurrency.

Session types for giving types to protocols. Session types have been studied for a wide range of process calculi, in particular, typed π-calculus. The idea is to describe two-party communication protocols as a type to ensure communication safety and progress [HVK98]. This has been extended to multi-party asynchronous channels [HYC08], multi-role types [DY11] which informally model topics of actor-based message-passing and dependent session types allowing quantification over messages [TCP11]. Our socket protocol definitions are quite similar to the multi-party asynchronous session types with progress encoded by having suitable ghost-assertions and using the magic wand. Actris [HBK20] is a logic for session-type based reasoning about message-passing in actor-based languages.

Hoare-style seasoning about distributed systems. Disel [SWT18] is a Hoare Type Theory for distributed program verification in Coq with ideas from separation logic. It provides the novel protocol-tailored rules WithInv and Frame which allow for modularity of proofs under the condition of an inductive invariant and distributed systems composition. In Disel, programs can be extracted into runnable OCaml programs, which is on our agenda for future work.

IronFleet [Haw+15] allows for building provably correct distributed systems by combining TLA-style state-machine refinement with Hoare-logic verification in a layered approach, all embedded in Dafny [Lei10]. IronFleet also allows for liveness assertions. For a comparison of Disel and IronFleet to Aneris from a modularity point of view we refer to the Introduction section.

Other distributed verification efforts. Verdi [Wil+15] is a framework for writing and verifying implementations of distributed algorithms in Coq, providing a novel approach to network semantics and fault models. To achieve compositionality, the authors introduced *verified system transformers*, that is, a function that transforms one implementation to another implementation with different assumptions about its environment. This makes vertical composition difficult for clients of proven protocols and in comparison AnerisLang seems more expressive.

EventML [Rah+15; Rah+17] is a functional language in the ML family that can be used for coding distributed protocols using high-level combinators from the Logic of Events, and verify them in the Nuprl interactive theorem prover. It is not quite clear how modular reasoning works, since one works within the model, however, the notion of a central main observer is akin to our distinguished system node.

3.9 Conclusion

Distributed systems are ubiquitous and hence it is essential to be able to verify them. In this chapter we presented Aneris, a framework for writing and verifying distributed systems in Coq built on top of the Iris framework. From a programming point of view, the important aspect of AnerisLang is that it is feature-rich: it is a concurrent ML-like programming language with network primitives. This allows individual nodes to internally use higher-order heap and concurrency to write efficient programs.

Module	Implementation	Specification	Proofs
Load balancer (Section 3.4)			
Load balancer	18	78	95
Addition service (Section 3.1.3)			
Server	11	15	38
Client	9	14	26
Adequacy (1 server, 2 clients)	5	12	62
Adequacy w. Load Balancing (3 servers, 2 clients)	16	28	175
Two-phase commit (Section 3.6)			
Coordinator	18	181	265
Participant	11		280
Replicated logging (Section 3.7)			
Instantiation of TPC	-	85	-
Logger	22	19	95
Database	24	20	190
Adequacy (2 dbs, 1 coordinator, 2 clients)	13	-	137

Table 3.1: Sizes of implementations, specifications, and proofs in lines of code. When proving adequacy, the system must be closed.

The Aneris logic provides node-local reasoning through socket protocols. That is, we can reason about individual nodes in isolation as we reason about individual threads. We demonstrate the versatility of Aneris by studying interesting distributed systems both implemented and verified within Aneris. The adequacy theorem of Aneris implies that these programs are safe to run.

Relating the verification sizes of the modules from Table 3.1 to other formal verification efforts in Coq indicates that it is easier to specify and verify systems in Aneris. The total work required to prove two-phase commit with replicated logging is 1,272 lines which is just half of the lines needed for proving the inductive invariant for TPC in other works [SWT18]. However, extensive work has gone into Iris Proof Mode thus it is hard to conclude that Aneris requires less verification effort and does not just have richer tactics.

4 Distributed Causal Memory

Abstract

We present the first specification and verification of an implementation of a causally-consistent distributed database that supports modular verification of full functional correctness properties of clients and servers. We specify and reason about the causally-consistent distributed database in Aneris, a higher-order distributed separation logic for an ML-like programming language with network primitives for programming distributed systems. We demonstrate that our specifications are useful, by proving the correctness of small, but tricky, synthetic examples involving causal dependency and by verifying a session manager library implemented on top of the distributed database. We use Aneris's facilities for modular specification and verification to obtain a highly modular development, where each component is verified in isolation, relying only on the specifications (not the implementations) of other components. We have used the Coq formalization of the Aneris logic to formalize all the results presented in the chapter in the Coq proof assistant.

The ubiquitous distributed systems of the present day internet often require highly available and scalable distributed data storage solutions. The CAP theorem [GL02] states that a distributed database cannot at the same time provide *consistency*, *availability*, and *partition (failure) tolerance*. Hence, many such systems choose to sacrifice aspects of data consistency for the sake of availability and fault tolerance [Bai+13; Cha+08; Llo+11; Tyu+19]. In those systems different replicas of the database may, at the same point in time, observe different, inconsistent data. Among different notions of weaker consistency guarantees, a popular one is *causal consistency*. With causal consistency different replicas can observe different data, yet, it is guaranteed that data are observed in a causally related order: if a node n observes an operation x originating at node m, then node n must have also observed the effects of any other operation that took place on node m before x. Causal consistency can, for instance, be used to ensure in a distributed messaging application that a reply to a message is never seen before the message itself.

Two simple, illustrative examples of causal dependency are depicted in Figure 4.1; programs executed on different nodes are separated using double vertical bars. Notice that in our setting all keys are uninitialized at the beginning and the read operation returns an optional value indicating whether or not the key is initialized. In both examples, the `read`(x) command returns the value 37, as indicated by the comment in the code, as the preceding `wait` command waits for the effects of `write`$(y, 1)$ to be propagated. In the example on the left (illustrating direct causal dependence) the `write`$(x, 37)$ command immediately precedes the `write`$(y, 1)$ command on the same node; hence any node that observes 1 for key y should also observe 37 for key x. However, in the example on the right, the `write`$(y, 1)$ command is executed on

$$\begin{array}{l|l} \texttt{write}(x, 37) & \texttt{wait}(y = 1) \\ \texttt{write}(y, 1) & \texttt{read}(x) \text{ // reads Some 37} \end{array}$$

$$\begin{array}{l|l|l} \texttt{write}(x, 0) & \texttt{wait}(x = 37) & \texttt{wait}(y = 1) \\ \texttt{write}(x, 37) & \texttt{write}(y, 1) & \texttt{read}(x) \text{ // reads Some 37} \end{array}$$

Figure 4.1: Two examples of causal dependency: direct (left) and indirect (right, [Llo+11]).

the middle node *only after* the value of 37 is observed for key x on that node; hence, in this example too, any node that observes 1 for key y should also observe 37 for key x.

Programming distributed systems is challenging and error-prone [Guo+13], especially in the presence of weaker consistency models like causal consistency which allow concurrent (causally independent) writes [BA12]. Consequently, there have been several efforts in recent years to provide tools for analysis and verification of distributed database systems with weaker notions of consistency, *e.g.*, Gotsman et al. [Got+16], Kaki et al. [Kak+18], Lesani et al. [LBC16], and Xiong et al. [Xio+19]. Those works give a high-level model of a (programming) language with primitives for reading from and writing to a distributed database and provide semantics for that language. The semantics is then used to build sound program analysis tools or to verify correctness of implementations of distributed databases and/or their clients. The semantics presented in the aforementioned works usually keeps track of the history of operations and (directly or indirectly) their dependence graph. A common aspect of these works is that the developed systems, be it a program logic, a program analysis tool, or both, are designed with the express purpose of verifying correctness of closed programs w.r.t. a specific notion of consistency. Thus they do not support general modular verification where components of the program are verified separately, although Lesani et al. [LBC16] do support the verification of databases and their clients independently. Moreover, importantly, the aforementioned works do not scale to verification of full functional correctness of programs and do not scale to a larger setting where the replicated database is just one component of a distributed system.

In this work, we present the first specification and verification of an implementation of a causally-consistent distributed database that supports modular verification of full functional correctness of clients and servers. In the rest of the introduction we briefly discuss the implementation, our verification methodology, and the examples of clients that we have verified against our specification.

Implementation, programming language, and program logic. We implement the pseudo code presented in the seminal work of [Aha+95] for a replicated, distributed database in AnerisLang. AnerisLang [Kro+20a] is a concurrent (multiple threads on each node) ML-like programming language with network primitives (UDP-like sockets) designed for programming distributed systems.[1] Our implementation makes use of a heap-allocated dictionary for storing the key-value pairs and uses networking primitives to propagate updates among replicas. Each replica has an *in-queue* and an *out-queue*. On each replica, there are three concurrently running threads: the send thread, the receive thread, and the apply thread. The send thread sends updates from the out-queue (enqueued by the write operation) to other replicas. The

[1] AnerisLang as presented by Krogh-Jespersen et al. [Kro+20a] features duplicate protection, *i.e.*, every sent message is received at most once. This feature has since been relaxed.

receiver thread receives updates from other replicas over the network and enqueues them in the in-queue. The apply thread applies updates from the in-queue to the local key-value store.

The operational semantics of AnerisLang is formally defined in the Coq proof assistant together with the Aneris program logic [Kro+20a]. The Aneris program logic is a *higher-order distributed concurrent separation logic* which facilitates modular specification and verification of partial correctness properties of distributed programs. The Aneris logic itself is defined on top of the Iris program logic framework [Jun+16; Jun+18b; Jun+15]. We have used the Aneris logic, Iris, and the Iris proof mode [Kre+18; KTB17] to formalize all the results presented in this work in the Coq proof assistant.

Mathematical model and specification. Our Aneris specification of the distributed database is based on a mathematical model tracking the abstract state of the local key-value stores, *i.e.*, the history of updates. Our specification represents this model using Iris's ghost theory to track auxiliary state (state that is *not* physically present at runtime and only tracked logically for verification purposes). We then use Iris invariants to enforce that the physical state of each replica is *consistent* with the tracked history at all times. We further enforce that the histories tracked by the ghost state are *valid*. We will define validity later; for now it suffices to say that, in our work, viewed at a high level of abstraction, causal consistency is a property of *valid* histories.

The history of updates in our mathematical model consists of the following information:

1. For each replica, we track a *local history* of all memory updates that the replica has observed since its initialization. It includes both local write operations (which are observed immediately) and updates due to synchronization with other replicas.

2. We also track an *abstract global memory* that, for each key, keeps track of all write events to that key (by any replica in the system).

We refer to the elements of local histories as *apply events* and to the elements of the abstract global memory as *write events*. Both apply and write events carry all the necessary information about the original update, including the logical time of the corresponding apply or write operation. We model logical time using a certain partial order; see Section 7.5 for more details. The ordering is defined such that it reflects causal order: if the time of event e is strictly less than the time of e', then e' *causally depends* on e, and if the time of e and e' are incomparable, then e and e' are *causally independent*. This allows us to formulate the causal consistency of the distributed database as follows:

> *If a node observes an apply event a, it must have already observed all write events of the abstract global memory that happened before (according to logical time) the write event corresponding to a.* (Causal consistency)

Moreover, we can prove that this property is a consequence of the validity of histories.

The specifications we give to the read and write operations essentially reflect the behaviors

of these operations into the histories. The intuitive reading of our specifications is as follows:[2]

- *Either,* `read`(k) *returns nothing, in which case we know that the local history contains no observed events for key* k*.*
- *Or,* `read`(k) *returns some value* v*, in which case we know that there is an apply event* a *in the local history with value* v*;* a *has a corresponding write event in the global memory; and* a *is a maximal element (w.r.t. time and hence causality) in the local history.* (Read spec)

After the write operation `write`(k, v) *there is a new write event* w *added to the global memory and a new apply event* a *corresponding to it in the local history, and* a *is the* maximum *element (w.r.t. time and hence causality) of the local history, i.e., the event* a *causally depends on all other events in the local history.* (Write spec)

Note that the specification does not say anything about the inter-replica communication, neither does it refer to some explicit causal relation. It merely assert properties of the history tracked in the ghost state. Indeed, it is our invariants that associate the ghost state of a valid history with the physical states of the replicas that allow us to reason about causal consistency. Crucially, this indirection through histories enables us to use the above specifications modularly. This is essentially because our specifications only refer to the relevant parts of the history, *i.e.*, the global memory for the key in question and the local history for the replica performing the operation.

We present our formal specifications for the read and write operations in Section 4.3. However, the specification for write presented in Section 4.3, and used throughout most of the chapter, is not general enough to support modular Iris-style reasoning about clients because it does not support reasoning about concurrent accesses to keys. The reason is that the write operation is not atomic, as required for Iris-style (and, more generally, concurrent-abstract-predicate-style [Din+10]) reasoning using invariants. The read operation is, of course, not atomic either, but the specification for it only involves so-called persistent predicates and hence is general enough. The issue with the write operation is an instance of the known challenge of how to give modular specifications for concurrent modules, see, *e.g.*, Dinsdale-Young et al. [DRG18] and Birkedal and Bizjak [BB17, Section 13]. Hence we use one of the solutions to this modularity challenge and present our full specification for the write operation in so-called HOCAP-style [SBP13], see Section 4.7. With our HOCAP-style specification we do indeed support modular reasoning about clients using Iris invariants. (It does take a little while to get used to HOCAP-style specifications; that is why we present the official specification for the write operation relatively late in the chapter.) Note however that the specification for the write operation given in Section 4.3 is *not* wrong but only weaker than the general HOCAP-style specs given in Section 4.7, and can in fact be derived from it. This rule, as we will see in Section 4.3, can be used in situations where there are no concurrent accesses to the key being written to.

Clients verified. We demonstrate the utility of our specifications by verifying a number of interesting examples, including the two examples presented in Figure 4.1. As a more realistic case study, we implement a *session manager* library that allows clients to communicate with a replica over the network, on top of our distributed database; we use our specifications to

[2] Notice that the read operation returns an optional value.

prove that the session manager satisfies four *session guarantees* for client-centric consistency [Ter+94].

Contributions. In summary, we make the following contributions:

- We present a simple and novel mathematical model of distributed causal memory amenable to building appropriate abstractions for reasoning about implementing and using such memory.
- On top of this model, we define high-level modular specifications for reading from and writing to a causally-consistent distributed database. Our specifications are *node-local* and *thread-local*: in the client's code where multiple threads (possibly on different nodes) interact with the database, all components can be verified separately from each other.
- We show that those high-level specifications are actually met by the original description of a causally-consistent distributed database from the 1995 seminal paper [Aha+95] which we have implemented in a realistic ML-like language with explicit network primitives.
- We show that our specifications provide the expected causality guarantees on standard examples from the literature. Moreover, we implement a session manager library that allows clients and replicas to run on different nodes, and show that our specifications imply the session guarantees for library clients.
- We have formalized all results on top of the Aneris Logic in the Coq proof assistant.

Outline. We start by presenting our implementation of a causally-consistent distributed database in AnerisLang in Section 4.1. This allows us to match the intuition behind the key ideas of our approach with concrete code. Then, in Section 7.5 we present those parts of our model of causality that are needed for client-side reasoning. In Section 4.3 we show how to turn the model into abstract program logic predicates and present the specifications of the distributed database operations. We also show how the specifications can be used to reason about the client program examples presented above. In Section 4.5 we present a more extensive case study of a client program, a sesssion-manager library, and show how we can use our specifications to reason about session guarantees. In Section 4.6 we then prove that the implementation of the distributed database meets our specification. In Section 4.7 we present the HOCAP-style specification for the write operation. After discussing related work in Section 5.7 we conclude and discuss future work in Section 7.9.

4.1 A causally-consistent distributed database

In this section we present our AnerisLang implementation[3] of the causally-consistent distributed database described by Ahamad et al. [Aha+95]. See Figure 4.2 for the implementation. Conceptually, the implementation can be split into two parts:

[3] AnerisLang is an ML-like language with a syntax close to OCaml. For readability purposes, the code we show here makes use of some OCaml constructs that are formally slightly different in AnerisLang (*e.g.*, we write [], :: for lists, which are implemented in AnerisLang using pairs).

```
let init l i =
  let db = ref (Dict.empty ()) in
  let t = ref (VC.make (length l) 0) in
  let (iq, oq) = (ref [], ref []) in
  let lock = newlock () in
  let skt = socket () in
  socketbind skt (List.nth l i);
  fork (apply db t lock iq i);
  fork (send_thread i skt lock l oq);
  fork (receive_thread skt lock iq);
  (read db lock, write db t oq lock i)

let read db lock k =
  acquire lock;
  let r = Dict.lookup k !db in
  release lock; r

let write db t oq lock i k v =
  acquire lock;
  t := VC.incr !t i;
  db := Dict.insert k v !db;
  oq := (k, v, !t, i) :: !oq;
  release lock

let receive_thread skt lock iq =
  listen skt (fun handler m from ->
    acquire lock;
    iq := (we_deser m) :: !iq;
    release lock;
    listen skt handler)
```

```
let check t i w =
  let (wt, wo) = (pi3 w, pi4 w) in
  let rec aux l r j = match (l, r) with
    | a :: l', b :: r' ->
      (if j = wo then a = b + 1 else a <= b)
      && aux l' r' (j + 1)
    | [], [] -> true
    | _      -> false
  in (i != wo) && (wo < length t) && (aux wt t 0)

let rec apply db t lock iq i =
  acquire lock;
  match (find (check !t i) !iq) with
   | Some (w, iq') ->
      iq := iq';
      db := Dict.insert (pi1 w) (pi2 w) !db;
      t := VC.incr !t (pi4 w)
      | None -> ()
  end;
  release lock; apply db t lock iq i ()

let rec send_thread i skt lock l oq =
  acquire lock;
  match !oq with
  | [] -> release lock
  | w :: oq' ->
    oq := oq'; release lock;
    broadcast skt (we_ser w) i l
  end;
  send_thread i skt lock l oq
```

Figure 4.2: Implementation of a causally-consistent distributed database replica.

- Three operations, `init`, `read`, and `write` allow clients respectively to initialize a local database, and read from and write to the distributed database on a particular replica i.
- Three other operations, `send_thread`, `receive_thread`, and `apply` are spawned during the initialization as non-terminating concurrently running threads that propagate updates between initialized replicas and which enforce that every replica applies locally all other replicas' updates in some order that respects causal dependencies.

Because `init` returns to the user a pair of partially applied `read` and `write` functions, the user only needs to supply `read` with a key on which the local store `db` should be read and to supply `write` with a key and value for which the `db` should be updated. The sending, receiving, and apply threads, which are running in a loop concurrently with the user's calls to `read` and `write`, are hidden from the user, who does not need to know about the underlying message-passing implementation. Thus once a replica is initialized, the user will access the local memory on the replica as if they were manipulating locally one global distributed memory.

Each replica has a local heap on which it allocates and further makes use of the following data:

- a dictionary `db` for storing the key-value pairs to implement causal memory locally;
- a vector clock `t` to timestamp outgoing updates and check dependencies of incoming updates. A vector clock t consists of a vector of n natural numbers, where the j^{th} number $t[j]$ indicates how many updates has been applied locally so far from the replica j;

- an in-queue `iq` and an out-queue `oq` for receiving/sending local updates among replicas;
- a UDP socket `skt` bound to the socket address of the replica;
- and a `lock` to control sharing of above-mentioned data among different concurrent threads.

Vector clocks are the key mechanism by which the implementation enforces that the order in which updates are applied locally on each replica respects the causal order of the entire system. This is enforced in the following way.

To make use of vector clocks, each `write k v` call is associated with a *write event* (k, v, t, i) where projection $t[i]$ describes the number of calls to `write` executed on a replica i (including the current call itself), and all other projections $t[j]$ describe the number of updates received from replica j and applied locally on replica i at the time when the update `write k v` takes place.

When the dictionary `db` and the vector clock `t` are updated by the call `write k v` on a replica i, before the call terminates, it adds the associated write event (k, v, t, i) to the outgoing queue `oq`. Once the `send_thread` acquires the lock to get access to the queue `oq`, it serializes the write event into a message and broadcasts to all other replicas.

To receive those update messages, each replica runs a `receive_thread` which listens on the socket `skt`, and when it gets a new message `msg` from the replica o, it deserializes it back to a write event $w = (k, v, t, o)$, and adds it to the incoming queue `iq`. As a matter of notation, for a write event $w = (k, v, t, o)$, we write $w.k$ for the key k, $w.v$ for the value v, *etc.*

When the `apply` operation acquires the lock, it consults the incoming queue `iq` in search of a write event that can be applied locally. To this end, it calls the `find (check !t i) !iq` subroutine with the current value of the vector clock `!t` and the index of the replica i. The idea is to search through the queue `iq` until a write event w that passes the test is found (`check !t i w` holds) and retrieved from `iq`, which is the case when the following conditions are satisfied:

1. The origin $w.o$ of the event w must be different from i, so that w corresponds indeed to an external write operation, and be within bounds $[0, n[$ (recall that `t` is a vector of length n).
2. For the projection $j = w.o$ (the event w's own origin), the number $w.t[j]$ must be equal to $\texttt{!t}[j] + 1$, which captures the intuition that the event w must be *the most recent write from the replica j that the replica i did not yet observe.*
3. For all other projections p different from j, the condition $w.t[p] \leq \texttt{!t}[p]$ should hold, which captures the intuition that if the write event w passes the dynamic check, it means that *any memory update on which w causally depends has already been locally applied by the replica i.*

We remark that the pseudo-code in the original paper by Ahamad et al. [Aha+95] requires a reliable network, *e.g.*, that network communication happens using TCP. This is important for showing liveness properties (*e.g.*, every replica eventually applies all messages from other replicas). In this work, we focus on safety properties and the properties we show (*e.g.*, on any replica, all updates that have been applied are causally consistent) are met by our implementation regardless of whether the network is reliable or not.

Events

$$
\begin{array}{lcl}
(k, v, t, o) & \in & \mathit{WriteEvent} \triangleq \mathit{Keys} \times \mathsf{Value} \times \mathsf{Time} \times \mathbb{N} \\
(k, v, t, o, m) & \in & \mathit{ApplyEvent} \triangleq \mathit{Keys} \times \mathsf{Value} \times \mathsf{Time} \times \mathbb{N} \times \mathbb{N} \\
\mathit{Maximals(X)} & \triangleq & \{x \mid x \in X \land \forall y \in X.\, \neg(x.t < y.t)\} \\
\mathit{Maximum(X)} & \triangleq & \begin{cases} \mathsf{Some}\ x & \text{if } x \in X \land \forall y \in X.\, x \neq y \implies y.t < x.t \\ \mathsf{None} & \text{otherwise} \end{cases} \\
\mathit{Observe} & : & \wp^{\mathsf{fin}}(\mathit{ApplyEvent}) \xrightarrow{\mathsf{fin}} \mathit{ApplyEvent} \\
\lfloor \cdot \rfloor & : & \mathit{ApplyEvent} \xrightarrow{\mathsf{fin}} \mathit{WriteEvent}
\end{array}
$$

Memory

$$
\begin{array}{lcl}
s_i & \in & \mathit{LocalHistory} \triangleq \wp^{\mathsf{fin}}(\mathit{ApplyEvent}) \\
M & \in & \mathit{GlobalMemory} \triangleq \mathit{Keys} \xrightarrow{\mathsf{fin}} \wp^{\mathsf{fin}}(\mathit{WriteEvent})
\end{array}
$$

States

$$
\begin{array}{lcl}
\{|\, M\, ; s_1, \ldots, s_n \,|\} & \in & \mathit{GlobalState} \triangleq \mathit{GlobalMemory} \times \mathit{LocalHistory} \times \ldots \times \mathit{LocalHistory} \\
\mathit{Valid}_G & : & \mathit{GlobalState} \to \mathit{iProp}
\end{array}
$$

Properties of Valid States

$$
\begin{array}{ll}
\textit{(Local Extensionality)} & \forall a_1, a_2 \in s_i.\, vc(a_1) = vc(a_2) \implies a_1 = a_2 \\
\textit{(Global extensionality)} & \forall k_1, k_2 \in \mathsf{dom}(M), w_1 \in M(k_1), w_2 \in M(k_2). \\
& \quad vc(w_1) = vc(w_2) \implies w_1 = w_2 \\
\textit{(Causal consistency)} & \forall k \in \mathsf{dom}(M), w \in M(k). \\
& \quad (\exists a \in s_i.\, vc(w) < vc(a)) \implies \exists a' \in s_i.\, \lfloor a' \rfloor = w \\
\textit{(Origin of write events)} & \forall k \in \mathsf{dom}(M), w \in M(k). \\
& \quad \exists i \in \{0..n-1\}, a \in s_i.\, i = \mathit{origin}(w) \land \lfloor a \rfloor = w \\
\textit{(Origin of apply events)} & \forall a \in s_i.\, \exists k \in \mathsf{dom}(M), w \in M(k).\, \lfloor a \rfloor = w
\end{array}
$$

Figure 4.3: Mathematical terminology for a distributed causal memory with an abstract notion of validity.

4.2 Mathematical model

In this section we formalize the key ideas of our mathematical model of causality. Figure 4.3 shows the model definitions and properties needed to reason about clients.

In the model, a write event is represented much as in the implementation, namely as a four-tuple (k, v, t, o) consisting of a key, a value, the time, and the index of the replica on which the write event happened. In the concrete implementation time is represented using vector clocks, but to reason about client code, all we need is an abstract notion of time, and therefore our model uses a notion of logical time, represented by a partial order $\leq$ (we write $<$ for the strict version of it). We can decide whether two write events w_1 and w_2 are causally

related by comparing their times: if $w_1.t < w_2.t$, then w_1 must have *happened before* w_2, and w_2 is *causally dependent* on w_1. When $w_1.t$ and $w_2.t$ are incomparable, then the events w_1 and w_2 are *causally unrelated*.

To account for how write events are applied locally on each replica we use the notion of an *apply event*. Thus an apply event only makes sense in the context of a particular replica. Formally, given a replica i, an *apply event* is represented by a five-tuple $a = (k, v, t, o, m)$, where m is the number of write events applied on replica i. We refer to m as the *sequence identifier* of a. When $i = o$, the apply event corresponds to a write operation invoked on the replica itself, whereas if $i \neq o$, then the apply event corresponds to a write event received from replica i. Given an apply event $a = (k, v, t, o, m)$, we denote by $\lfloor a \rfloor$ the write event (k, v, t, o), which we refer to as the *erasure of* a.

As explained in the introduction, we keep track of all write and apply events. The *local history* of replica i, written s_i, is the set of all apply events observed by the replica since its initialization. The *abstract global memory*, written M, is a finite map from keys to finite sets of write events. We model the local key-value store for a replica i simply as a finite map from keys to values.

Given a set X of write or apply events, $\mathrm{Maximals}(X)$ (resp. $\mathrm{Maximum}(X)$) denotes the set of maximal events (resp. the maximum event) w.r.t. the time ordering. Note that, for any events $e, e' \in \mathrm{Maximals}(X)$, the time of e and e' are incomparable and hence e and e' are causally unrelated. Given a non-empty set of apply events A, the event $\mathsf{Observe}(A)$ is the maximum element of A w.r.t. the ordering of sequence identifiers. (If A is empty, we let $\mathsf{Observe}(A)$ be some default apply event).

The global state $\{|\, M \,;\, s_1, \ldots, s_n \,|\}$ consists of the abstract global memory and the local histories of all replicas. Just keeping track of apply and write events is obviously not enough; we also need to make sure that the local histories are always in a consistent state w.r.t. the abstract global memory. This will be expressed using a notion of *validity*. The client need not know the precise definition of validity, but only that there is some predicate Valid_G on global states, and that valid global states satisfy the properties listed in the figure. The local and global extensionality properties express that apply events and write events are uniquely identified by their times. The causal consistency property formalizes the intuitive description of causality from the introduction. The origin of write property expresses that for every write event there is at least one corresponding apply event on the replica where the write occurred. The origin of apply event property says that every apply event must also be recorded in the abstract global memory.

4.3 Specification

As discussed in the introduction, we use the Aneris program logic built on top of the Iris program logic framework for our verification. In this section we present the Aneris specifications of our distributed database operations: `read`, `write`, and `init`. A summary of the full specification presented in the following is found in Figure 4.4.

In Section 4.3.1 we call to mind those aspects of Aneris and Iris that are necessary for following the rest of the chapter and introduce the abstract Iris predicates that are provided to clients and used in the specification of the database operations. In Section 4.3.2 we present some laws that hold for the abstract predicates and which the client may use for client-side reasoning; the laws are the program logic version of the mathematical model from the previ-

$$
\begin{aligned}
&\exists\ \mathsf{GlobalInv} : \mathit{iProp}.\\
&\exists\ (\cdot) \rightharpoonup_s (\cdot) : \mathit{Keys} \to \wp^{\mathsf{fin}}(\mathit{WriteEvent}) \to \mathit{iProp}.\\
&\exists\ (\cdot) \rightharpoonup_u (\cdot) : \mathit{Keys} \to \wp^{\mathsf{fin}}(\mathit{WriteEvent}) \to \mathit{iProp}.\\
&\exists\ \mathsf{Seen} : \mathbb{N} \to \mathit{LocalHistory} \to \mathit{iProp}.\\
&\exists\ \mathsf{Snap} : \mathit{Keys} \to \wp^{\mathsf{fin}}(\mathit{WriteEvent}) \to \mathit{iProp}.\\
&\exists\ \mathsf{initToken} : \mathbb{N} \to \mathit{iProp}.\\
&\exists\ \Phi_{\mathsf{DB}} : \mathit{Message} \to \mathit{iProp}.\\
&\qquad \mathsf{Snap}(x, h) * \mathsf{Snap}(x, h') \vdash \mathsf{Snap}(x, h \cup h')\\
&\wedge \quad x \rightharpoonup_u h \vdash x \rightharpoonup_u h * \mathsf{Snap}(x, h)\\
&\wedge \quad \ldots\\
&\wedge \quad \boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i, s) * x \rightharpoonup_u h \vdash\\
&\qquad \Rrightarrow_{\mathcal{E}} \forall a \in s, w \in h.\ vc(w) < vc(a) \Rightarrow \exists a' \in s_{|x}.\ \lfloor a' \rfloor = w\\
&\wedge \quad \mathsf{True} \vdash \Rrightarrow_{\mathcal{E}} \boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \Big(\mathop{\Large *}_{0 \le i < \mathit{length}(\mathsf{Addrlist})} \mathsf{initToken}(i) \Big) *\\
&\qquad \Big(\mathop{\Large *}_{k \in \mathit{Keys}} k \rightharpoonup_u \varnothing \Big) * \mathsf{initSpec}(\texttt{init})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{initSpec}(\texttt{init}) \triangleq\\
&\quad \left\{ \begin{array}{l} \mathsf{initToken}(i) * \mathsf{Fixed}(A) * * \mathsf{Addrlist}[i] = (ip_i, p) *\\ (ip_i, p) \in A * \mathit{isList}(\mathsf{Addrlist}, v) * \mathsf{FreePorts}(ip_i, \{p\}) * \mathop{\Large *}_{z \in \mathsf{Addrlist}} z \Rrightarrow \Phi_{\mathsf{DB}} \end{array} \right\}\\
&\qquad \langle ip_i;\ \texttt{init}(i, v) \rangle\\
&\quad \left\{ (\mathsf{rd}, \mathsf{wr}).\ \mathsf{Seen}(i, \varnothing) * \mathsf{readSpec}(\mathsf{rd}, i) * \mathsf{writeSpec}(\mathsf{wr}, i) \right\}\\
&\mathsf{readSpec}(\mathsf{read}, i) \triangleq\\
&\quad \{\mathsf{Seen}(i, s)\}\\
&\qquad \langle ip_i;\ \mathsf{read}(x) \rangle\\
&\quad \left\{ v.\ \begin{array}{l} \exists s' \supseteq s.\ \mathsf{Seen}(i, s') * (v = \mathsf{None} \wedge s'_{|x} = \varnothing) \vee\\ (\exists a \in s'_{|x}.\ v = \mathsf{Some}\ a.v * \mathsf{Snap}(x, \{\lfloor a \rfloor\}) * a \in \mathrm{Maximals}(s'_{|x}) * \mathsf{Observe}(s'_{|x}) = a) \end{array} \right\}\\
&\mathsf{writeSpec}(\texttt{write}, i) \triangleq\\
&\quad \forall \mathcal{E}, k, v, s, P, Q.\ \mathcal{N}_{\mathsf{GI}} \subseteq \mathcal{E} \Rightarrow\\
&\quad \Box \left(\begin{array}{l} \forall s', a.\ (s \subseteq s' * a \notin s' * a.k = k * a.v = v * P)\\ \qquad {}^{\top}\!\Rrightarrow\!\!\!\!\!\!\ast_{\mathcal{E}}\ \forall h.\ \left(\begin{array}{l} \lfloor a \rfloor \notin h * \lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\}) * k \rightharpoonup_s h * \\ \mathsf{Seen}(i, s' \uplus \{a\}) * \mathrm{Maximum}(s' \uplus \{a\}) = a \end{array} \right)\\ \qquad \Rrightarrow\!\!\!\!\!\!\ast_{\mathcal{E} \setminus \mathcal{N}_{\mathsf{GI}}}\ k \rightharpoonup_s h \uplus \{\lfloor a \rfloor\} * {}_{\mathcal{E}}\!\Rrightarrow_{\top} Q\ a\ h\ s' \end{array} \right) \mathrel{-\!\!*}\\
&\quad \{P * \mathsf{Seen}(i, s)\}\ \langle ip_i; \texttt{write}(k, v) \rangle\ \{v.v = () * \exists h, s', a.\ s \subseteq s' * Q\ a\ h\ s'\}
\end{aligned}
$$

Figure 4.4: Summary of the full database specification.

$$
\begin{aligned}
P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \Rightarrow Q \mid P \vee Q \\
& \mid \forall x.\ P \mid \exists x.\ P \mid \cdots && \text{(higher-order logic)} \\
& \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_n v \mid \{P\}\ \langle n; e\rangle\ \{x.\ Q\} && \text{(separation logic)} \\
& \mid \Box P \mid \boxed{P}^{\mathcal{N}} \mid {}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P && \text{(Iris)} \\
& \mid z \Mapsto \Phi \mid \mathsf{Fixed}(A) \mid \mathsf{FreePorts}(ip, \mathcal{P}) && \text{(Aneris)}
\end{aligned}
$$

Figure 4.5: The fragment of Iris and Aneris relevant for this chapter.

ous section. Then we present the specifications for read and write operations in Section 4.3.3; these specifications are node local and do not involve any distributed-systems-specific aspects. In Section 4.3.4 we explain how the distributed database is initialized and present the specification for the initialization operation; this specification naturally involves distributed-systems-specific aspects. Finally, in Section 4.4 we give a proof sketch of the client programs from the introduction.

4.3.1 *Iris, Aneris, resources, and tracking the state of the distributed database*

Figure 4.5 contains the fragment of the Iris and Aneris logics that is relevant for this work. Here P and Q range over Iris propositions. We write *iProp* for the universe of Iris propositions. Iris is a higher-order logic that features separation logic primitives: separating conjunction, $*$, and magic wand, $-\!\!*$, also known as the separating implication. The points-to proposition, $\ell \mapsto_n v$, expresses exclusive ownership of memory location ℓ with value v in the heap of the node n. The *separating* nature of the separating conjunction, and the *exclusive* nature of the points-to propositions can be seen in the rule $\ell \mapsto_n v * \ell' \mapsto_n v' \vdash \ell \neq \ell'$, where $\vdash$ is the entailment relation on Iris propositions. This rule states that separating conjuncts assert ownership over *disjoint* parts of the heap. The Hoare triple $\{P\}\ \langle n; e\rangle\ \{x.\ Q\}$ is used for partial correctness verification of distributed programs. Intuitively, if the Hoare triple $\{P\}\ \langle n; e\rangle\ \{x.\ Q\}$ holds, then whenever the precondition P holds, e is safe to execute *on node* n and whenever e reduces to a value v on node n then that value should satisfy the postcondition $Q[v/x]$; note that x is a binder for the resulting value. The proposition $\boxed{P}^{\mathcal{N}}$ is an Iris invariant: it asserts that the proposition P should hold at all times. The invariant name $\mathcal{N}$ is used for bookkeeping, to prevent the same invariant from being reopened in a nested fashion which is unsound. The update modality, ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P$, asserts that Iris resources can be updated in such a way that P would hold. The masks $\mathcal{E}_1$ and $\mathcal{E}_2$ (sets of invariant names) indicate which invariants can be accessed during this update (those in $\mathcal{E}_1$) and which invariants should remain accessible after the update ($\mathcal{E}_2$). Whenever both masks of an update modality are the same mask $\mathcal{E}$, which is most often the case, we write $\Rightarrow_{\mathcal{E}}$ instead of ${}_{\mathcal{E}}\!\Rightarrow_{\mathcal{E}}$. We write $P \ {}_{\mathcal{E}_1}\!\Rrightarrow_{\mathcal{E}_2}\ Q$ and $P \Rrightarrow_{\mathcal{E}} Q$ as a shorthand for $P \mathrel{-\!\!*} {}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} Q$ and $P \mathrel{-\!\!*} \Rightarrow_{\mathcal{E}} Q$, respectively. We write $\top$ for the mask that allows access to all invariants. We will explain the Aneris specific propositions later on.

Ephemeral versus persistent propositions. Iris, and by extension Aneris, is a logic of resources. That is to say that propositions can assert (exclusive) ownership of resources. In this regards, propositions can be divided into two categories: *ephemeral* propositions and *persistent* propo-

Table 4.1: Propositions to track the state of the key-value store.

Proposition	**Intuitive meaning**
$\mathsf{Seen}(i, s)$	The set s is *a causally closed subset of* the local history of replica i
$\mathsf{Snap}(k, h)$	The set h is *a subset of* the global memory for key k
$k \rightharpoonup_u h$	The global memory for key k is *exactly* h

We say s is a *causally closed* subset of s' if:
$$s \subseteq s' \land \forall a_1, a_2 \in s'.\ vc(a_1) < vc(a_2) \land a_2 \in s \Rightarrow a_1 \in s.$$

sitions. Ephemeral propositions represent transient facts, *i.e.*, facts that later stop from being true. The quintessential ephemeral proposition is the points-to proposition; the value of the memory location ℓ can change by performing a write on ℓ—this can be seen in the specs for writing to a memory location:

$$\{\ell \mapsto_n v\}\ \langle n; \ell := w\rangle\ \{x.\ x = () * \ell \mapsto_n w\}$$

One important aspect of ephemeral propositions is that they give us precise information about the state of the program: having $\ell \mapsto_n v$ implies that the value stored in memory location ℓ on node n is v. The upshot of this is that while we own a points-to proposition for a location ℓ, no other concurrently running thread can update the value of ℓ. Hence, to allow concurrent accesses to a location, its points-to proposition should be shared, *e.g.*, using Iris invariants. Persistent propositions, as opposed to ephemeral propositions, express knowledge; these propositions never cease to be true. The persistently modality $\Box$ is used to assert persistence of propositions: $\Box P$ holds, if P holds, and P is persistent. Hence, the logical entailments $\Box P \vdash P$ and $\Box P \vdash \Box\Box P$ hold in Iris. Formally, we say a proposition is persistent if $P \dashv\vdash \Box P$, where $\dashv\vdash$ is the logical equivalence of Iris propositions. Persistent propositions, unlike ephemeral ones, can be freely duplicated, *i.e.*, $\Box P \dashv\vdash \Box P * \Box P$. The quintessential persistent propositions in Iris are Iris invariants. In addition, Hoare triples are also defined to always be persistent. This intuitively means that all the requirements for the correctness of the program with respect to the postcondition are properly captured by the precondition.

Iris predicates to represent the state of the key-value store. Recall the intuitive specifications that we gave for the read and write operations on our distributed database in the introduction. These specs only assert that certain write/apply events are added to the global memory/local history. Hence, it suffices to have a persistent proposition in the logic that asserts the partial information that certain events are indeed part of the local history or global memory. For this purpose, we introduce the persistent abstract predicates Seen and Snap which intuitively assert knowledge of a subset of the local history, and global memory, respectively. These abstract predicates and their intuitive meaning are presented in Table 4.1. Notice that the Seen predicates assert knowledge of a subset of the local history that is *causally closed* as defined in the figure. We will discuss the significance of causal closure later.

In addition to the partial knowledge about the global memory represented using the Snap predicate, it is also useful to track the precise contents of the global memory for each key—see the example presented in Section 4.4. We do this using the ephemeral abstract proposition $k \rightharpoonup_u h$ which, intuitively, asserts that the set of *all* write events for the key k is h. We can track the precise contents of the global memory because all write events in the global memory

Properties of global memory, *i.e.,* Snap and $\rightharpoonup_u$ predicates:

$$\mathsf{Snap}(k,h) * \mathsf{Snap}(k,h') \vdash \mathsf{Snap}(k, h \cup h') \qquad \text{(Snap union)}$$

$$k \rightharpoonup_u h \vdash k \rightharpoonup_u h * \mathsf{Snap}(k,h) \qquad \text{(Take Snap)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * k \rightharpoonup_u h * \mathsf{Snap}(k,h') \vdash \Rrightarrow_{\mathcal{E}} k \rightharpoonup_u h * h' \subseteq h \qquad \text{(Snap inclusion)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Snap}(k,h) * \mathsf{Snap}(k,h') \vdash \Rrightarrow_{\mathcal{E}} \forall w \in h, w' \in h'.\ vc(w) = vc(w') \Rightarrow w = w' \qquad \text{(Snap ext.)}$$

Properties of local histories, *i.e.,* the Seen predicate:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Seen}(i,s') \vdash \Rrightarrow_{\mathcal{E}} \mathsf{Seen}(i, s \cup s') \qquad \text{(Seen union)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Seen}(i',s') \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, a' \in s'.\ vc(a) = vc(a') \Rightarrow a.k = a'.k \wedge a.v = a'.v \qquad \text{(Seen glo. ext.)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Seen}(i,s') \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, a' \in s'.\ vc(a) = vc(a') \Rightarrow a = a' \qquad \text{(Seen loc. ext.)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * a \in s \vdash \Rrightarrow_{\mathcal{E}} \exists h.\ \mathsf{Snap}(a.k, h) * \lfloor a \rfloor \in h \qquad \text{(Seen provenance)}$$

Causality in terms of resources and predicates:

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \mathsf{Seen}(i,s) * \mathsf{Snap}(k,h) \vdash \Rrightarrow_{\mathcal{E}} \forall a \in s, w \in h.\ vc(w) < vc(a) \Rightarrow \exists a' \in s_{|k}.\ \lfloor a' \rfloor = w \qquad \text{(Causality)}$$

Figure 4.6: Laws governing database resources. The mask $\mathcal{E}$ is any arbitrary mask that includes $\mathcal{N}_{\mathsf{GI}}$. $s_{|k}$ denotes the set of apply events in s with key k: $s_{|k} \triangleq \{a \in s \mid a.k = k\}$

can only originate from a write operation on the distributed database. On the other hand, we cannot have precise knowledge about local histories because at any point in time, due to the concurrent execution of a replica's apply function, a replica may observe new events.

In addition to the abstract predicates just discussed, the client will also get access to a global invariant $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$ which, intuitively, asserts that there is a valid global state, and that the predicates Seen, Snap, and $\rightharpoonup_u$ track this global state.[4] Clients need not know the definition of this invariant and can just treat it as an abstract predicate.

(For Iris experts we remark that the abstract predicates in Table 4.1 are all timeless, which simplifies client-side reasoning [Jun+18b]).

4.3.2 *Laws governing database resources*

The laws governing the predicates Seen, Snap, and $\rightharpoonup_u$, are presented in Figure 4.6. The laws presented in this figure, with the exception of one law that we will discuss in Section 4.7, are *all* the laws that are necessary for client-side reasoning about our distributed database. Notice

[4] $\mathcal{N}_{GI}$ is a fixed name of the global invariant.

that most of these laws only hold under the assumption that the global invariant holds. This can also be seen in the fact that they are expressed in terms of an update modality with a mask that enables access to the global invariant. All of these laws make intuitive sense based on the intuitive understanding of the predicates Seen, Snap, and $\rightharpoonup_u$. For instance, the law (Snap union) asserts that if we know that the sets h and h' are both subsets of the global memory for a key k, then so must the set $h \cup h'$. The extensionality laws essentially state that events are uniquely identifiable with their time: if two events have the same time, then they are the same event. The only caveat is that in case of the law (Seen global extensionality): if two apply events *on two different replicas* have the same time, then they must agree on their key and value, but not on their sequence identifiers which represent the order in which events are applied locally on the replica. Note that the law (Seen union), as opposed to the law (Snap union), requires access to the invariant. This is because, we need to establish causal closure (see Table 4.1) for $s \cup s'$ in the conclusion of the law with respect to the local history tracked in the global invariant.

The most important law in Figure 4.6 is the law (Causality). This law allows us to reason about causality: if a replica i has observed an event a that has a time greater than a write event w, w causally depends on a, then replica i must have also observed w (it must have a corresponding apply event a'). Notice that the *causal closure* property of local histories s for which we have $\mathsf{Seen}(i, s)$ in Table 4.1 is crucial for the (Causality) law to hold.

4.3.3 Specs for the read and write operations

Read specification. The read specification looks as follows.

$$
\begin{array}{l}
\textsc{ReadSpec} \\
\{\mathsf{Seen}(i, s)\} \\
\quad \langle ip_i;\ \mathtt{read}(k) \rangle \\
\left\{ \begin{array}{l}
\quad \exists s' \supseteq s.\ \mathsf{Seen}(i, s') * (v = \mathsf{None} \wedge {s'}_{|k} = \varnothing) \vee \\
v.\ (\exists a \in {s'}_{|k}.\ v = \mathsf{Some}\ a.v * \mathsf{Snap}(k, \{\lfloor a \rfloor\}) * \\
\quad a \in \mathrm{Maximals}({s'}_{|k}) * \mathsf{Observe}({s'}_{|k}) = a)
\end{array} \right\}
\end{array}
$$

The post condition of the operation states formally, in the language or Aneris logic, the intuitive explanation that we described in the introduction. It asserts that the client gets back a set of apply events s', $\mathsf{Seen}(i, s')$, observed by replica i performing the read operation such that $s' \supseteq s$. The reason for the $s' \supseteq s$ relation is that during the time since performing the last operation by replica i, *i.e.*, when we had observed the set s, some write events from other replicas may have been applied locally.

When $\mathtt{read}(k)$ is executed on a replica i, it either returns None or Some v for a value v. If it returns None, then the local memory does not contain any values for key k. Hence the local history s' restricted to key k, ${s'}_{|k}$ should be empty *cf.* the definition of ${s'}_{|k}$ in Figure 4.6. Otherwise, if it returns Some v, then the local memory contains the value v for key k. This can happen only if the local memory of the replica at the key k has been updated, and the latest update for that key has written the value v. Consequently, the local history s' must have recorded this update as *the latest apply event* a for the key k, *i.e.*, $\mathsf{Observe}({s'}_{|k}) = a$. Hence $a \in \mathrm{Maximals}({s'}_{|k})$.

One may wonder why a is not the maximum element, but only in the set of maximal elements. To see why, suppose that just before the read operation was executed, two exter-

nal causally-unrelated writes have been applied locally on replica i, so that the local history recorded them as two distinct apply events whose times are incomparable. Naturally, one of two writes must have been applied before the other and the latest observed apply event *must* correspond to the subsequent second write event. However, as the apply operation is hidden from the client, there is no way for the client to observe which of the two writes was the last. Consequently, all the client can know is that the last observed event a is *one of the most recent* local updates for key k, *i.e.*, among the maximal elements. Naturally, the write event $\lfloor a \rfloor$ should be in the abstract global memory. This is expressed logically by the proposition $\mathsf{Snap}(k, \{\lfloor a \rfloor\})$.

Write specification. The (simplified) write specification looks as follows.

$$
\begin{array}{l}
\textsc{WriteSpec} \\
\{\mathsf{Seen}(i, s) * k \rightharpoonup_u h\} \\
\quad \langle ip_i; \texttt{write}(k, v) \rangle \\
\left\{ \begin{array}{l} \exists s' \supseteq s.\, \exists a.\, k = a.k * v = a.v * \mathsf{Seen}(i, s' \uplus \{a\}) * k \rightharpoonup_u h \uplus \{\lfloor a \rfloor\} * \\ a = \mathrm{Maximum}(s' \uplus a) * \lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\}) \end{array} \right\}_{-}
\end{array}
$$

The postcondition of the specification ensures that after the execution of $\texttt{write}(k, v)$, the client gets back the resources $k \rightharpoonup_u h \uplus \{\lfloor a \rfloor\}$ and $\mathsf{Seen}(i, s' \uplus \{a\})$, where a and $\lfloor a \rfloor$ are respectively the apply and write events that model the effect of the write operation. The mathematical operation $\uplus$ is the disjoint union operation on sets; $A \uplus B$ is undefined if $A \cap B \neq \varnothing$.

As for read, the new set of apply events s' can be a superset of s. Contrary to read, the postcondition for write states that $a = \mathrm{Maximum}(s' \uplus a)$, *i.e.*, that a is actually *the most recent* apply event. This matches the intuition that the update $\texttt{write}(k, v)$ causally depends on any other apply event previously observed at this replica.

While a is the maximum apply event locally, its erasure $\lfloor a \rfloor$ is only guaranteed to be among the maximal write events, *i.e.*, $\lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\})$. Intuitively, this is because there can be other write events in h, performed by other replicas, that we have not yet locally observed. As those events are not observed on our replica, the newly added write event $\lfloor a \rfloor$ does not causally depend on them and hence does not have a strictly greater time—in practice those write events have times that are incomparable to that of $\lfloor a \rfloor$ as neither depend on the other.

4.3.4 Initializing the distributed database

Our distributed database must be initialized before it is used. Initialization takes place in two phases:

1. Initialization of resources and establishing the global invariant. This phase is a *logical* phase, *i.e.*, it does not correspond to any program code.
2. Initialization of the replica. This phase corresponds to the execution of the `init` function—see Figure 4.2.

Importantly, in the spirit of modular verification, our methodology for verifying client programs is to *assume* that the library is initialized when we verify parts of the client program that interact with the read and write functions. We only later *compose* these proofs with the

InitSpec

$$\left\{\begin{array}{l}\mathsf{initToken}(i) * \mathsf{Fixed}(A) * * \mathsf{Addrlist}[i] = (ip_i, p) * \\ (ip_i, p) \in A * \mathit{isList}(\mathsf{Addrlist}, v) * \mathsf{FreePorts}(ip_i, \{p\}) * \ast_{z \in \mathsf{Addrlist}}\ z \Rrightarrow \Phi_{\mathsf{DB}}\end{array}\right\}$$

$$\langle ip_i;\ \texttt{init}(i, v)\rangle$$

$$\left\{(\mathsf{rd}, \mathsf{wr}).\ \ \mathsf{Seen}(i, \varnothing) * \mathsf{readSpec}(\mathsf{rd}, i) * \mathsf{writeSpec}(\mathsf{wr}, i)\right\}$$

Figure 4.7: Specification for `init`.

proof corresponding to the initialization of the system—we have indeed followed this methodology in verifying all the examples discussed in this chapter. Below we discuss initialization, starting with phase 2.

The specification of the `init` function (Phase 2). Figure 4.7 shows the specification for the `init` function. The postcondition states that the new replica has not observed any events, *i.e.*, $\mathsf{Seen}(i, \varnothing)$. Furthermore, the `init` function returns a pair of functions that satisfy, respectively, the read and write specifications discussed earlier. This is formally written as $\mathsf{readSpec}(rd, i)$, and $\mathsf{writeSpec}(wr, i)$, respectively. The precondition of the `init` function is slightly more elaborate. Addrlist is the list of socket addresses (pairs of an ip address and a port) of all replicas of the database, *including* the replica being initialized. Hence, the i^{th} element of the list should be the socket address (ip_i, p) which this replica uses for communication with other replicas; this is indicated in the precondition by the assertion $\mathsf{Addrlist}[i] = (ip_i, p)$. The predicate $\mathit{isList}(\mathsf{Addrlist}, v)$ asserts that the AnerisLang value v is a list of values corresponding to the mathematical list Addrlist. The `init` function requires an *initialization token*, $\mathsf{initToken}(i)$; initialization tokens are produced by the first initialization phase. Distributed systems modeled in Aneris always have a distinguished set A of fixed socket addresses, written using the persistent proposition $\mathsf{Fixed}(A)$. The socket address (ip_i, p) used by replica i should be a fixed address. Moreover, on the ip address ip_i, the port p must be free (*i.e.*, must not have any socket bound to it), as indicated by the ephemeral proposition $\mathsf{FreePorts}(ip_i, \{p\})$. In Aneris, fixed socket addresses, as opposed to dynamic socket addresses, are those that have globally fixed so-called socket protocols (explained in the following). Finally, the precondition of the `init` function requires the knowledge that all socket addresses participating in the distributed database follow the same socket protocol Φ_{DB}. This is written as the persistent Aneris proposition $z \Rrightarrow \Phi_{\mathsf{DB}}$. In Aneris, a socket protocol is simply a predicate over messages which restricts what messages can be sent over and/or may be received through a socket. The protocol Φ_{DB} asserts that any message sent over the socket is the serialization of a write event $w = (k, v, t, o)$ that has been recorded in the abstract global memory, *i.e.*, for which $\mathsf{Snap}(k, \{w\})$ holds.

Phase 1. The purely logical nature of the first phase can be seen in the fact that it is expressed in terms of an update modality:

InitSetup

$$\mathsf{True} \vdash \Rrightarrow_{\mathcal{E}} \boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} * \left(\ast_{0 \le i < \mathit{length}(\mathsf{Addrlist})} \mathsf{initToken}(i)\right) * \left(\ast_{k \in \mathit{Keys}}\ k \rightharpoonup_u \varnothing\right) * \mathsf{initSpec}(\texttt{init})$$

The InitSetup rule simply asserts that resources can be allocated so as to initialize the abstract global memory with an empty set for all keys (*i.e.*, elements of *Keys*), to establish the global in-

variant, and to provide initialization tokens for all declared replicas (*i.e.*, elements of Addrlist). Furthermore, after the InitSetup update is performed, we have that the `init` function meets the specification given in Figure 4.7, formally written as initSpec(`init`).

4.4 Client reasoning about causality

To illustrate how clients can reason about interactions with the distributed database, we give a proof sketch for the example of direct causal dependency from Figure 4.1. The core part of the proof is sketched in Figure 4.8 and assumes local replicas have already been properly initialized, that the specifications hold for the read and write functions, and that writing is an atomic operation. Afterwards, we will show how to initialize local replicas and compose the distributed system. Notice that in this example, while y is being accessed concurrently, x is *not*; reading x on the right happens after the write to y on the left, and hence also after the write to x. Therefore, we use the rule WriteSpec to reason about $\texttt{write}(x, 37)$ while we use the HOCAP-style specification (which we present in Section 4.7) for reasoning about $\texttt{write}(y, 1)$; see the accompanying Coq formalization for the full formal proof. Appendix A provides a proof sketch of the example of indirect causal dependency from Figure 4.1.

We remark that the proof of the example is quite similar in structure to the proof of a similar message-passing example, *but for a weak memory model* with release-aquire and non-atomic accesses [Kai+17a]; see Section 5.7 for a comparison to this related work.

For our presentation here, as a convention, we will only mention persistent assertions (such as invariants and equalities) once and use them freely later.

Both nodes operate on the key y concurrently so ownership of y is put into an invariant $\boxed{\mathsf{Inv}_y}$. The invariant essentially says that y is in one of two states: either 1 has been written to y or not. It will be the responsibility of node i to write 1 and advance the state. When node j reads 1, it will expect to be able to gain ownership of key x. Thus when writing 1 to the key y, node i will have to establish another invariant $\boxed{\mathsf{Inv}_x(w')}$ about x, for some write event w' that has happened *before* the write of 1 to y.

The invariant $\boxed{\mathsf{Inv}_x(w')}$ asserts either ownership of x, and that the maximum event is w' with value 37, or a token $\boxed{\diamond}$. The token is a uniquely ownable piece of ghost state (*i.e.*, $\boxed{\diamond} * \boxed{\diamond} \vdash \mathsf{False}$). Intuitively, when the first node establishes the invariant, the ownership of x is transferred into the invariant. The unique token is given to the second node and when it learns of the existence of $\boxed{\mathsf{Inv}_x(w')}$ it can safely swap out the token for the ownership of x.

Node i initially has knowledge of its local history s and ownership of key x with history h_x such that $h_x \subseteq \lfloor s \rfloor$ where $\lfloor s \rfloor \triangleq \{\lfloor a \rfloor \mid a \in s\}$. Intuitively, this means that all writes to x have been observed at node i and that any future writes to x will be causally dependent on these. Using the specification for the write function, we obtain updated resources for the local history and the key x, *i.e.*, $\mathsf{Seen}(i, s' \uplus \{a_x\})$ and $x \rightharpoonup_u (h_x \uplus \{\lfloor a_x \rfloor\})$, such that $\mathrm{Maximum}(s' \uplus \{a_x\}) = \mathsf{Some}\ a_x$. From Snap extensionality, Seen global extensionality, and the definition of Maximum we conclude $\mathrm{Maximum}(h_x \uplus \{\lfloor a_x \rfloor\}) = \mathsf{Some}\ \lfloor a_x \rfloor$, which suffices for establishing the invariant for x, $\boxed{\mathsf{Inv}_x(\lfloor a_x \rfloor)}$. We then open the invariant for y in order to write 1 to y (this is where we cheat in this sketch and use the assumption that write is atomic); using the invariant we just established for x it is straightforward to reestablish the invariant for y after writing 1 to y as $vc(\lfloor a_x \rfloor) < vc(\lfloor a_y \rfloor)$ follows from $a_x \in s''$ and a_y being the maximum of $s'' \uplus \{a_y\}$, *cf.* global extensionality of Seen.

Invariants

$$\mathsf{Inv}_x(w) \triangleq (\exists h.\, x \rightharpoonup_u h * \mathrm{Maximum}(h) = \mathsf{Some}\ w * w.v = 37) \vee \boxed{\diamond}$$
$$\mathsf{Inv}_y \triangleq \exists h.\, y \rightharpoonup_u h * \forall w \in h.\, w.v = 1 \Rightarrow \exists w'.\, vc(w') < vc(w) * \boxed{\mathsf{Inv}_x(w')}$$

Node i, proof outline

$$\begin{array}{l}
\{\mathsf{Seen}(i, s) * x \rightharpoonup_u h_x * h_x \subseteq \lfloor s \rfloor * \boxed{\mathsf{Inv}_y}\} \\
\quad \mathtt{write}(x, 37) \\
\left\{\begin{array}{l}\exists a_x, s' \supseteq s.\, \mathsf{Seen}(i, s' \uplus \{a_x\}) * x \rightharpoonup_u (h_x \uplus \{\lfloor a_x \rfloor\}) * \\ \mathrm{Maximum}(s' \uplus \{a_x\}) = \mathsf{Some}\ a_x * a_x.v = 37\end{array}\right\} \\
\{\mathsf{Seen}(i, s' \uplus \{a_x\}) * \boxed{\mathsf{Inv}_x(\lfloor a_x \rfloor)}\} \\
\quad \left|\begin{array}{l}\text{open } \mathsf{Inv}_y \\ \end{array}\right. \begin{array}{l}\{\mathsf{Seen}(i, s' \uplus \{a_x\}) * y \rightharpoonup_u h_y * \ldots\} \\ \quad \mathtt{write}(y, 1) \\ \left\{\begin{array}{l}\exists a_y, s'' \supseteq s' \uplus \{a_x\}\,.\, \mathsf{Seen}(i, s'' \uplus \{a_y\}) * y \rightharpoonup_u (h_y \uplus \{\lfloor a_y \rfloor\}) * \\ a_y.v = 1 * \mathrm{Maximum}(s'' \uplus \{a_y\}) = \mathsf{Some}\ a_y\end{array}\right\}\end{array} \\
\{\mathsf{Seen}(i, s'' \uplus \{a_y\}) * \boxed{\mathsf{Inv}_y}\}
\end{array}$$

Node j, proof outline

$$\begin{array}{l}
\{\mathsf{Seen}(j, s) * \boxed{\mathsf{Inv}_y} * \boxed{\diamond}\} \\
\quad \mathtt{wait}(y = 1) \\
\{\exists s' \supseteq s, a_y \in s', w_x.\, \mathsf{Seen}(j, s') * \boxed{\mathsf{Inv}_x(w_x)} * \boxed{\diamond} * vc(w_x) < vc(\lfloor a_y \rfloor)\} \\
\{\mathsf{Seen}(j, s') * x \rightharpoonup_u h_x * \mathrm{Maximum}(h_x) = \mathsf{Some}\ w_x * w_x.v = 37\} \\
\quad \mathtt{read}(x) \\
\{v.\ \exists s'' \supseteq s'.\, \mathsf{Seen}(j, s'') * x \rightharpoonup_u h_x * v = \mathsf{Some}\ 37\}
\end{array}$$

Figure 4.8: Proof sketch, example with direct causal dependency.

Node j initially has knowledge of its local history s and ownership of the token $\boxed{\diamond}$. After repeatedly reading y until we read 1 (call to the `wait` function[5]), the specification for the read function gives us $\mathsf{Snap}(y, \{\lfloor a_y \rfloor\})$ such that $a_y.v = 1$. By Snap inclusion and by opening the invariant for y we get $\boxed{\mathsf{Inv}_x(w_x)}$ such that $vc(w_x) < vc(\lfloor a_y \rfloor)$. We can now open the invariant for x and swap out the token for the ownership of key x and knowledge about its maximum write event. Intuitively, due to causality, as $vc(w_x) < vc(\lfloor a_y \rfloor)$ and a_y has been observed, we are guaranteed to read *something* when reading from x; as $\mathrm{Maximum}(h) = \mathsf{Some}\ w_x$ and $w_x.v = 37$ the value we read has to be 37. Formally, this argument follows from extensionality of Seen and Snap, Snap inclusion, and the definition of $\mathrm{Maximum}$ and $\mathrm{Maximals}$—we refer to the Coq formalization for all the details.

[5]The `wait`$(k = n)$ operation is just a simple loop that repeatedly reads k until the read value is n. In particular, there are no locks/other synchronization code in the `wait` implementation.

The proof sketched in Figure 4.8 verifies the two nodes individually, assuming local replicas have been properly initialized. To set up a distributed system, we spawn two nodes that each initialize a local replica using the `init` function:

$$\texttt{let}\ (\texttt{read}, \texttt{write}) = \texttt{init}(i, ips)\ \texttt{in} \quad \texttt{write}(x, 37);\ \texttt{write}(y, 1) \quad \Big\| \quad \texttt{let}\ (\texttt{read}, \texttt{write}) = \texttt{init}(j, ips)\ \texttt{in} \quad \texttt{wait}(y = 1);\ \texttt{read}(x)$$

where ips is the list of ip addresses of the participating replicas. The proof of the combined system follows from the specification for the `init` function (*cf.* Figure 4.7) and the proof sketch just given. In particular, the specification of `init` ensures that both the history for x and the history for y are empty and hence the precondition for node i holds.[6]

Now we have a complete proof of the client program, under the assumption that the specifications for the distributed database operations hold. By combining this proof with the proof of the implementation (in Section 4.6) we get a closed correctness proof of the whole distributed system in Aneris. This means that we can apply the adequacy theorem of Aneris [Kro+20a, Section 4.2] to conclude that the whole system is safe, *i.e.*, that nodes and threads cannot get stuck (crash) in the operational semantics. (Safety is enough to capture the intuitive desired property for this example: if we included an assert statement after the read of x that would crash if the return value is not 37, then the adequacy theorem would guarantee that this would never happen. We have included such an assert statement in the Coq formalization.)

4.5 Case study: towards session guarantees for client-centric consistency

In the examples in the introduction and Section 4.4, each client program is co-located on the *same* node as the database replica that it reads from and writes to. By contrast, in a *client-server* architecture, a client might interact with *multiple* replicas (servers), and clients and replicas are located on *different* nodes.

The client-server setting is interesting for at least two reasons. First, it is a prevalent mode of use of databases within cloud computing (*e.g.*, MongoDB [CD10] and DynamoDB [Siv12]), where client applications transparently interact with a geo-replicated database running in the cloud. Second, there are consistency models that are tailored to the client-server setting [TS07]. In particular, *session*[7] *guarantees* (*read your writes*, *monotonic reads*, *monotonic writes*, and *writes follow reads*) [Ter+94] describe properties that programmers can use to reason about client-server interactions. For example, the monotonic writes (MW) guarantee ensures that writes happening within a session are propagated to *all* replicas *in program order*.

In this section, we show that our distributed database can be used *in a client-server setting*. Specifically, we build a *session manager library* that exposes the database's `read` and `write` operations to external clients. The library consists of two components: a client *stub* that proxies requests to the server, and a *request handler* that handles the requests server-side. Figure 4.9 illustrates how clients (C1A, C1B, and C2) on different nodes communicate with database replicas (DB1 and DB2) via the session manager stub (SM) and request handler (RH).

We give specifications for the session manager library that rely exclusively on *persistent* resources (as opposed to the database specifications WriteSpec and ReadSpec, which use the

[6] In AnerisLang there is a distinguished system node, which starts all the nodes in the distributed system. Technically, phase 1 of the initialization happens in the proof of the system node, which is then composed with the proof sketch given above for the two nodes.

[7] A *session* is a consecutive sequence of reads and writes issued by a particular client.

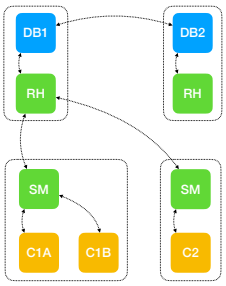


Figure 4.9: Clients using the distributed database via the session manager library.

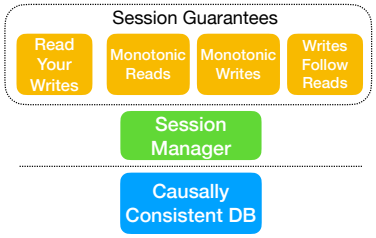


Figure 4.10: Vertical compositionality of specifications.

exclusive ownership of the global memory predicate $k \rightharpoonup_u h$). This is important as multiple clients could be interacting with the same replica concurrently (and from different nodes) in an uncoordinated way.

In spite of being weaker than the underlying replicated database specifications, our session manager specifications are strong enough to prove versions of the four previously-mentioned *session guarantees*. This result is in line with prior work showing that, at the model level, causal consistency implies all four guarantees [BSW04]. In our case, we are able to establish this connection while reasoning about concrete programs.

We illustrate the guarantees with four examples that use the session manager library. In this way, the case study additionally illustrates the modularity of our approach, in particular the *vertical composability* of our specifications, cf. Figure 4.10: we are able to verify each layer using only the *specifications* of the layer below.

Session manager library. As previously mentioned, the session manager exposes the database's operations to the network. The client stub provides its user with three operations: `sconnect`, `sread`, and `swrite`. The client calls `sconnect` ip_i to start interacting with the replica located at ip_i. Reading returns an option with the retrieved value, or None if the key is not populated. All three operations are synchronous at the client-side and every call is blocking while waiting for a server to reply.

Specifications. Figure 4.11 presents a high-level view of the session manager specifications, focusing on how the resources are updated; the full specifications are found in Appendix B. The client, located on the network node at address ip_{client}, reasons about session manager operations using the snapshot predicates $\mathsf{Seen}(i, s)$ and $\mathsf{Snap}(k, h)$ for local and global histories, respectively. For example, to reason about the write $\texttt{swrite}(ip_i, k, v)$, the user provides $\mathsf{Seen}(i, s)$ and $\mathsf{Snap}(k, h)$. Once the write operation completes (is processed by the server and a reply is received), the user gets back *updated* snapshots $\mathsf{Seen}(i, s')$ and $\mathsf{Snap}(k, h')$, such that $s \subseteq s'$, $h \subseteq h'$, and the written value v is stored in an apply (resp. write) event that is part of s' (resp. h'). This captures the idea that if we know that a replica observed *at least* a set s of writes, then after we write v the replica will have observed at least the set $s' = s \uplus \{a\}$, where $a.v = v$. We can then reuse the updated snapshots in the precondition of subsequent operations (we get the initial snapshots from the postcondition of `sconnect`).

$$
\begin{array}{lll}
\{\top\} & \langle ip_{client};\ \texttt{sconnect}(ip_i)\rangle & \left\{\exists s.\mathsf{Seen}(i,s) * \mathop{\Large *}_{k\in\mathsf{Keys}} \exists h_k.\mathsf{Snap}(k,h_k)\right\} \\
\{\mathsf{Seen}(i,s) * \mathsf{Snap}(k,h)\} & \langle ip_{client};\ \texttt{sread}(ip_i,k)\rangle & \{\exists s' \supseteq s, h' \supseteq h.\mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') * \ldots\} \\
\{\mathsf{Seen}(i,s) * \mathsf{Snap}(k,h)\} & \langle ip_{client};\ \texttt{swrite}(ip_i,k,v)\rangle & \{\exists s' \supset s, h' \supset h.\mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') * \ldots\}
\end{array}
$$

Figure 4.11: Simplified specifications of session manager operations. Full specifications are found in Appendix B.

Table 4.2: The four session guarantees.

Guarantee	**Program**	**Description**
read your writes	`swrite(ip,k,v); sread(ip,k)`	Reads observe writes not older than preceding writes.
monotonic reads	`sread(ip,k); sread(ip,k)`	Reads observe writes not older than writes observed by preceding reads.
monotonic writes	`swrite(ip,k1,v1); swrite(ip,k2,v2)`	Writes propagate to all replicas in program order.
writes follow reads	`sread(ip,k1); swrite(ip,k2,v)`	Writes and writes observed through reads propagate to all replicas in program order.

SM-MONOTONIC-WRITES

$$
\begin{array}{l}
\{ip_i \Rightarrow \Phi_i\} \\
\quad \langle ip_{client};\ \texttt{sconnect}(ip_i); \texttt{swrite}(ip_i,k_1,v_1); \texttt{swrite}(ip_i,k_2,v_2)\rangle \\
\left\{ () .\ \begin{array}{l}
\exists s_1, a_1, a_2.\ a_1.k = k_1 * a_1.v = v_1 * a_2.k = k_2 * a_2.v = v_2 \\
\quad * \mathsf{Seen}(i,s_1) * a_1, a_2 \in s_1 * a_1.t < a_2.t \\
\quad * (\forall a, s', j.\ \mathsf{Seen}(j,s') * a \in s' * a_2.t \leq a.t \\
\qquad \Rrightarrow \mathop{\Large *}_{\top} \exists a_1', a_2'.\ \lfloor a_1' \rfloor = \lfloor a_1 \rfloor * \lfloor a_2' \rfloor = \lfloor a_2 \rfloor * a_1', a_2' \in s' * a_1'.t < a_2'.t)
\end{array} \right\}
\end{array}
$$

Figure 4.12: Specification for monotonic writes example.

Session guarantees. Table 4.2 shows the four session guarantees and corresponding client programs we verify. Each guarantee describes what a client can infer from observing the effect of a pair of (read or write) operations within the same session. We only describe our *monotonic writes* (MW) example in detail and refer to Appendix B or the Coq formalization for the remaining guarantees.

Figure 4.12 shows a simplified specification for the MW example (we omit some network-related predicates from the precondition), which involves two consecutive writes to a replica located at address ip_i. The precondition for the Hoare triple is just knowledge that address ip_i behaves according to socket protocol Φ_i. The definition of this socket protocol is a key part of verifying the session manager library, since it allows us to tie physical client requests to their logical counterparts, but is relegated to the Coq formalization for space reasons. Let us now unpack the postcondition. We obtain a snapshot $\mathsf{Seen}(i, s_1)$ that represents the replica state after the two writes. Both writes are recorded in the local history s through apply events a_1 and a_2 that respect program order ($a_1.t < a_2.t$). We ensure that the writes are propagated in the same order to *all* replicas through the following implication. Suppose we observe the snapshot $\mathsf{Seen}(j, s')$ of a replica j. Now suppose that *enough time has passed* so that there

exists an event $a \in s'$ such that $a_2.t \leq a.t$; that is, the event observed at a_2 is *as recent* as the second write at a_1. Using the causality property from Figure 4.3 we show that the two writes at node i have been propagated to node j as the apply events a'_1 and a'_2 that respect program order ($a'_1.t < a'_2.t$). This way we express the MW guarantee.

We remark that our session manager library delegates the replica selection to clients. This is a simplification w.r.t practical implementations, where replica selection is done transparently by the session manager. In this simplified setting, our notion of causality is strong enough to provide session guarantees for the clients. Extending the case study to the general setting with transparent replica selection will require exposing a notion of time (*e.g.*, vector clocks) to the session manager and clients.

4.6 Verification of the implementation

So far we have described how to use our specifications for client reasoning. In this section we show that our implementation from Section 4.1 does satisfy our specifications. Conceptually, the proof of the implementation can be split into the following three stages:

1. We define a concrete notion of validity tying together all layers of the model (abstraction of the replicas' physical states, local histories, and the abstract global memory), and show that validity is preserved by the write and apply operations.

2. We define the meaning of the abstract predicates ($\mathsf{Seen}(i, s)$, $\mathsf{Snap}(k, h)$, $k \rightharpoonup_u h$, $\boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}}$) using Iris ghost state.

3. We define the lock invariant that governs replica-local shared data (the key-value dictionary, vector clock, in- and out-queues) and prove the correctness of the implementation of each operation.

In this section we discuss the key aspects of these three stages.

4.6.1 Local and global validity

Obviously, the local history s_i must be consistent with the physical state of the replica i for which it tracks the updates. We model the physical state for replica i as a *local state* (δ_i, t_i, s_i) defined as

$$(\delta_i, t_i, s_i) \in \mathit{LocalState} \triangleq (\mathit{Keys} \xrightarrow{\text{fin}} \overset{?}{\mathsf{Value}}) \times \mathsf{VectorClock} \times \mathit{LocalHistory}.$$

Here δ_i is a model of the local key-value store `db` used by the implementation at replica i, and t_i is the vector clock stored in the reference `vc` in the implementation.

We express consistency of the physical state as a validity property of the corresponding local state. To this end, we first observe that a local history at replica i can be partitioned into *sections* according to the origin of the apply events. Concretely, given a local history s_i, we define its j^{th} *section*, denoted by $s_{i,j}$, to be the subset of s_i events whose origin is j, *i.e.*, $s_{i,j} \triangleq \{a \in s_i \mid \mathit{origin}(a) = j\}$. Intuitively, for $j \in \{0..n-1\} \setminus \{i\}$, each section $s_{i,j}$

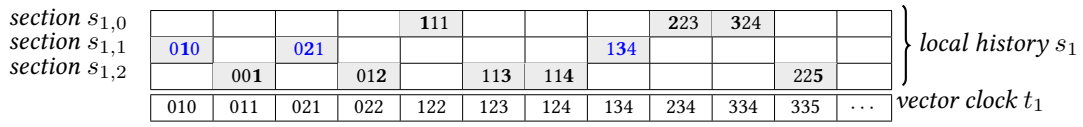


Figure 4.13: A valid history s_1 partitioned into sections $s_{1,0}$, $s_{1,1}$, $s_{1,2}$, and a vector clock t_1 evolving through time. Each cell contains the time of apply events. For example, the apply events at section $s_{1,1}$ (in blue color) are those events that come from the writes of replica 1. For each $s_{1,j}$, the j^{th} component of the vector clock is depicted in bold. The numbers in bold reflect condition (2) in Definition 4.6.1.

describes the external updates from replica j applied locally on i, while the "diagonal" section $s_{i,i}$ describes the write operations executed on the replica i itself.[8]

Definition 4.6.1 (Valid Local Histories)**.** Local history s_i is *valid* if [9]

1. $\forall a_1, a_2 \in s_i.vc(a_1) = vc(a_2) \implies a_1 = a_2$
2. $\forall k.1 \leq k \leq |s_{i,j}| \implies \exists a \in s_{i,j}.vc(a)[j] = k$
3. $\forall a \in s_{i,i}.\forall j' \in \{0..n-1\} \backslash \{i\}\,.vc(a)[j'] = Sup\left\{vc(b)[j'] \,|\, b \in s_{i,j'} \wedge b.m < a.m\right\}$
4. $\forall a \in s_{i,j}.\forall j' \in \{0..n-1\} \backslash \{j\}\,.j \neq i \implies$
$$vc(a)[j'] \leq Sup\left\{vc(b)[j'] \,|\, b \in s_{i,j'} \wedge b.m < a.m\right\}$$

The conditions above capture the fact that all apply events in the local history must have valid times. For instance, condition (2) reflects the fact that the set of apply events of a given section $s_{i,j}$ is downwards closed and complete w.r.t. the projection j of the vector clocks they carry. The most subtle are conditions (3) and (4), which ensure that for any event a that we have in our local history, we have observed all the events it depends on before observing a. To see this, note that $vc(a)[j']$ corresponds to the number of write events originating from replica j' that a depends on, and $Sup\left\{vc(b)[j'] \,|\, b \in s_{i,j'} \wedge b.m < a.m\right\}$ corresponds to the number of events we have observed from replica j' before a. Figure 4.13 illustrates the notion of validity with a concrete example.

Valid histories satisfy the following theorem which expresses the causality relation of the origin and the time projection of apply events:

Theorem 4.6.2. *If local history s_i is valid, then the following properties hold:*

- $s_{i,j} = \varnothing \iff \forall a \in s_i.vc(a)[j] = 0$ *(Empty sect.)*
- $\forall a \in s_i.\forall j' \in \{0..n-1\}\,.\forall p \in \mathbb{N}^+.p \leq vc(a)[j'] \implies$
$\exists a' \in s_{i,j'}.vc(a')[j'] = p$ *(Local causality)*
- $\forall a_1, a_2 \in s_{i,j}.vc(a_1)[j] = vc(a_2)[j] \implies a_1 = a_2$ *(Component extensionality)*

[8] In the following, we assume that all local histories and sections are well-formed: when we write $s_{i,j}$, we assume $i, j \in \{0..n-1\}$, and for any $a \in s_i$, the vector clock $vc(a)$ is of length n, the key $a.k$ belongs to the fixed set of keys, and the sequence identifier $a.m$ is less than or equal to the size of s_i.

[9] Here Sup is the supremum function. Note that $Sup(\varnothing) = 0$.

- $\forall k.1 \leq k \leq |s_{i,j}| \iff \exists a \in s_{i,j}.vc(a)[j] = k$ *(Strong completeness)*

Having defined the validity of local histories, we can now state the validity for local state (δ_i, t_i, s_i) as a predicate $Valid_{\mathbf{L}}(\delta_i, t_i, s_i)$ defined below.

Definition 4.6.3 (Valid local states)**.** $Valid_{\mathbf{L}}(\delta_i, t_i, s_i)$ holds if the following conditions hold:

1. $\forall k \in \mathsf{dom}(\delta_i), \forall v.\ \delta_i(k) = \mathsf{Some}\ v \implies \exists a \in s_i.a = \mathsf{Observe}(s_{i|k}) \wedge a \in \mathrm{Maximals}(s_{i|k})$
2. $\forall k.\ \delta_i(k) = \mathsf{None} \implies s_{i|k} = \varnothing$
3. $\forall j \in \{0 \ldots n-1\}\ .t_i[j] = Sup\,\{vc(a)[j] \mid a \in s_{i,j}\}$
4. s_i is a valid local history

Here conditions (1) and (2) express the consistency of the local history s_i with the local store δ_i, capturing the correctness argument of the `read` specification. Condition (3) states that in the local time of replica i, each projection $t_i[j]$ is equal to the projection $vc(a)[j]$, where a is the most recent event from section $s_{i,j}$, if such a exists, otherwise $vc(a)[j] = 0$. In particular, we have $\forall a \in s_i.\ vc(a) \leq t_i$.

Using the notion of validity for local histories, we finally define validity for global states.

Definition 4.6.4 (Valid global states)**.** $Valid_{\mathbf{G}}(\{|\, M\,; s_1, \ldots, s_n \,|\})$ holds if

1. $(\forall k \in \mathsf{dom}(M), w \in M(k).\ \exists a \in s_{origin(w),origin(w)}.w = \lfloor a \rfloor) \wedge$
   $(\forall a \in \bigcup_{i=1}^{n} s_i.\ \exists w \in M(a.k).\ w = \lfloor a \rfloor)$
2. All local histories $s_1, \ldots, s_n$ are valid.

Condition (1) defines a provenance relation between apply and write events in both directions. All the properties listed for valid global states in Figure 4.3 follow from this definition.

Crucially, the correctness argument for the `write` and `apply` operations relies on the following validity preservation theorem (which we here state only informally; see the Coq development for the formal statement):

Theorem 4.6.5 (Validity preservation)**.** *Consider a valid global state $\{|\, M\,; s_1, \ldots, s_n \,|\}$ and a replica i whose local state (δ_i, t_i, s_i) is valid. The effects of both write and apply operations intuitively described below* ***preserve both local and global validity****.*

- *The effect of write event: adding a new apply event a with time $\mathsf{incr}_i(t_i)$ to s_i and a new write event $\lfloor a \rfloor$ to $M(a.k)$, where $\mathsf{incr}_j(t_i)$ is t_i with j^{th} incremented.*
- *The effect of apply event: adding a new apply event a to s_i such that a has passed the dynamic check and $\lfloor a \rfloor$ is already in $M(a.k)$.*

Table 4.3: Predicates tracking abstract state of the distributed database defined in terms of resources.

Predicate	**Intuitive definition**
$\mathsf{Snap}(k, h)$	ownership of $\circ_{\mathbb{S}}\ \{(k, h)\}$
$k \rightharpoonup_u h$	ownership of $\circ_{\mathbb{M}}\ \{(k, h)\}$ and $\circ_{\mathbb{S}}\ \{(k, h)\}$
$\mathsf{GM}(M)$	ownership of $\bullet_{\mathbb{M}}\ M$ and $\bullet_{\mathbb{S}}\ M$

Predicate	**Intuitive definition**
$\mathsf{Seen}(i, s)$	ownership of $\circ_{\mathbb{C}_i}\ s$ and $\circ_{\mathbb{L}_i}\ s$
$\mathsf{LH}_{\mathbf{G}}(i, s)$	ownership of $\circ_{\mathbb{C}_i}\ s$ and $\bullet_{\mathbb{L}_i}\ s$
$\mathsf{LH}_{\mathbf{L}}(i, s)$	ownership of $\bullet_{\mathbb{C}_i}\ s$ and $\circ_{\mathbb{L}_i}\ s$

4.6.2 Ghost state

The theory of resource algebras in Iris [Jun+16] can be used to define so-called *ghost theories*, *i.e.*, to define resources and Iris propositions that assert ownership over resources. The exact combination of resource algebras and how they are used to define Iris propositions determines the properties of the ghost theories, *e.g.*, which propositions are persistent/ephemeral, the way resources can be updated, *e.g.*, allowing monotonic growth, allowing (de)allocation, *etc.* We refer the reader to Jung et al. [Jun+18b] and discussions therein for details of how resource algebras work.

One of the most important and versatile resource algebras is the so-called *authoritative* resource algebra, $\mathit{Auth}(\mathbb{A})$, where $\mathbb{A}$ is itself a resource algebra. The elements of the authoritative resource algebra are resources that are divided into two parts: the full part, of the form $\bullet_{\mathbb{A}}\ m$, and the fragment part, of the form $\circ_{\mathbb{A}}\ m$, for a resource $m \in \mathbb{A}$. The idea is that $\bullet_{\mathbb{A}}\ m$ is the central authoritative view of the ghost state, while $\circ_{\mathbb{A}}\ m'$ represents fragments of $\bullet_{\mathbb{A}}\ m$; we write this as $m' \preceq_{\mathbb{A}} m$, where $\preceq_{\mathbb{A}}$ is the resource inclusion relation for resources in $\mathbb{A}$. Hence, owning resource $\bullet_{\mathbb{A}}\ m$ is ephemeral, while $\circ_{\mathbb{A}}\ m$ can possibly be split up into multiple parts, depending on how elements of $\mathbb{A}$ can be split. Moreover, the ownership of $\circ_{\mathbb{A}}\ m$ may be ephemeral or persistent depending on whether ownership of elements of $\mathbb{A}$ is ephemeral or persistent.

Abstract global memory. We use two instance of the authoritative resource algebra, namely $\mathit{Auth}(\mathbb{S})$ and $\mathit{Auth}(\mathbb{M})$, for modeling the abstract global memory. The resource algebra $\mathbb{S}$ is the resource algebra of finite maps from keys to finite sets of write events. It is defined so that the inclusion relation $M' \preceq_{\mathbb{S}} M$ holds if, and only if, $\forall x.\ x \in \mathsf{dom}(M') \implies M'(x) \subseteq M(x)$. That is, in $\mathit{Auth}(\mathbb{S})$ fragments track *lower bounds* of the sets of write events tracked in the authoritative part. Hence, ownership of fragments in $\mathit{Auth}(\mathbb{S})$ is persistent. The resource algebra $\mathbb{M}$ is the resource algebra of finite maps from keys to *exclusive* finite sets of write events. It is defined so that the inclusion relation $M' \preceq_{\mathbb{M}} M$ holds if, and only if, $\forall x.\ x \in \mathsf{dom}(M') \implies M'(x) = M(x)$. That is, in $\mathit{Auth}(\mathbb{M})$ fragments track *precisely* the sets of write events tracked in the authoritative part. Ownership of fragments in $\mathit{Auth}(\mathbb{M})$ is thus ephemeral. The authoritative parts of $\mathit{Auth}(\mathbb{S})$ and $\mathit{Auth}(\mathbb{M})$ are used to define $\mathsf{GM}(M)$ which is used in the global invariant to track the abstract global memory. The fragments are used to define $\mathsf{Snap}(k, h)$ and $k \rightharpoonup_u h$. Table 4.3 gives the intuitive definition of these predicates.

$$\mathsf{GM}(M) * \mathsf{Snap}(k, h) \vdash \exists h'. M(k) = h' \land h \subseteq h' \qquad \text{(Snapshot inclusion)}$$
$$\mathsf{LH}_{\mathbf{G}}(i, s') * \mathsf{Seen}(i, s) \vdash s \text{ is a causally-closed subset of } s' \qquad \text{(Seen inclusion)}$$
$$\mathsf{LH}_{\mathbf{L}}(i, s) * \mathsf{LH}_{\mathbf{G}}(i, s') \vdash s = s' \qquad \text{(Local hist. agreement)}$$
$$\mathsf{GM}(M) * k \rightharpoonup_u h * h \subseteq h' \vdash \Rrightarrow_{\mathcal{E}} \mathsf{GM}(M[k := h']) * k \rightharpoonup_u h' \qquad \text{(Global mem. update)}$$
$$\mathsf{LH}_{\mathbf{L}}(i, s) * \mathsf{LH}_{\mathbf{G}}(i, s) * a \in \mathrm{Maximals}(s \cup \{a\}) \vdash \Rrightarrow_{\mathcal{E}} \mathsf{LH}_{\mathbf{L}}(i, s \cup \{a\}) * \mathsf{LH}_{\mathbf{G}}(i, s \cup \{a\}) \qquad \text{(Local hist. update)}$$

Figure 4.14: Selected laws of the concrete ghost state.

Note that fragments of both $\mathit{Auth}(\mathbb{S})$ and $\mathit{Auth}(\mathbb{M})$ are used in the definition of $k \rightharpoonup_u h$. This is why we can prove the rule (Take Snap) in Figure 4.6. An excerpt of the laws governing the use of predicates tracking ghost resources are presented in Figure 4.14.

Local history. We track local histories using two different kinds of resource algebras, $\mathit{Auth}(\mathbb{C})$ and $\mathit{Auth}(\mathbb{L})$, one instance of each per replica. When necessary, we write $\mathbb{C}_i$ and $\mathbb{L}_i$ instead of just $\mathbb{C}$ and $\mathbb{L}$ to distinguish instances used for replica i. Both $\mathbb{C}$ and $\mathbb{L}$ are similar in that they track sets of apply events. Moreover, the ownership of the fragments in both $\mathit{Auth}(\mathbb{C})$ and $\mathit{Auth}(\mathbb{L})$ are persistent. The main difference between $\mathbb{C}$ and $\mathbb{L}$ is in their inclusion relation:

$$s' \preceq_{\mathbb{L}} s \text{ if and only if } s' \subseteq s$$
$$s' \preceq_{\mathbb{C}} s \text{ if and only if } s' \text{ is a } \textit{causally-closed} \text{ subset of } s$$

We need to track local history of a replica both in the global invariant and in the local lock invariant of the replica (see below). For this reason we define propositions $\mathsf{LH}_{\mathbf{G}}(i, s)$, and $\mathsf{LH}_{\mathbf{L}}(i, s)$, respectively. Table 4.3 gives the intuitive definition of these predicates as well as that of the $\mathsf{Seen}(i, s)$. Note that $\mathsf{LH}_{\mathbf{G}}(i, s)$, and $\mathsf{LH}_{\mathbf{L}}(i, s)$ each have the full part of one of the two resource algebras and the fragment of the other. This fact, together with the inclusion relations above, is why we can prove the rule (Local hist. agreement) in Figure 4.14.

Global invariant. The global invariant defined below simply states that there should exist an abstract global memory M and local histories $s_1, \ldots, s_n$ that we track using Iris resources such that the global state $\{|\, M\,;\, s_1, \ldots, s_n \,|\}$ is valid.

$$\mathsf{GlobalInv} \triangleq \exists M, s_1, \ldots, s_n.\ \mathit{Valid}_{\mathbf{G}}(\{|\, M\,;\, s_1, \ldots, s_n \,|\}) * \mathsf{GM}(M) * \mathop{\Large *}_{i=1}^{n} \mathsf{LH}_{\mathbf{G}}(i, s_i)$$

4.6.3 *Proof of the implementation*

To verify the operations of the implementation, a crucial aspect is the choice of lock invariant. We use an Aneris lock module, which itself is implemented as a spin lock, and whose specification is very similar to a standard Iris lock, see, *e.g.*, Birkedal and Bizjak [BB17, Section 7.6]. The lock module uses an abstract predicate $\mathrm{isLock}(ip, \ell, P)$ to assert that the memory location ℓ is a lock on node ip protecting resources described by P.

The lock invariant for our distributed database is defined by the predicate

$$\Psi(i, \mathtt{db}, \mathtt{vc}, \mathtt{iq}, \mathtt{oq}) \triangleq$$
$$\exists v_d, v_t, v_{iq}, v_{oq}.\ \exists \delta_i, t_i, s_i.\ \exists q_{in}, q_{out}.\ \exists ip_i, p.$$
$$\mathsf{Addrlist}[i] = (ip_i, p) * \mathtt{db} \mapsto_{ip_i} v_d * \mathtt{vc} \mapsto_{ip_i} v_t * \mathtt{iq} \mapsto_{ip_i} v_{iq} * \mathtt{oq} \mapsto_{ip_i} v_{oq} *$$
$$isDictionary(v_d, \delta_i) * isVectorClock(v_t, t_i) * InQueue(v_{iq}, q_{in}) *$$
$$OutQueue(v_{oq}, q_{out}) * \mathsf{LH}_{\mathbf{L}}(i, s) * Valid_{\mathbf{L}}(\delta_i, t_i, s_i)$$

The predicate asserts that the dictionary `db`, the vector clock `vc`, and the queues `iq`, and `oq` are all allocated in the local heap of the replica i with values v_d, v_t, v_{iq}, and v_{oq}, respectively. It also enforces that the representation predicates $isDictionary$, $isVectorClock$, $InQueue$, $OutQueue$ tie together those program values with their logical counterparts δ_i, t_i, q_{in}, and q_{out}, respectively. Moreover, the predicate Ψ asserts that the local history s tracked for replica i (*cf.* $\mathsf{LH}_{\mathbf{L}}(i, s)$) together with the dictionary δ_i and vector clock t_i forms a valid local state, *i.e.*, $Valid_{\mathbf{L}}(\delta_i, t_i, s_i)$ holds. The $InQueue(v_{iq}, q_{in})$ and $OutQueue(v_{oq}, q_{out})$ predicates enforce that the contents of both queues are write events a for which we have $\mathsf{Snap}(a.k, \{a\})$.

With the lock invariant defined as above, we verify all operations of the database. The `init` function can use its precondition to establish the lock invariant—in fact, $\mathsf{initToken}(i)$ is defined as $\mathsf{LH}_{\mathbf{L}}(i, \varnothing)$. For the `write` operation we essentially need to prove that, given the precondition, it preserves the lock invariant and the global invariant, and that we can establish the post condition afterwards. The bulk of the proof, apart from reasoning about Iris resources, involves showing preservation of validity which follows directly from Theorem 4.6.5. For the `read` operation we only need to access the lock invariant and the global invariant in order to establish the postcondition—the lock invariant and the global invariant are trivially preserved as we do not change the state of the database. Recall that the postcondition of the `read` function almost follows from the definition of local state validity (Definition 4.6.3).

For the `apply` operation we essentially need to prove that it preserves the lock invariant and the global invariant. This follows from Theorem 4.6.5.

For the send thread we only need to show that the write events we send over the network adhere to the socket protocol Φ_{DB}. This immediately follows from $OutQueue(v_{oq}, q_{out})$ in the lock invariant. For the receive thread we need to show that the write events we receive over the network can be enqueued in q_{in}, *i.e.*, these are write events w for which we have $\mathsf{Snap}(w.k, \{w\})$. This immediately follows from the socket protocol Φ_{DB}.

4.7 HOCAP-style specification for the write operation

In this section we present our HOCAP-style specification for the write operation, *cf.* the earlier discussion in the introduction and in Section 4.3. Recall that the need for this more general specification comes from the fact that the natural specification of `write` involves ephemeral resources (the $k \rightharpoonup_u h$ resource used in the WriteSpec), which the clients should be able to govern by an Iris invariant in case the clients concurrently access keys. For a client to use invariants, the write operation must be atomic since otherwise the client cannot open and close invariants around the write operation. But since the write operation is not atomic, we need to use another approach—and thus we use the HOCAP-style specification approach (see [BB17] for an introduction to this style of specification).

In the HOCAP-style approach, there are in fact *two* views of the abstract state of the global memory of the key-value store: the client view $k \rightharpoonup_u h$, which we have seen before, and the

module view $k \rightharpoonup_s h$; both of the abstract predicates are provided to the client as part of the modular specification interface of the replicated database. These two views always agree on the abstract state of the global memory, *i.e.*, $k \rightharpoonup_s h * k \rightharpoonup_u h' \vdash h = h'$. One of the key ideas is that neither the client nor the module can update its own view of the abstract global memory on their own. Instead, the module delegates updating the abstract global memory to the client (so that the client can control what happens in case the client needs to coordinate concurrent accesses using invariants). Thus the HOCAP-style write specification is parametrized by view shifts (update modalities) which the client has to prove and which allow the client to update the module's view of the abstract global memory $k \rightharpoonup_s h$ by combining it with its own view $k \rightharpoonup_u h$ of the abstract global memory. The latter is done using the following law

$$\forall w, \mathcal{E}.\ k \rightharpoonup_s h * k \rightharpoonup_u h \vdash \Rightarrow_{\mathcal{E}} k \rightharpoonup_s h \uplus \{w\} * k \rightharpoonup_u h \uplus \{w\} \qquad \text{(System User Update)}$$

which we provide to the client as part of the modular interface describing the laws governing database resources. This law is in fact the single law that is missing in Figure 4.6 from Section 4.3.2.[10]

With this defined, the HOCAP-style specification for `write` is formally stated as follows:

$$\begin{array}{l}\forall \mathcal{E}, k, v, s, P, Q.\ \mathcal{N}_{\mathsf{GI}} \subseteq \mathcal{E} \Rightarrow \\ \square \left(\begin{array}{l} \forall s', a.\ (s \subseteq s' * a \notin s' * a.k = k * a.v = v * P) \\ \qquad {}_{\top}\Rightarrow\!\!\!*_{\mathcal{E}}\ \forall h.\ \left(\begin{array}{l} \lfloor a \rfloor \in \mathrm{Maximals}(h \uplus \{\lfloor a \rfloor\}) * k \rightharpoonup_s h * \\ \mathsf{Seen}(i, s' \uplus \{a\}) * \mathrm{Maximum}(s' \uplus \{a\}) = a \end{array} \right) \\ \qquad \Rightarrow\!\!\!*_{\mathcal{E} \setminus \mathcal{N}_{\mathsf{GI}}}\ k \rightharpoonup_s h \uplus \{\lfloor a \rfloor\} * {}_{\mathcal{E}}\Rightarrow_{\top} Q\ a\ h\ s' \end{array} \right) \mathrel{-\!\!*} \\ \{P * \mathsf{Seen}(i, s)\}\ \langle ip_i; \mathtt{write}(k, v) \rangle\ \{v.v = () * \exists h, s', a.\ s \subseteq s' * Q\ a\ h\ s'\}\end{array}$$

where P : *iProp* and Q : *ApplyEvent* $\to \wp^{\mathsf{fin}}(\mathit{WriteEvent}) \to \mathit{LocalHistory} \to \mathit{iProp}$.

Consider the view shifts before the Hoare triple for `write`. The client has to show, for the client's choice of predicates P and Q, that, given P and an apply event a corresponding to the write, if the client opens up invariants from the mask $X = \top \setminus \mathcal{E}$ and then additionally gets access to the module's view of the abstract state $k \rightharpoonup_s h$, then the client must be able to (1) update the abstract state to $k \rightharpoonup_s h \uplus \{\lfloor a \rfloor\}$, and, in doing so, they may open (and close) all invariants in $\mathcal{E}$, *except* for the global invariant $\mathcal{N}_{\mathsf{GI}}$ (which makes sense since that invariant is used internally by the implementation), and (2) close the invariants in X and then establish Q.

We remark that this use of view shifts is slightly more advanced than standard HOCAP-style specifications because here the client is allowed to open some invariants (those in X) before updating the abstract state and then close the invariants in X again to establish the postcondition Q.

This HOCAP-style specification of the write operations is the actual specification and the one we have proved and use in our Coq formalization to verify client programs in a modular way.

As we mentioned earlier, the simplified specification for the write operation (WriteSpec) is derivable from the HOCAP-style specification above. To prove this, take $\mathcal{E}$ to be $\top$, let

[10] We define the meaning of these abstract predicates using appropriate Iris resource algebras and prove this law when verifying the implementation of the database.

$P \triangleq k \rightharpoonup_u h$ (provided in the precondition of WriteSpec) and let

$$Q\ a\ h'\ s' \triangleq h = h' * a.k = k * a.v = v * \lfloor a \rfloor \in \mathrm{Maximals}(h' \uplus \{\lfloor a \rfloor\}) * \\ k \rightharpoonup_u h' \uplus \{\lfloor a \rfloor\} * \mathsf{Seen}(i, s' \uplus \{a\}) * a = \mathrm{Maximum}(s' \uplus \{a\}).$$

We can prove the equality $h = h'$ in Q because we have $k \rightharpoonup_u h$ (as P) and we know that the client and the module always agree on the view of the abstract global memory. Additionally, we use the rule System User Update with $w = \lfloor a \rfloor$ and $\mathcal{E} = \top \setminus \mathcal{N}_{\mathsf{GI}}$ to prove the update modality $\Rrightarrow\!\!\!\!\!\!*_{\mathcal{E} \setminus \mathcal{N}_{\mathsf{GI}}}$ in HOCAP-style spec above and obtain $k \rightharpoonup_s h' \uplus \{\lfloor a \rfloor\}$ and $k \rightharpoonup_u h' \uplus \{\lfloor a \rfloor\}$ (needed to prove Q).

4.8 Related work

Lesani et al. [LBC16] present an abstract causal operational semantics for replicated key-value store implementations and their client programs. Through a refinement argument, two implementations in Coq's functional language, Gallina, are shown to realize this semantics. As a result of their approach, client programs can automatically be verified by model checking. In comparison, our work allows both the distributed database and clients to be implemented in a realistic ML-like language and verified using a separation logic in a completely modular way. This mean we can build libraries and provide abstractions on top of clients, as exemplified by our session manager library, and compose the database with other components to build and verify larger distributed systems. It is unclear how the approach of Lesani et al. [LBC16] would scale to a larger setting where a key-value store is just one component of a distributed system.

Several approaches exist for reasoning about weaker consistency models of distributed databases and their clients, including declarative approaches [ALO00; Aha+95; Bur+12; CBG15; CGY17; Coo+08; Got+16] as well as operational approaches [Cro+17; Kak+18; SZ18; Xio+19]. Common for all these works is that they reason about high-level models of distributed replicated databases and protocols with tools tailored for reasoning about databases, specific combinations of consistency models, and specific consistency guarantees. In constrast, our approach is aimed at the verification of concrete implementations and allows databases and clients to be composed with other components to build and verify larger distributed systems *while also* allowing us to reason about the weak consistency offered by the implementation.

Formal specification and verification of distributed systems and algorithms has been carried out by means of model checking [Hol97; Kil+07; Lam92; Pnu77] and, more recently, using a variety of program logics: Disel [SWT18] is a Hoare Type Theory for distributed program verification in Coq with ideas from separation logic. IronFleet [Haw+15] allows for building provably correct distributed systems by combining TLA-style state-machine refinement with Hoare-logic verification in a layered approach, all embedded in Dafny [Lei10]. Verdi [Wil+15] is a framework for writing and verifying implementations of distributed algorithms in Coq. Here we build on the Aneris logic, which supports horizontal and vertical composability of distributed systems implemented using sockets, node-local state and concurrency, and higher-order functions. Moreover, we rely on the Coq formalization of Aneris to mechanize all of our program correctness proofs.

In recent years, there has been a lot of work on formally specifying memory models of modern processors [Alg+10; Arm+19; CS15; LB20; Mad+12; Sev+11], and there has also been work on program logics for formal reasoning on top of such memory models [AM16a; BAP15; DV16; TVD14; VN13]. In particular, Kaiser et al. [Kai+17a] provide a framework for proving

programs in a fragment of C11 containing release-acquire (RA) and non-atomic (NA) accesses. Their specifications for read and write rely on a global view of the weak memory and a local view of each thread. While our specifications of read and write are at a very different level (for an implementation of a distributed database rather than for an operational semantics model of a processor), our specifications follow a similar pattern, as we track both the abstract global memory and the local history of each replica in our specifications. However, in Kaiser et al. [Kai+17a] each update is explicitly tracked only globally, and the local thread view only associates each location with the time of its latest update. We further note that their consistency model corresponds to RA consistency of the weak memory, while our model describes causal consistency for a distributed system implementation. Bouajjani et al. [Bou+17] show that causal consistency is equivalent to a WRA (weak release acquire) model which is strictly weaker than RA consistency. According to Lahav [Lah19], understanding how concurrent separation logics for the RA model can be weakened to the causal consistency is an interesting research question, and we hope that our specifications may serve as inspiration for future investigations in that direction.

4.9 Conclusion

We have presented a modular formal specification of a causally-consistent distributed database in Aneris, a higher-order distributed separation logic, and proved that a concrete implementation of the distributed algorithm due to Ahamad et al. [Aha+95] meets our specification. We have demonstrated that our specifications are useful, by proving the correctness of small, but tricky, synthetic examples involving causal dependency and by verifying a sesssion-manager library implemented on top of the distributed database. For the session-manager we have, moreover, verified formal program logic versions of the session guarantees known from the distributed systems literature.

We have relied on Aneris's facilities for modular specification and verification, in particular node-local reasoning *qua* socket protocols, to achieve a highly modular development, where each component is verified in isolation, relying only on the specifications (not the implementations) of other components. In particular, the distributed database is specified in the same style as other libaries and data structures are specified in distributed/concurrent separation logics, and thus it can be freely combined with other client programs and libraries (as evidenced by the session-manager library case study).

Future work includes implementing and verifying a strengthened session-manager library with transparent replica selection. It would also be interesting to verify other implementations of causally-consistent databases.

5 Trillium

Abstract

Formal verification systems such as TLA$^+$ have been widely used to design, model, and verify complex concurrent and distributed systems. In many of these tool suites, systems are modeled as state transition systems, and both safety and liveness properties can usually be checked. This enables users to reason about abstract system specifications and uncover design flaws, but it offers no guarantees about the *implementations* of systems nor about the relation between an implementation and its abstract specification.

In this work, we show how to connect concrete implementations of concurrent and distributed systems to abstract system models. We develop Trillium, a separation logic framework for establishing *history-sensitive* refinement relations between programs and models. We use our logic to prove correctness of implementations of two-phase commit and single-decree Paxos by showing that they refine their abstract TLA$^+$ specifications. We further use our notion of refinement to transfer fairness assumptions on program executions to model traces and then transfer liveness properties of fair model traces back to program executions. This enables us to prove liveness properties such as strong eventual consistency of a concrete implementation of a Conflict-Free Replicated Data Type and fair termination of a concurrent program.

Formal verification systems such as SPIN [Hol97] and TLA$^+$ [Lam92] have been widely used to design, model, and verify complex concurrent and distributed systems with successful industrial applications in organizations like NASA [HLP01], Intel [Bee08], Amazon [New14; New+15], and Microsoft [Lar17]. In many of these tool suites, systems are modeled as state transition systems and the tools can usually check both safety and liveness properties. This enables users to reason about abstract system specifications and uncover design flaws, but it offers no guarantees about the *implementations* of systems nor about the relation between an implementation and its abstract specification.

Concurrent separation logic [OHe07] and its modern variants, such as Iris [Jun+16; Jun+18b; Jun+15; Kre+17], provide powerful and modular reasoning principles for concurrent programs, thanks to mechanisms such as *ghost state* and *invariants*. They have been used to verify a wide range of implementations of sophisticated fine-grained concurrent data structures [Car+22; Cha+21; Kri+20; MJ21; VPJ20; VB21; VFB22] and distributed systems [Gon+21a; Kro+20a].

In this work, we show how to connect concrete implementations of concurrent and distributed systems to abstract system models. We develop Trillium, a modular language-agnostic separation logic framework for establishing *history-sensitive* refinement relations between programs and models, which relate *traces* of program executions to *traces* of a model. Not only does the refinement relation establish a formal connection and hence a specification of the program in its own right, but it makes it possible to transfer both safety and liveness properties of

```
let rec inc_loop () =
  let n = !ℓ in
  cas(ℓ, n, n + 1);
  inc_loop ()
in
  inc_loop () || inc_loop ()
```

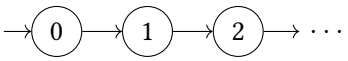


Figure 5.1: A concurrent counter `inc` and its corresponding model $\mathcal{M}_{\mathsf{inc}}$.

a model to its implementation. Consequently, this allows us to exploit existing properties of abstract models to establish properties about programs that are not expressible using ordinary concurrent separation logics.

Our development is *foundational* [App01], in that all our results, including the operational semantics, the models, and the logic, are formalized in the Coq proof assistant on top of the Iris base logic while using the Iris Proof Mode [KTB17] for reasoning within the logic. In Section 5.4 we show how concrete implementations of two distributed protocols, *two-phase commit* [Gra78] and *single-decree Paxos* [Lam98; Lam01], formally refine their abstract TLA$^+$ specifications, and how safety properties of the models can be transferred to the implementations. Moreover, we explain in Section 5.5 and Section 5.6 how to prove liveness properties of distributed and concurrent programs through refinement: we prove strong eventual consistency of an implementation of a *Conflict-Free Replicated Data Type* (CRDT) [Sha+11] and fair termination of a concurrent program.

The Trillium methodology. Consider the example in Figure 5.1 that shows a program `inc` written in an ML-like imperative language. The program uses a global reference ℓ, with an initial value of zero, that it increments concurrently in two infinite loops using *compare-and-set* operations. Our goal is to prove that the reference ℓ takes successively the values 0, 1, 2, ... without skipping any number. To do so, we will establish a refinement between the program and an abstract model $\mathcal{M}_{\mathsf{inc}}$, depicted on the right-hand side of Figure 5.1. A *model* in Trillium is an arbitrary state transition system $\mathcal{M} = (A_{\mathcal{M}}, \rightharpoonup_{\mathcal{M}})$ where $A_{\mathcal{M}}$ denotes a set of states and $\rightharpoonup_{\mathcal{M}} \subseteq A_{\mathcal{M}} \times A_{\mathcal{M}}$ the transition relation on states. In this particular model, at each step, either the state does not change, or it is increased by one. The refinement will express that, at all times, the value stored at location ℓ is equal to the current value of the model and no value is skipped. Note that prior techniques for proving refinements in Iris could not express the fact that the program never skips a number (see Section 5.2.1 for a more elaborate discussion) and that our techniques also allow us to prove the same property about programs that allocate locations dynamically (see Section 5.3).

As Trillium is a *separation logic*, propositions denote not only facts but *ownership* of resources. For example, the proposition $\ell \mapsto v$ asserts exclusive ownership of location ℓ storing value v. When, as in the example of Figure 5.1, two threads access the same location, the resource P can be shared by placing it into an *invariant*, *e.g.*, $\boxed{P}$. Invariants are guaranteed to hold a every computation step and they are duplicable, *i.e.*, $\boxed{P} \vdash \boxed{P} * \boxed{P}$, which means invariants can be shared among multiple threads. The *separating conjunction* $P * Q$ holds for resources that can be split into two disjoint sub-resources which satisfy P and Q respectively.

When instantiated with a programming language, Trillium provides a new notion of Hoare triple $\{P\}\ e\ \{Q\}^{\mathcal{M}}$ and a special resource $\mathsf{Model}_\circ(\delta : \mathcal{M})$ that represents the current state of the abstract model $\mathcal{M}$ and asserts that it is equal to δ. We omit $\mathcal{M}$ from the connectives

when it is clear from the context. The refinement we want to show can be established in Trillium by proving, using the logical predicates and rules of the Trillium logic, the validity of the following Hoare triple:

$$\{\boxed{\exists n.\, \ell \mapsto n * \mathsf{Model}_\circ(n)}\}\ \texttt{inc}\ \{\mathsf{False}\}^{\mathcal{M}_{\mathsf{inc}}}$$

where `inc` is the example program. The invariant ties the contents of the reference ℓ to the current state of the model and enforces that they always agree. The postcondition of the Hoare triple is False as the program does not terminate.

Intuitively, a Hoare triple like the above means the program is *safe*, *i.e.*, it does not crash, and that the post condition holds for its end-state. To prove this formally, Trillium comes with an *adequacy theorem*. In Trillium, however, the adequacy theorem additionally concludes a history-sensitive refinement relation between all traces generated by the program and the model and, as such, that the model state and the reference ℓ progress in a lock-step fashion.

Definition 5.0.1 (History-Sensitive Refinement)**.** Let τ be a program execution trace, κ a model trace, and ξ a binary relation on traces. τ is a *history-sensitive refinement* of κ under ξ whenever $\tau \precsim_\xi \kappa$ holds, where $\tau \precsim_\xi \kappa$ is coinductively defined by:

$$\tau \precsim_\xi \kappa \triangleq \xi(\tau, \kappa) \wedge \forall c.\, last(\tau) \to c \Rightarrow \exists \delta.\, \tau :: c \precsim_\xi \kappa :: \delta$$

where :: denotes trace extension, and $\to$ is the stepping-relation of the operational semantics of the programming language.

That is, an execution trace τ is a history-sensitive refinement of a model trace κ under ξ if $\xi(\tau, \kappa)$ holds and for all configurations that the last configuration of τ may step to, there exists a model state δ such that the extended traces refine each under ξ as well. This relation straightforwardly extends to all finite prefixes of all possibly-infinite traces generated by program execution as detailed in Section 5.2.2. This will be crucial for properties, such as liveness, that we consider. Figure 5.2 graphically illustrates history-sensitive refinement.

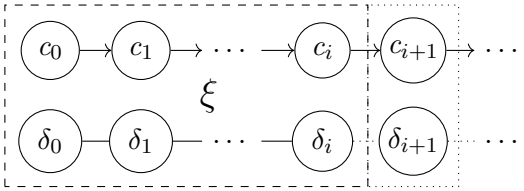


Figure 5.2: Illustration of Definition 5.0.1. Trace $\tau = c_0 :: c_1 :: \ldots :: c_i$ refines $\kappa = \delta_0 :: \delta_1 :: \ldots :: \delta_i$ under ξ, written $\tau \precsim_\xi \kappa$ if $\xi(\tau, \kappa)$ and for any c_{i+1} that c_i may step to there exists δ_{i+1} such that $\tau :: c_{i+1} \precsim_\xi \kappa :: \delta_{i+1}$ holds.

For our running example, we pick the predicate ξ_{inc} to express that the value of the reference ℓ in the current state of the program (the last state of the trace) is equal to the current state of the model and does not skip model states:

$$\xi_{inc}(\tau, \kappa) \quad \triangleq \quad heap(last(\tau))(\ell) = last(\kappa) \wedge stuttering(\kappa)$$

where the operator $last$ returns the last element of a (non-empty) trace, $heap(c)$ returns the heap component of a program configuration c, and

$$stuttering(\kappa) = \begin{cases} last(\kappa') = \delta \vee last(\kappa') \rightharpoonup_{\mathcal{M}_{\mathsf{inc}}} \delta & \text{if } \kappa = \kappa' :: \delta \\ \mathsf{True} & \text{otherwise} \end{cases}$$

In this simple example, the relation ξ_{inc} only depends on the last elements of the two traces, in which case history-sensitive refinement reduces to the usual notion of simulation. We will see in Section 5.5 and Section 5.6 situations where the additional expressive power of history-sensitive relations is needed.

We can now state a simplified version of the adequacy theorem, saving the full statement of the adequacy theorem for Section 5.2.

Theorem 5.0.2 (Simplified adequacy)**.** *Let e be a program, σ_0 a program state, and Φ an Iris predicate on values. Let $\delta_0 \in \mathcal{M}$ be a model state and ξ a finitary binary relation on program and model traces. Suppose*

$$\{\boxed{P_1} * \cdots * \boxed{P_k}\}\ e\ \{\Phi\}^{\mathcal{M}} \qquad \text{and} \qquad AlwaysHolds(\xi, e, \sigma_0, \delta_0)$$

can be proved in the Trillium logic*, then $(e, \sigma_0) \precsim_\xi \delta_0$ holds in the meta-logic, e.g. Coq, and the program e is safe when started in the initial heap σ_0. The Trillium predicate $AlwaysHolds(\xi, e, \sigma_0, \delta_0)$ expresses that the invariants $\boxed{P_1}, \ldots, \boxed{P_k}$ imply that ξ holds at the current state.*

We emphasize that the Trillium Hoare triple is an extension of the usual Iris-style Hoare triple and we can seamlessly reuse Iris program logic proofs in Trillium. The proof rules of Trillium include all the usual rules of program logics built on top of Iris; there is just one additional proof rule for reasoning about refinement. It also means that the *same* Hoare triple can be used to capture refinement as well as functional correctness. In particular, it is possible to use properties of the model when proving a Hoare triple as it is embedded in the logic as a resource. The specification can, *e.g.*, be used by clients to show safety, which might rely on properties of the model; see Section 5.4.3 for an example and a more detailed discussion.

Composing refinements. One of the aspects that makes refinements a powerful tool in studying programs and computer systems is their composability. That is, one can prove that a system A refines B and that B refines C in order to establish A refines C. This concept is useful especially in cases where the gap, *e.g.*, in implementation details, between A and B, and between B and C, are smaller than the gap between A and C which makes those refinements easier to establish. What we are interested in, especially in verification of distributed systems, is a tower of composable refinements where one end is a low-level, realistic program implementation while the other end is a high-level, abstract state transition system (STS) that captures the essence of the implementation.

The literature on reasoning about both programs and systems includes many works on proving refinements between pairs of programs [BST12; KTB17; TDB13; Tur+13], including reasoning in program logics, and pairs of STSs [Lam92]. In this work, we focus on the *transition-point refinement* where we only consider refinements between programs and STSs. See Section 5.7 for a comparison of the present work with other works that also involve transition-point refinements. In this regard, Trillium enables incorporation of different tools as different refinements need not be proven in the same formalism in order for them to be composed. An example of this fact is illustrated in our example where we show refinement between the implementation of single-decree Paxos and its TLA$^+$ spec (see Section 5.4). The TLA$^+$ spec that we use in this example is in fact Lamport's low-level TLA$^+$ spec [Lam19] which itself is shown to refine another high-level TLA$^+$ model. Hence, our proof also establishes that the single-decree Paxos implementation we consider refines this more high-level TLA$^+$ model.

We believe, but have not formally established, that Trillium can be combined with other approaches for refinements between a pair of programs, *e.g.*, Krebbers et al. [KTB17], in order to establish a refinement between a program and an STS.

Instantiations. In Section 5.2.3 we instantiate Trillium with AnerisLang, an OCaml-like imperative programming language with network socket primitives, and recover an extension of the Aneris program logic [Kro+20a] for reasoning about distributed systems. The extension inherits from Aneris both (1) *horizontal modularity* via node-local and thread-local reasoning, which allows one to verify—and now refine—distributed systems by verifying each thread and each node in isolation, and (2) *veritical modularity* via separation logic features such as the frame rule and the bind rule, which allow one to compose proofs of different components within each node. Using the Trillium instantiation, we extend these principles to history-sensitive refinement which additionally allows us to prove liveness properties of distributed systems. In Section 5.6 we consider an instantiation of Trillium with HeapLang, a concurrent (non-distributed) language, and show how Trillium can also be used to reason about termination of concurrent programs by establishing a *fair* termination-preserving refinement of a suitable model. Figure 5.3 gives an overview of the different components and concepts developed in this work, how they depend on each other, and which formal system is used to reason about each part.

Contributions. In summary, we make the following contributions:

- We introduce Trillium, a language-agnostic separation logic framework for establishing history-sensitive refinement among traces of program executions and traces of abstract models through Hoare-style reasoning.
- We instantiate Trillium with AnerisLang to get an extension of Aneris that allows us prove refinements for distributed systems. We prove soundness of a wide collection of proof rules, including all the earlier proof rules of Aneris, with respect to the notion of Hoare triple provided by Trillium.
- We use this instantiation to prove the correctness of concrete implementations of two distributed protocols, two-phase commit and single-decree Paxos, by showing that they refine abstract TLA$^+$ specifications. To the best of our knowledge, this is the first foundationally verified proof that a concrete implementation of a distributed protocol correctly implements an abstract TLA$^+$ specification. We also demonstrate how to use the *same* specification to prove properties about the implementation, by relying on correctness theorems of the TLA$^+$ specification, and to show functional correctness of clients.
- We further show functional correctness and strong eventual consistency of a concrete implementation of a Conflict-Free Replicated Data Type (CRDT). To the best of our knowledge, this is the first such proof that takes into account the inter-replica communication at the level of the implementation; the concurrent interactions with the user-exposed operations makes it non-trivial to reason about eventual consistency.
- We instantiate Trillium with HeapLang, a higher-order non-distributed concurrent imperative programming language and use the resulting logic to show fair termination of a concurrent program by establishing a fair termination-preserving refinement of a model.

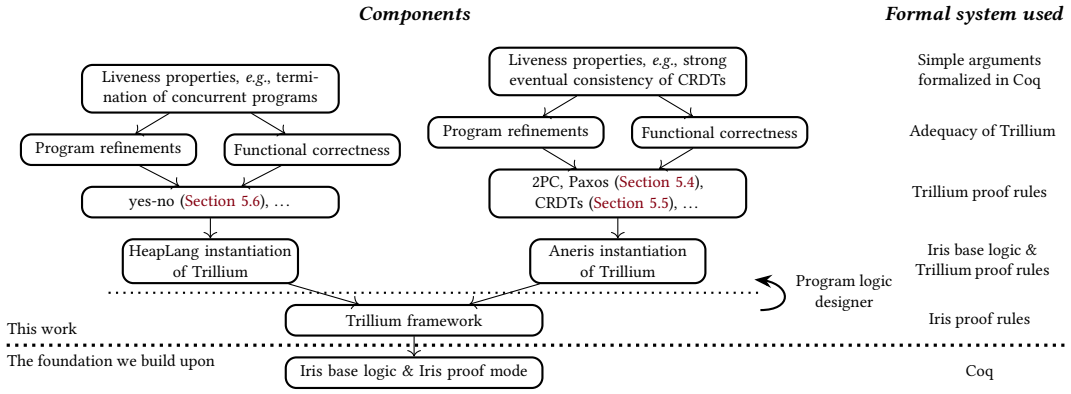


Figure 5.3: An overview of the components described and developed in this work with arrows indicating dependency. The column to the right describes the formal system used to reason about the components in the row.

5.1 A trace program logic framework

The Trillium program logic is language agnostic and is defined with respect to an operational semantics given by a notion of expression $e \in \mathit{Expr}$, value $v \in \mathit{Val} \subseteq \mathit{Expr}$, evaluation context $K \in \mathit{Ectx}$, program state $\sigma \in \mathit{State}$ (a model of, *e.g.*, the heap and/or the network), and a *primitive reduction relation* $e_1, \sigma_1 \rightarrow_{\mathsf{h}} e_2, \sigma_2, \vec{e_f}$ that relates an expression e_1 and a state σ_1 to an expression e_2, a state σ_2, and a list $\vec{e_f}$ of expressions, corresponding to the threads forked by the reduction. The top-level reduction relation $c \rightarrow c'$ on *system configurations* $c, c' \in \mathit{Cfg} = \mathit{List}(\mathit{Expr}) \times \mathit{State}$ is a standard interleaving small-step semantics using evaluation contexts, where the first component of a configuration is a list of expressions which denotes the threads of the system.

When concerned with program execution traces, we will only be interested in *valid traces*, written $\mathit{valid}(\tau)$, over configurations where each configuration reduces to the next according to the reduction relation of the operational semantics.

5.1.1 Program logic

When instantiated with a programming language, the Trillium program logic comes with a set of low-level proof rules which relate the validity of some Hoare triples with the semantics of the language. When Trillium is instantiated with a concrete programming language, such as AnerisLang, the usual high-level proof rules are proved using the low-level rules. Once this work has been done, which happens only once for a given language, Trillium can be used like any Hoare-style logic.

A selection of the proof rules for the AnerisLang instantiation is shown in Figure 5.4; the notion of expression in Aneris also includes an ip address on which the expression is running. AnerisLang is an OCaml-like programming language with network primitives for creating and binding network sockets as well as sending (`sendto`) and receiving (`receivefrom`) messages. The operational semantics is designed so that the primitives closely model Unix sockets and UDP networking.

The proof rules include all of the earlier proof rules of Aneris. We write $\{P\}\, e\, \{v.Q\}^{\mathcal{M}}$ to explicate that the post condition is a predicate $\lambda v.\, Q$ on the result of evaluating e. HT-FRAME

and HT-BIND constitute the quintessential rules for modular reasoning in separation logic: the frame rule says that executing e for which we know $\{P\}\ e\ \{Q\}^{\mathcal{M}}$ cannot possibly affect parts of the heap that are separate from its *footprint* and the bind rule lifts a local specification to a more global specification in evaluation context K. The rules HT-ALLOC and HT-LOAD are for reasoning about allocating and reading from references, respectively, on a node with ip-address ip. The rule HT-SENDTO reasons about sending a message m over a socket z; both the pre- and postcondition contains Aneris-specific network resources concerned with the status of socket handles, which messages that have been sent (T) and received (R) at a particular address (a), and Aneris communication protocols (Φ). We refer to Krogh-Jespersen et al. [Kro+20a] for a detailed description of the Aneris resources and more proof rules.

HT-FRAME
$$\frac{\{P\}\ \langle ip;\ e\rangle\ \{Q\}^{\mathcal{M}}}{\{P * R\}\ \langle ip;\ e\rangle\ \{Q * R\}^{\mathcal{M}}}$$

HT-BIND
$$\frac{\{P\}\ \langle ip;\ e\rangle\ \{v.Q\}^{\mathcal{M}} \qquad \forall v.\{Q\}\ \langle ip;\ K[v]\rangle\ \{R\}^{\mathcal{M}}}{\{P\}\ \langle ip;\ K[e]\rangle\ \{R\}^{\mathcal{M}}}$$

HT-ALLOC
$$\{\mathsf{True}\}\ \langle ip;\ \texttt{ref}(v)\rangle\ \{v.\ \exists \ell.\ v = \ell * \ell \mapsto_{ip} v\}^{\mathcal{M}}$$

HT-LOAD
$$\{\ell \mapsto_{ip} w\}\ \langle ip;\ !\,\ell\rangle\ \{v.v = w * \ell \mapsto_{ip} w\}^{\mathcal{M}}$$

HT-SENDTO
$$\{z \hookrightarrow_{ip} \mathsf{Some}\ a * a \rightsquigarrow (R, T) * to \Rrightarrow \Phi * \Phi(m, a, to) * a.\mathsf{ip} = ip\}$$
$$\langle ip;\ \texttt{sendto}\ z\ m\ to\rangle$$
$$\{v.\ v = |to| * z \hookrightarrow_{ip} \mathsf{Some}\ a * a \rightsquigarrow (R, T \cup \{(m, a, to)\})\}^{\mathcal{M}}$$

Figure 5.4: Selected proof rules of the Aneris instantiation of Trillium.

To reason about model refinement, Trillium admits *one* additional proof rule. For convenience, we show its AnerisLang instantiation HT-TAKE-STEP.

HT-TAKE-STEP
$$\frac{\{P\}\ e\ \{Q\}^{\mathcal{M}} \qquad \delta \rightharpoonup_{\mathcal{M}} \delta' \qquad \mathrm{Atomic}(e) \qquad e \notin \mathit{Val}}{\{P * \mathsf{Model}_\circ(\delta)\}\ e\ \{Q * \mathsf{Model}_\circ(\delta')\}^{\mathcal{M}}}$$

The rule allows the state of the model to be updated during atomic operations by updating the $\mathsf{Model}_\circ(\delta)$ resource according to the stepping relation of the model. The rule can, naturally, be combined with other rules, such as HT-INV-OPEN which we show later in this section, to access the $\mathsf{Model}_\circ(\delta)$ resource from inside invariants. Note how the rule also *requires* the program to take a single step: it is atomic, so it takes *at most* a single step, and it is not a value, hence it *must* take a step. Symmetrically, the rule also prevents us from taking two consecutive steps in the model.

As a consequence of building on the Iris framework, the Trillium logic features all the usual connective and rules of high-order separation logic, some of which are summarized in Figure 6.5; the type of Iris propositions is *iProp*.[1]

[1]To avoid the issue of reentrancy, invariants are annotated with a *namespace* and Hoare triples and update

$$
\begin{aligned}
\sigma &::= 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathit{iProp} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma \mid \ldots && \text{(Types)}\\
P, Q &::= x \mid \lambda x : \sigma.\, P \mid \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q && \text{(Propositional logic)}\\
&\mid \forall x : \sigma.\, P \mid \exists x : \sigma.\, P \mid t = u && \text{(Higher-order logic)}\\
&\mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v && \text{(Separation logic)}\\
&\mid \boxed{P} \mid \triangleright P \mid \Rightarrow\!\!\!\Rightarrow P \mid \boxed{a}^{\gamma} \mid \ldots && \text{(Iris connectives)}\\
&\mid \{P\}\, e\, \{v.Q\}^{\mathcal{M}} \mid \mathsf{Model}_{\circ}(\delta : \mathcal{M}) && \text{(Trillium connectives)}
\end{aligned}
$$

Figure 5.5: Syntax of Iris and Trillium. t and u represent arbitrary terms.

As as a separation logic, it has a separating conjunction $P*Q$ and the corresponding notion of implication, the *magic wand* $P \mathrel{-\!\!*} Q$, in that it satisfies *modus ponens*: $P * (P \mathrel{-\!\!*} Q)$ entails Q.

The Iris base logic allows one to allocate invariants $\boxed{P}$ and user-defined ghost state $\boxed{a}^{\gamma}$. The logical support for both user-defined invariants and ghost state allows us to relate (ghost and physical) resources to each other. This is crucial for our specifications as already exemplified in the introduction.

The contents of invariants may be accessed in a carefully restricted way in Trillium using HT-INV-OPEN while verifying a program e. The rule permits us to access the contents of an invariant, which involves acquiring ownership of P before the verification of e and giving back ownership of P afterwards. Crucially, e is required to be *atomic*, meaning that the computation completes in a single step. Notice that invariants are just another kind of proposition and it can be used anywhere that normal propositions can be used, including invariants themselves, referred to as *impredicativity*. This is the reason P appears under a "later" modality $\triangleright P$; without, the rule for opening invariants is unsound. However, if one does not store invariants inside invariants or make similar impredicative constructions, one can generally ignore the later modality as we will do throughout the chapter.

$$
\textsc{Ht-inv-open}\quad
\frac{\mathrm{Atomic}(e) \qquad \{\triangleright P * Q_1\}\, e\, \{\triangleright P * Q_2\}^{\mathcal{M}}}{\left\{\boxed{P} * Q_1\right\} e \left\{\boxed{P} * Q_2\right\}^{\mathcal{M}}}
$$

User-definable ghost state can be introduced via the proposition $\boxed{a}^{\gamma}$ which asserts ownership of a piece of ghost state a at ghost location γ. Resources can be *updated* through the *update modality* $\Rightarrow\!\!\!\Rightarrow P$. The update modality provides a way, inside the logic, to talk about the resources we *could own* after performing an update to what we *do own*. The intuition is that $\Rightarrow\!\!\!\Rightarrow P$ holds for a resource r, if from r we can do an update to some r' that satisfies P while not invalidating any existing resources. Using GHOST-UPDATE, we can update a user-defined ghost resource from $\boxed{a}^{\gamma}$ to $\boxed{b}^{\gamma}$ under the update modality if the particular user-defined ghost *resource algebra* permits it ($a \rightarrowtail b$). In a similar spirit, INV-ALLOC allocates an invariant $\boxed{P}$ by

modalities with *masks*. We omit both for the sake of presentation in most of the paper as they are orthogonal to the novelties of Trillium.

giving up ownership of P.

$$\textsc{Ghost-update}\ \frac{a \rightarrow b}{\boxed{a}^{\gamma} \vdash \Rrightarrow \boxed{b}^{\gamma}} \qquad \textsc{Inv-alloc}\ P \vdash \Rrightarrow \boxed{P}$$

We refer to Jung et al. [Jun+18b] for a thorough treatment of how user-defined ghost state is constructed in Iris. In the remainder of this chapter, we will simply describe user-defined ghost state in terms of the rules that govern the concrete instantiations that we use.

5.2 The semantics of Hoare triples

In Trillium, Hoare triples are defined using more primitive notions as:

$$\{P\}\, e\, \{Q\}^{\mathcal{M}} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}^{\mathcal{M}}\, e\, \{Q\})$$

where $\mathsf{wp}^{\mathcal{M}}\, e\, \{Q\}$ is the *weakest precondition* that is required for e to terminate safely in a value v satisfying Q *and* such that the execution of e refines the model $\mathcal{M}$. A technicality is that it is necessary to wrap the implication in a *persistence modality* $\Box$ to ensure that Hoare triples are duplicable and can be used repeatedly.

The Trillium weakest precondition is defined using the Iris base logic, similarly to how the weakest precondition of the Iris program logic is defined using the Iris base logic [Jun+18b]. The key idea and novelty of the Trillium weakest precondition is to track a model trace alongside a program execution trace and enforce that whenever the program takes a step according to the operational semantics, there is a state in the model that corresponds to the program step. Importantly, it does *not* mention an explicit model state or traces as the relationship between the program and the model trace is encapsulated inside the definition of the weakest precondition.

In order to avoid reentrancy issues, where invariants are opened in a nested (and unsound) fashion, Iris features *invariant namespaces* $\mathcal{N} \in \mathit{InvName}$ and *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$. Up until now, these matters have been omitted but to fully state and comprehend the definition of the weakest precondition—and consequently our adequacy theorem—they are necessary evils. We emphasize that their use is entirely standard and identical to the use in the definition of the Iris weakest precondition.

The Iris base logic annotates each invariant $\boxed{P}^{\mathcal{N}}$ with a namespace $\mathcal{N}$ to identify the invariant and we annotate the weakest precondition with a mask $\mathcal{E}$ to keep track of which invariants are enabled and may be opened. In order to work with invariants formally in Iris, the update modality is annotated with two masks: ${}_{\mathcal{E}_1}\Rrightarrow_{\mathcal{E}_2}$. We write $\Rrightarrow_{\mathcal{E}}$ when $\mathcal{E}_1 = \mathcal{E}_2 = \mathcal{E}$ and $\Rrightarrow$ when $\mathcal{E} = \top$, the set of all masks. As discussed earlier, the update modality is used to reason about ghost state akin to how a weakest precondition is used to reason about physical state. On top of this, the mask annotations $\mathcal{E}_1$ and $\mathcal{E}_2$ denote which invariants are enabled and may be opened before and after the modality is introduced, respectively. Intuitively, the proposition ${}_{\mathcal{E}_1}\Rrightarrow_{\mathcal{E}_2} P$ holds for resources that—given the invariants in $\mathcal{E}_1$ are enabled—can be updated to resources that satisfy P—with the invariants in $\mathcal{E}_2$ enabled—without violating the environment's ownership of resources. The contents of invariants may be accessed in a carefully restricted way (Inv-open-upd): to prove $\Rrightarrow_{\mathcal{E}} Q$ we map open an invariant with namespace

$\mathcal{N}$ and assume $\triangleright P$ as long as $\mathcal{N} \in \mathcal{E}$ and we can re-establish the invariant as well.

$$\textsc{Inv-open-upd}\quad \frac{\mathcal{N} \in \mathcal{E}}{\boxed{P}^{\mathcal{N}} * (\triangleright P \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} (\triangleright P * Q)) \vdash \Rrightarrow_{\mathcal{E}} Q}$$

As we will see, the careful placement of the update modalities in the definition of the weakest precondition will require all invariants in the mask annotation $\mathcal{E}$ to be enabled after every physical step in the operational semantics, corresponding to the intuition that invariants can only be opened atomically. For more details on invariants and the update modality in Iris we refer to Birkedal and Bizjak [BB17] and Jung et al. [Jun+18b].

As a consequence of the Trillium framework's generality and parameterization by both a language and a model, we have no knowledge of the physical state, model, and how they might be related and reflected in the logic. Therefore, we parameterize the weakest preconditions by a *trace interpretation* relation $S : \mathit{Trace}(\mathit{Cfg}) \times \mathit{Trace}(A_{\mathcal{M}}) \to \mathit{iProp}$ that ties program execution traces and model traces to Iris resources. In contrast, the standard Iris weakest precondition uses a state interpretation predicate $S : \mathit{State} \to \mathit{iProp}$ for reflecting the physical state, such as a heap, as resources in the logic to give meaning to, for example, the points-to connective $\ell \mapsto v$. The trace interpretation relation subsumes this notion and is able to reflect both the current physical state and model state, but also their histories and relation, as resources.

Given a *trace interpretation* S, the definition of the weakest precondition is defined by guarded recursion in the Iris base logic as follows (highlighting the novel aspects in blue):

$$\begin{aligned}
\mathsf{wp}^{\mathcal{M}}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq\ & \big(e \in \mathit{Val} * \Rrightarrow_{\mathcal{E}} \Phi(e)\big) \vee \\
& \Big(e \notin \mathit{Val} * \forall \tau, \tau', \kappa, \sigma, K, T_1, T_2. \\
& \quad \mathit{valid}(\tau) * \tau = (\tau' :: (T_1 +\!\!+ K[e] +\!\!+ T_2, \sigma)) * S(\tau, \kappa) \mathrel{-\!\!*} {}_{\mathcal{E}}\Rrightarrow_{\emptyset} \\
& \quad\ \mathit{reducible}(e, \sigma) * \\
& \quad\ \Big(\forall e_2, \sigma_2, \vec{e_f}.\ (e, \sigma) \rightsquigarrow (e_2, \sigma_2, \vec{e_f}) \mathrel{-\!\!*} \triangleright {}_{\emptyset}\Rrightarrow_{\mathcal{E}} \\
& \quad\quad \exists \delta.\ S(\tau :: (T_1 +\!\!+ K[e_2] +\!\!+ T_2 +\!\!+ \vec{e_f}, \sigma'), \kappa :: \delta) * \\
& \quad\quad \mathsf{wp}^{\mathcal{M}}_{\mathcal{E}}\, e_2\, \{\Phi\} * \mathop{\Large *}_{e' \in \vec{e_f}} \mathsf{wp}^{\mathcal{M}}_{\mathcal{E}}\, e'\, \{\mathsf{True}\}\Big)\Big)
\end{aligned}$$

where $\mathcal{E}$ is a mask, $\mathcal{M}$ is a model, and $\Phi \in \mathit{Val} \to \mathit{iProp}$ a predicate on values. The definition is by case distinction. If the program has already terminated (*i.e.*, e is a value) the postcondition should hold. If the program is not a value, then for all model traces κ and valid execution traces τ where e is about to take a step on some thread *and* the trace interpretation holds, there are two requirements. First, the program should be *reducible*, which means it is able to take a thread-local step, ensuring safety. Second, with access to all the invariants in $\mathcal{E}$, for any possible configuration that e might step to, there should exist a model state δ such that the trace interpretation holds for the extended trace. Additionally, if the program makes a step, then the weakest precondition must hold for the reduced program as well as for all threads it might have forked-off.

In summary, the Trillium weakest precondition is a conservative generalization of the usual Iris-style weakest precondition that admits all the usual rules that you would expect from program logics built on top of Iris. In addition, it offers Hoare-style reasoning about the

relationship between program execution traces and model traces through the addition of a single rule HT-TAKE-STEP.

5.2.1 Adequacy

Given a weakest precondition for a program e and binary relation ξ on execution traces and model traces, the adequacy theorem of Trillium concludes that e is a history-sensitive refinement of the model and that ξ holds for all possible traces generated by the program and the model. Importantly, the refinement property only relies on the definition of the operational semantics and the traces over model states: when a refinement property is established using Trillium, one does not need to trust Iris nor Trillium, but only that the operational semantics and the model are as intended.

The adequacy theorem shown below involves a technical condition that requires the ξ relation to be *finitary*: the set $\{\delta \mid \xi(\tau :: c, \kappa :: \delta)\}$ has to be finite, for any τ, κ, and c. This condition is necessary as the underlying model of the base logic of Iris is step-indexed over the natural numbers.[2] The property is generally straightforward to prove: often, as in all our examples, either the model itself is finitely branching or the program configuration determines exactly the current state of the model.

Theorem 5.2.1 (Adequacy). *Let e be a program, σ_0 a program state, and Φ an Iris predicate on values. Let $\delta_0 \in \mathcal{M}$ be a model state and ξ a finitary binary relation on execution traces and traces of $\mathcal{M}$. If*

$$\Rrightarrow_{\top} S((e, \sigma_0), \delta_0) * \mathsf{wp}^{\mathcal{M}}_{\top}\, e \left\{\Phi\right\} * \mathit{AlwaysHolds}(\xi, e, \sigma_0, \delta_0)$$

then $(e, \sigma_0) \precsim_{\xi} \delta_0$ holds in the meta-logic. Here $\mathit{AlwaysHolds}(\xi, e, \sigma, \delta)$ is the Trillium predicate

$$\forall \tau, \kappa. \begin{pmatrix} S(\tau, \kappa) * \mathit{valid}(\tau) * |\tau| = |\kappa| * \mathit{first}(\tau) = (e, \sigma_0) * \mathit{first}(\kappa) = \delta_0 * \\ \left(\forall \tau', \kappa', c', \delta'.\ \tau = \tau' :: c' \wedge \kappa = \kappa' :: \delta' \Rightarrow \xi(\tau', \kappa')\right) * \\ \begin{pmatrix} \forall e_1, \ldots, e_n, \sigma.\ \mathit{last}(\tau) = (e_1, \ldots, e_n; \sigma) \Rightarrow \\ (\forall 1 \le i \le n.\ e_i \in \mathit{Val} \vee \mathit{reducible}(e_i, \sigma)) \wedge (e_1 \in \mathit{Val} \Rightarrow \Phi(e_1)) \end{pmatrix} \end{pmatrix} \twoheadrightarrow {}_{\top}\Rrightarrow_{\emptyset}\ \xi(\tau, \kappa)$$

Intuitively, the $\mathit{AlwaysHolds}$ predicate simply says that ξ should follow from all invariants, *cf.*, the update modality masks. When proving this, one is additionally allowed to assume that the trace interpretation S holds, the execution trace is valid and the trace starts from e, σ_0, and δ_0; ξ holds for the prefixes of the traces; none of the threads at the current execution point are stuck; and if the first thread (corresponding to e) has evaluated to a value, then the postcondition Φ holds.

5.2.2 Refinements for infinite executions

The adequacy theorem concludes a refinement relation for finite program executions, however, it straightforwardly extends to infinite executions as well. A *possibly-infinite trace* (over

[2] The finiteness condition does not restrict the properties we can transport along a refinement, see Section 5.5 and Section 5.6. One possible approach to avoid the finiteness condition is to use the recently proposed Transfinite Iris [Spi+21] as the base logic. However, it is not obvious that one can carry out our development in Transfinite Iris, since Transfinite Iris does not include all of the basic reasoning principles of standard Iris; in particular, it lacks commutation rules for the later modality.

some set) is a finite or infinite sequence (of elements from the set). For possibly-infinite traces, refinement is a predicate on pairs of traces: a finite trace, corresponding, intuitively, to the trace up until now and a possibly-infinite trace corresponding to the remaining execution. Let τ and κ denote finite traces and $\dot{\tau}$ and $\dot{\kappa}$ possibly-infinite traces. We define history-sensitive refinement between possibly-infinite execution traces and model traces coinductively as follows:

$$(\tau, \dot{\tau}) \mathrel{\dot{\precsim}_\xi} (\kappa, \dot{\kappa}) \triangleq \begin{cases} \xi(\tau, \kappa) & \text{if } \dot{\tau} = \emptyset \text{ and } \dot{\kappa} = \emptyset \\ \xi(\tau, \kappa) \wedge (\tau :: c, \dot{\tau}') \mathrel{\dot{\precsim}_\xi} (\kappa :: \delta, \dot{\kappa}') & \text{if } \dot{\tau} = c :: \dot{\tau}' \text{ and } \dot{\kappa} = \delta :: \dot{\kappa}' \end{cases}$$

Intuitively, this states that all finite prefixes of the possibly-infinite traces preserve ξ.

Corollary 5.2.2 (Possibly-Infinite Refinement)**.** *Let τ, κ be finite traces such that $\tau \precsim_\xi \kappa$, and let $\dot{\tau}$ be a possibly-infinite program trace such that τ and $\dot{\tau}$ are valid. Then there exists a possibly-infinite model trace $\dot{\kappa}$ such that $(\tau, \dot{\tau}) \mathrel{\dot{\precsim}_\xi} (\kappa, \dot{\kappa})$ holds.*

Note that it is *not* be possible to extract history-sensitive refinement relations from the usual weakest precondition in an Iris-style program logic. To see why, consider the example from the introduction. Using Iris's mechanism for user-defined ghost state, one can easily definite ghost state modelling a monotonically increasing counter, *e.g.*, a resource $\boxed{n}^\gamma$ such that

$$\boxed{n}^\gamma \vdash \Rightarrow \boxed{n+1}^\gamma$$

is the only way it can be updated. Using this ghost theory, one *could* also prove a similarly-looking specification of the shape

$$\left\{ \boxed{\exists n.\, \ell \mapsto n * \boxed{n}^\gamma} \right\} \texttt{inc} \{\ldots\}$$

using the standard weakest precondition theory from Iris. However, this specification would *also* be satisfied by, for example, an implementation with threads that increment the counter by two, which clearly does not satisfy our refinement notion. If the counter represents a digital clock, for example, it would be far from ideal if it was allowed to skip every other minute.

The fundamental problem is that resource updates are *transitive* which means that if a particular resource algebra allows a resource update from a to b and from b to c it will also allow it to be updated directly from a to c; in particular, a ghost theory that allows our counter to progress as 0-1-2-… will also allow it to progress directly from 0 to 2. In Trillium, the model is encapsulated in the weakest precondition whose definition forces us to match up exactly *one* model step per computation step. In the Aneris instantiation (as detailed in the next section) we consider the reflexive closure, however in Section 5.6 we do not, which is crucial for the soundness of our method for showing fair termination-preserving refinement.

5.2.3 Aneris Instantiation of Trillium

To instantiate Trillium with AnerisLang, we define the trace interpretation using three components:

$$S(\tau, \kappa) \triangleq \mathsf{physSI}(\mathit{last}(\tau)) * \mathsf{Model}_\bullet(\mathit{last}(\kappa)) * \mathit{stuttering}(\kappa)$$

The physSI predicate corresponds to a state interpretation and associates the physical Aneris state, *i.e.*, the heap and the network, to Aneris resources, akin to how the heap is associated to

```
let rec incr_loop l =
  let n = !l in
  cas(l, n, n + 1);
  incr_loop l
in
let l = ref<s> 0 in
fork (incr_loop l); incr_loop l
```

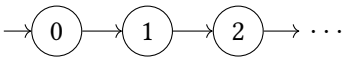


Figure 5.6: A concurrent counter $\texttt{inc}'$ with dynamically allocated memory and its model $\mathcal{M}_{\mathsf{inc}}$.

the $\ell \mapsto v$ resource in standard Iris instantiations. The $\mathsf{Model}_\bullet$ predicate ties the *current* state of the model to an instance of the *authoritative resource algebra* [Jun+18b]. The $\mathsf{Model}_\bullet$ predicate (the authoritative part of the model) comes with a counterpart $\mathsf{Model}_\circ$ (the fragmental part of the model) satisfying

$$\mathsf{Model}_\bullet(\delta) * \mathsf{Model}_\circ(\delta') \vdash \delta = \delta'.$$

As discussed in the introduction, this gives the user a way of connecting the model's current state to other separation logic resources through the $\mathsf{Model}_\circ(\delta)$ resource. The *stuttering* predicate is defined as

$$\mathit{stuttering}(\kappa) = \begin{cases} \mathit{last}(\kappa') = \delta \lor \mathit{last}(\kappa') \rightharpoonup_{\mathcal{M}} \delta & \text{if } \kappa = \kappa' :: \delta \\ \mathsf{True} & \text{otherwise} \end{cases}$$

which allows the model trace to *stutter*: a model state in the trace is either the same as the previous or it is related to the previous state by a *single step* of the transition relation. When refining implementations of distributed systems, it is natural to allow stuttering on the model side given that we wish to relate a detailed program execution to a more abstract model.

5.3 Events

Trace properties and relations among traces may depend on certain *events* to have happened to be meaningful. Figure 5.6 shows an augmented version of our initial examples from the Introduction that uses a dynamically allocated reference cell in favor of a global reference. We would like to express that the contents of the reference refine the model, however, a priori, we do not know which concrete memory location gets assigned during allocation. Depending on scheduling, concurrently running code may or may not allocate memory beforehand and hence affect the concrete location. To remedy this issue, we introduce *trace events*. Concretely, we introduce a notion of *labels* that may be used to annotate operations, such as reference allocation, *cf.* the label s in Figure 5.6 together with a resource for reasoning about them. We can then express that the contents of the reference allocated at label s refines the model.

We introduce two predicates $\mathsf{AllocEvs}_s(\mathit{evs})$ and $\mathsf{eventSI}(\tau)$ satisfying

$$\mathsf{AllocEvs}_s(\mathit{evs}) * \mathsf{eventSI}(\tau) \vdash \mathit{TraceAllocs}_s(\tau) = \mathit{evs}$$

where $\mathit{TraceAllocs}_s$ is a function that maps an execution trace to the list of its allocation steps labeled by s. By keeping $\mathsf{eventSI}(\tau)$ in the trace interpretation, intuitively, $\mathsf{AllocEvs}_s(\mathit{evs})$ means evs is the list of allocations labeled by s that have taken place so far during the execution.

To establish a refinement relation for the example we use the following invariant:

$$I_{\mathsf{incr}} \triangleq \boxed{\begin{array}{l}(\mathsf{AllocEvs}_s([]) * \mathsf{IncrLoc}_\bullet(\mathsf{None}) * \mathsf{Model}_\circ(0)) \lor \\ (\exists \ell, n.\, \mathsf{AllocEvs}_s([\mathit{allocated}(\ell)]) * \mathsf{IncrLoc}_\bullet(\mathsf{Some}\ \ell) * \ell \mapsto n * \mathsf{Model}_\circ(n))\end{array}}^{\mathcal{N}_{incr}}$$

The invariant states that either there has been no allocation with label s, in which case the model has value 0, or there has been exactly one location ℓ allocated with label s, and its value corresponds to the state of the model. The predicates $\mathsf{IncrLoc}_\bullet$ and $\mathsf{IncrLoc}_\circ$ are user-defined ghost state that is used for tracking in the proof whether the location has been allocated or not: by owning $\mathsf{IncrLoc}_\circ(\mathsf{None})$ one can deduce that the location has not been allocated yet. The two predicates satisfy the following rules:

INCR-LOC-CREATE
$$\vdash \Rrightarrow \mathsf{IncrLoc}_\bullet(\mathsf{None}) * \mathsf{IncrLoc}_\circ(\mathsf{None})$$

INCR-LOC-AGREE
$$\mathsf{IncrLoc}_\bullet(a) * \mathsf{IncrLoc}_\circ(b) \vdash a = b$$

INCR-LOC-UPDATE
$$\mathsf{IncrLoc}_\bullet(\mathsf{None}) * \mathsf{IncrLoc}_\circ(\mathsf{None}) \vdash \Rrightarrow \mathsf{IncrLoc}_\bullet(\mathsf{Some}\ \ell) * \mathsf{IncrLoc}_\circ(\mathsf{Some}\ \ell)$$

Using these rules together with the proof rules of the Trillium instantiation it is straightforward to prove a specification

$$\{I_{\mathsf{incr}} * \mathsf{IncrLoc}_\circ(\mathsf{None})\}\ \texttt{inc}'\ \{\mathsf{False}\}^{\mathcal{M}_{\mathsf{inc}}}$$

where `inc'` is the program from Figure 5.6. As before, the postcondition can be False as the program loops indefinitely.

5.4 Refinement of TLA⁺ specifications

In the following, we discuss how to establish a refinement relation between implementations of two classical distributed algorithms, two-phase commit and single-decree Paxos (SDP), and their TLA⁺ models. As simple corollaries of the refinement relation and our adequacy theorems we establish using the *same* modular specification

1. that clients are *safe*, *i.e.*, they do not crash,
2. a formal proof that the AnerisLang implementation correctly implements the protocol specification, and
3. correctness of the implementation by leveraging already-established correctness properties of the models.

Both the TLA⁺ specification of transaction commit[3] and the TLA⁺ specification of single-decree Paxos[4] can also be found in the official TLA⁺-examples repository on GitHub. In our formalization, we have manually translated the TLA⁺ system specifications into transition systems in Coq and proved their correctness properties. We argue that the translations are faithful and straightforward, however, it is not crucial for soundness that the translation is correct as the translated models have been proven correct independently in Coq.[5]

[3] https://github.com/tlaplus/Examples/blob/master/specifications/transaction_commit
[4] https://github.com/tlaplus/Examples/tree/master/specifications/Paxos
[5] A user that does not aim to be foundational could, however, reuse the existing TLA⁺ proofs.

5.4.1 Two-phase commit

The two-phase commit protocol [Gra78] is one of the best-known practical algorithms for solving the *transaction-commit problem* where a collection of processes, called *resource managers*, have to agree on whether a transaction ought to be *committed* or *aborted*. A protocol that solves the problem has to ensure *agreement*, *i.e.*, no two processes may decide differently.

In this development, we consider a TLA⁺ model of transaction commit and show that a distributed implementation of the two-phase commit protocol refines it. As a corollary of the refinement and the agreement theorem for the model we show that the implementation also ensures agreement.

Model. The transaction commit model is summarized in Figure 5.7. The model is parameterized by a set RMs of resource managers that are each in either an initial Working state, a preparation state Prepared, or in a final state Committed or Aborted. The full state of the model is a finite mapping from resource managers to one of these states. The transition relation allows resource managers to transition freely from the Working to the Prepared state (TC-Prepare). A resource manager may transition from the Prepared to the Committed state if all resource managers are either in the Prepared or the Committed state (TC-Commit), and from the Working or Prepared state to the Aborted state if no resource manager is in the Committed state (TC-Abort).

$$\begin{aligned}
\mathrm{RMStates} &\triangleq \{\mathsf{Working}, \mathsf{Prepared}, \mathsf{Committed}, \mathsf{Aborted}\} \\
\delta &\in \mathrm{RMs} \xrightarrow{\text{fin.}} \mathrm{RMStates} \\
\mathrm{CanCommit}(\delta) &\triangleq \forall r \in \mathrm{RMs}.\ \delta(r) = \mathsf{Prepared} \vee \delta(r) = \mathsf{Committed} \\
\mathrm{NotCommitted}(\delta) &\triangleq \forall r \in \mathrm{RMs}.\ \delta(r) \neq \mathsf{Committed}
\end{aligned}$$

TC-Prepare
$$\frac{\delta(r) = \mathsf{Working}}{\delta \rightharpoonup_{\mathsf{TC}} \delta[r \mapsto \mathsf{Prepared}]}$$

TC-Commit
$$\frac{\delta(r) = \mathsf{Prepared} \qquad \mathrm{CanCommit}(\delta)}{\delta \rightharpoonup_{\mathsf{TC}} \delta[r \mapsto \mathsf{Committed}]}$$

TC-Abort
$$\frac{\delta(r) = \mathsf{Working} \vee \delta(r) = \mathsf{Prepared} \qquad \mathrm{NotCommitted}(\delta)}{\delta \rightharpoonup_{\mathsf{TC}} \delta[r \mapsto \mathsf{Aborted}]}$$

Figure 5.7: TLA⁺ specification of the transaction-commit (TC) problem.

By induction on the transition relation one can easily show that the model satisfies the agreement property when starting from an initial state where all resource managers are in the Working state.

Theorem 5.4.1 (Agreement of TC). *Let* $\delta_{\mathsf{Init}}(r) \triangleq \mathsf{Working}$. *If* $\delta_{\mathsf{Init}} \rightharpoonup^*_{\mathsf{TC}} \delta$ *then for all* $r_1, r_2 \in \mathrm{RMs}$ *it is* not *the case that* $\delta(r_1) = \mathsf{Committed}$ *and* $\delta(r_2) = \mathsf{Aborted}$.

Implementation. The two-phase commit protocol relies on a *transaction manager* to orchestrate the agreement process; the transaction manager may either be a distinguished resource manager or a separate process. Listing 5.1 and Listing 5.2 show implementations in AnerisLang of the transaction manager and resource manager roles, respectively. The transaction manager uses a simple library functionality for removing duplicate messages; the functionality is initialized with the `nodup_init` invocation that returns a wrapped `receivefrom` primitive that removes duplicate messages but is otherwise not important.

Listing 5.1: Transaction manager.

```
let wait_receive skt test =
  let rec loop () =
    let msg = receivefrom skt in
    if test msg then msg else loop ()
  in loop ()

let resource_manager RM TM =
  let skt = socket () in
  socketbind skt RM;
  let (m, _) = receivefrom skt in
  if m = "ABORT"
  then sendto skt "ABORTED" TM
  else
    let local_abort = coin_flip () in
    if local_abort
    then sendto skt "ABORTED" TM
    else
      sendto skt "PREPARED" TM;
      let (decision, _) =
        wait_receive skt
          (fun (_, m) => m = "COMMIT" ||
                         m = "ABORT") in
      if decision = "COMMIT" then
        sendto skt "COMMITTED" TM
      else
        sendto skt "ABORTED" TM
```

Listing 5.2: Resource manager.

```
let broadcast skt X msg =
  Set.iter (fun x -> sendto skt msg x) X

let recv_resps recv skt RMs =
  let rec loop prepared =
    Set.equal prepared RMs ||
      let (msg, sndr) = recv skt in
      msg = "PREPARED" &&
        loop (Set.add sndr prepared) in
  loop (Set.empty ()) in

let transaction_manager TM RMs =
  let skt = socket () in
  socketbind skt TM;
  let recv = nodup_init () in
  broadcast skt RMs "PREPARE";
  let ready = recv_resps recv skt RMs in
  if ready then
    broadcast skt RMs "COMMIT";
    receive_all skt recv RMs;
    "COMMITTED"
  else
    broadcast skt RMs "ABORT";
    "ABORTED"
```

The transaction manager implementation starts by allocating a socket and binding it to the socket address `TM` given as argument. It continues by sending a `"PREPARE"` message to all the resource manager socket addresses given in `RMs`, asking the resource managers to transition to the preparation phase. If all the resource managers respond with `"PREPARED"`—signifying that they are all ready to commit—the transaction manager continues by sending a `"COMMIT"` message, telling the resource managers the decision is to commit, after which it awaits their responses and returns. If a single resource manager responds with `"ABORTED"`, the transaction manager stops receiving responses, relays the information, and returns.

The resource manager implementation starts by allocating a socket and binding it to the socket address `RM` given as argument. It continues by listening for an initial request from the transaction manager; in case another resource manager already aborted and this information arrived prior to the initial `"PREPARE"` request, the resource manager aborts. If asked to prepare, the resource manager makes a local decision—here with a nondeterministic coin flip–and sends the decision to the transaction manager. If the resource manager decides to abort, it immediately returns; otherwise it awaits the final decision from the transaction manager, confirms the transition, and returns.

Refinement. To show that the two-phase commit implementation refines the transaction-commit model we instantiate the Aneris logic with the model; this gives us a handle to the current model state δ that we can manipulate through the separation logic resource $\mathsf{Model}_\circ(\delta)$. The key proof strategy is to keep this resource in an invariant that ties together the model state and the physical with enough information such that the continued simulation is strong enough for proving our final correctness theorem (Corollary 5.4.2). In this development, we will tie sending a message (such as `"COMMITTED"`) from resource manager r to the corresponding transition in the model (such as TC-Commit). Additionally, the invariant will have to keep sufficient ghost resources and information for us to establish the conditions ($\mathrm{CanCommit}(\delta)$ and $\mathrm{NotCommitted}(\delta)$) for progressing the model.

To state a sufficient invariant for the two-phase commit refinement we will make use of two resource algebras: a variation of the *oneshot* algebra [Jun+18b] with *discardable fractions* [VB21] as well as a monotone ghost map algebra.

The oneshot algebra with discardable fractions allows us to define resources $\mathsf{pending}(q)$, $\mathsf{discarded}$, and $\mathsf{shot}(a)$ governed by the rules listed below.

$$
\begin{aligned}
&\mathsf{pending}(q) \vdash \Rightarrow\!\!\!|\, \mathsf{discarded} \\
&\mathsf{shot}(a) * \mathsf{pending}(q) \vdash \mathsf{False} \\
&\mathsf{shot}(a) * \mathsf{discarded} \vdash \mathsf{False} \\
&\mathsf{shot}(a) * \mathsf{shot}(b) \vdash a = b \\
&\mathsf{pending}(1) \vdash \Rightarrow\!\!\!|\, \mathsf{shot}(a) \\
&\mathsf{pending}(p) * \mathsf{pending}(q) \dashv\vdash \mathsf{pending}(p+q) \\
&\mathsf{shot}(a) * \mathsf{shot}(a) \dashv\vdash \mathsf{shot}(a) \\
&\mathsf{discarded} * \mathsf{discarded} \dashv\vdash \mathsf{discarded}
\end{aligned}
$$

Intuitively, $\mathsf{pending}(q)$ corresponds to owning a q-sized share in making some decision; only by owning all shares a unique decision can be made as witnessed by owning $\mathsf{shot}(a)$. By discarding a share a party can ensure that no decision can ever be made. Notice how this construction can be used to model the two-phase commit protocol (in particular the condition $\mathrm{CanCommit}(\delta)$ and $\mathrm{NotCommitted}(\delta)$) by picking the decision value a to be the unit value: each party initially owns an evenly sized share of the decision and transfers this share to the transaction manager when preparing to commit. By receiving a share from all resource managers, the transaction manager can make the decision to commit. By discarding a share, a resource manager can ensure that no decision to commit will ever be made and safely abort.

Using the *monotone resource algebra* [TB21], we construct a logical points-to connective $r \overset{q}{\mapsto}_\bullet s$ that will track q-fractional ownership of the current model state s of resource manager r, but where s may only evolve monotonically according to the internal resource manager transition relation given by $\mathsf{Working} \rightsquigarrow \mathsf{Prepared} \rightsquigarrow \mathsf{Committed}$ and $\mathsf{Working}, \mathsf{Prepared} \rightsquigarrow \mathsf{Aborted}$. The construction is accompanied by a duplicable $r \mapsto_\circ s$ resource that gives a *lower-bound* on the current state of resource manager r as seen from the rules below.

$$
\begin{aligned}
&r \overset{q}{\mapsto}_\bullet s * r \mapsto_\circ s' \vdash s' \rightsquigarrow^* s \\
&r \overset{1}{\mapsto}_\bullet s * s \rightsquigarrow^* s' \vdash \Rightarrow\!\!\!|\, r \overset{1}{\mapsto}_\bullet s' * r \mapsto_\circ s' \\
&r \mapsto_\circ s * r \mapsto_\circ s \dashv\vdash r \mapsto_\circ s
\end{aligned}
$$

Equipped with the two constructions from above we can define the refinement invariant for the two-phase commit implementation:

$$I_{\mathsf{TPC}} \triangleq \exists \delta.\, \mathsf{Model}_\circ(\delta) * \mathop{\Large *}_{r \in \mathrm{RMs}} \exists R, T, s.\ \begin{array}{l} r \overset{\frac{1}{2}}{\mapsto}_\bullet s * \delta(r) = s * \mathrm{TokenCoh}(s) * \\ r \rightsquigarrow^{\phi_{\mathsf{RM}}}_{\square} (R, T) * \mathrm{ModelCoh}(r, s, T) \end{array}$$

The invariant owns the current model state δ and for each resource manager r it owns half of the corresponding monotone points-to connective for some state s such that $\delta(r) = s$; the resource manager itself will own the remaining half. This ensures that the resource manager itself knows exactly which state it is in and that the resource cannot be updated without updating the model as well. We moreover tie being in the model states Committed and Aborted to ownership of, respectively, the shot and discarded resources as given by $\mathrm{TokenCoh}(s)$ below.

$$\mathrm{TokenCoh}(s) \triangleq \begin{cases} \mathsf{shot} & \text{if } s = \mathsf{Committed} \\ \mathsf{discarded} & \text{if } s = \mathsf{Aborted} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

The remaining two clauses constitute the key component in connecting the model to the physical state; the persistent socket protocol $r \rightsquigarrow^{\phi_{\mathsf{RM}}}_{\square} (R, T)$ tracks the history T of sent messages from resource manager r and $\mathrm{ModelCoh}(r, s, T)$ requires that if the resource manager r is in state s then a corresponding message must have been sent to the transaction manager t and if a message corresponding to a state s' has been sent, the resource manager must be in *at least* that state:

$$\mathrm{MessageCoh}(r, s, T) \triangleq \begin{cases} (r, t, \text{"PREPARED"}) \in T & \text{if } s = \mathsf{Prepared} \\ (r, t, \text{"COMMITTED"}) \in T & \text{if } s = \mathsf{Committed} \\ (r, t, \text{"ABORTED"}) \in T & \text{if } s = \mathsf{Aborted} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathrm{ModelCoh}(r, s, T) \triangleq \mathrm{MessageCoh}(r, s, T) \wedge \forall s'.\, \mathrm{MessageCoh}(r, s', T) \to s' \rightsquigarrow^* s$$

The socket protocol ϕ_{TM} governing the communication with the transaction manager is defined below. It follows the intuitive description given earlier: when preparing to commit, the pending resource is transferred to the transaction manager, and in order to commit or abort, the resources shot and discarded must be transferred as well, respectively. Moreover, the resource manager has to prove that its model state has (at least) been progressed to the corresponding states. The socket protocol for ϕ_{RM} for the resource manager follows a similar pattern.

$$\begin{aligned} \phi_{\mathsf{TM}}(r, t, b) \triangleq\ & r \in \mathrm{RMs} * \\ & \Big(b = \text{"PREPARED"} * \mathsf{pending}(\tfrac{1}{|\mathrm{RMs}|+1}) * r \mapsto_\circ \mathsf{Prepared}\Big) \vee \\ & (b = \text{"COMMITTED"} * \mathsf{shot} * r \mapsto_\circ \mathsf{Committed}) \vee \\ & (b = \text{"ABORTED"} * \mathsf{discarded} * r \mapsto_\circ \mathsf{Aborted}) \\ \phi_{\mathsf{RM}}(r, t, b) \triangleq\ & b = \text{"PREPARE"} \vee \\ & \big(b = \text{"COMMIT"} * \mathsf{shot} * \mathop{\Large *}_{r \in \mathrm{RMs}} r \mapsto_\circ \mathsf{Prepared}\big) \vee \\ & (b = \text{"ABORT"} * \mathsf{discarded}) \end{aligned}$$

The transaction manager implementation can be given the specification below; notice how it does not rely on the refinement invariant but only on the socket protocols and resources as described.

$$\left\{\begin{array}{l}\mathsf{Fixed}(A) * t \in A * \mathsf{FreePort}(t) * t \rightsquigarrow (\emptyset, \emptyset) * \\ \mathsf{pending}(\frac{1}{|\mathrm{RMs}|+1}) * t \Rrightarrow \phi_{\mathsf{TM}} * \mathop{\Large\ast}_{r \in \mathrm{RMs}} r \Rrightarrow \phi_{\mathsf{RM}}\end{array}\right\}$$
$$\langle t;\ \texttt{transaction\_manager}\ t\ \mathrm{RMs}\rangle$$
$$\left\{v.\ \begin{array}{l}(v = \texttt{"COMMITTED"} * \mathop{\Large\ast}_{r \in \mathrm{RMs}} r \mapsto_\circ \mathsf{Committed}) \vee \\ (v = \texttt{"ABORTED"} * \exists r \in \mathrm{RMs}.\ r \mapsto_\circ \mathsf{Aborted})\end{array}\right\}$$

The specification for the resource manager as seen below, however, relies on the invariant as well as fractional ownership of the resource manager's model state.

$$\left\{\begin{array}{l}\mathsf{Fixed}(A) * r \in A * \mathsf{FreePort}(r) * \boxed{I_{\mathsf{TPC}}}^{\mathcal{N}_{\mathsf{TPC}}} * \\ r \Rrightarrow \phi_{\mathsf{RM}} * t \Rrightarrow \phi_{\mathsf{TM}} * \mathsf{pending}(\frac{1}{|\mathrm{RMs}|+1}) * r \overset{\frac{1}{2}}{\mapsto}_\bullet \mathsf{Working}\end{array}\right\}$$
$$\langle r;\ \texttt{resource\_manager}\ r\ t\rangle$$
$$\{\mathsf{True}\}$$

By combining the adequacy theorem (Theorem 5.2.1) with our model correctness theorem (Theorem 5.4.1) we obtain the following corollary that *only* talks about the execution of the two-phase commit implementation.

Corollary 5.4.2. *If* $(e;\emptyset) \to^* (T;\Sigma)$ *and* $m_{s_1}, m_{s_2} \in \mathit{messages}(\Sigma)$ *such that* m_{s_i} *is the physical message corresponding to state* s_i *then it is not the case that* $s_1 = \mathsf{Committed}$ *and* $s_2 = \mathsf{Aborted}$.

5.4.2 Single-decree Paxos

The Paxos algorithm is a consensus protocol and its single-decree version allows a set of distributed nodes to reach agreement on a single value by communicating through message-passing over an unreliable network. In the following we omit network-related Aneris resources and focus on the core parts of showing the refinement. We note that the approach is similar to the one taken for two-phase commit.

In SDP, each node in the system adopts one or more of the roles of either *proposer*, *acceptor*, or *learner*. A value is chosen when a learner learns that a *quorum* (*e.g.*, a majority) of acceptors have accepted a value proposed by some proposer. The algorithm works in two phases: in the first phase, a proposer tries to convince a quorum of acceptors to promise that they will later accept its value. If it succeeds, it continues to the second phase where it asks the acceptors to fulfill their promise and accept its value. To satisfy the requirements of consensus, each attempt to decide a value is distinguished with a unique totally-ordered round number or *ballot*. Each acceptor stores its current ballot and the last value it might have accepted, if any. Acceptors will only give a promise to proposers with a ballot greater than their current one, and in that case they switch to the proposer's ballot; proposers only propose values that ensure consistency, if chosen. By observing that a quorum of acceptors have accepted a value for the same ballot, learners will learn that a value has been chosen. We refer to Lamport [Lam01] for an elaborate textual description of the protocol.

Model. The TLA⁺ model of SDP is summarized in Figure 5.8. The model is parameterized over a set of acceptors, Acceptor, and a type of values, Value, among which values are chosen. The state of the model consists of a set of sent messages $\mathcal{S} \in \wp(\mathsf{PaxosMessage})$ and two maps $\mathcal{B} : \mathsf{Acceptor} \to \mathsf{Ballot}^?$ and $\mathcal{V} : \mathsf{Acceptor} \to \mathsf{Ballot} \times \mathsf{Value}^?$ that for each acceptor record the greatest ballot promise and the last accepted value together with its ballot, respectively. The message type is defined using a datatype-like notation as

$$\mathsf{PaxosMessage} \triangleq \mathsf{msg1a}(b) \mid \mathsf{msg1b}(a, b, o) \mid \mathsf{msg2a}(b, v) \mid \mathsf{msg2b}(a, b, v)$$

where $a \in \mathsf{Acceptor}$, $b \in \mathsf{Ballot}$, $v \in \mathsf{Value}$, and $o \in \mathsf{Ballot} \times \mathsf{Value}^?$.

$$\begin{aligned}
Q1bv(\mathcal{S}, Q, b) &\triangleq \{m \in \mathcal{S} \mid \exists a, v.\ m = \mathsf{msg1b}(a, b, \mathsf{Some}\ v) \wedge a \in Q\} \\
HavePromised(\mathcal{S}, Q, b) &\triangleq \forall a \in Q.\ \exists m \in \mathcal{S}, o.\ m = \mathsf{msg1b}(a, b, o) \\
IsMaxVote(\mathcal{S}, Q, b, v) &\triangleq \exists m_0 \in Q1bv(\mathcal{S}, Q, b), a_0, b_0.\ m = \mathsf{msg1b}(a_0, b, \mathsf{Some}\ b_0, v) \wedge \\
&\qquad \forall m' \in Q1bv(\mathcal{S}, Q, b). \\
&\qquad\quad \exists a', b', v'.\ m' = \mathsf{msg1b}(a', b, \mathsf{Some}\ b', v') \wedge b_0 \geq b' \\
ShowsSafeAt(\mathcal{S}, Q, b, v) &\triangleq HavePromised(\mathcal{S}, Q, b)\ \wedge (Q1bv(\mathcal{S}, Q, b) = \emptyset \vee \\
&\qquad\qquad IsMaxVote(\mathcal{S}, Q, b, v))
\end{aligned}$$

SDP-Phase1a

$$\frac{}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg1a}(b)\}, \mathcal{B}, \mathcal{V}}$$

SDP-Phase1b

$$\frac{\mathsf{msg1a}(b) \in \mathcal{S} \qquad b > \mathcal{B}(a) \qquad \mathcal{V}(a) = o}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg1b}(a, b, o)\}, \mathcal{B}[a \mapsto \mathsf{Some}\ b], \mathcal{V}}$$

SDP-Phase2a

$$\frac{\nexists v'.\ \mathsf{msg2a}(b, v') \in \mathcal{S} \qquad Quorum(Q) \qquad ShowsSafeAt(\mathcal{S}, Q, b, v)}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg2a}(b, v)\}, \mathcal{B}, \mathcal{V}}$$

SDP-Phase2b

$$\frac{\mathsf{msg2a}(b, v) \in \mathcal{S} \qquad b \geq \mathcal{B}(a)}{\mathcal{S}, \mathcal{B}, \mathcal{V} \rightharpoonup_{\mathsf{SDP}} \mathcal{S} \cup \{\mathsf{msg2b}(a, b, v)\}, \mathcal{B}[a \mapsto \mathsf{Some}\ b], \mathcal{V}[a \mapsto \mathsf{Some}\ b, v]}$$

Figure 5.8: TLA⁺ specification of single-decree Paxos (SDP).

The SDP-Phase1a transition adds a $\mathsf{msg1a}(b)$ message to the set of sent messages; this corresponds to the proposer asking the acceptors to *not* accept values for ballots smaller than b. If a $\mathsf{msg1a}(b)$ message has been sent and b is greater than acceptor a's current ballot $\mathcal{B}(a)$ then the SDP-Phase1b transition updates a's state and sends a $\mathsf{msg1b}(a, b, o)$ message where o is a's last accepted value, if any. This corresponds to an acceptor responding to a proposer's promise request.

The second phase is initiated using the SDP-Phase2a transition that corresponds to the proposer proposing a value v for ballot b by sending a $\mathsf{msg2a}(b, v)$ message. However, the transition can only be made if no value has previously been proposed for ballot b and if a quorum Q of acceptors exists such that the $ShowsSafeAt(\mathcal{S}, Q, b, v)$ predicate holds; this predicate is at the heart of the Paxos algorithm. Intuitively, the predicate holds if all acceptors in Q have promised not to accept values for any ballot less than b ($HavePromised(\mathcal{S}, Q, b)$) and *either*

none of the acceptors have accepted any value for all ballots less than b *or* v is the value of the largest ballot that acceptors from Q have accepted. Following the SDP-Phase2b transition, acceptor a may accept a proposal for value v and ballot b by sending a $\mathsf{msg2b}(a, b, v)$ message and updating its state to reflect this fact. A value v has been chosen when a quorum of acceptors have sent a $\mathsf{msg2b}(a, b, v)$ message for some ballot b:

$$Chosen(\mathcal{S}, v) \triangleq \exists b, Q.\ Quorum(Q) \land \forall a \in Q.\ \mathsf{msg2b}(a, b, v) \in \mathcal{S}$$

As follows from the theorem below, it is not possible for the protocol to choose two different values at the same time and hence SDP solves the consensus problem.

Theorem 5.4.3 (Consistency, SDP model). *Let* $\iota_{\mathsf{SDP}} = (\emptyset, \lambda\_.\ \mathsf{None}, \lambda\_.\ \mathsf{None})$. *If* $\iota_{\mathsf{SDP}} \rightharpoonup^*_{\mathsf{SDP}} (\mathcal{S}, \mathcal{B}, \mathcal{V})$ *and both* $Chosen(\mathcal{S}, v_1)$ *and* $Chosen(\mathcal{S}, v_2)$ *hold then* $v_1 = v_2$.

Implementation. Figure 5.9 shows an implementation in AnerisLang of the acceptor, proposer, and learner roles.

The acceptor implementation receives as input a set of learner socket addresses and an address to communicate on. It creates a fresh socket, binds it to the address, and allocates two local references to keep track of its current ballot and last accepted value. In a loop, it listens for the two different kinds of messages that it may receive from the proposers. Given a phase one message, it only considers the message if the ballot is greater than its current ballot in which case it responds with its last accepted value. Given a phase two message, it only considers the message if the ballot is greater than or equal to its current ballot in which case it accepts the value and broadcasts the fact to all the learners. The learner implementation simply waits for such a message for the same ballot from a majority of acceptors.

The proposer implementation receives as input a set of acceptor socket addresses, a bound socket, a ballot number and a value to (possibly) propose in the ballot. First phase is initiated by sending a message to all the acceptors and after receiving a response from a majority of the acceptors it continues to the second phase. In the second phase it picks the value of the maximum ballot among the responses; if no such value exist, it picks its own. The candidate is finally sent to all acceptors.

Note that this proposer implementation only proposes a value for a single ballot; typically, proposers will issue new ballots when learning that no decision has been reached due to messages being dropped or nodes crashing. Moreover, it is crucial that proposers do not issue proposals for the same ballot. In our Coq formalization, proposer p repeatedly issues new ballots of the form $k \cdot |\mathsf{Proposer}| + p$ for $k \in \mathbb{N}$ by keeping track of the last issued k in a local reference.

5.4.3 Consensus by refinement

To show that the SDP implementation refines the SDP model we instantiate the Aneris logic with the model; the key part of the proof is to keep the $\mathsf{Model}_\circ(\delta)$ resource in a global invariant that ties together the model state and the physical state with enough information to verify the implementation and for the refinement relation established through the adequacy theorem to be strong enough for proving our final correctness theorem (Corollary 5.4.4). Under this invariant we will *modularly* verify each Paxos role and each component in isolation.

To state the invariant, we use three kinds of resources corresponding to:

```
let acceptor learners addr =
  let skt = socket () in
  socketbind skt addr;
  let maxBal = ref None in
  let maxVal = ref None in
  let rec loop () =
    let (m, sndr) = receivefrom skt in
    match acceptor_deser m with
    | inl bal =>
      if !maxBal = None ||
         Option.get !maxBal < bal then
        maxBal := Some bal;
        sendto skt
          (proposer_ser (bal, !maxVal)) sndr
      else ()
    | inr (bal, v) =>
      if !maxBal = None ||
         Option.get !maxBal <= bal then
        maxBal := Some bal;
        maxVal := Some accept;
        broadcast skt learners
          (learner_ser (bal, v))
      else ()
      end; loop () in loop ()
```

```
let learner acceptors addr client =
  let skt = socket () in
  socketbind skt addr;
  let majority =
    Set.cardinal acceptors / 2 + 1 in
  let votes = ref (Map.empty ()) in
  let rec go () =
    let (m, sndr) = receivefrom skt in
    let (bal, v) = learner_deser m in
    let bal_votes =
      match Map.find_opt bal !votes with
      | Some vs => vs
      | None => Set.empty ()
      end in
    let bal_votes' = Set.add sndr bal_votes in
    if Set.cardinal bal_votes' = majority
    then (bal, v)
    else
      votes <- Map.add bal bal_votes' votes;
      go () in
  let result = go () in
  sendto skt (client_ser result) client
```

```
let recv_promises skt n bal0 =
  let promises = ref (Set.empty ()) in
  let senders = ref (Set.empty ()) in
  let rec loop () =
    if Set.cardinal !senders = n
    then !promises
    else
      let (m, sndr) = receivefrom skt in
      let (bal, mval) = proposer_deser m in
      if bal = bal0 then
        senders <- Set.add !senders sndr;
        promises <- Set.add !promises mval
      else ();
      loop ()
  in loop ()

let find_max_promise s =
  let max_promise acc promise =
    match promise, acc with
    | Some (b1, _), Some (b2, _) =>
      if b1 < b2 then acc else promise
    | None, Some _ => acc
    | _, _ => promise
    end
  in Set.fold max_promise s None

let proposer acceptors skt bal v =
  broadcast skt acceptors
    (acceptor_ser (inl bal));
  let majority =
    (Set.cardinal acceptors) / 2 + 1 in
  let promises =
    recv_promises skt majority bal in
  let max_promise =
    find_max_promise promises in
  let av = Option.value max_promise v in
  broadcast skt acceptors
    (acceptor_ser (inr (bal, av)))
```

Figure 5.9: Implementation of the acceptor, proposer, and learner roles of the Paxos protocol.

1. sets of messages with predicates $\mathsf{Msgs}_\bullet(\mathcal{S})$ and $\mathsf{Msgs}_\circ(m)$ such that

$$\mathsf{Msgs}_\bullet(\mathcal{S}) * \mathsf{Msgs}_\circ(m) \vdash m \in \mathcal{S}$$
$$\mathsf{Msgs}_\bullet(\mathcal{S}) \vdash \Rrightarrow (\mathsf{Msgs}_\bullet(\mathcal{S} \cup m) * \mathsf{Msgs}_\circ(m))$$

2. maps, *e.g.*, with predicates $\mathsf{MaxBal}_\bullet(\mathcal{B})$ and $\mathsf{MaxBal}_\circ(a, b)$ such that

$$\mathsf{MaxBal}_\bullet(\mathcal{B}) * \mathsf{MaxBal}_\circ(a, b) \vdash \mathcal{B}(a) = b$$
$$\mathsf{MaxBal}_\bullet(\mathcal{B}) * \mathsf{MaxBal}_\circ(a, b) \vdash \Rrightarrow \left(\mathsf{MaxBal}_\bullet(\mathcal{B}[a \mapsto b']) * \mathsf{MaxBal}_\circ(a, b')\right)$$

3. ballots with predicates $\mathsf{pending}(b)$ and $\mathsf{shot}(b, v)$ such that

$$\mathsf{pending}(b) * \mathsf{shot}(b, v) \vdash \mathsf{False}$$
$$\mathsf{pending}(b) * \mathsf{pending}(b) \vdash \mathsf{False}$$
$$\mathsf{pending}(b) \vdash \Rrightarrow \mathsf{shot}(b, v)$$
$$\mathsf{shot}(b, v_1) * \mathsf{shot}(b, v_2) \vdash v_1 = v_2$$

Equipped with these resource we can state the invariant:

$$I_{\mathsf{SDP}} \triangleq \boxed{\begin{array}{r}\exists \mathcal{S}, \mathcal{B}, \mathcal{V}.\ \mathsf{Model}_\circ(\mathcal{S}, \mathcal{B}, \mathcal{V}) * \mathsf{Msgs}_\bullet(\mathcal{S}) * \mathsf{MaxBal}_\bullet(\mathcal{B}) * \\ \mathsf{MaxVal}_\bullet(\mathcal{V}) * \mathit{BalCoh}(\mathcal{S}) * \mathit{MsgCoh}(\mathcal{S})\end{array}}$$

The first part of the invariant ties the current state of the model $(\mathcal{S}, \mathcal{B}, \mathcal{V})$ to its logical *authoritative* counterparts which means that by owning a *fragmental* part you own a piece of the model: *e.g.*, by owning $\mathsf{MaxBal}_\circ(a, b)$ you may open the invariant and conclude $\mathcal{B}(a) = b$ where $\mathcal{B}$ is the current map of ballots. Intuitively, we will give acceptor a exclusive ownership of the parts of the model that should correspond to its local state (through resources $\mathsf{MaxBal}_\circ(a, b)$ and $\mathsf{MaxVal}_\circ(a, o)$). Similarly, by owning $\mathsf{Msgs}_\circ(m)$ one may conclude that the message m has in fact been added to the set of messages in the model; this predicate we will transfer when sending physical messages corresponding to m.

In the last part of the invariant, the $\mathit{BalCoh}(\mathcal{S})$ predicate simply requires that if $\mathsf{msg2a}(b, v) \in \mathcal{S}$ then $\mathsf{shot}(b, v)$ holds. This implies that by owning $\mathsf{pending}(b)$ you are the only entity that may propose a value for ballot b and it may never change. The $\mathit{MsgCoh}(\mathcal{S})$ predicate ties the physical state of the program to the model using Aneris-specific predicates for tracking the state of the network in a similar was as for the two-phase commit verification. This, for instance, forces acceptors and proposers to also add to the model state $\mathcal{S}$ any message they send over the network. Hence, to verify a proposer or an acceptor that sends a message, the proof must open the invariant, use HT-TAKE-STEP to take a step in the model, and update the corresponding logical resources to close the invariant. Following this methodology, we give specifications of the following shape to the proposer and acceptor components:

$$\{I_{\mathsf{SDP}} * \mathsf{MaxBal}_\circ(a, \mathsf{None}) * \mathsf{MaxBal}_\circ(a, \mathsf{None}) * \ldots\}\ \langle ip;\ \texttt{acceptor}\ L\ a\rangle\ \{\mathsf{False}\}$$
$$\{I_{\mathsf{SDP}} * \mathsf{pending}(b) * \ldots\}\ \langle ip;\ \texttt{proposer}\ A\ skt\ b\ v\rangle\ \{\mathsf{True}\}$$

omitting Aneris-specific network connectives in the precondition; the postcondition for `acceptor` may be False as it does not terminate. We give a similar specification to the learner. Working in a modular program logic, we can compose these specifications to get a single Hoare

```
let client addr =
let skt = socket () in
socketbind skt addr;
let (m1, sndr1) = receivefrom skt in
let (_, v1) = client_deser m1 in
let (m2, _) = wait_receive skt
(fun (_, sndr2), sndr2 <> sndr1) in
let (_, v2) = client_deser m2 in
assert (v1 = v2); v1.
```

Figure 5.10: An example of a client implementation.

triple for a distributed system with both proposers, acceptors, and learners. By applying the adequacy theorem to this specification we get that the implementation indeed refines the TLA⁺ model of SDP.[6]

Consensus for the implementation. Given the specification has been established for the implementation, we can state and prove that the consistency property holds for all executions by transporting the consistency property of the model. Let

$$ChosenI(M, v) \triangleq \exists b, Q.\ Quorum(Q) \land \forall a \in Q.\ \exists m \in M.\ m \sim \mathsf{msg2b}(a, b, v)$$

where $\mathcal{M}$ is a set of physical messages and $m \sim s$ holds when m is the serialization of the message s. By picking a trace relation ξ_{SDP} that requires messages in the model to correspond to messages in the program state (as implied by $MsgCoh(\mathcal{S})$):

$$\xi_{\mathsf{SDP}}(\tau, \kappa) \triangleq \exists \mathcal{S}.\ last(\kappa) = (\mathcal{S}, \_, \_) \land messages(last(\tau)) \sim \mathcal{S} \land stuttering(\kappa)$$

we combine the adequacy theorem (Theorem 5.2.1) with our model correctness theorem (Theorem 5.4.3) to obtain the following corollary that *only* talks about the execution of the SDP implementation.

Corollary 5.4.4. *Let e be a distributed system obtained by composing n proposers, m acceptors, and k learners. For any T and Σ, if $(e; \emptyset) \to^* (T; \Sigma)$ and both $ChosenI(messages(\Sigma), v_1)$ and $ChosenI(messages(\Sigma), v_2)$ hold then $v_1 = v_2$.*

Functional correctness of clients. Corollary 5.4.4 is a meta-logic theorem (*e.g.*, in Coq) that only talks about the program execution and it follows as a consequence of the adequacy theorem and the model theorem. However, it is not only in the meta-logic that we can exploit properties of the model to prove properties about programs as the model is also embedded as a *resource* in the logic.

Figure 5.10 shows a client application that receives a message from two different learners and asserts that the two values are equal; if the two values do not agree, the program crashes. We can prove a specification for the client of the shape $\{I_{\mathsf{SDP}} * \ldots\}\ \langle ip;\ \texttt{client}\ a\rangle\ \{\ldots\}$. From the adequacy theorem it follows that the program is safe, *i.e.*, it does not crash, which means the asserted statement must always hold. In the proof of this specification, the client will receive ghost resources from the learners conveying that v_1 and v_2 have been chosen (*i.e.*, that

[6] The full Coq proof amounts to about 1100 lines of proof scripts.

a quorum of acceptors have accepted v_i). By opening the invariant I_{SDP} and hence obtaining the model resource $\mathsf{Model}_\circ(\mathcal{S}, \mathcal{B}, \mathcal{V})$, we can combine this knowledge with Theorem 5.4.3—a property exclusively of the model—and conclude that $v_1 = v_2$. Naturally, we may still compose a distributed system containing the client together with proposers, acceptors, and learner nodes and derive a specification for the full system. This single specification for the full distributed system entails *both* the refinement of the TLA$^+$ model and the safety of the programs running on all nodes.

5.5 Specification and verification of CRDTs using refinement

According to the CAP theorem [Bre00] no distributed system can, simultaneously, satisfy all the three desired properties of distributed systems: consistency (all replicas always agree), availability (responsiveness), and partition tolerance (can function even if some nodes have crashed/disconnected). Hence, different distributed systems choose to sacrifice (parts of) one of these three properties. Conflict-free replicated data types [Sha+11] weaken the consistency of the system to so-called *eventual consistency* [Vog09], which, loosely speaking, states that all replicas are guaranteed to be consistent once they have received the same set of updates from other replicas.

In this section we use Trillium to reason about a CRDT called G-Counter (a grow-only replicated counter). Despite their simplicity, G-Counters illustrate subtle and salient aspects of specification and verification of eventual consistency of CRDTs when *(a)* it is done fully formally *(b)* for an actual implementation including replicas' intercommunication, *(c)* along with specifying and proving node-local functional correctness within the same formal setting (in Aneris and Coq).[7]

Implementation. The implementation of the G-Counter in Figure 5.11 consists of the following:

- The `install` method is used to initialize instances of G-Counter on different replicas and returns two methods: `query`, to read the value, and `incr`, to increment it.
- The `broadcast` loop, forked by `install`, repeatedly sends the local state to other replicas.
- The `apply` loop, also forked by `install`, repeatedly updates the local state based on the states of the other replicas it receives over the network.

In the code of the `install` function the `s` in `ref<s>` is the so-called *label* of allocation used to identify so-called allocation events. See Section 5.3 for more details on allocation events and how they allow us to state and prove properties regarding memory locations even before they are allocated. For instance, here we enforce that the state of a replica in the model should be zero before the local state of the replica is physically allocated and that it must match the state stored physically on the replica after its allocation.

The state of a G-Counter replicas is a vector (an array), one element for each replica (including themselves), with the j^{th} element of the vector tracking the number of increments performed on the j^{th} replica. Note how the `incr` method on the i^{th} replica increments the i^{th} element of the vector, and the `query` function returns the sum of the vector. The `apply` method

[7] The full Coq proof amounts to about 2000 lines of proof scripts.

updates the local state by taking, for each replica, the maximum value of its current state and the value it has received from the network—the `vect_join_max` function computes point-wise maximum of two vectors.

```
let install addrlst i s =
  let n = List.length addrlst in
  let m = ref<s> (vect_mk n 0) in
  let sh = socket () in
  socketbind sh (List.nth addrlst i);
  fork (apply m sh);
  fork (broadcast m sh addrlst i);
  (query m, incr m i)


let rec incr m i () =
  let t = !m in
  if cas m t (vect_inc t i)
  then ()
  else incr m i ()

let query m () = vect_sum !m
```

```
let rec perform_merge m m2 =
  let t = !m in
  if cas m t (vect_join_max t m2) then ()
  else perform_merge m m2

let apply m sh =
  let rec loop () =
    let (b, _) = receivefrom sh in
    let m2 = vect_deserialize b in
    perform_merge m m2; loop ()
  in loop ()

let broadcast m sh nodes i =
  let rec loop () =
    let msg = vect_serialize !m in
    send_to_all sh msg nodes i; loop ()
  in loop ()
```

Figure 5.11: Implementation of a global counter.

Note the inherent node-local concurrency in this implementation; the `broadcast` and `apply` methods running concurrently alongside the client code which invokes increment and query methods. Hence, in this example we use advanced features of Aneris, *e.g.*, support for node-local concurrency. The idea of eventual consistency for G-Counters, which we will make formal later, is that if at some point no increment operation takes place on any replica, assuming some fairness properties about the network and scheduling, the states of all replicas will converge.

Functional correctness. Unsurprisingly, the *node-local guarantees* that clients can get for querying and incrementing are much weaker than for two-phase commit or Paxos. In the absence of coordination, the G-Counter merely enforces that each client always observes the effect of its calls to increment, but cannot know the exact value of the counter; we only know that it is monotonically increasing. Figure 5.12 shows the formal specifications for `incr` and `query` that we have proved and used to prove both safety and eventual consistency of CRDTs and their clients.[8] In the specs for both methods, the local state of the i^{th} replica is represented by an *abstract predicate* $\mathsf{gcounter}(i, k)$ where k is an under approximation of its current value (the sum of all elements of the vector).

$$\{\mathsf{gcounter}(i,k)\}\ \langle ip_i;\ \texttt{query}()\rangle\ \{m.\ k \le m * \mathsf{gcounter}(i,m)\} \qquad \textsc{QuerySpec}$$
$$\{\mathsf{gcounter}(i,k)\}\ \langle ip_i;\ \texttt{incr}()\rangle\ \{().\ \exists m.\ k < m * \mathsf{gcounter}(i,m)\} \qquad \textsc{IncrSpec}$$

Figure 5.12: G-Counter query and increment node-local specification.

[8] We omit the spec for `install` whose postcondition is straightforward (it returns a pair of methods (qr, ic) that satisfy the specifications QuerySpec and IncrSpec respectively), but whose precondition contains network-specific initialization conditions which are not relevant here, *e.g.* the address List.nth addrlst i being a pair (ip, p) where p is a free port.

5.5.1 Specifying and proving eventual consistency by refinement

In this section we show that G-Counter has the eventual consistency property. That is, any execution trace that is *network-fair*, defined below, and has a *stability point*, a point after which there is no increment, also has a *convergence point*, a point after which all replicas have the same local state.

The eventual consistency property is a property about the entire execution of a program. Hence, we need to be able to formally define traces that capture the entire execution of a program, or its corresponding trace in the model, which can also include infinite executions. For this reason, we introduce possibly-infinite traces. These traces, like finite traces we have seen before, are sequences of elements with the difference that they can be finite, or infinite, or even empty. We will use possibly-infinite traces in conjunction with finite traces, often as a pair. The idea is that this captures the program execution (or a model trace) up to a certain point, the finite trace being the past while the possibly-infinite trace is the future. We denote possibly-infinite traces with letters with a dot on top, *i.e.*, $\dot{\tau}$ and $\dot{\kappa}$ for possibly-infinite program and model traces, respectively. We will also use similar notation to finite traces for operations on possibly-infinite traces, *e.g.*, we write $c :: \dot{\tau}$ for extending the possibly-infinite trace $\dot{\tau}$ with configuration c.

The high-level idea of our proof is as follows. We consider a simple model for G-Counter. We define eventual consistency (stability point implies convergence point) for both possibly-infinite execution traces and possibly-infinite model traces. We show that any possibly-infinite trace of the model that is *model-fair*, defined below, is eventually consistent. The refinement relation that we obtain between possibly-infinite traces of the program and the model allows us to show

- That any possibly-infinite model trace corresponding to a possibly-infinite execution trace that satisfies network fairness properties is model-fair.
- That given a possibly-infinite execution trace and its corresponding model trace, if the model trace is eventually consistent, then so is the execution trace.

Putting all of the above together we can conclude that all program traces that satisfy network fairness properties are eventually consistent.

Model and model fairness. The state of the model we take for G-Counter with n replicas is simply a vector (of length n) of vectors (of length n), *i.e.* a square matrix. That is, for each replica we take a vector representing its local state. As expected, the initial state for our model is a square matrix where all elements are 0. We write ι_n^{GC} for this initial state where n is the number of G-Counter replicas. The model STS has two kinds of transitions (Figure 5.13) corresponding to the two state-changing operations on G-Counter: incrementing and merging a message received from the network. The GC-IncrStep transition updates the state δ such that the i^{th} element of the i^{th} vector, $\delta_{i,i}$ is incremented. This is precisely what happens in the program during the increment operation. The GC-ApplyStep transition, on the other hand, updates the i^{th} vector to be the result of merging (point-wise maximum) of the i^{th} vector with some vector $\vec{v}$ which is, point-wise, less than ($\sqsubseteq$) the vector for some other (j^{th}) replica. The idea is that the vector being merged corresponds to the state of j^{th} replica in the past—the j^{th} replica could have been incremented after its state was sent over the network and before getting merged.

$$\text{GC-IncrStep}\ \frac{}{\delta \to_{\mathsf{GC}} \delta[i \mapsto \delta_i[i \mapsto \delta_{i,i} + 1]]} \qquad \text{GC-ApplyStep}\ \frac{\vec{v} \sqsubseteq \delta_j}{\delta \to_{\mathsf{GC}} \delta[i \mapsto \delta_i \sqcup \vec{v}]}$$

Figure 5.13: Transition relation for the global counter model.

In order to support simpler and more compact writing we introduce the following notation. Given a finite trace t and a possibly-infinite trace $\dot{t}$, we write $Unroll_n(t, \dot{t})$ for the finite trace obtained by taking the first n elements of $\dot{t}$ (if there are n elements, otherwise as many as available) and appending them on t. (Recall the intuition we gave when we introduced possibly-infinite traces. The $Unroll_n$ function simply computes the execution trace n steps into the future.) As an example, we have $Unroll_1(t, a :: \dot{t}) = t :: a$. We write $Drop_n$ for dropping the first n elements of a possibly-infinite trace. For example, we have $Drop_1(a :: \dot{t}) = \dot{t}$.

We define fairness for a model trace and a possibly-infinite model trace as follows:

$$ModelFair(\kappa, \dot{\kappa}) \triangleq \forall i, j, k.\ \exists k'.\ last(Unroll_k(\kappa, \dot{\kappa}))_i \sqsubseteq last(Unroll_{k'}(\kappa, \dot{\kappa}))_j$$

Note that here we write v_i for the i^{th} row of the matrix v and $v_{i,j}$ for the j^{th} component of the i^{th} vector. This definition simply states that for any replicas i and j, for any number of steps k, there is a k' such that the vector for replica j at step k' is greater than or equal to the vector for replica i at step k. In other words, it is *always* the case that the current state of i^{th} replica is *eventually* merged into the j^{th} replica. We define eventual consistency, stability point, and convergence point as follows:

$$\begin{aligned}
ModelStab_{\vec{v}}(\kappa, \dot{\kappa}) &\triangleq \exists k.\ \forall k'.\ \forall i.\ last(Unroll_{k+k'}(\kappa, \dot{\kappa}))_{i,i} = \vec{v}_i \\
ModelConv_{\vec{v}}(\kappa, \dot{\kappa}) &\triangleq \exists k.\ \forall k'.\ \forall i.\ last(Unroll_{k+k'}(\kappa, \dot{\kappa}))_i = \vec{v} \\
ModelEvCons(\kappa, \dot{\kappa}) &\triangleq \forall \vec{v}.\ ModelStab_{\vec{v}}(\kappa, \dot{\kappa}) \Rightarrow ModelConv_{\vec{v}}(\kappa, \dot{\kappa})
\end{aligned}$$

The predicate $ModelStab_{\vec{v}}$ states that there exists a point k after which the diagonal of the state is exactly $\vec{v}$. Similarly, the predicate $ModelConv_{\vec{v}}$ states that there is a point k after which all local states are exactly $\vec{v}$.

Theorem 5.5.1 (Model Eventual Consistency)**.** *For all κ and $\dot{\kappa}$, if $ModelFair(\kappa, \dot{\kappa})$ then $ModelEvCons(\kappa, \dot{\kappa})$.*

Closed system, network fairness, and eventual consistency. For the rest of this section we assume that we have a closed system consisting of a number of nodes where each node runs a client of G-Counters after initializing a local instance. That is, each node in the system runs a program

```
let (qr, ic) = install addrlst i i in client_i qr ic
```

where `addrlist` is the list of socket addresses of all replicas and $\texttt{client}_i$ is some arbitrary code that runs on the i^{th} node as a client of the G-Counter—note how the label of the allocated location for the state of the i^{th} node is i. For clients we only assume that they satisfy a Hoare triple where the precondition requires the query and the increment functions satisfy their

specs given in Figure 5.12. We write c_n^{GC} for the initial configuration of the closed system of n replicas of G-Counter.

We now proceed to show that any closed system of G-Counters has the eventual consistency property. As expected from earlier high-level informal proofs [Sha+11], this is based on a fairness assumption on the network. Since we are considering a concrete implementation here, we additionally assume some liveness properties of the implementation (including fairness of schedulers on different nodes in the system), *e.g.*, that a message is eventually received if the network has not dropped it. The assumptions are as follows:

$$
\begin{aligned}
\mathit{NetFairSend}(\tau, \dot\tau) &\triangleq \forall i, j, n.\ \exists k.\ n \leq |\mathrm{TraceSends}_{i,j}(\mathit{Unroll}_k(\tau, \dot\tau))| \\
\mathit{NetFairRec}(\tau, \dot\tau) &\triangleq \forall i, n.\ \exists k.\ n \leq |\mathrm{TraceRecs}_{i}(\mathit{Unroll}_k(\tau, \dot\tau))| \\
\mathit{NetFairDel}(\tau, \dot\tau) &\triangleq \forall i, j, \mathit{sev}, k.\ \mathit{sev} \in \mathrm{TraceSends}_{i,j}(\mathit{Unroll}_k(\tau, \dot\tau)) \Rightarrow \\
&\quad \exists k', \mathit{sev}', \mathit{rev}.\ \mathrm{same\_or\_happens\_after}(\mathit{sev}', \mathit{sev}) \wedge \mathit{msg}(\mathit{sev}') = \mathit{msg}(\mathit{rev}) \wedge \\
&\quad\quad \mathit{sev}' \in \mathrm{TraceSends}_{i,j}(\mathit{Unroll}_{k+k'}(\tau, \dot\tau)) \wedge \\
&\quad\quad\quad \mathit{rev} \in \mathrm{TraceRecs}_{j}(\mathit{Unroll}_{k+k'}(\tau, \dot\tau)) \\
\mathit{NetFair}(\tau, \dot\tau) &\triangleq \mathit{NetFairSend}(\tau, \dot\tau) \wedge \mathit{NetFairRec}(\tau, \dot\tau) \wedge \\
&\mathit{NetFairDel}(\tau, \dot\tau)
\end{aligned}
$$

where $\mathrm{TraceSends}_{i,j}$ is the list of all send events from the i^{th} replica to the j^{th} replica and $\mathrm{TraceRecs}_i$ is the list of all receive events on i^{th} replica. The fairness criterion $\mathit{NetFairDel}$ simply says that for any send event sev from i^{th} replica to j^{th} replica, there is a send event sev' also sent from i^{th} replica to j^{th} replica that is received by the j^{th} node. Moreover, sev' is either the same as sev or it is sent after it. Note how this definition allows for messages to be dropped but essentially only requires that *always eventually* a message is delivered from any node to any other node.

We define eventual consistency, stability point, and convergence point for a closed system just as we defined them for the model; instead of the state of the model we refer to the values stored on the heap of each replica. However, this is not immediately expressible as at the beginning of execution the memory is not allocated. Hence, we follow an approach similar to the example in Section 5.3 using allocation events. The eventual consistency theorem that we prove about our implementation is as follows:

Theorem 5.5.2 (Eventual Consistency)**.** *Let $\dot\tau$ be a valid possibly-infinite trace starting from c_n^{GC} such that $\mathit{NetFair}(c_n^{GC}, \dot\tau)$ holds. Then there exist $k \in \mathbb{N}$, $\dot\kappa$, and n locations $\ell_1, \ldots, \ell_n$ such that after k steps of computation (of the entire distributed system) all the locations storing local states of all replicas are allocated and these are exactly locations $\ell_1, \ldots, \ell_n$ (ℓ_i storing the state of the i^{th} replica). Furthermore, we have*

$$\mathit{EvCons}_{\ell_1,\ldots,\ell_n}\left(\mathit{Unroll}_k\left(c_n^{GC}, \dot\tau\right), \mathit{Drop}_k(\dot\tau)\right)$$

where $\mathit{EvCons}_{\ell_1,\ldots,\ell_n}$ is the predicate stating that if there is a stability point (a point after which the j^{th} component of the state stored in ℓ_j does not change for any j) then there is a convergence point (a point after which the values stored in all locations is the same).

This theorem essentially says that there eventually is a point where all replicas have allocated their locations and that as of that point if there is a stability point, there must also be a convergence point.

Proof sketch of Theorem 5.5.2. The proof is divided into two parts. We first show that a certain relation GcMainRel holds between the program trace and the model trace; this is essentially a simple consequence of the refinement relation that we have established, and which we obtain by the adequacy theorem. Afterwards, we prove that GcMainRel together with the network fairness properties implies *EvCons*.

We begin by showing that there exists $k \in \mathbb{N}$, $\dot{\kappa}$, and n locations $\ell_1, \ldots, \ell_n$ as described in the theorem and that the following holds:

$$\forall k'.\ \mathsf{GcMainRel}\left((\ell_1, \ldots, \ell_n), \mathit{Unroll}_{k+k'}\left(c_n^{GC}, \dot{\tau}\right), \mathit{Unroll}_{k+k'}\left(\iota_n^{GC}, \dot{\kappa}\right)\right)$$

This simply states that GcMainRel holds for ever after the k^{th} step. Intuitively, the relation GcMainRel holds when:

1. For any vector $\vec{v}$ sent from node i to node j, $\vec{v}$ is point-wise greater than or equal to *the vector stored on the heap* at node i at the time of all previous sent messages from node i to node j.
2. At all times, the vector stored on the heap of node i is point-wise greater than or equal to all vectors received by node i, *except possibly for the very last received message.*
3. At all times, the vectors stored on the heap and the model agree.

Note the subtlety of condition (1) as it is capturing precisely the interaction between the program scheduler and the network steps: the vector on heap at the time of a send operation might be larger than the vector being sent as increment operations can take place in between the reading and the sending operations. Also, for condition (2), since the program receives messages in a loop and then subsequently merges them it might be that the last vector received is not yet merged.

As for the second part of the proof, i.e., proving $\mathit{EvCons}_{\ell_1,\ldots,\ell_n}$, note how GcMainRel together with $\mathit{NetFair}$ implies $\mathit{ModelFair}$. Assume that $\vec{v}$ is the current state of i^{th} replica (both in the model and the heap as they agree). At some point in the future, there is a vector $\vec{v'}$ sent from i^{th} replica to j^{th} replica that is received, such that $\vec{v} \sqsubseteq \vec{v'}$. On the other hand, the j^{th} replica keeps receiving messages and once it gets a message after $\vec{v'}$, its state is guaranteed to be greater than or equal to $\vec{v'}$ and therefore also greater than or equal to $\vec{v}$. Moreover, since the vectors on the heap and the vectors in the model correspond at all times, the stability point and convergence point of the program and the model also correspond.

The invariant. The invariant that we use for establishing the refinement relation between G-Counters and their models is as follows:

$$\boxed{\exists \mathit{locs}, \delta.\ \mathsf{Model}_{\circ}(\delta) * \mathit{HeapRel}(\mathit{locs}, \delta) * \mathit{SentRel}(\mathit{locs}, \delta) * \mathit{RecRel}(\mathit{locs}, \delta)}$$

Here the predicate $\mathit{HeapRel}$ states that either the i^{th} replica has allocated its location and it stores exactly δ_i or δ_i is all 0's. See the example in Section 5.3 for a detailed example where allocation events are used to establish a property similar to this property. This invariant captures the relation GcMainRel above using allocation, send, and receive events. Here, locs is a list of optional locations (instead of a list of locations in GcMainRel). The $\mathit{SentRel}$ and RecRel predicates, respectively, use send and receive events to state the criteria (2) and (3) in the explanation above for GcMainRel. In order to maintain the invariant above, we use the rule HT-TAKE-STEP to update the state of the model whenever the `cas` operations in the `perform_merge` or `incr` methods succeed—note that the `cas` operation is an atomic operation.

```
let rec yes b n =
  if cas b 1 0 then n := !n-1;
  if !n > 0 then yes b n

let rec no b m =
  if cas b 0 1 then m := !m-1;
  if !m > 0 then no b m

let start k =
  let b = ref 0 in
  (yes b (ref k) || no b (ref k))
```

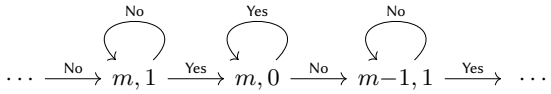


Figure 5.14: The Yes and No threads and the model $\mathcal{F}_{\text{YN}}$.

5.6 Fair termination of concurrent programs

We now consider an instantiation of Trillium to the HeapLang language, a concurrent higher-order language without network capabilities. We explain how instantiating Trillium with a suitable model allows proving fair termination of concurrent programs.

In a concurrent setting, the generally relevant notion of termination is *fair termination*, as most concurrent programs only terminate if the scheduler is fair. For example, the program presented in Figure 5.14, where two threads `yes` and `no` flip back and forth a shared Boolean `b`, does intuitively terminate. However, it does not terminate if, after some point, only one thread is ever scheduled; this should not happen under a reasonable scheduler. By definition, fair termination of a program means that all its *fair traces* are finite. An execution trace $(T_1, \sigma_1) \xrightarrow{\text{tid}_1} (T_2, \sigma_2) \xrightarrow{\text{tid}_2} \cdots$, whose transitions are labeled with the indices tid of the threads which take steps, is *fair* if it is finite, or if every reducible thread *eventually* takes a step.

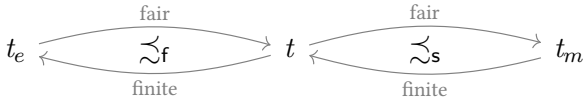
Fair termination is a liveness property, and hence we cannot prove it directly in a step-indexed logic such as Iris (as discussed by Spies et al. [Spi+21] and Tassarotti et al. [TJH17]). Our solution is to prove a *reduction* from the fair termination of an abstract model $\mathcal{F}$ to that of the program e:

$$\mathcal{F} \text{ is fairly terminating } \land \ e \text{ refines } \delta \in \mathcal{F} \quad \Longrightarrow \quad e \text{ is fairly terminating} \tag{5.1}$$

where the fact that e refines δ is proved in Trillium. To express fairness, we use an instantiation of Trillium where models are labeled transition systems. Thus the model $\mathcal{F}$ above should be a *fairness model*: an STS labeled with *roles* which act similarly to thread ids in the definition of *fair traces* of $\mathcal{F}$. Each state $\delta \in \mathcal{F}$ has a finite set of enabled roles. For the example above, we define in Figure 5.14 a model $\mathcal{F}_{\text{YN}}$ with two roles, Yes and No corresponding to the two threads, and whose states are pairs $(m, b) \in \mathbb{N} \times \mathbb{B}$ which represent respectively the value of `m` and of `b`. Intuitively, the states of $\mathcal{F}_{\text{YN}}$ summarize the states of the program; note that if, initially, $n = m = k$ and $b = 0$, then $n = m + b$. Loops in $\mathcal{F}_{\text{YN}}$ represent failed `cas` operations and rightward arrows successful ones. This model is fairly terminating: at each state, one of the two roles decreases the state ordered lexicographically.

We need the refinement relation between the program e and the state δ of the fairness model $\mathcal{F}$ to induce a relation $\precsim$ on their traces which entails that (5.1) holds, since fair termination is a trace property. The relation $\precsim$ is complicated, as it needs to maintain an evolving mapping between thread ids and roles and to ensure finiteness of stuttering. We define it as

the composition of two simpler relations $\precsim_{\mathsf{f}}$ and $\precsim_{\mathsf{s}}$ on traces so that $t_e \precsim t_m$ iff there exists a trace t of an intermediate labeled STS $\mathsf{Live}(\mathcal{F})$ such that the two following refinements hold:



(5.2)

The first relation $\precsim_{\mathsf{f}}$ is defined as history-sensitive refinement of a model $\mathsf{Live}(\mathcal{F})$ for a certain fixed relation ξ. The flexibility of Trillium's notion of refinement means that, by carefuly choosing the model and the relation ξ, it can be made to be a fairness-preserving termination-preserving refinement. A state of $\mathsf{Live}(\mathcal{F})$ is a triple (δ, F, T) where $\delta \in \mathcal{F}$, F associates a natural number $F(\rho)$ to each role ρ enabled in δ which is called its *fuel*, and T associates each role with a thread id of the program. The idea is that a stutter step of thread tid decreases the fuels of all the roles associated to it according to T, and that a tid-step in the program which corresponds to a ρ step in $\mathcal{F}$, with $F(\rho) = \mathsf{tid}$, can *increase* the fuel of ρ but must *decrease* the fuel of all the other roles $\rho' \in T^{-1}(\mathsf{tid})$. This decreasing-fuel discipline ensures that there exists a computable function which extracts a trace t_m in $\mathcal{F}$ from a trace t of $\mathsf{Live}(\mathcal{F})$ by ignoring stuttering steps. The relation $\precsim_{\mathsf{s}}$ is the graph of this function.

The correctness of this construction is represented by the gray arrows in Equation (5.2): for example, the arrow from t_e to t means that if $t_e \precsim_{\mathsf{f}} t$ and if t_e is fair, then t is fair. Thus if t_e is fair, then t_m is fair as well. If $\mathcal{F}$ is fairly terminating, we then get that t_m is finite, and therefore t_e is finite.

Coming back to our example, since $\mathcal{F}_{\text{YN}}$ is fairly terminating, it only remains to establish the refinement $t_e \precsim_{\mathsf{f}} t$, which we do by proving a weakest precondition for the program e in $Trillium$ instantiated with the model $\mathsf{Live}(\mathcal{F})$. We use an instantiation of Trillium where the weakest precondition is parameterized by the current thread id (allowing to match thread ids to roles). We make use of ghost resources corresponding to the states of $\mathsf{Live}(\mathcal{M})$: in particular, $\rho \mapsto_F f$ means that role ρ has fuel f, and $\mathsf{tid} \mapsto_T R$ means that the thread tid is associated to the set R of roles. Finally, $\mathsf{Model}_\circ(\delta)$ states that the current state of the underlying fairness model $\mathcal{F}$ is δ. Because fuel needs to decrease at each step, every program step of thread tid needs to be justified by owning the predicate $\mathsf{tid} \mapsto_T R$ (with $R \neq \emptyset$) and $\rho \mapsto_F f_\rho + 1$ for each $\rho \in R$; the $+1$ is consumed when tid takes a step. We can specialize the adequacy theorem of Trillium and use Equation (5.2) to get:

Theorem 5.6.1. *Given a program e, a finitely branching fairness model $\mathcal{F}$, a state $\delta_0 \in \mathcal{F}$, if*

$$\left\{ \mathsf{Model}_\circ(\delta_0) * 0 \mapsto_T R * \mathop{\Large *}_{\rho \in R} \rho \mapsto_F f_{init} \right\} \; e \; @ \; 0 \; \{0 \mapsto_T \emptyset\}^{\mathcal{F}}$$

holds for any f_{init}, R, and if $\mathcal{F}$ is fairly terminating, then e is fairly terminating. (0 above is the initial thread id).

We remark that the hypothesis that $\mathcal{F}$ is fairly terminating can be proved without quantifying over all fair traces: there is a simple criterion presented by Stefanesco [Ste21] based on a well-founded order which can be checked locally by considering transitions individually.

A technical inconvenience is that threads need to have at least one role to take a step, but must have none when they end. In turn, this means that the last step of tid must take a step to a state in the model where its roles are not enabled. For our example, this leads to adding

to the model $\mathcal{F}_{\text{YN}}$ depicted in Figure 5.14 two Booleans ye and ne to the states, where $ye = 0$ means Yes has finished, and $ne = 0$ means No has.

We use the theorem above to prove that the program in Figure 5.14 is fairly terminating by establishing the weakest precondition, with $R :- \{\text{Yes}, \text{No}\}$ and $\mathsf{f}_{init} :- 30$. The proof of the weakest precondition is fairly simple and follows the methodology explained for the minimal example in the introduction. We refer to Stefanesco [Ste21] for full definitions and more details on the methodology.

The approach presented here is similar in spirit to the one in the work of Tassarotti et al. [TJH17] but for reasoning about refinement of general concurrent programs with respect to abstract models. To the best of our knowledge, the expressiveness of the logics is roughly similar. The main difference is that Tassarotti et al. [TJH17] augments the Iris base logic with *linear* propositions, which requires modifying the definition of resource algebra to add a transition relation. We achieve similar results without heavy modifications, using that the authoritative state of the model is threaded through the weakest precondition, and by putting an exclusive structure on the set of roles owned by a thread, which prevents the weakening of $\mathsf{tid} \mapsto_T R_1 \cup R_2$ to $\mathsf{tid} \mapsto_T R_1$, a limited form of linearity.

5.7 Related work

We discuss some further related work not already discussed throughout the chapter.

Refinement-based verification of distributed systems. There is a lot of work on verification of high-level models of distributed systems, but here we focus on works that, as ours, aim at proving that concrete implementations refine abstract models. The most closely related works are IronFleet [Haw+15] and Igloo [Spr+20]. IronFleet uses the Dafny verifier to verify the implementation of a system and encode the relation to the STS being refined in preconditions and postconditions of programs. IronFleet does not support node-local concurrency and hence would not be applicable to our CRDT example. IronFleet uses a pen-and-paper argument for proving liveness of simple programs (programs that consist of a simple event loop which calls event handlers that are terminating), which does not scale to proving eventual consistency of our CRDT example. Igloo proves only safety properties about programs and not liveness properties like eventual consistency. Igloo starts with a high-level STS which is refined (possibly in multiple steps) to a more low-level STS for each node of the system. These STSs are annotated with IO operations which are used to generate IO specifications for network communications of the node in the style of Penninckx et al. [PJP15]. The program (each node) is then verified against this generated specification. Hence, the relationship between the implementation and the model considered in Igloo is a fixed relation, *i.e.* producing the same IO behavior. In contrast, our work allows an arbitrary (history-sensitive) refinement relation to be specified and established between the program and the model. In contrast to both IronFleet and Igloo our verification approach is foundational: the operational semantics of the distributed programming language, the abstract models, and the model of the program logic are all formally defined in Coq, and through adequacy theorems of the program logic, the end result of a verification is a formal theorem expressed only in terms of the operational semantics of the programming language and the model.

Refinement in Iris. There has been earlier work on proving refinements using Iris. Most of this work, however, has focused on *contextual* refinement, where a (higher-order concurrent imperative, but not distributed) program is related to another program [FKB18; KTB17; KSB17; Spi+21; Tim+18] or termination-preserving refinements among programs [Gäh+22; Spi+21; TJH17]. Perennial [Cha+19] defines correctness of a system using *concurrent recovery refinement*, requiring that the (possibly crashing) implementation and specification STS has the same external I/O. This notion of refinement is much coarser and does not allow you to prove, *e.g.*, fair termination. Tassarotti and Harper [TH19] relates concurrent probabilistic programs to abstract specifications denoting indexed valuations, exhibiting a probabilistic coupling when assuming that the implementation terminates.

Non-refinement-based verification of distributed systems. [Woo+16] verify the Raft consensus protocol [OO14] in the Verdi framework [Wil+15] for implementing and verifying distributed systems in Coq. In Verdi, the programmer provides a specification, implementation, and proof of a distributed system under an idealized network model in a high-level language. The application is automatically transformed into one that handles faults via verified system transformers: this makes vertical composition difficult for clients and the high-level language does not include features such as node-local concurrency. The Disel framework [SWT18] also allows users to implement distributed systems using a domain specific language and verify them using a Hoare-style program logic in Coq; the work includes a case study on two-phase commit. Disel struggles with node-local reasoning as the use of internal mutable state in nodes must be exposed in the high-level system protocol and state changes are tied to sending and receiving messages.

Paxos verification efforts. Paxos and its multiple variants have been considered by many verification efforts using, *e.g.*, automated theorem provers and model checkers [CLS16; JM05; Kel04; Kra+20; MSB17; Pad+17]. These efforts all consider abstract *models* or specifications in high-level domain-specific languages of Paxos(-like) protocols and not actual implementations in a realistic and expressive programming language.

García-Pérez et al. [Gar+18] devise composable specifications for a pseudo-code implementation of Single-Decree Paxos and semantics-preserving optimizations to the protocol on pen-and-paper but without a formal connection to their implementation in Scala; it would be interesting future work to implement and verify the same optimizations in our setting.

CRDTs. Zeller et al. [ZBP14] present an Isabelle/HOL framework for verifying state-based CRDTs, including verifying that a CRDT implementation refines its specification. Unlike in our work, CRDT implementations are defined at a high level of abstraction using state-transition systems. Zeller et al. [ZBP14] do not reason about inter-replica communication. Gomes et al. [Gom+17] present the first mechanized proof of eventual consistency of operation-based CRDTs but do not consider network communications as part of the program. Moreover, unlike our work on a state-based CRDT, Gomes et al. [Gom+17] do not consider functional correctness. Nair et al. [NPS20] present proof rules to reason about functional correctness of several state-based CRDTs that have richer safety guarantees than the CRDT we have studied because some operations of those CRDTs require coordination between replicas. However, they show safety and eventual consistency based on an abstract operational semantics which ignores inter-replica communication and node-local concurrency. Liang and Feng [LF21] propose an

approach to verify implementation of operation-based CRDTs where they show both functional correctness and strong eventual consistency within the same theoretical framework. They use a rely-guarantee-style program logic to reason about client programs, but do so at a higher "algorithmic" level of abstraction than our work, ignoring inter-replica communication. Furthermore, Liang and Feng [LF21] do not mechanize their work in a proof assistant. On the other hand, they consider many more examples of CRDTs than we do. Here, we have just focused on a single example, to illustrate how Trillium may be used to reason about CRDTs. In future work, it would be interesting to apply Trillium to other, more complex examples as well.

Fair termination of concurrent programs. We have already discussed the most closely related work on fair termination via termination-preserving refinement in Section 5.6. Liang and Feng [LF16; LF18] have also used refinement to show a wider range of liveness properties of concurrent programs, including programs with partial methods, but focusing on first-order logic and first-order programs. It would be interesting to investigate if Trillium could serve as a basis for generalizing the verification methods of Liang and Feng [LF16; LF18] to higher-order logic and higher-order programs.

5.8 Conclusion

We have introduced Trillium, a mechanized generic program logic that unifies Hoare-style reasoning with local reasoning about history-sensitive refinement relations among execution traces and traces of a model. We have shown how to use an instantiation of Trillium to a distributed higher-order concurrent imperative programming language to give modular proofs of correctness of concrete implementations of two-phase commit and single-decree Paxos by showing that they refine their abstract TLA$^+$ specifications. Moreover, we have shown how our notion of refinement can be used to reason about liveness properties such as strong eventual consistency of a concrete implementation of a CRDT and fair termination of concurrent programs.

6 *Mechanized Logical Relations for Termination-Insensitive Noninterference*

Abstract

We present an expressive information-flow control type system with recursive types, existential types, label polymorphism, and impredicative type polymorphism for a higher-order programming language with higher-order state. We give a novel semantic model of this type system and show that well-typed programs satisfy termination-insensitive noninterference. Our semantic approach supports compositional integration of syntactically well-typed and syntactically ill-typed—but semantically sound—components, which we demonstrate through several interesting examples. We define our model using logical relations on top of the Iris program logic framework; to capture termination-insensitivity, we develop a novel language-agnostic theory of Modal Weakest Preconditions. We formalize all of our theory and examples in the Coq proof assistant.

Systems for information-flow control put restrictions on how a program's outputs are related to its inputs. Such systems establish various notions of *noninterference* [GM82], conveying that observable aspects of the program's behavior is independent of its sensitive inputs. Information-flow control enforcement is often specified as a static type system (*e.g.*, [Aba+99; AM16b; HR98; LC15; Mye99; Sim03b]) or via an encoding into an existing type system (*e.g.*, [AR17; GTA19; LZ06; PS03; Rus15; RCH08; Vas+18]). Modern programming languages have rich type systems featuring, *e.g.*, higher types, reference types, and abstract types, which are all essential for modern software engineering practice and for implementing reusable software components. Naturally, modern practical information-flow secure languages have to meet the same demands, but as the complexity of the type system increases, so does the burden of proving the type system sound.

In this work, we prove soundness of an expressive information-flow control type system for a higher-order language with higher-order state. The type system is an extension of the fine-grained type system of Rajani and Garg [RG20] and the type system of Flow Caml [Sim03a] with recursive types, existential types, and impredicative type polymorphism (in addition to existing reference types and function types). The main high-level goal of our work is to prove that the type system satisfies *termination-insensitive noninterference* using a semantic model. Since such type soundness results for expressive type systems involve myriads of details (as exhibited by a 100 pp. chapter in a technical appendix [RG20]), we formalize our model in a proof assistant and use it to give a full mechanization of all our technical results.

Even with a very expressive type system, any static type system is necessarily overly conservative. This entails that there is a large body of programs that cannot be type-checked while

still being information-flow secure for reasons too subtle for the type system to verify. This includes, *e.g.*, low-level implementations of data structures that are optimized for efficiency and systems governed by security policies that rely on value-dependency or dynamic run-time information. Our *semantic* approach to establishing noninterference enables compositional integration of syntactically well-typed components with syntactically ill-typed but semantically sound components: only the syntactically ill-typed parts need to be carefully verified to show that the entire program enjoys the security property.

To meet our goals, we define a novel logical-relations model of our proposed type system. We define our logical-relations model in the Iris separation logic framework [Jun+16; Jun+18b; Jun+15; Kre+17]. We do this to

1. define and reason about our logical-relations model at a high level of abstraction,
2. side-step the well-known problem of *type-world circularity*[1] [Ahm04; AAV02; Bir+11] when defining logical-relations models of programming languages with higher-order state in the presence of impredicative polymorphism, and
3. to leverage the Coq formalization and the MoSeL framework [Kre+18] to fully mechanize all examples and technical results.

Challenges. Extending the earlier type systems is mostly straightforward: similarly to how ordinary functions in languages with effects may have latent effects, polymorphic functions may also have latent effects and thus they must be annotated with a label expressing a lower bound on these effects. So what is new and challenging about our semantic model? In summary, we address three major challenges:

1. combining unary and binary logical-relations models in the presence of impredicative polymorphism, and
2. constructing "logical" [DAB09] logical-relations models for termination-insensitive reasoning while
3. soundly allowing syntactically ill-typed but semantically secure programs to be composed with syntactically well-typed programs.

To construct a logical-relations model of a termination-insensitive information-flow control type system in the presence of state, it is necessary to combine both a unary and a binary model; when branching on high-labeled information, it is crucial that that the two branches, independently, do not modify low-labeled references, to avoid implicit leaks through the store. This is commonly know as the *confinement lemma* in proofs of noninterference.

When developing logical-relations models for languages with state in the presence of impredicative polymorphism, one needs to work with so-called step-indexed recursive Kripke worlds which are used to describe the semantics of the contents of the heap [Ahm04; Bir+11]. These step-indexed Kripke worlds imply that both the binary and the unary logical relations

[1] To ensure that heap updates are type-preserving, the model of mutable references types $\mathsf{ref}(\tau)$ needs to keep track of the semantics of τ; this is usally done using a store typing (a World) Θ mapping locations to types. The model of $\mathsf{ref}(\tau)$ then needs to refer to Θ to check whether a location maps to the appropriate type. This in turn means the model of all types has to take Θ as an argument, introducing what is known as the type-world circularity—this kind of recursive domain equation has no solution in the category of sets.

have to be step-indexed. Binary logical relations usually tie the logical steps of the recursive Kripke worlds to the physical steps taken by only one of the two programs in the relation. However, this causes a mismatch in the number of steps when we want to combine the individual unary logical relatedness of two programs to conclude that they are in the binary relation. To solve this problem, one novelty of our binary logical-relations model is that we count the steps taken by the programs on *both sides* of the relation. Rajani and Garg [RG20] circumvent this problem by using syntactic worlds which does not scale to impredicative polymorphism. This also means that their logical relations are defined over syntactically well-typed programs and hence cannot be used for reasoning about syntactically ill-typed but semantically well-typed programs as we do in this work.

The idea of using a more expressive logic to simplify the definition of logical-relations models is not novel and goes back to Plotkin and Abadi [PA93] who used second order logic for modeling System F and Dreyer et al. [DAB09] who used a logic with step-indexing to model recursive types. It has since been used for defining logical relations models for a variety of programming languages and features, *e.g.*, an ML-style language with concurrency [KTB17], a Haskell style ST monad [Tim+18], a concurrent ML-style language featuring continuations [TB19], and the Rust programming language [Jun+18a]. All these models are either unary logical-relations models used for proving type safety or binary logical-relations models for proving traditional contextual program refinement. Intuitively e contextually refinenes e' if whenever e terminates with some value v, then e' must also terminate with some value v' and v and v' should be suitably related. In symbols:

$$e \Downarrow v \Rightarrow e' \Downarrow v' \land v \approx v'.$$

This is crucially different from the idea of *termination-insensitive* noninterference where two programs are equivalent if, assuming that *both* e and e' terminate, then their resulting values should be suitably related:

$$e \Downarrow v \land e' \Downarrow v' \Rightarrow v \approx v'.$$

The termination-insensitive nature of the equivalence is the reason why the approaches taken heretofore on expressing logical-relations models in program logics cannot be extended to support reasoning about termination-insensitive noninterference. Moreover, these works do not consider logical-relations models that incorporate both a unary and a binary relation.

The core challenge here is to properly hide the details of step-indexing and recursive Kripke worlds. To this end, the base logic of Iris provides modalities to reason about step-indices and ghost resources (logical counterparts of recursive Kripke worlds). Yet, using these logical facilities directly, while hiding a lot of details, still requires us to think and work in terms of step-indices and explicit resource updates (manipulating ghost resources). Previous work [KTB17; TB19; TDB13], addressed this problem by defining the logical relation models using Iris' weakest precondition predicates, which themselves are defined using logical step-indexing and ghost resource modalities but which, importantly, come with high-level reasoning principles that hide those details. Iris' weakest precondition predicates were a good match for contextual refinement: we can express "if e terminates then so does e'" as a weakest precondition for e where the post condition states that e' terminates, *i.e.*, wp $e\,\{v.\ e' \Downarrow v \land v = v'\}$. As discussed above, this is crucially different from termination-insensitive noninterference: "if both programs terminate then …". This prevents us from using weakest preconditions to model our logical relations. One might be tempted to consider nested weakest preconditions:

wp e $\{v.$ wp e' $\{v'.$ $v = v'\}\}$. This formulation does indeed imply that if both programs terminate then their results are equal, however, this formulation is too strong and in particular makes it impossible to employ the kind of modular reasoning that is essential to proving the fundamental theorem of logical relations. Intuitively, this is because such a formulation requires us to reason about the execution of e' only *after* the full execution of e. Technically, this formulation does not admit the so-called binary bind rule (see Lemma 6.2.3).

In place of weakest preconditions we introduce and use a novel program logic construct that we call *Modal Weakest Preconditions* (MWP). Our Modal Weakest Precondition theory is language agnostic, parameterized by a modal operator, and general enough to allows us to define both a unary and a binary predicate for reasoning about computations using the same theory. Indeed, the generality is one of the key strengths of our theory. Different instantiations automatically inherit a set of basic structural proof rules that hold irrespective of the particular modality and programming language. For particular instantiations, one can then prove more specific proof rules, *e.g.*, for heap-manipulating operations and for how the instantiation interacts with other instantiations with different modal operators. We use three different instantiations for our logical-relations model and two more for concrete examples. The generality of our MWP theory allows us to define our binary logical relations model to be weak enough so as to allow us to reason modularly as discussed above. Yet, the interaction between different instantiations of MWP's (which is proven generally and not particularly for our programming language) allows us to strengthen this definition, in order to combine unary and binary logical relations (see Lemma 6.2.4) and to prove certain examples that require stronger reasoning principles (see Section 6.4).

Another challenge worth noting is the modeling of reference types. Intuitively, two values are related at the reference type $\mathsf{ref}(\tau)$ if they are both locations that invariantly store values that are related at type τ. Previous work used Iris invariants to formalize this idea. In our case, we can only use Iris invariants for our binary logical relation; for the unary logical relation we need to use a more refined approach as discussed in Section 6.2.

Contributions. In summary, we make the following contributions:

- We present the first logical-relations model of an information-flow type system with recursive types, existential types, and impredicative polymorphism for a language with higher-order state. To the best of our knowledge, this is also the first soundness proof of such an expressive information-flow type system irrespective of method.
- We present the first "logical" logical-relations model that incorporates both a unary and a binary relation and termination-insensitive reasoning.
- We introduce a new theory of *Modal Weakest Preconditions* (MWP) that allows us to construct novel logical-relations models for proving relational properties of programs that were out of reach of existing techniques.
- We propose a methodology that allows us to establish *termination-insensitive noninterference* of syntactically ill-typed but semantically secure programs while allowing these programs to be composed with syntactically well-typed programs and showcase multiple examples.
- We show that our logical-relations model allows us to prove "*free theorems*" for our information-flow control type system.

- We formalize all of the theory and examples on top of the Iris program logic framework in the Coq proof assistant using the MoSeL framework [Kre+18].

6.1 The λ_{sec} language

We present the syntax and operational semantics of λ_{sec}, the subject of our study: a higher-order functional call-by-value language with higher-order state which we equip with an information-flow control type system with recursive types, existential types, label polymorphism, and impredicative type polymorphism.

6.1.1 Syntax and semantics

The syntax of λ_{sec} is defined by the BNF below.

$$
\begin{aligned}
\odot &::= + \mid - \mid * \mid = \mid < \\
e \in \mathit{Expr} &::= x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid n \in \mathbb{N} \mid e \odot e \mid \lambda x.\, e \mid e\ e \mid \Lambda\, e \mid \Lambda\!\!\backslash\, e \mid e\ \_ \mid \\
&\quad \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (e, e) \mid \pi_i\ e \mid \mathsf{inj}_i\ e \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inj}_i\ \Rightarrow e_i\ \mathsf{end} \\
&\quad \mid \mathsf{ref}(e) \mid\ !\, e \mid e \leftarrow e \mid \mathsf{fold}\ \ e \mid \mathsf{unfold}\ \ e \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e \\
v \in \mathit{Val} &::= () \mid \mathsf{true} \mid \mathsf{false} \mid n \in \mathbb{N} \mid \ell \in \mathit{Loc} \mid \lambda x.\, e \mid \Lambda\, e \mid \Lambda\!\!\backslash\, e \mid \mathsf{fold}\ \ v \mid \mathsf{pack}\ v \\
&\quad \mid (v, v) \mid \mathsf{inj}_i\ v \\
\iota &::= \kappa \mid l \in \mathcal{L} \mid \iota \sqcup \iota \\
\tau \in \mathit{LType} &::= t^{\iota} \\
t \in \mathit{Type} &::= \alpha \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \tau \times \tau \mid \tau + \tau \mid \tau \xrightarrow{\iota} \tau \mid \forall_{\iota}\, \alpha.\, \tau \mid \forall_{\iota}\, \kappa.\, \tau \mid \exists \alpha.\, \tau \mid \mu\, \alpha.\, \tau \mid \mathsf{ref}(\tau)
\end{aligned}
$$

The term language is mostly standard but note that there are no types in terms; we write $\Lambda\ e$ for (unlabeled) type abstraction and $e\ \_$ for type application. Similarly, we write $\Lambda\!\!\backslash\ e$ for label abstraction and $e\ \_$ for label application. $\mathsf{fold}\ \ e$ and $\mathsf{unfold}\ \ e$ are the special term constructs for iso-recursive types. $\mathsf{ref}(e)$ allocates a new reference, $!\, e$ dereferences the location e evaluates to, and $e_1 \leftarrow e_2$ assigns the result of evaluating e_2 to the location that e_1 evaluates to. We introduce syntactic sugar for let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ defined as $(\lambda x.\, e_2)(e_1)$, and sequencing $e_1; e_2$ defined as $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$.

The state is modeled as a finite partial map $\sigma \in \mathit{Loc} \xrightarrow{\mathsf{fin}} \mathit{Val}$. Using evaluation contexts

$$
\begin{aligned}
K \in \mathit{Ectx} \quad ::= {} & - \mid K \odot e \mid v \odot K \mid \mathsf{if}\ K\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (K, e) \mid (v, K) \\
& \mid \pi_1\ K \mid \pi_2\ K \mid \mathsf{inj}_1\ K \mid \mathsf{inj}_2\ K \mid \mathsf{match}\ K\ \mathsf{with}\ \mathsf{inj}_i\ \Rightarrow e_i\ \mathsf{end} \mid K\ e \mid v\ K \\
& \mid \mathsf{ref}(K) \mid\ !\, K \mid K \leftarrow e \mid v \leftarrow K \mid \mathsf{fold}\ K \mid \mathsf{unfold}\ K \\
& \mid \mathsf{pack}\ K \mid \mathsf{unpack}\ K\ \mathsf{as}\ x\ \mathsf{in}\ e
\end{aligned}
$$

we define a call-by-value small-step operational semantics $(\sigma, e) \to (\sigma', e')$ in Figure 6.1.

The set of types of λ_{sec} is parameterized over an arbitrary bounded join-semilattice $\mathcal{L}$ with ordering $\sqsubseteq$. The lattice ordering $\sqsubseteq$ defines the security policy: if $\iota_1 \sqsubseteq \iota_2$ and $\iota_2 \not\sqsubseteq \iota_1$ then information with label ι_1 may influence information with label ι_2 but not the other way around. We write $\bot$ for the least element. Syntactically, a label ι is either a label variable κ, a label l drawn from the lattice $\mathcal{L}$, or the formal least upper bound (join) of two labels.

Types are syntactically either labeled or unlabeled; we use τ to range over labeled types and t to range over unlabeled types. A term of (labeled) type t^{ι} is a term of the (unlabeled) type

$$
\begin{aligned}
v \circledcirc v' &\overset{\text{pure}}{\rightsquigarrow} v'' && \text{if } v'' = v \circledcirc v' \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 &\overset{\text{pure}}{\rightsquigarrow} e_1 \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 &\overset{\text{pure}}{\rightsquigarrow} e_2 \\
\pi_i\,(v_1, v_2) &\overset{\text{pure}}{\rightsquigarrow} v_i && i \in \{1, 2\} \\
\mathsf{match\ inj}_i\ v\ \mathsf{with\ inj}_i\ \Rightarrow e\ \mathsf{end} &\overset{\text{pure}}{\rightsquigarrow} e[v/x] && i \in \{1, 2\} \\
(\lambda x.\, e)\, v &\overset{\text{pure}}{\rightsquigarrow} e[v/x] \\
(\Lambda\, e)\, \_ &\overset{\text{pure}}{\rightsquigarrow} e \\
(\Lambda\!\!\backslash\, e)\, \_ &\overset{\text{pure}}{\rightsquigarrow} e \\
\mathsf{unfold}\ (\mathsf{fold}\ v) &\overset{\text{pure}}{\rightsquigarrow} v \\
\mathsf{unpack}\,(\mathsf{pack}\, v)\ \mathsf{as}\ x\ \mathsf{in}\ e &\overset{\text{pure}}{\rightsquigarrow} e[v/x] \\
(\sigma, e) &\to_{\mathsf{h}} (\sigma, e') && \text{if } e \overset{\text{pure}}{\rightsquigarrow} e' \\
(\sigma, \mathsf{ref}(v)) &\to_{\mathsf{h}} (\sigma[\ell \mapsto v], \ell) && \text{if } \ell \notin \mathsf{dom}(\sigma) \\
(\sigma, !\,\ell) &\to_{\mathsf{h}} (\sigma, \sigma(\ell)) && \text{if } \ell \in \mathsf{dom}(\sigma) \\
(\sigma, \ell \leftarrow v) &\to_{\mathsf{h}} (\sigma[\ell \mapsto v], ()) && \text{if } \ell \in \mathsf{dom}(\sigma)
\end{aligned}
$$

$$
\frac{(\sigma, e) \to_{\mathsf{h}} (\sigma', e')}{(\sigma, K[e]) \to (\sigma', K[e'])}
$$

Figure 6.1: Small-step operational semantics $(\sigma, e) \to (\sigma', e')$ for λ_{sec}.

t labeled with the security label ι. Note that type abstraction, existential types, and recursive types abstract over unlabeled types.

The unlabeled types of λ_{sec} include basic types such as the unit type, Booleans, natural numbers, products, and sums. The function type $\tau \xrightarrow{\iota} \tau$ is annotated with a label ι. This label, which we refer to as a *latent effect label*, denotes a *lower bound* on the write effects of the function body. The type system will ensure that any reference that the function may write to has a label that is ι or higher according to the lattice ordering. This is necessary to prevent implicit information leaks through the store where programs have write effects that conditionally depend on sensitive information. Let ℓ be a reference with contents of type $\mathbb{N}^{\bot}$ and h a variable of type $\mathbb{B}^{\top}$ with $\top \not\sqsubseteq \bot$. When control flow depends on h, invoking a function like $f \triangleq \lambda\_.\, \ell \leftarrow 1$ implicitly leaks h through the store, *e.g.*,

$$\mathsf{if}\ h\ \mathsf{then}\ f\ ()\ \mathsf{else}\ ()$$

by subsequently observing whether the write effects happened or not. The label $\bot$ is a lower bound of the side-effects of f and it may not be invoked when control flow depends on h with label $\top$ as $\top \not\sqsubseteq \bot$. For the same reason, type-polymorphic types $\forall_\iota\, \alpha.\, \tau$ and label-polymorphic types $\forall\!\!\!/_\iota\, \kappa.\, \tau$ also include a latent effect label annotation ι. Finally, types also include existential types $\exists \alpha.\, \tau$, recursive types $\mu\, \alpha.\, \tau$, and the type $\mathsf{ref}\,(\tau)$ of memory locations storing values of type τ.

6.1.2 *Information-flow control type system*

The type system of λ_{sec} is mostly similar to the fine-grained type system of Rajani and Garg [RG20] and the type system of Flow Caml [Sim03a] but extended with recursive types, existential types, and impredicative polymorphic types. We write $\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau$ for the syntactic typing judgment which expresses that expression e has type τ under typing contexts Γ, Ξ, and Ψ. A typing context Γ maps free variables that may appear in e to their types. The type-level contexts Ξ and Ψ are sets of free type and label variables, respectively, that may appear in τ and Γ. The annotation pc is a label, often called the *program counter* label, denoting a lower bound on the write effects of e, *cf.*, how function types and the polymorphic types are annotated with a latent effect label.

The typing relation is shown in Figure 6.2 and 6.3 and we discuss some of the most important rules below. The syntactic label ordering relation $\Psi \vdash \iota_1 \sqsubseteq \iota_2$ is straightforward and shown below.

$$\textsc{F-refl}\ \frac{\mathrm{FV}(\iota) \subseteq \Psi}{\Psi \vdash \iota \sqsubseteq \iota} \qquad \textsc{F-trans}\ \frac{\Psi \vdash \iota_1 \sqsubseteq \iota_2 \qquad \Psi \vdash \iota_2 \sqsubseteq \iota_3}{\Psi \vdash \iota_1 \sqsubseteq \iota_3} \qquad \textsc{F-bottom}\ \frac{\mathrm{FV}(\iota) \subseteq \Psi}{\Psi \vdash \bot \sqsubseteq \iota} \qquad \textsc{F-label}\ \frac{l_1 \sqsubseteq l_2}{\Psi \vdash l_1 \sqsubseteq l_2}$$

$$\textsc{F-join}\ \frac{\Psi \vdash \iota_1 \sqsubseteq \iota_3 \qquad \Psi \vdash \iota_2 \sqsubseteq \iota_3}{\Psi \vdash \iota_1 \sqcup \iota_2 \sqsubseteq \iota_3}$$

The *protected-at* relation $\tau \searrow \iota$ is defined as $t^{\iota'} \searrow \iota \triangleq \iota \sqsubseteq \iota'$, meaning that the label of the type is at least as high as ι.

When applying a function expression e_1 of type $\left(\tau_1 \xrightarrow{\iota_e} \tau_2\right)^{\iota}$ to an argument e_2 of type τ_1 the rule for function application (T-app) requires that the program counter label pc is lower than the latent effect label ι_e to avoid implicit leaks through the store. In addition, the label ι of the function value must be below ι_e and the return type τ_2 must be protected at ι in order to prevent implicit leaks arising from the identity of the function that e_1 evaluates to: If not, then, for example, given ℓ is a reference of type $\mathsf{ref}(\mathbb{N}^{\bot})$ and h a variable of type $\mathbb{B}^{\top}$ both programs Equation (6.1) and Equation (6.2) would be typeable at $\mathbb{N}^{\bot}$ while both leaking h.

$$\mathsf{let}\ f = \mathsf{if}\ h\ \mathsf{then}\ \lambda\_.\ 1\ \mathsf{else}\ \lambda\_.\ 0\ \mathsf{in}\ f\ () \tag{6.1}$$

$$\mathsf{let}\ f = \mathsf{if}\ h\ \mathsf{then}\ \lambda\_.\ \ell \leftarrow 1\ \mathsf{else}\ \lambda\_.\ \ell \leftarrow 0\ \mathsf{in}\ (f\ ();!\,\ell) \tag{6.2}$$

Our type system correctly handles these situations in two different ways: in Equation (6.1) the leak is captured in the output type of f, and in Equation (6.2) it is captured in the label of f itself. Indeed, according to our typing rules, since the identity of the function f in Equation (6.1) depends on h, the type of the output of f will have a label that is at least $\top$. On the other hand, in Equation (6.2), the latent effect label for f is $\bot$ but the label of the function value itself must be $\top$ as it depends on h and the function may not be invoked. Similar considerations apply to the rules T-tapp and T-lapp for type and label application.

The rules for case analysis (T-match and T-if) demand that both branches are typed with program counter label $pc \sqcup \iota$ to account for the fact that control flow depends on the information with label ι of the expression e being cased on. This ensures that the branches do not have write effects below ι, which would otherwise be dependent on more sensitive information. Similarly, the result type τ has to be protected at ι.

T-Var
$$\frac{x : \tau \in \Gamma}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} x : \tau}$$

T-unit
$$\Xi \mid \Psi \mid \Gamma \vdash_{pc} () : 1^{\bot}$$

T-bool
$$\frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} b : \mathbb{B}^{\bot}}$$

T-nat
$$\frac{n \in \mathbb{N}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} n : \mathbb{N}^{\bot}}$$

T-binop
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \mathbb{N}^{\iota_1} \qquad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \mathbb{N}^{\iota_2} \qquad \circledcirc : \mathbb{N} \times \mathbb{N} \Rightarrow t}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 \circledcirc e_2 : t^{\iota_1 \sqcup \iota_2}}$$

T-lam
$$\frac{\Xi \mid \Psi \mid \Gamma, x : \tau_1 \vdash_{\iota_e} e : \tau_2}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \lambda x.\, e : \left(\tau_1 \xrightarrow{\iota_e} \tau_2\right)^{\bot}}$$

T-app
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \left(\tau_1 \xrightarrow{\iota_e} \tau_2\right)^{\iota} \qquad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \tau_1 \qquad \Psi \vdash \tau_2 \searrow \iota \qquad \Psi \vdash pc \sqcup \iota \sqsubseteq \iota_e}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1\ e_2 : \tau_2}$$

T-tlam
$$\frac{\Xi, \alpha \mid \Psi \mid \Gamma \vdash_{\iota_e} e : \tau}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \Lambda\, e : \left(\forall_{\iota_e} \alpha.\, \tau\right)^{\bot}}$$

T-llam
$$\frac{\Xi \mid \Psi, \kappa \mid \Gamma \vdash_{\iota_e} e : \tau \qquad \mathrm{FV}(\iota_e) \subseteq \Psi \cup \{\kappa\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \Lambda\!\!\Lambda\, e : \left(\forall\!\!\forall_{\iota_e} \kappa.\, \tau\right)^{\bot}}$$

T-tapp
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \left(\forall_{\iota_e} \alpha.\, \tau\right)^{\iota} \qquad \Psi \vdash pc \sqcup \iota \sqsubseteq \iota_e \qquad \mathrm{FV}(t) \subseteq \Xi}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e\ \_ : \tau[t/\alpha]}$$

T-lapp
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \left(\forall_{\iota_e} \kappa.\, \tau\right)^{\iota} \qquad \Psi \vdash pc \sqcup \iota \sqsubseteq \iota_e[\iota'/\kappa] \qquad \Psi \vdash \tau[\iota'/\kappa] \searrow \iota \qquad \mathrm{FV}(\iota') \subseteq \Psi}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e\ \_ : \tau[\iota'/\kappa]}$$

T-if
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \mathbb{B}^{\iota} \qquad \forall i \in \{1,2\}\,.\, \Xi \mid \Psi \mid \Gamma \vdash_{pc \sqcup \iota} e_i : \tau \qquad \Psi \vdash \tau \searrow \iota}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

T-inj
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau_i \qquad i \in \{1,2\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{inj}_i\ e : (\tau_1 + \tau_2)^{\bot}}$$

T-pair
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \tau_1 \qquad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \tau_2}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} (e_1, e_2) : (\tau_1 \times \tau_2)^{\bot}}$$

T-proj
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : (\tau_1 \times \tau_2)^{\iota} \qquad \Psi \vdash \tau_i \searrow \iota \qquad i \in \{1,2\}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \pi_i\ e : \tau_i}$$

T-match
$$\frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : (\tau_1 + \tau_2)^{\iota} \qquad \forall i \in \{1,2\}\,.\, \Xi \mid \Psi \mid \Gamma, x : \tau_i \vdash_{pc \sqcup \iota} e_i : \tau \qquad \Psi \vdash \tau \searrow \iota}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inj}_i\ \Rightarrow e_i\ \mathsf{end} : \tau}$$

Figure 6.2: Typing relation, part 1.

$$\textsc{T-fold}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau[\mu\,\alpha.\,\tau/\alpha]}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{fold}\ \ e : (\mu\,\alpha.\,\tau)^{\bot}} \qquad \textsc{T-unfold}\ \frac{\Psi \vdash \tau[\mu\,\alpha.\,\tau/\alpha] \searrow \iota \qquad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e : (\mu\,\alpha.\,\tau)^{\iota}}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{unfold}\ \ e : \tau[\mu\,\alpha.\,\tau/\alpha]}$$

$$\textsc{T-pack}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau[t/\alpha]}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{pack}\ e : (\exists\alpha.\,\tau)^{\bot}}$$

$$\textsc{T-unpack}\ \frac{\Psi \vdash \tau \searrow \iota \qquad \Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{pack}\ e_1 : (\exists\alpha.\,\tau')^{\iota} \qquad \Xi, \alpha \mid \Psi \mid \Gamma, x : \tau' \vdash_{pc \sqcup \iota} e_2 : \tau}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{unpack}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \tau}$$

$$\textsc{T-alloc}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau \qquad \Psi \vdash \tau \searrow pc}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{ref}(e) : \mathsf{ref}(\tau)^{\bot}}$$

$$\textsc{T-store}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 : \mathsf{ref}(\tau)^{\iota} \qquad \Xi \mid \Psi \mid \Gamma \vdash_{pc} e_2 : \tau \qquad \Psi \vdash \tau \searrow pc \sqcup \iota}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e_1 \leftarrow e_2 : 1^{\bot}}$$

$$\textsc{T-load}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc} \mathsf{ref}(e_1) : \mathsf{ref}(\tau)^{\iota} \qquad \Xi \mid \Psi \vdash \tau <: \tau' \qquad \Psi \vdash \tau' \searrow \iota}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} !\, e : \tau'}$$

$$\textsc{T-sub}\ \frac{\Xi \mid \Psi \mid \Gamma \vdash_{pc'} e : \tau' \qquad \Psi \vdash pc \sqsubseteq pc' \qquad \Xi \mid \Psi \vdash \tau' <: \tau}{\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau}$$

Figure 6.3: Typing relation, part 2.

The rule for assignment (T-store) captures that the pc label acts as an effect lower bound. It requires that when assigning an expression of type τ to a reference of type $\mathsf{ref}(\tau)^{\iota}$ then the label of τ is protected at both pc and ι. The former enforces pc as a lower bound on effects and the latter prevents implicit leaks arising from the identity of the reference. If not, then, for example, given ℓ_1 and ℓ_2 are references of type $\mathsf{ref}(\mathbb{N}^{\bot})$ and h a variable of type $\mathbb{B}^{\top}$, the program in Equation (6.3) would be typeable at $\mathbb{N}^{\bot}$ while leaking h.

$$\ell_1 \leftarrow 0; \ell_2 \leftarrow 0; \mathsf{let}\ r = \mathsf{if}\ h\ \mathsf{then}\ \ell_1\ \mathsf{else}\ \ell_2\ \mathsf{in}\ r \leftarrow 1; !\,\ell_1 \tag{6.3}$$

Note that all expressions typed with an introduction rule gets a type with label $\bot$. Intuitively, introducing, *e.g.*, a pair with components τ_1 and τ_2 has no observable effect nor does it leak any information, and the label of τ_i already captures the information that may have influenced the component. The label can, however, freely be raised using T-sub and the subtyping relation in Figure 6.4. The rule S-labeled allows a term with label ι_1 to be treated as a term with label ι_2 if $\iota_1 \sqsubseteq \iota_2$; the rest of the subtyping rules are standard. Notice that T-sub also allows the pc label to be freely weakened.

$$\textsc{S-refl}\ \frac{\mathrm{FV}(t) \subseteq \Xi}{\Xi \mid \Psi \vdash t <: t} \qquad \textsc{S-trans}\ \frac{\Xi \mid \Psi \vdash t_1 <: t_2 \qquad \Xi \mid \Psi \vdash t_2 <: t_3}{\Xi \mid \Psi \vdash t_1 <: t_3}$$

$$\textsc{S-arrow}\ \frac{\Xi \mid \Psi \vdash \tau_1' <: \tau_1 \qquad \Xi \mid \Psi \vdash \tau_2 <: \tau_2' \qquad \Psi \vdash \iota_2 \sqsubseteq \iota_1}{\Xi \mid \Psi \vdash \tau_1 \xrightarrow{\iota_1} \tau_2 <: \tau_1' \xrightarrow{\iota_2} \tau_2'}$$

$$\textsc{S-tforall}\ \frac{\Psi \vdash \iota_2 \sqsubseteq \iota_1 \qquad \Xi, \alpha \mid \Psi \vdash \tau_1 <: \tau_2}{\Xi \mid \Psi \vdash \forall_{\iota_1} \alpha.\, \tau_1 <: \forall_{\iota_2} \alpha.\, \tau_2} \qquad \textsc{S-lforall}\ \frac{\Psi, \kappa \vdash \iota_2 \sqsubseteq \iota_1 \qquad \Xi \mid \Psi, \kappa \vdash \tau_1 <: \tau_2}{\Xi \mid \Psi \vdash \mathbb{\forall}_{\iota_1} \kappa.\, \tau_1 <: \mathbb{\forall}_{\iota_2} \kappa.\, \tau_2}$$

$$\textsc{S-prod}\ \frac{\Xi \mid \Psi \vdash \tau_1 <: \tau_1' \qquad \Xi \mid \Psi \vdash \tau_2 <: \tau_2'}{\Xi \mid \Psi \vdash \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'} \qquad \textsc{S-sum}\ \frac{\Xi \mid \Psi \vdash \tau_1 <: \tau_1' \qquad \Xi \mid \Psi \vdash \tau_2 <: \tau_2'}{\Xi \mid \Psi \vdash \tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

$$\textsc{S-labeled}\ \frac{\Psi \vdash \iota_1 \sqsubseteq \iota_2 \qquad \Xi \mid \Psi \vdash t_1 <: t_2}{\Xi \mid \Psi \vdash {t_1}^{\iota_1} <: {t_2}^{\iota_2}}$$

Figure 6.4: Subtyping relation.

6.2 Semantic model

In this section we define our semantic model of λ_{sec}'s type system. The model formalizes an *observer-sensitive equivalence* which only relates computations *from the perspective of some observer*. Concretely, the observer is modelled by a fixed but arbitrary label ζ drawn from the lattice $\mathcal{L}$. The intuition is that terms typed with a label higher than ζ are indistinguishable to the observer whereas terms typed with a label lower than ζ are *not*.

Our semantic model captures all invariants necessary to prove that the type system guarantees that well-typed programs satisfy noninterference (Theorem 6.3.4). In Section 6.4 we demonstrate that our model can also be used to prove that syntactically ill-typed programs are semantically secure—this allows us to safely compose syntactically ill-typed but semantically secure programs with syntactically well-typed programs. In Section 6.4.5, we show that the model can also be used to prove "free" theorems.

A central idea in the model is to interpret each type both as a binary relation (Figure 6.6) and as unary relation (Figure 6.8). The binary relation relates expressions that are observationally equivalent to a ζ observer, and the unary relation relates expressions that do not have any ζ-observable side-effects. The unary relation is used within the binary relation to relate terms *independently* when the label of the type is higher than the observer—such terms are indistinguishable to the observer as long as they do not have any visible side-effects.

In this section, we will show step-by-step how to define our model in Iris. The syntax of Iris is shown in Figure 6.5; Iris is a higher-order separation logic with propositions of type *iProp* and some custom connectives that we will explain as we go along.

We start by defining the binary and unary value relations (Section 6.2.1) followed by a brief intermezzo, where we present the Modal Weakest Precondition theory (Section 6.2.2). We then turn to the expression relation (Section 6.2.3), the fundamental theorem of logical

relations, and the soundness theorem (Section 6.3).

$$\begin{aligned}
\sigma &::= 0 \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathit{iProp} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma \mid \ldots && \text{(Types)}\\
P, Q &::= x \mid \lambda x : \sigma.\, t \mid t(u) \mid \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q && \text{(Propositional logic)}\\
&\mid \forall x : \sigma.\, P \mid \exists x : \sigma.\, P \mid t = u && \text{(Higher-order logic)}\\
&\mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_L v \mid \ell \mapsto_R v \mid \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M}}\, e\, \{\Phi\} && \text{(Separation logic)}\\
&\mid \Box P \mid \triangleright P \mid \mu x : \sigma.\, t \mid {}_{\mathcal{E}_1}\!\Rightarrow\!\!\!\!\!\!\Rightarrow_{\mathcal{E}_2} P \mid \boxed{P}^{\mathcal{N}} \mid \ldots && \text{(Iris-specific connectives)}
\end{aligned}$$

Figure 6.5: Syntax of Iris. t and u represent arbitrary terms.

6.2.1 Value relations

The binary value relation is an Iris relation of type $\mathit{Rel} \triangleq \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\Box}$ where $\mathit{iProp}_{\Box}$ denotes the class of *persistent propositions* in Iris:

$$\begin{aligned}
\mathit{iProp}_{\Box} &\triangleq \{P : \mathit{iProp} \mid \mathsf{persistent}(P)\}\\
\mathsf{persistent}(P) &\triangleq P \vdash \Box P
\end{aligned}$$

Similarly, the unary value relation is an Iris predicate of type $\mathit{Pred} \triangleq \mathit{Val} \to \mathit{iProp}_{\Box}$.

By default, since Iris is a separation logic, propositions denote sets of resources and $P * Q$ holds for resources that can be split into two disjoint parts satisfying P and Q, respectively. The proposition $P \mathrel{-\!\!*} Q$ describes those resources which, if we combine them with a disjoint resource satisfying P, satisfies Q. As such, Iris propositions assert *ownership* of ephemeral (non-persistent) resources. For example, the points-to connectives $\ell \mapsto_L v$ and $\ell \mapsto_R v$ asserts exclusive ownership of location ℓ storing value v in the state of the programs on the left- and right-hand side, respectively. Such proposition may cease to hold, *e.g.*, when ℓ is updated to point to some other value than v. Intuitively, persistent propositions are propositions that do *not* assert exclusive ownership of resources and once they hold, they hold forever. In Iris, this is expressed using the *persistence modality* $\Box$. The proposition $\Box P$ (read "persistently P") says P holds without asserting any ephemeral propositions and thus P can be freely duplicated, *i.e.*, $\Box P \vdash \Box P * \Box P$, and eliminated, *i.e.*, $\Box P \vdash P$. It is important that our value relations are defined using persistent predicates as our type system is intuitionistic, in the sense that it admits the usual structural rules, which, *e.g.*, means that the assumption that a value has a type τ may be used repeatedly.

Binary value relation. The binary value relations $[\![\tau]\!]^{\rho}_{\Theta}$ and $[\![t]\!]^{\rho}_{\Theta}$ for a labeled type τ and an unlabeled type t are defined by mutual induction on τ and t. Here $\rho : \mathit{LabelVar} \to \mathcal{L}$ is a semantic label environment mapping label variables to labels, and Θ is a semantic type environment for type variables, as is usual for interpretations of languages with parametric polymorphism. However, for every type variable we keep both a binary relation and two unary relations, one for each of the two sides:

$$\Theta : \mathit{TypeVar} \to \mathit{Rel} \times \mathit{Pred} \times \mathit{Pred}.$$

We use $\Theta_L, \Theta_R : \mathit{TypeVar} \to \mathit{Pred}$ as shorthand for $\pi_2 \circ \Theta$ and $\pi_3 \circ \Theta$, respectively, where $\pi_i(x)$ denotes the ith projection of x. It will be a property of the binary relation that the following binary-unary subsumption property (Lemma 6.3.2) holds:

$$\forall v, v'.\, [\![\tau]\!]^\rho_\Theta(v, v') \;-\!\!*\; [\![\tau]\!]^\rho_{\Theta_L}(v) * [\![\tau]\!]^\rho_{\Theta_R}(v'),$$

where $[\![\tau]\!]^\rho_{\Theta_L}(v)$ and $[\![\tau]\!]^\rho_{\Theta_R}(v')$ denote the unary interpretation of τ at v and v'. This property is crucial: Intuitively, even though two values are observationally *equivalent* to a ζ observer, they are also—independently—not supposed to have any latent ζ-observable side-effects. We elaborate further on this in Section 6.3 with more technical details. However, for the property to hold, the binary value relation has to be set up carefully, and the binary relation on open terms (explained in Section 6.2.3) requires that Θ is *coherent* in the following sense:

$$\mathit{Coh}(\Theta) \triangleq \mathop{\Large *}_{(\Phi,\Phi_L,\Phi_R)\in \mathrm{Im}(\Theta)} \square\left(\forall v, v'.\, \Phi(v, v') \;-\!\!*\; \Phi_L(v) * \Phi_R(v')\right).$$

The big iterated separating conjunction quantifies over all triples (Φ, Φ_L, Φ_R) in the image of Θ and demands that the binary-unary subsumption property holds for the relations. The full definition of the binary interpretation is shown in Figure 6.6.

The value interpretation of *labeled* types makes use of an interpretation $[\![\iota]\!]_\rho$ of syntactic labels ι defined as follows:

$$\begin{aligned} [\![\kappa]\!]_\rho &\triangleq \rho(\kappa) \\ [\![l]\!]_\rho &\triangleq l \\ [\![\iota_1 \sqcup \iota_2]\!]_\rho &\triangleq [\![\iota_1]\!]_\rho \sqcup [\![\iota_2]\!]_\rho. \end{aligned}$$

As above, ρ is an environment mapping label variables to labels. Notice that in the last equation, the $\sqcup$ on the left is the formal syntactic least upper bound whereas the $\sqcup$ on the right is the least upper bound in the lattice $\mathcal{L}$.

The interpretation of labeled types now follows the intuition given in the beginning of this section: low-labeled types (where $[\![\iota]\!]_\rho \sqsubseteq \zeta$) are distinguishable to the observer, and thus values should be related by the binary relation; high-labeled types (where $[\![\iota]\!]_\rho \not\sqsubseteq \zeta$) are *in*distinguishable to the observer and thus values should individually satisfy the unary interpretation to ensure that any latent effects will not be ζ-observable.

$$[\![t^\iota]\!]^\rho_\Theta(v, v') \triangleq \begin{cases} [\![t]\!]^\rho_\Theta(v, v') & \text{if } [\![\iota]\!]_\rho \sqsubseteq \zeta \\ [\![t]\!]^\rho_{\Theta_L}(v) * [\![t]\!]^\rho_{\Theta_R}(v') & \text{if } [\![\iota]\!]_\rho \not\sqsubseteq \zeta \end{cases}$$

This is the key point of interaction between the unary and binary relation.

The value interpretation of *unlabeled* types follows a structure that readers familiar with previous logical-relations models in Iris will find familiar. However, we also need to guarantee that the relation satisfies the binary-unary subsumption property.

If t is an (unlabeled) ground type (1, $\mathbb{B}$, or $\mathbb{N}$), two values are related at t if they are equal and compatible with the type. For products $\tau_1 \times \tau_2$, two values are related if they are both pairs with components related at their respective types. Similarly for sums $\tau_1 + \tau_2$, two values are related if they are both inj_i for the same i and their contents are related at τ_i.

Value relation

$$
\begin{aligned}
[\![\alpha]\!]^{\rho}_{\Theta} &\triangleq \pi_1\,(\Theta(\alpha))\\
[\![1]\!]^{\rho}_{\Theta}(v,v') &\triangleq v = v' = ()\\
[\![\mathbb{B}]\!]^{\rho}_{\Theta}(v,v') &\triangleq v = v' \in \{\mathsf{true},\mathsf{false}\}\\
[\![\mathbb{N}]\!]^{\rho}_{\Theta}(v,v') &\triangleq v = v' \in \mathbb{N}\\
[\![\tau_1 \times \tau_2]\!]^{\rho}_{\Theta}(v,v') &\triangleq \exists v_1, v_2, v_1', v_2'.\, v = (v_1,v_2) * v' = (v_1',v_2') * [\![\tau_1]\!]^{\rho}_{\Theta}(v_1,v_1') * [\![\tau_2]\!]^{\rho}_{\Theta}(v_2,v_2')\\
[\![\tau_1 + \tau_2]\!]^{\rho}_{\Theta}(v,v') &\triangleq \bigvee_{i\in\{1,2\}} \exists w, w'.\, v = \mathsf{inj}_i\, w * v' = \mathsf{inj}_i\, w' * [\![\tau_i]\!]^{\rho}_{\Theta}(w,w')\\
[\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^{\rho}_{\Theta}(v,v') &\triangleq \Box\,(\forall w, w'.\, [\![\tau_1]\!]^{\rho}_{\Theta}(w,w') \mathrel{-\!\!*} \mathcal{E}[\![\tau_2]\!]^{\rho}_{\Theta}(v\,w, v'\,w')) *\\
&\quad [\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^{\rho}_{\Theta_L}(v) * [\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^{\rho}_{\Theta_R}(v')\\
[\![\forall_{\iota_e}\,\alpha.\,\tau]\!]^{\rho}_{\Theta}(v,v') &\triangleq \Box\,\big(\forall \Phi : \mathit{Rel}.\, \forall \Phi_L, \Phi_R : \mathit{Pred}.\\
&\quad \Box\,(\forall v, v'.\, \Phi(v,v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v')) \mathrel{-\!\!*} \mathcal{E}[\![\tau]\!]^{\rho}_{\Theta,\alpha\mapsto(\Phi,\Phi_L,\Phi_R)}(v\ \_, v'\ \_)\big) *\\
&\quad [\![\forall_{\iota_e}\,\alpha.\,\tau]\!]^{\rho}_{\Theta_L}(v) * [\![\forall_{\iota_e}\,\alpha.\,\tau]\!]^{\rho}_{\Theta_R}(v')\\
[\![\forall_{\iota_e}\,\kappa.\,\tau]\!]^{\rho}_{\Theta}(v,v') &\triangleq \Box\left(\forall l \in \mathcal{L}.\, \mathcal{E}[\![\tau]\!]^{\rho,\kappa\mapsto l}_{\Theta}(v\ \_, v'\ \_)\right) * [\![\forall_{\iota_e}\,\kappa.\,\tau]\!]^{\rho}_{\Theta_L}(v) * [\![\forall_{\iota_e}\,\kappa.\,\tau]\!]^{\rho}_{\Theta_R}(v')\\
[\![\exists\alpha.\,\tau]\!]^{\rho}_{\Theta}(v,v') &\triangleq \Box\,\big(\exists \Phi : \mathit{Rel}.\, \exists \Phi_L, \Phi_R : \mathit{Pred}.\\
&\quad \Box\,(\forall v, v'.\, \Phi(v,v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v')) *\\
&\quad \exists w, w'.\, v = \mathsf{pack}\, w * v' = \mathsf{pack}\, w' * [\![\tau]\!]^{\rho}_{\Theta,\alpha\mapsto(\Phi,\Phi_L,\Phi_R)}(w,w')\big)\\
[\![\mu\,\alpha.\,\tau]\!]^{\rho}_{\Theta} &\triangleq \mu\,\Phi : \mathit{Rel}.\, \lambda(v,v').\exists w, w'.\, v = \mathsf{fold}\, w * v' = \mathsf{fold}\, w' *\\
&\quad \triangleright [\![\tau]\!]^{\rho}_{\Theta,\alpha\mapsto(\Phi,[\![\mu\,\alpha.\,\tau]\!]^{\rho}_{\Theta_L},[\![\mu\,\alpha.\,\tau]\!]^{\rho}_{\Theta_R})}(w,w')\\
[\![\mathsf{ref}(\tau)]\!]^{\rho}_{\Theta}(v,v') &\triangleq \exists \ell, \ell'.\, v = \ell * v' = \ell' * \boxed{\exists w, w'.\, \ell \mapsto_L w * \ell' \mapsto_R w' * [\![\tau]\!]^{\rho}_{\Theta}(w,w')}^{\mathcal{N}_{root}.(\ell,\ell')}\\
[\![t^{\iota}]\!]^{\rho}_{\Theta}(v,v') &\triangleq \begin{cases} [\![t]\!]^{\rho}_{\Theta}(v,v') & \text{if } [\![\iota]\!]_{\rho} \sqsubseteq \zeta\\ [\![t]\!]^{\rho}_{\Theta_L}(v) * [\![t]\!]^{\rho}_{\Theta_R}(v') & \text{if } [\![\iota]\!]_{\rho} \not\sqsubseteq \zeta \end{cases}
\end{aligned}
$$

Expression relation

$$
\mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e,e') \triangleq \mathsf{mwp}\ e \sim e'\ \{[\![\tau]\!]^{\rho}_{\Theta}\}
$$

Environment relation

$$
\begin{aligned}
\mathcal{G}[\![\cdot]\!]^{\rho}_{\Theta}(\epsilon,\epsilon) &\triangleq \mathsf{True}\\
\mathcal{G}[\![\Gamma, x:\tau]\!]^{\rho}_{\Theta}(\vec{v}w, \vec{v'}w') &\triangleq \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta}(\vec{v},\vec{v'}) * [\![\tau]\!]^{\rho}_{\Theta}(w,w')
\end{aligned}
$$

Semantic typing judgment

$$
\begin{aligned}
\mathit{Coh}(\Theta) &\triangleq \mathop{\Large *}_{(\Phi,\Phi_L,\Phi_R)\in\Theta} \Box\,(\forall v, v'.\, \Phi(v,v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v'))\\
\Xi \mid \Psi \mid \Gamma \vDash e \approx_{\zeta} e' : \tau &\triangleq \Box \begin{pmatrix} \forall \Theta, \rho, \vec{v}, \vec{v'}.\, \mathsf{dom}(\Xi) \subseteq \mathsf{dom}(\Theta) * \mathsf{dom}(\Psi) \subseteq \mathsf{dom}(\rho) \mathrel{-\!\!*} \\ \mathit{Coh}(\Theta) * \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta}(\vec{v},\vec{v'}) \mathrel{-\!\!*} \mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}]) \end{pmatrix}
\end{aligned}
$$

Figure 6.6: Binary interpretations.

The first clause of the interpretation of the function type is a slight variation of the classical function type interpretation in logical-relations models: two values v and v' are related at type $\tau_1 \xrightarrow{\iota_e} \tau_2$ if they map inputs related at τ_1 to related results at τ_2.

$$\begin{aligned}[\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^\rho_\Theta(v, v') \triangleq \Box\big(\forall w, w'.\, [\![\tau_1]\!]^\rho_\Theta(w, w') \mathrel{-\!\!*} \mathcal{E}[\![\tau_2]\!]^\rho_\Theta(v\ w, v'\ w')\big) * \\ [\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^\rho_{\Theta_L}(v) * [\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^\rho_{\Theta_R}(v').\end{aligned}$$

Note that we ignore the latent effect label and that we wrap the clause in a persistence modality in order to ensure that the relation is persistent. The former will only be important for the unary interpretation. The two following clauses require that v and v' individually satisfy the unary interpretation to ensure that the binary-unary subsumption property holds.

For type-polymorphic types we use the semantic type environment which maps type variables to triples consisting of an Iris relation on values and two unary relations. We define the interpretation $[\![\alpha]\!]^\rho_\Theta$ of type variable α by looking up the variable in Θ and taking the first projection.

Universal types are interpreted using logical propositions that are also universally quantified but over *semantic predicates*, heavily relying on Iris's support for higher-order impredicative quantification.

$$\begin{aligned}[\![\forall_{\iota_e}\, \alpha.\, \tau]\!]^\rho_\Theta(v, v') \triangleq \Box\big(\forall \Phi : \mathit{Rel}.\, \forall \Phi_L, \Phi_R : \mathit{Pred}. \\ \Box\big(\forall v, v'.\, \Phi(v, v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v')\big) \mathrel{-\!\!*} \\ \mathcal{E}[\![\tau]\!]^\rho_{\Theta, \alpha \mapsto (\Phi, \Phi_L, \Phi_R)}(v\ \_, v'\ \_)\big) * \\ [\![\forall_{\iota_e}\, \alpha.\, \tau]\!]^\rho_{\Theta_L}(v) * [\![\forall_{\iota_e}\, \alpha.\, \tau]\!]^\rho_{\Theta_R}(v').\end{aligned}$$

However, we quantify not only over a binary relation but also two unary relations for which we require the subsumption property to hold. This ensures that the semantic type environment is coherent. Two value v and v' are then related at type $\forall_{\iota_e}\, \alpha.\, \tau$ when type applications v _ and v' _ are related at τ in a semantic environment mapping α to the binary relation and the two unary relations. As in the case for the function type, we also require that v and v' satisfy the unary interpretation.

Label abstraction is interpreted following a similar pattern:

$$[\![\forall\!\!/_{\iota_e}\, \kappa.\, \tau]\!]^\rho_\Theta(v) \triangleq \Box\left(\forall l \in \mathcal{L}.\, \mathcal{E}[\![\tau]\!]^{\rho, \kappa \mapsto l}_\Theta(v\ \_)\right).$$

We quantify over *semantic labels*—which are just labels from the lattice $\mathcal{L}$—and express that v and v' are related at type $\forall\!\!/_{\iota_e}\, \kappa.\, \tau$ when the applications v _ and v' _ are related at type τ in an extended semantic label environment mapping κ to the label l.

The interpretation of existential types $\exists \alpha.\, \tau$ quantifies existentially over a binary relation and two unary relations satisfying the subsumption property and relates values of the form pack w and pack w' if w and w' are related at type τ in an extended semantic type environment.

$$\begin{aligned}[\![\exists \alpha.\, \tau]\!]^\rho_\Theta(v, v') \triangleq \Box\big(\exists \Phi : \mathit{Rel}.\, \exists \Phi_L, \Phi_R : \mathit{Pred}. \\ \Box\big(\forall v, v'.\, \Phi(v, v') \mathrel{-\!\!*} \Phi_L(v) * \Phi_R(v')\big) * \\ \exists w, w'.\, v = \mathsf{pack}\ w * v' = \mathsf{pack}\ w' * [\![\tau]\!]^\rho_{\Theta, \alpha \mapsto (\Phi, \Phi_L, \Phi_R)}(w, w')\big)\end{aligned}$$

To interpret recursive types we make use of Iris's *guarded recursive predicates*. The guarded fixed-point operator $\mu x : \sigma.\, t$ of Iris can be used to define recursive predicates (without restrictions on variance for occurrences of x) by requiring that all recursive occurrences of x are

guarded by a *later modality* $\triangleright$. Intuitively, the later modality asserts that something holds "one step of computation later". It is monotone ($P \vdash Q$ implies $\triangleright P \vdash \triangleright Q$) and can be introduced ($P \vdash \triangleright P$). In Iris, with the restriction of guardedness, the fixed-point operator satisfies the expected equation: $\mu x : \sigma.\, t = t[\mu x : \sigma.\, t/x]$. The key proof principle associated with the later modality is the Löb rule: $(\triangleright P \Rightarrow P) \Rightarrow P$, which is used to prove the binary-unary subsumption property (Lemma 6.3.2) in the case of recursive types.

Using this fixed-point property, two values are related at type $\mu\,\alpha.\,\tau$ if they are of the form $\mathsf{fold}\ w$ and $\mathsf{fold}\ w'$, and if w and w' are related at τ (under a later modality) in an extended type environment mapping α to the triple $(\llbracket \mu\,\alpha.\,\tau \rrbracket^{\rho}_{\Theta}, \llbracket \mu\,\alpha.\,\tau \rrbracket^{\rho}_{\Theta_L}, \llbracket \mu\,\alpha.\,\tau \rrbracket^{\rho}_{\Theta_R})$.

$$\llbracket \mu\,\alpha.\,\tau \rrbracket^{\rho}_{\Theta} \triangleq \mu\,\Phi : \mathit{Rel}.\ \lambda(v, v').\exists w, w'.\ v = \mathsf{fold}\ w * v' = \mathsf{fold}\ w' * \triangleright \llbracket \tau \rrbracket^{\rho}_{\Theta, \alpha \mapsto (\Phi, \llbracket \mu\,\alpha.\,\tau \rrbracket^{\rho}_{\Theta_L}, \llbracket \mu\,\alpha.\,\tau \rrbracket^{\rho}_{\Theta_R})}(w, w').$$

The unary relations again ensure that the extended semantic type environment is coherent.

Recall that the binary relation is intended to relate terms that are observationally equivalent to a ζ-observer. Hence related values of reference type $\mathsf{ref}(\tau)$ should be locations ℓ and ℓ' such that their contents may change but the contents should always stay related at type τ. To express this requirement, we make use of Iris's *invariant assertion*

$$\boxed{P}^{\mathcal{N}}$$

which expresses the (persistent) knowledge that a proposition P holds at all times. In order to avoid reentrancy issues, where invariants are opened in a nested (and unsound) fashion, Iris features *invariant namespaces* $\mathcal{N} \in \mathit{InvName}$ and *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$. Iris annotates invariants with a namespace $\mathcal{N}$ to identify the invariant and, as we shall explain later, we annotate modal weakest preconditions $\mathsf{mwp}^{\mathcal{M}}_{\mathcal{E}}\ e\ \{\Phi\}$ with a mask $\mathcal{E}$ to keep track of which invariants are enabled and may be opened. If the mask is omitted we consider the modal weakest precondition with mask $\top$, the set of all invariant names.

In order to work with invariants formally in Iris we make use of the *update modality* ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2}$. We write $\Rightarrow_{\mathcal{E}}$ if $\mathcal{E}_1 = \mathcal{E}_2 = \mathcal{E}$. Akin to how a weakest precondition is used to reason about physical state, the update modality is used to reason about ghost state. The update modality is annotated with masks $\mathcal{E}_1$ and $\mathcal{E}_2$ that denote which invariants are enabled and may be opened before and after the modality is introduced. Intuitively, the proposition ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P$ holds for resources that (given the invariants in $\mathcal{E}_1$ are enabled) can be updated to resources that satisfy P (with the invariants in $\mathcal{E}_2$ enabled) without violating the environment's knowledge or ownership of resources.

Some formal rules for invariants in Iris can be found in Figure 6.7. An invariant can be al-

INV-ALLOC
$$\triangleright P \vdash \Rightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

INV-PERSIST
$$\boxed{P}^{\mathcal{N}} \vdash \Box \boxed{P}^{\mathcal{N}}$$

INV-OPEN-UPD
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * (\triangleright P \mathrel{-\!\!*} \Rightarrow_{\mathcal{E} \setminus \mathcal{N}} (\triangleright P * Q)) \vdash \Rightarrow_{\mathcal{E}} Q}$$

Figure 6.7: Rules for invariants.

located (INV-ALLOC) by giving up ownership of P, a possibly ephemeral proposition. Invariants are persistent (INV-PERSIST) and hence duplicable. The contents of invariants may be accessed

in a carefully restricted way (INV-OPEN-UPD): to prove $\Rightarrow\!\!\!\!\triangleright_{\mathcal{E}} Q$, we may open an invariant and assume $\triangleright P$ as long as we re-establish the invariant $\triangleright P$. For more details on invariants in Iris, including the role of the later modality in the rules, see Birkedal and Bizjak [BB17] and Jung et al. [Jun+18b].

With the invariant connective at hand, the binary relation for reference types $\mathsf{ref}(\tau)$ is straightforward and relates locations that invariantly have contents related at type τ.

$$[\![\mathsf{ref}(\tau)]\!]^{\rho}_{\Theta}(v, v') \triangleq$$
$$\exists \ell, \ell'.\, v = \ell * v' = \ell' * \boxed{\exists w, w'.\, \ell \mapsto_L w * \ell' \mapsto_R w' * [\![\tau]\!]^{\rho}_{\Theta}(w, w')}^{\mathcal{N}.(\ell,\ell')}.$$

Here $\mathcal{N}.(\ell, \ell')$ is some namespace designated to the invariant on the locations ℓ and ℓ'.

Unary value relation. The unary value relations $[\![\tau]\!]^{\rho}_{\Delta}$ and $[\![t]\!]^{\rho}_{\Delta}$ for a labeled type τ and an unlabeled type t are defined by mutual induction on τ and t; however, the label on labeled types is ignored since, as mentioned earlier, the point of the unary relation is to ensure that computations embedded in values have no ζ-observable side-effects.

$$[\![t^{\iota}]\!]^{\rho}_{\Delta}(v) \triangleq [\![t]\!]^{\rho}_{\Delta}(v).$$

Here Δ is a semantic type environment mapping type variables to unary relations of type *Pred* and ρ is a semantic label environment mapping label variables to labels. The full relation is shown in Figure 6.8.

The only values of ground type are values compatible with the type. Similarly, values of type $\tau_1 \times \tau_2$ are pairs with components inhabiting the interpretation of τ_1 and τ_2, respectively. Values of type $\tau_1 + \tau_2$ are inj_i with contents related at τ_i.

The unary interpretation of function type $\tau_1 \xrightarrow{\iota_e} \tau_2$ follows the canonical pattern and takes related input at τ_1 to related results at τ_2.

$$[\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^{\rho}_{\Delta}(v) \triangleq \square\left(\forall w.\, [\![\tau_1]\!]^{\rho}_{\Delta}(w) \mathrel{-\!\!*} \mathcal{E}_{\iota_e}[\![\tau_2]\!]^{\rho}_{\Delta}(v\ w)\right)$$

However, notice that the unary expression relation is indexed with the latent effect label of the function. The unary relation is only concerned with expressions in high-labeled contexts; low-labeled contexts are ζ-observable and the unary relation poses no requirements on these. We will return to these matters in Section 6.2.3 when discussing the expression relations.

Both type- and label-polymorphic types are interpreted by quantifying over their semantic counterparts and a value v inhabits the polymorphic type if application v _ inhabits τ in the extended semantic environment. As for function types, the expression relation is indexed with the latent effect label of the polymorphic binder. The interpretation of existential types and recursive types follows the same pattern as in the binary interpretation.

$$[\![\forall_{\iota_e}\, \alpha.\, \tau]\!]^{\rho}_{\Delta}(v) \triangleq \square\left(\forall \Phi : \mathit{Pred}.\, \mathcal{E}_{\iota_e}[\![\tau]\!]^{\rho}_{\Delta,\alpha\mapsto\Phi}(v\ \_)\right)$$
$$[\![\forall_{\iota_e}\, \kappa.\, \tau]\!]^{\rho}_{\Delta}(v) \triangleq \square\left(\forall l \in \mathcal{L}.\, \mathcal{E}_{\iota_e}[\![\tau]\!]^{\rho,\kappa\mapsto l}_{\Delta}(v\ \_)\right)$$
$$[\![\exists \alpha.\, \tau]\!]^{\rho}_{\Delta}(v) \triangleq \square\left(\exists \Phi : \mathit{Pred}.\, \exists w.\, v = \mathsf{pack}\ w * [\![\tau]\!]^{\rho}_{\Delta,\alpha\mapsto\Phi}(w)\right)$$
$$[\![\mu\, \alpha.\, \tau]\!]^{\rho}_{\Delta} \triangleq \mu\, \Phi : \mathit{Pred}.\, \lambda v. \exists w.\, v = \mathsf{fold}\ w * \triangleright [\![\tau]\!]^{\rho}_{\Delta,\alpha\mapsto\Phi}(w)$$

The interpretation of reference types is the central part of the unary interpretation and states that terms have no ζ-observable side-effects. Intuitively, a reference containing data

Value relation

$$
\begin{aligned}
[\![\alpha]\!]^\rho_\Delta &\triangleq \Delta(\alpha)\\
[\![1]\!]^\rho_\Delta(v) &\triangleq v = ()\\
[\![\mathbb{B}]\!]^\rho_\Delta(v) &\triangleq v \in \{\mathsf{true}, \mathsf{false}\}\\
[\![\mathbb{N}]\!]^\rho_\Delta(v) &\triangleq v \in \mathbb{N}\\
[\![\tau_1 \times \tau_2]\!]^\rho_\Delta(v) &\triangleq \exists v_1, v_2.\, v = (v_1, v_2) * [\![\tau_1]\!]^\rho_\Delta(v_1) * [\![\tau_2]\!]^\rho_\Delta(v_2)\\
[\![\tau_1 + \tau_2]\!]^\rho_\Delta(v) &\triangleq \bigvee_{i\in\{1,2\}} \exists w.\, v = \mathsf{inj}_i\, w * [\![\tau_i]\!]^\rho_\Delta(w)\\
[\![\tau_1 \xrightarrow{\iota_e} \tau_2]\!]^\rho_\Delta(v) &\triangleq \Box\,(\forall w.\, [\![\tau_1]\!]^\rho_\Delta(w) \mathrel{-\!\!*} \mathcal{E}_{\iota_e}[\![\tau_2]\!]^\rho_\Delta(v\ w))\\
[\![\forall_{\iota_e}\, \alpha.\, \tau]\!]^\rho_\Delta(v) &\triangleq \Box\left(\forall f : \mathit{Pred}.\, \mathcal{E}_{\iota_e}[\![\tau]\!]^\rho_{\Delta,\alpha\mapsto f}(v\ \_)\right)\\
[\![\forall_{\iota_e}\, \kappa.\, \tau]\!]^\rho_\Delta(v) &\triangleq \Box\left(\forall l \in \mathcal{L}.\, \mathcal{E}_{\iota_e}[\![\tau]\!]^{\rho,\kappa\mapsto l}_\Delta(v\ \_)\right)\\
[\![\exists\alpha.\, \tau]\!]^\rho_\Delta(v) &\triangleq \Box\left(\exists \Phi : \mathit{Pred}.\, \exists w.\, v = \mathsf{pack}\ w * [\![\tau]\!]^\rho_{\Delta,\alpha\mapsto\Phi}(w)\right)\\
[\![\mu\,\alpha.\, \tau]\!]^\rho_\Delta &\triangleq \mu\,\Phi : \mathit{Pred}.\, \lambda v.\exists w.\, v = \mathsf{fold}\ w * \triangleright [\![\tau]\!]^\rho_{\Delta,\alpha\mapsto f}(w)\\
[\![\mathsf{ref}(t^\iota)]\!]^\rho_\Delta(v) &\triangleq \exists \ell, \mathcal{N}.\, v = \ell * \mathcal{R}(\Delta, \rho, \ell, \iota, \mathcal{N})\\
\mathcal{R}(\Delta, \rho, \ell, \iota, \mathcal{N}) &\triangleq \begin{cases}
\Box\forall\mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rightarrow & \\
\quad \left(\mathcal{E} \Mapsto_{\mathcal{E}\setminus\mathcal{N}} \triangleright \left(\begin{array}{l}\exists w.\, \ell \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) *\\ \left((\triangleright\, \ell \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w)) \mathrel{-\!\!*}\ {}_{\mathcal{E}\setminus\mathcal{N}}\Mapsto_{\mathcal{E}} \mathsf{True}\right)\end{array}\right)\right) & \text{if } [\![\iota]\!]_\rho \sqsubseteq \zeta\\
\Box\forall\mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rightarrow & \\
\quad \left(\mathcal{E} \Mapsto_{\mathcal{E}\setminus\mathcal{N}} \triangleright \left(\begin{array}{l}\exists w.\, \ell \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) *\\ \left((\triangleright \exists w'.\, \ell \mapsto_i w' * [\![\tau]\!]^\rho_\Delta(w')) \mathrel{-\!\!*}\ {}_{\mathcal{E}\setminus\mathcal{N}}\Mapsto_{\mathcal{E}} \mathsf{True}\right)\end{array}\right)\right) & \text{if } [\![\iota]\!]_\rho \not\sqsubseteq \zeta
\end{cases}\\
[\![t^\iota]\!]^\rho_\Delta(v) &\triangleq [\![t]\!]^\rho_\Delta(v)
\end{aligned}
$$

Expression relation

$$
\mathcal{E}_{pc}[\![\tau]\!]^\rho_\Delta(e) \triangleq [\![pc]\!]_\rho \not\sqsubseteq \zeta \Rightarrow \mathsf{mwp}^{\mathcal{M}\Mapsto} e\, \{[\![\tau]\!]^\rho_\Delta\}
$$

Environment relation

$$
\begin{aligned}
\mathcal{G}[\![\cdot]\!]^\rho_\Delta(\epsilon) &\triangleq \mathsf{True}\\
\mathcal{G}[\![\Gamma, x : \tau]\!]^\rho_\Delta(\vec{v}w) &\triangleq \mathcal{G}[\![\Gamma]\!]^\rho_\Delta(\vec{v}) * [\![\tau]\!]^\rho_\Delta(w)
\end{aligned}
$$

Semantic typing judgment

$$
\Xi \mid \Psi \mid \Gamma \vDash_{pc} e : \tau \triangleq \Box \begin{pmatrix}\forall \Delta, \rho, \vec{v}.\, \mathsf{dom}(\Xi) \subseteq \mathsf{dom}(\Delta) * \mathsf{dom}(\Psi) \subseteq \mathsf{dom}(\rho) \mathrel{-\!\!*}\\ \mathcal{G}[\![\Gamma]\!]^\rho_\Delta(\vec{v}) \mathrel{-\!\!*} \mathcal{E}_{pc}[\![\tau]\!]^\rho_\Delta(e[\vec{v}/\vec{x}])\end{pmatrix}
$$

Figure 6.8: Unary interpretations.

with a label lower than ζ is *not* allowed to change when execution conditionally depends on higher-labeled information as this would implicitly leak the high-labeled information through the state. The contents of references with a label higher than ζ, however, can always be modified as long as the new contents are compatible with the types.

In order to state this intuition formally in Iris, *while* at the same time ensuring that the binary-unary subsumption property holds, we make use of the update modality to encode a relaxed form of semantic invariants. Instead of using an Iris invariant to capture the meaning of a reference type, we essentially use the key properties of Iris invariants (that they can be opened and closed again) and, depending on the label of the contents of the reference, we can express whether the value stored in the reference is allowed to change or not. As such, values v of type $\mathsf{ref}(t^\iota)$ are locations for which there exists a namespace $\mathcal{N}$ such that $\mathcal{R}(\Delta, \rho, \ell, \iota, \mathcal{N})$ holds.

$$[\![\mathsf{ref}(t^\iota)]\!]^\rho_\Delta(v) \triangleq \exists \ell, \mathcal{N}.\, v = \ell * \mathcal{R}(\Delta, \rho, \ell, \iota, \mathcal{N}).$$

The namespace $\mathcal{N}$ is *some* namespace associated with ι. The $\mathcal{R}(\Delta, \rho, \ell, \iota, \mathcal{N})$ proposition states that if the content of the reference is of a low-labeled type ($[\![\iota]\!]_\rho \sqsubseteq \zeta$) then the content of ℓ is not allowed to change in an observable way:

$$\square \forall \mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rightarrow \left({}_{\mathcal{E}}\Rightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright \left(\begin{array}{l} \exists w.\, \ell \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) * \\ \left(\left(\triangleright\, \ell \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) \right) \mathrel{-\!\!*} {}_{\mathcal{E}\setminus\mathcal{N}}\Rightarrow_{\mathcal{E}} \mathsf{True} \right) \end{array} \right) \right).$$

If we ignore the later modalities, intuitively, this says that if the namespace $\mathcal{N}$ is currently enabled we can, by disabling $\mathcal{N}$, get ownership of the points-to connective $\ell \mapsto_i w$ with $i \in \{L, R\}$ such that w inhabits $[\![\tau]\!]^\rho_\Delta$. Moreover, the namespace $\mathcal{N}$ can only be enabled again by giving back the ownership of the points-to connective with *unmodified* contents w.

In a similar fashion, if the content of the reference is of a high-labeled type ($[\![\iota]\!]_\rho \not\sqsubseteq \zeta$) then the content *is* allowed to change:

$$\square \forall \mathcal{E}.\, \mathcal{N} \subseteq \mathcal{E} \Rightarrow \left({}_{\mathcal{E}}\Rightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright \left(\begin{array}{l} \exists w.\, \ell \mapsto_i w * [\![\tau]\!]^\rho_\Delta(w) * \\ \left(\left(\triangleright\, \exists w'.\, \ell \mapsto_i w' * [\![\tau]\!]^\rho_\Delta(w') \right) \mathrel{-\!\!*} {}_{\mathcal{E}\setminus\mathcal{N}}\Rightarrow_{\mathcal{E}} \mathsf{True} \right) \end{array} \right) \right).$$

Intuitively, as before, if the namespace $\mathcal{N}$ is currently enabled we can, by disabling $\mathcal{N}$, get ownership of the points-to connective $\ell \mapsto_i w$ such that w inhabits τ. However, we can enable $\mathcal{N}$ again by giving back the ownership of the points-to connective with *any* content w' as long as it still inhabits type τ.

6.2.2 Modal Weakest Precondition

We now turn to the theory of the *Modal Weakest Precondition (MWP)* connective. Recall from the Introduction that due its termination-insensitive nature, existing approaches using Iris' weakest preconditions do not suffices for defining our expression relations. Moreover, we will need both a binary and a unary connective that interact in a reasonable way. As will be clear by the end of this section, the MWP theory accommodates all of this while providing high-level reasoning principles.

Similarly to how the standard weakest precondition in Iris is defined [Kre+17], our new definition of a modal weakest precondition is language agnostic; it is not tied to a particular

programming language and it is defined generically over any suitable notion of expression, state, and reduction relation. As a consequence of this generality, we do not make any assumptions on how the state of the programming language is defined; instead, as for standard Iris weakest preconditions, we parameterize modal weakest preconditions by a *state interpretation* $S : \mathit{State} \to \mathit{iProp}$. The S predicate interprets the state of the programming language using Iris propositions, *e.g.*, as a resource for modeling the heap of the program.

The modal weakest precondition connective is also parameterized by a modal operator and, indeed, one of the key strengths of the the connective is its generality and the fact that instantiations of it automatically inherit a set of basic structural proof rules (*cf.* Figure 6.9) that hold irrespective of the particular modality and programming language. For a particular instantiation of the connective, one can then prove soundness of more specific proof rules, *e.g.*, for heap-manipulating operations (*cf.* Lemma 6.2.1) and for the interaction with other instantiations with different modal operators (*cf.* Lemma 6.2.4).

In this work, we will use the generality of modal weakest preconditions to reason about the λ_{sec} programming language: we will use three different instantiations with three different modalities for our logical-relations model; in fact, one of these modalities will be defined in terms of an earlier instantiation. We start by giving a simplified presentation before giving the definition in its full generality. Finally, we use the theory of modal weakest preconditions in the subsequent Section 6.2.3 to define and reason about the expression relations of our logical-relations model.

Modal Weakest Precondition (simplified)

We define a predicate $\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\ e\ \{\Phi\}$ which intuitively says that if program e reduces to a value v in n steps then $\Phi(v, n)$ holds under modality M. The predicate is parameterized over a mask $\mathcal{E} \in \mathit{Masks} = \wp(\mathit{InvName})$ and the modality $\mathsf{M} : \mathit{Masks} \to \mathbb{N} \to \mathit{iProp} \to \mathit{iProp}$. The modality M is indexed by a mask $\mathcal{E}$ and a natural number n. The invariant names in $\mathcal{E}$ are those invariants the modality may allow to be opened, if the modality allows the use of invariants at all. The number n is the number of logical steps that the modality allows which we tie to the physical steps of the program execution in the definition of $\mathsf{mwp}^{\mathcal{M}}_{\mathcal{E}}\ e\ \{\Phi\}$; the preliminary definition is as follows:

$$\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\ e\ \{\Phi\} \triangleq \forall \sigma_1, \sigma_2, v, n.\ (e, \sigma_1) \to^n (v, \sigma_2) \mathrel{-\!\!*} S(\sigma_1) \mathrel{-\!\!*} \mathsf{M}_{\mathcal{E};n}(\Phi(v, n) * S(\sigma_2)).$$

The predicate expresses that if (e, σ_1) reduces to (v, σ_2) in n steps and $S(\sigma_1)$ holds then under modality M both $\Phi(v, n)$ and $S(\sigma_2)$ will hold. Note that the predicate does not require that the program is *safe* to execute, nor that it terminates. In particular, if the program gets stuck or diverges then $\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\ e\ \{\Phi\}$ holds trivially.

The connective can be used for a range of different modalities; we only require that the modality M satisfies two conditions:

$$\mathcal{E} \subseteq \mathcal{E}' \Rightarrow P \mathrel{-\!\!*} Q \vdash \mathsf{M}_{\mathcal{E};n}(P) \mathrel{-\!\!*} \mathsf{M}_{\mathcal{E}';n}(Q) \qquad \text{(monotone)}$$
$$\mathsf{M}_{\mathcal{E};0}(P) \vdash \mathsf{M}_{\mathcal{E};n}(P) \qquad \text{(introducable)}$$

We say that the modality M is *valid* if it satisfies the two conditions.

Given a valid modality, the $\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\ e\ \{\Phi\}$ predicate satisfies several general structural rules. We present a selection of such rules in Figure 6.9. Most of these are self-explanatory, but we point out the importance of the MWP-BIND rule which is crucial for local reasoning.

$$\frac{\text{MWP-PURE-STEP}\quad \forall \sigma.\, (\sigma, e) \to (\sigma', e') \qquad \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e'\, \{\Phi\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}} \qquad \frac{\text{MWP-VALUE}\quad \mathsf{M}_{\mathcal{E};0}(\Phi(v, 0))}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, v\, \{\Phi\}}$$

$$\frac{\text{MWP-MONO}\quad \forall v, n.\, \Phi(v, n) \mathrel{-\!\!*} \Psi(v, n) \qquad \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Psi\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}} \qquad \frac{\text{MWP-MASK-MONO}\quad \mathcal{E} \subseteq \mathcal{E}' \qquad \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e\, \{\Phi\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}'}\, e\, \{\Phi\}}$$

$$\frac{\text{MWP-BIND}\quad \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, e \left\{ v, n.\, \mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, K[v]\, \{w, m.\, \Phi(w, n + m)\} \right\}}{\mathsf{mwp}^{\mathsf{M}}_{\mathcal{E}}\, K[e]\, \{\Phi\}}$$

Figure 6.9: Excerpt of rules for the modal weakest precondition connective given a valid modality.

Example 1 (MWP instance: Update modality). Let $\mathsf{M}^{\Rightarrow}_{\mathcal{E},n}(P) \triangleq \Rightarrow_{\mathcal{E}} P$. This is a valid modality. The modality does not allow any logical steps (and ignores its index n). When proving $\mathsf{mwp}^{\mathsf{M}^{\Rightarrow}}_{\mathcal{E}}\, e\, \{\Phi\}$, however, all invariants in $\mathcal{E}$ may be opened before establishing the post condition Φ but must be immediately closed.

The simplified presentation given so far suffices for defining the modal weakest precondition instance that we will use for the unary expression interpretation. This is the point of the following example.

Example 2 (MWP instance: Step-taking update modality). Let

$$\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}(P) \triangleq \left({}_{\mathcal{E}}\Rightarrow_{\emptyset} \triangleright\, {}_{\emptyset}\Rightarrow_{\mathcal{E}}\right)^n \Rightarrow_{\mathcal{E}} P.$$

where $\left({}_{\mathcal{E}}\Rightarrow_{\emptyset} \triangleright\, {}_{\emptyset}\Rightarrow_{\mathcal{E}}\right)^n$ is n times repetition of ${}_{\mathcal{E}}\Rightarrow_{\emptyset} \triangleright\, {}_{\emptyset}\Rightarrow_{\mathcal{E}}$. The modality $\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}$ is valid and can be thought of as a *step-taking update modality*. Intuitively, $\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}(P)$ expresses that n steps into the future, we can update our resources to satisfy P, and, moreover, for every step, all invariants in $\mathcal{E}$ may be opened to reason about progress as long as they are immediately closed afterwards. In practice, the later modality allows stripping later modalities from assumptions that we get when opening invariants.

Using the structural rules for MWP in Figure 6.9, in particular the MWP-BIND rule, one can see that the modality distributes over compound expressions such that when proving $\mathsf{mwp}^{\mathsf{M}^{\Rightarrow\triangleright}}_{\mathcal{E}}\, e\, \{\Phi\}$, one is allowed to open invariants *atomically*, *i.e.*, for the duration of a single atomic step.

When instantiating the modal weakest precondition theory with λ_{sec} and the $\mathsf{M}^{\Rightarrow\triangleright}_{\mathcal{E},n}$ modality we can derive the following properties for reasoning about heap-manipulating operations.

Lemma 6.2.1 (Properties of step-taking update MWP with λ_{sec})**.**

1. $\triangleright \forall \ell.\, \ell \mapsto_i v \mathrel{-\!\!*} Q\, \ell \vdash \mathsf{mwp}^{\mathcal{M}^{\Rightarrow\triangleright}}_{\mathcal{E}}\, \mathsf{ref}(v)\, \{v.\, Q\}$
2. $\triangleright \ell \mapsto_i v * \triangleright(\ell \mapsto_i v \mathrel{-\!\!*} Q\, v) \vdash \mathsf{mwp}^{\mathcal{M}^{\Rightarrow\triangleright}}_{\mathcal{E}}\, !\ell\, \{v.\, Q\}$

3. $\triangleright \ell \mapsto_i v * \triangleright(\ell \mapsto_i w \mathrel{-\!\!*} Q\ ()) \vdash \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M}\Rrightarrow} \ell \leftarrow w\ \{v.\ Q\}$

Lemma 6.2.1 state properties that allow us to allocate, read, and modify the heap. They all express that the postcondition Q will hold if the resources needed are given and Q holds for the updated resources.

Modal Weakest Precondition (full definition)

The definition of the modal weakest precondition connective presented so far suffices for unary reasoning about programs. A specific instance of it has already been used in previous work by Timany et al. [Tim+18] who considered an instantiation with a so-called future modality. However, in order to facilitate termination-insensitive reasoning about two programs at the same time, we generalize the definition further such that we can use an MWP connective *as the modality* of another MWP connective. In Section 6.2.3, we will see how this general connective is particularly useful for defining and working with our binary logical-relations model.

The key idea behind the generalization is to let the modality—apart from the number of steps of the execution and the mask—have its own "state" embodied in an index and to let the proposition that the modality acts on be parameterized over some information provided by the modality. For unary reasoning, both of these indices will just be the unit type, meaning the modality has no state and provides no information to the postcondition (in which case we recover the simplified presentation from the above). However, when used for binary reasoning, as in our binary logical-relations model, the index of the modality will be the second program, and the postcondition parameter will be the return value of the second program.

Formally, we parameterize the modality by two types, A and B, and a predicate $\mathsf{BindCond}$ that determines when and how the modality will change "when the binding lemma applies" (explained below). We bundle these parameters together with the modality $\mathsf{M} : A \to \mathit{Masks} \to \mathbb{N} \to (B \to \mathit{iProp}) \to \mathit{iProp}$ as a tuple $\mathcal{M}$.

Definition 6.2.2 (Modal Weakest Precondition)**.** Let $\mathcal{M} = (A, B, \mathsf{M}, \mathsf{BindCond})$ and $a \in A$. Then

$$\mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};a}\ e\ \{\Phi\} \triangleq \forall \sigma_1, \sigma_2, v, n.\ (e, \sigma_1) \to^n (v, \sigma_2) \mathrel{-\!\!*} S(\sigma_1) \mathrel{-\!\!*} \mathsf{M}_{\mathcal{E};n}^{a}(\lambda b.\ \Phi(v, n, b) * S(\sigma_2)).$$

For the modality defined by $\mathcal{M}$ to be valid it has to satisfy the two conditions from above (monotonicity and introducability) and, moreover, whenever $\mathsf{BindCond}(a, a', f, g)$ holds, then we should also have

$$\mathsf{M}_{\mathcal{E};n}^{a'}(\lambda b.\ \mathsf{M}_{\mathcal{E};m}^{f(b)}(\lambda b'.\ \Phi(g(b, b')))) \vdash \mathsf{M}_{\mathcal{E};n+m}^{a}(\Phi). \qquad \text{(binding)}$$

Intuitively, $\mathsf{BindCond}(a, a', f, g)$ defines when and how the modality can be chained together through binding; a modality with index a and $n+m$ logical steps can be split into the sequence of the modality with index a' and n steps followed by the modality with index $f(b)$ and m steps given the postcondition is updated according to g. This will allow us to suitably generalize MWP-BIND to take into account the new indices and how the modality may evolve.

MWP-BIND-GEN

$$\frac{\mathsf{BindCond}(a, a', f, g) \qquad \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};a'}\ e \left\{v, n, b.\ \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};f(b)}\ K[v]\ \{w, m, b'.\ \Phi(w, n+m, g(b, b'))\}\right\}}{\mathsf{mwp}_{\mathcal{E}}^{\mathcal{M};a}\ K[e]\ \{\Phi\}}$$

The generalized modal connective allows us to use a modal weakest precondition connective as the modality of another modal weakest precondition. This not only allows us to define a relational predicate on two computations (as we will see below), but also to have a collection of proof rules (*cf.* Figure 6.9) for reasoning about the individual computations.

Example 3 (MWP instance: Binary step-taking update modality). The relational predicate used in our binary logical-relations model has the following shape when unfolding the definition:

$$\begin{aligned}\mathsf{mwp}_{\mathcal{E}}\, e_1 \sim e_2\, \{v, w.\, \Phi\} = \forall \sigma_1, \sigma_1', v, n.\, (e_1, \sigma_1) \to^n (v, \sigma_1') \mathrel{-\!\!*} S_1(\sigma_1) \mathrel{-\!\!*} \\ \forall \sigma_2, \sigma_2', w, m.\, (e_2, \sigma_2) \to^m (w, \sigma_2') \mathrel{-\!\!*} S_2(\sigma_2) \mathrel{-\!\!*} \\ \left({}_{\mathcal{E}}\Rightarrow_{\emptyset} \triangleright {}_{\emptyset}\Rightarrow_{\mathcal{E}}\right)^{n+m} \Rightarrow_{\mathcal{E}} \left(\Phi(v, w) * S_1(\sigma_1') * S_2(\sigma_2')\right) \end{aligned} \tag{6.4}$$

and is, as it seems, a binary version of the instance from Example 2. Intuitively, if e_1 terminates in n steps with value v and e_2 terminates in m steps with value w then $n + m$ steps into the future, we can update our resources to satisfy $\Phi(v, w)$ while being able to open all invariants in $\mathcal{E}$ atomically during every step. Note that this is a *termination-insensitive* relation; we assume both relations terminate and then the postcondition should hold. This is in contrast to the earlier relational models in Iris which have been *termination-sensitive* and definable using the standard weakest preconditions of Iris. Moreover, notice that we count the steps taken on *both* sides of the relation by including later modalities for both executions—Rajani and Garg [RG20] and earlier relational models in Iris only count steps for one of the programs.

Formally, we define this predicate using two modal weakest precondition instances where the latter is defined in terms of the former. We use this approach rather than defining the binary predicate directly as it will allow us to re-use the proof rules for modal weakest preconditions to reason about the individual programs when arguing binary relatedness. The definitions are somewhat technical and can easily be skipped on a first reading.

Let $\mathcal{M}_I$ be defined by

$$\begin{aligned}\mathsf{M}^m_{\mathcal{E};n}(\Phi) &\triangleq \left({}_{\mathcal{E}}\Rightarrow_{\emptyset} \triangleright {}_{\emptyset}\Rightarrow_{\mathcal{E}}\right)^{n+m} \Rightarrow_{\mathcal{E}} \Phi() \\ \mathsf{BindCond}(n, m, f, g) &\triangleq m \leq n \wedge \forall x, f(x) = m - n \wedge \lambda\_, g = id.\end{aligned}$$

The modality's index is a natural number m that adds m extra logical steps to the step-taking update modality and the postcondition parameter is unit. The bind condition ensures that the logical steps "add up" and that the post condition is otherwise unmodified. The modality defined by $\mathcal{M}_I$ is valid.

Now, let $\mathcal{M}_{\times\Rightarrow\triangleright}$ be defined by

$$\begin{aligned}\mathsf{M}^e_{\mathcal{E};n}(\Phi) &\triangleq \mathsf{mwp}^{\mathcal{M}_I;n}_{\mathcal{E}}\, e\, \{w, m.\, \Phi(w, m)\} \\ \mathsf{BindCond}(e_1, e_2, f, g) &\triangleq \exists K.\, e_1 = K[e_2] \wedge\ g = \lambda(v_1, n_1), (v_2, n_2).(v_2, n_1 + n_2) \wedge \\ &\qquad \forall v, k.\, f(v, k) = K[v].\end{aligned}$$

The modality is the MWP connective instantiated with $\mathcal{M}_I$. The bind condition reflects the preconditions for the binding lemma for the inner connective and that the steps taken are propagated to the postcondition. The modality defined by $\mathcal{M}_{\times\Rightarrow\triangleright}$ is valid. We now define $\mathsf{mwp}_{\mathcal{E}}\, e \sim e'\, \{v, w.\, \Phi\}$ to be $\mathsf{mwp}^{\mathcal{M}_{\times\Rightarrow\triangleright}, e'}_{\mathcal{E}}\, e\, \{v, \_, (w, \_).\, \Phi\}$; by unfolding the definitions one can see that it indeed satisfies the desired relational predicate in Equation (6.4).

A crucial property of the binary relation defined in the above example is the following *binary* version of the bind rule, which intuitively means that we can do *relational* reasoning in a local way.

Lemma 6.2.3 (Binary step-taking update MWP - bind)**.**

$$\frac{\mathsf{mwp}\ e \sim e' \left\{v, v'.\ \mathsf{mwp}\ K[v] \sim K'[v']\ \{\Phi\}\right\}}{\mathsf{mwp}\ K[e] \sim K'[e']\ \{\Phi\}}$$

At the same time, essential for our logical-relations model, we have all the proof rules for reasoning about the two computations individually. This is embodied in Lemma 6.2.4 that allows us to reason about each computation using the unary modal weakest precondition instance from Example 2.

Lemma 6.2.4 (Unary-binary step-taking update MWP)**.**

$$\mathsf{mwp}_{\mathcal{E}}^{\mathcal{M}\Rrightarrow} e_1 \left\{v.\ \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M}\Rrightarrow} e_2\ \{w.\ \Phi(v,w)\}\right\} \mathrel{-\!\!*} \mathsf{mwp}\ e_1 \sim e_2\ \{\Phi\}$$
$$\mathsf{mwp}_{\mathcal{E}}^{\mathcal{M}\Rrightarrow} e_2 \left\{w.\ \mathsf{mwp}_{\mathcal{E}}^{\mathcal{M}\Rrightarrow} e_1\ \{v.\ \Phi(v,w)\}\right\} \mathrel{-\!\!*} \mathsf{mwp}\ e_1 \sim e_2\ \{\Phi\}$$

Recall that the modal weakest precondition connective is defined as a proposition in Iris of type *iProp*. To demonstrate that the theory actually makes the expected statements about program execution *in the meta logic*, once and for all, for *any* language and for *any* modality, we prove a general adequacy theorem for the modal weakest precondition theory. The details of this general theorem is relegated to the Coq formalization. For concrete languages and modalities, specific adequacy theorems such as the following hold as simple corollaries.

Theorem 6.2.5 (Adequacy of binary step-taking update MWP with λ_{sec})**.** *Let φ be a first-order (meta-logic) predicate over values. Suppose* $\mathsf{mwp}_{\mathcal{E}}\ e_1 \sim e_2\ \{\varphi\}$ *is derivable. If* $(\sigma_1, e_1) \to^* (\sigma_1', v_1)$ *and* $(\sigma_2, e_2) \to^* (\sigma_2', v_2)$ *then* $\varphi(v_1, v_2)$ *holds at the meta-level.*

6.2.3 Expression relations

We now return to the expression relations of our logical-relations model, which are defined using modal weakest preconditions; see Figure 6.8 and Figure 6.6.

The binary interpretation relates expressions at τ that only terminate with related values at τ. This is defined directly using the binary connective derived in Example 3.

$$\mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e, e') \triangleq \mathsf{mwp}\ e_1 \sim e_2\ \{[\![\tau]\!]^{\rho}_{\Theta}\}.$$

Recall that the unary interpretation is intended to be inhabited by terms that have no ζ-observable side-effects. We observe that only expressions in high-labeled contexts (where control flow depends on high-labeled data) are critical; low-labeled contexts are ζ-observable and should not be considered. To incorporate this observation, the unary expression relation is annotated with a pc label. The unary value interpretation of the function type and the polymorphic types pass on the latent effect label as this parameter. If pc is a high label ($[\![pc]\!]_{\rho} \not\sqsubseteq \zeta$), then e is in the expression interpretation of τ if e satisfies the unary modal weakest precondition from Example 2.

$$\mathcal{E}_{pc}[\![\tau]\!]^{\rho}_{\Delta}(e) \triangleq [\![pc]\!]_{\rho} \not\sqsubseteq \zeta \Rightarrow \mathsf{mwp}^{\mathcal{M}\Rrightarrow} e\ \{[\![\tau]\!]^{\rho}_{\Delta}\}.$$

With the value and expression relations for closed values and expressions defined, logical relatedness for open terms is now defined by closing them by related substitutions, as is usual for logical relations. Substitutions are related using the environment relation interpretations denoted $\mathcal{G}$ in Figure 6.8 and Figure 6.6.

The *unary semantic typing judgment* (logical relation) is defined as

$$\Xi \mid \Psi \mid \Gamma \vDash_{pc} e : \tau \triangleq \Box \left(\begin{array}{c} \forall \Delta, \rho, \vec{v}.\, \mathsf{dom}(\Xi) \subseteq \mathsf{dom}(\Delta) * \mathsf{dom}(\Psi) \subseteq \mathsf{dom}(\rho) \mathrel{-\!\!*} \\ \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Delta}(\vec{v}) \mathrel{-\!\!*} \mathcal{E}_{pc}[\![\tau]\!]^{\rho}_{\Delta}(e[\vec{v}/\vec{x}]) \end{array} \right)$$

and the *binary semantic typing judgment* as

$$\Xi \mid \Psi \mid \Gamma \vDash e \approx_{\zeta} e' : \tau \triangleq \\ \Box \left(\begin{array}{c} \forall \Theta, \rho, \vec{v}, \vec{v'}.\, \mathsf{dom}(\Xi) \subseteq \mathsf{dom}(\Theta) * \mathsf{dom}(\Psi) \subseteq \mathsf{dom}(\rho) \mathrel{-\!\!*} \\ \mathit{Coh}(\Theta) * \mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta}(\vec{v}, \vec{v'}) \mathrel{-\!\!*} \mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}]) \end{array} \right).$$

Notice that the binary judgment additionally requires the semantic type environment to be coherent.

6.3 Fundamental theorems and soundness

It is straightforward to show the unary fundamental theorem by structural induction on the typing derivation. All proofs are carried out at an abstraction level similar to the structural rules shown in this chapter. This is enabled by our formulation of the modal weakest precondition theory and the MoSeL framework [Kre+18] for manipulating the Iris connectives.

Theorem 6.3.1 (Unary fundamental theorem)**.**

$$\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau \Rightarrow \Xi \mid \Psi \mid \Gamma \vDash_{pc} e : \tau$$

Similarly, the binary fundamental theorem also follows by structural induction in the typing derivation and the structural rules of the binary modal weakest precondition and its interaction with the unary modal weakest precondition. However, it also relies heavily on the binary-unary subsumption property which is the content of the following lemma.

Lemma 6.3.2 (Binary-unary subsumption)**.**

$$\mathit{Coh}(\Theta) * [\![\tau]\!]^{\rho}_{\Theta}(v, v') \mathrel{-\!\!*} [\![\tau]\!]^{\rho}_{\Theta_L}(v) * [\![\tau]\!]^{\rho}_{\Theta_R}(v').$$

To see why this property is crucial and to exemplify how the binary and unary relations interact, consider the compatibility lemma for conditional expressions. This lemma concludes

$$\Xi \mid \Psi \mid \Gamma \vDash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \approx_{\zeta} \mathsf{if}\ e'\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2' : \tau$$

given well-typed sub-terms and $\Xi \mid \Psi \mid \Gamma \vDash e \approx_{\zeta} e' : \mathbb{B}^{\iota}$, $\Xi \mid \Psi \mid \Gamma \vDash e_i \approx_{\zeta} e_i' : \tau$, and $\tau \searrow \iota$, *cf.*, T-IF for conditional expressions in Figure 6.2.

Unfolding the definition of the binary semantic typing judgment, this means that given related substitutions $\vec{v}$ and $\vec{v'}$, *i.e.*, $\mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta}(\vec{v}, \vec{v'})$, and $\mathit{Coh}(\Theta)$ it suffices to show

$$\mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(\mathsf{if}\ e[\vec{v}/\vec{x}]\ \mathsf{then}\ e_1[\vec{v}/\vec{x}]\ \mathsf{else}\ e_2[\vec{v}/\vec{x}], \mathsf{if}\ e'[\vec{v'}/\vec{x}]\ \mathsf{then}\ e_1'[\vec{v'}/\vec{x}]\ \mathsf{else}\ e_2'[\vec{v'}/\vec{x}]).$$

The proof continues by considering whether label ι of the branch condition is ζ-observable or not, *i.e.*, whether $\iota \sqsubseteq \zeta$ or $\iota \not\sqsubseteq \zeta$. In the case where the label is *not* observable this means that given $\tau = t^{\iota'}$ then $\iota' \not\sqsubseteq \zeta$ as well and hence the values that e and e' evaluate to are (potentially) different, *cf.*, the binary value interpretation of labeled Booleans. In turn, this means evaluation of the two conditional expressions might continue through different branches, *i.e.*, we end up having to show $\mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e_1[\vec{v}/\vec{x}], e_2'[\vec{v'}/\vec{x}])$ and $\mathcal{E}[\![\tau]\!]^{\rho}_{\Theta}(e_2[\vec{v}/\vec{x}], e_1'[\vec{v'}/\vec{x}])$.

Using Lemma 6.2.4 we can reason about the two expressions individually, and the statements follow from the unary fundamental theorem (Theorem 6.3.1). However, the assumption $\mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta}(\vec{v}, \vec{v'})$ on substitutions $\vec{v}$ and $\vec{v'}$ is *binary*—in order to use the unary fundamental theorem, the related substitutions individually need to satisfy the unary environment interpretations, *i.e.*, $\mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta_L}(\vec{v})$ and $\mathcal{G}[\![\Gamma]\!]^{\rho}_{\Theta_R}(\vec{v'})$. This fact follows from Lemma 6.3.2.

Theorem 6.3.3 (Binary fundamental theorem)**.**

$$\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \tau \Rightarrow \Xi \mid \Psi \mid \Gamma \vDash e \approx_{\zeta} e : \tau.$$

By composing the binary fundamental theorem and the adequacy theorem for the binary modal weakest precondition instance (Theorem 6.2.5) we show our final soundness theorem, which shows that our type system does indeed imply termination-insensitive noninterference.

Theorem 6.3.4 (Termination-Insensitive Noninterference)**.** *Let ζ, $\top$ and $\bot$ be labels drawn from a join-semilattice such that $\bot \sqsubseteq \zeta$ and $\top \not\sqsubseteq \zeta$. If*

$$\cdot \mid \cdot \mid x : \mathbb{B}^{\top} \vdash_{\bot} e : \mathbb{B}^{\bot},$$
$$\cdot \mid \cdot \mid \cdot \vdash_{\bot} v_1 : \mathbb{B}^{\top}, \text{ and } \cdot \mid \cdot \mid \cdot \vdash_{\bot} v_2 : \mathbb{B}^{\top}$$

then

$$(\emptyset, e[v_1/x]) \rightarrow^* (\sigma_1, v_1') \wedge (\emptyset, e[v_2/x]) \rightarrow^* (\sigma_2, v_2') \Rightarrow v_1' = v_2'.$$

6.4 Examples of semantic typing

By the soundness theorem (Theorem 6.3.4) we now know that any syntactically well-typed program satisfies noninterference. Our model also allows us to *semantically* type programs that are not syntactically well-typed but are nevertheless secure, for reasons too subtle for the syntactic type system to discover. Semantically well-typed programs can then be safely composed with *syntactically* well-typed programs while maintaining noninterference. To see how this works in practice, we will first examine a few small programs that are safe to execute but untypable in our static type system. Later, we will move on to more intricate examples and show how we can prove that these are secure and therefore also safe to compose with other syntactically typed programs. The examples in this section thus illustrate some of the strengths of our semantic approach to noninterference. The proofs of the examples rely both on our semantic model of types (in particular abstract types) and also on our ability to use Iris ghost state and Iris invariants to reason about intricate invariants on local state. Proof can be found in the accompanying Coq formalization.

In the following examples we will often omit labels on composite types to simplify the presentation. An omitted label should always be read as being label $\bot$.

To start off, consider the trivial program $\lambda\, v.\, v * 0$ that multiplies its input by zero. Syntactically, it cannot be typed at $\mathbb{N}^{\top} \to \mathbb{N}^{\bot}$ as its output seemingly depends on its input which is at a high security label. But, by simple arithmetic, the output is always constant and the function can thus be shown to be in the semantic interpretation $[\![\mathbb{N}^{\top} \to \mathbb{N}^{\bot}]\!]$ of the type. Hence it can be safely composed with any syntactically well-typed code that relies on a function of this type.

Next, consider the programs shown in Equation (6.5) and Equation (6.6)

$$\mathsf{let}\ x = !\,\ell\ \mathsf{in}\ \ell \leftarrow !\,h;\ \ldots;\ \ell \leftarrow x \tag{6.5}$$

$$(\mathsf{if}\ !\,h = 42\ \mathsf{then}\ \ell \leftarrow 0\ \mathsf{else}\ \ell \leftarrow 1);\ \ell \leftarrow 0 \tag{6.6}$$

which both *temporarily* store information (both explicitly and implicitly) from a sensitive reference h into a public reference ℓ. Due to the flow-insensitive nature of the syntactic type system, both of the programs cannot be type checked, as sensitive information is not allowed to flow into a public reference. However, by restoring public information in the reference, both programs are in fact secure. In both cases, location ℓ inhabits type $\mathsf{ref}(\mathbb{N}^{\bot})^{\bot}$ which, *cf.* Figure 6.6, means that its contents must invariantly be binary related at $\mathbb{N}^{\bot}$. To prove that these examples are semantically well-typed, it is necessary to keep the invariant open for the full execution of the program. Recall that the modal weakest precondition (Example 3) used to define the binary expression relation only allows invariants to be opened atomically during every step, so it seems that it might be difficult to show semantic well-typedness. But, fortunately, we can prove semantic well-typedness of these examples using a binary version of the modal weakest precondition instance from Example 1 that allows invariants to stay open for the full execution of the program.

Lemma 6.4.1 (Binary update MWP implies binary step-update MWP)**.** *If either e_1 or e_2 are able to make progress then*

$$\left({}_{\mathcal{E}_1}\!\Rrightarrow_{\mathcal{E}_2} \triangleright\, \mathsf{mwp}_{\mathcal{E}_2}^{\mathcal{M}_{\times \Rrightarrow ; e_2}}\, e_1\, \{v, n, b.\ {}_{\mathcal{E}_2}\!\Rrightarrow_{\mathcal{E}_1} \Phi(v, n, b)\}\right) \mathrel{-\!\!*} \mathsf{mwp}_{\mathcal{E}_1}\, e_1 \sim e_2\, \{\Phi\}.$$

We relegate the details to the Coq formalization, however, we emphasize that this example illustrates the generality and flexibility offered by our modal weakest precondition theory.

6.4.1 *Static semantic typing instead of dynamic enforcement*

We now consider an example adapted from Fennell and Thiemann [FT13], namely a report processing application containing security-typed operations that process reports by reference. The example contains code fragments that the type system of Fennell and Thiemann cannot statically type check. Instead, they propose to use a gradual security type system where security levels are checked at run-time. Those code fragments cannot be type checked by our syntactic type system either, but we *can* prove that they are semantically well-typed. Not only does this prove the example secure, it also avoids unnecessary run-time cost while still allowing the code to be composed with the remainder of the syntactically well-typed report processing application.

The basic operations of the report processing application include

$$\mathit{sendToManager} : \mathsf{ref}(\mathit{Report}^{\top}) \xrightarrow{\top} 1$$

$$\mathit{sendToFacebook} : \mathsf{ref}(\mathit{Report}^{\bot}) \xrightarrow{\bot} 1$$

where the idea is that *sendToManager* can process sensitive reports and send those to trusted managers (by assigning to a reference, *cf.* the $\top$ latent effect label) whereas *sendToFacebook* can only process public reports and thus has a $\bot$ latent effect label.

Fennell and Thiemann consider an extension of the application with a utility function *addPrivileged*, which adds privileged information to a report before passing it to a *worker* (like one of the basic operations in the above):

$$\begin{aligned} \mathit{addPrivileged} \triangleq \lambda\, \mathit{isPrivileged}, \mathit{worker}, \mathit{report}.\ & \\ & \mathsf{if}\ \mathit{isPrivileged}\ \mathsf{then}\ \mathit{report} \leftarrow\, !\,\mathit{report} + !\,h\ \mathsf{else}\ () \\ & \mathit{worker}\ \mathit{report} \end{aligned}$$

The flag *isPrivileged* indicates whether the *worker* argument has a sufficient security level to handle a privileged report. If the flag is true, sensitive information is retrieved from a global reference h and appended to the report. Otherwise, the *worker* is invoked with an unmodified report. Both *addPrivileged* itself and the application *addPrivileged* true *sendToManager* syntactically type checks, the former at the type $\mathsf{ref}(\mathit{Report}^{\top}) \xrightarrow{\top} 1$, as *sendToManager* can safely operate on sensitive information. However, the code fragment *addPrivileged* false *sendToFacebook* does *not* type check, even though it is safe and no sensitive information is leaked. The code does not type check because the type system does not track the dependency between the *isPrivileged* flag and the *worker*'s security clearance. Using our model, however, we can prove that it can be semantically typed at type $\mathsf{ref}(\mathit{Report}^{\bot}) \xrightarrow{\bot} 1$, meaning that the code can be composed with other syntactically well-typed report operations, without introducing any runtime labels.

Proposition 6.4.2. *Let* $\mathit{addPFB} \triangleq \mathit{addPrivileged}\ \mathsf{false}\ \mathit{sendToFacebook}$ *then*

$$\cdot \mid \cdot \mid \cdot \vDash \mathit{addPFB} \approx_{\zeta} \mathit{addPFB} : \mathsf{ref}(\mathit{Report}^{\bot}) \xrightarrow{\bot} 1^{\bot}$$

The proof is straightforward and follows by symbolic execution of the program.

6.4.2 *Value-dependent classification and modularity*

Traditionally, information-flow control systems partition the heap into compartments for each security level. This is impractical for realistic settings where resources, such as the heap, can be shared. To address this issue, some recent information-flow systems [GTA19; LC15; Mur+16; NBG11; ZM07] support *value-dependent classification* policies. These policies describe a relationship between two values, such that the value of one decides the classification-level of the other. We now demonstrate that our semantic model supports reasoning about value-dependent classification policies; we also use this example to show an application of existential types to increase modularity by hiding the value dependency.

Consider the example of a program with value-dependent classification below.

$$\begin{aligned} \mathit{valDep} \triangleq \lambda\, f.\ & \mathsf{let}\ \mathit{dep} = \mathsf{ref}(\mathsf{true}, \mathit{secret})\ \mathsf{in} \\ & f\ \mathit{dep}; \\ & \mathsf{let}\ \mathit{tmp} = !\,\mathit{dep}\ \mathsf{in} \\ & \mathsf{if}\ \pi_1\ \mathit{tmp}\ \ \mathsf{then}\ 42\ \mathsf{else}\ \pi_2\ \mathit{tmp} \end{aligned}$$

The program allocates a reference *dep* which points to a pair consisting of a Boolean and a number. If the Boolean is true, the contents of the second component should be regarded as secret; otherwise public. The reference is passed to the function f which therefore must uphold this invariant for the program to be secure. Finally, the contents of *dep* is inspected and if the Boolean is true (*i.e.*, the content is secret), we ignore the second component and return 42 and otherwise it is safe to return the second component. Ideally, we would like to show that given a function f, the pair $(\mathit{valDep}\ f, \mathit{valDep}\ f)$ is in the binary interpretation $[\![\mathbb{N}^{\bot}]\!]$. Obviously this does not hold for an arbitrary function f; to prove it we need to know that f maintains the following invariant on *dep*:

$$\exists b, d_L, d_R. \mathit{dep}_L \mapsto_L (b, d_L) * \mathit{dep}_R \mapsto_R (b, d_R) * [\![\mathbb{N}^{\mathsf{if}\ b\ \mathsf{then}\ \top\ \mathsf{else}\ \bot}]\!](d_L, d_R)$$

The invariant ensures that f cannot write a secret to *dep* without also setting the Boolean to true.

This example shows how we can encode value-dependent classifications in our system but with the cost of burdening the client of the above program with showing that f upholds the invariant. The issue is that the client's code gets direct access to the reference with the classification, but the static type system is oblivious to the semantic meaning of it.

To alleviate this problem, we can instead hide the reference in an existential package. This allows us to only expose accessor- and mutator methods to the client, such that the client only needs to statically type check against these methods. The code for this variant is seen below.

$$\begin{aligned}
\mathit{valDepPack} \triangleq\ & \mathsf{let}\ \mathit{get} = \lambda\ \mathit{dep}. \\
& \quad \mathsf{let}\ c = !\ \mathit{dep}\ \mathsf{in}\ \mathsf{if}\ \pi_1\ c\ \mathsf{then}\ \mathsf{inj}_1\ (\pi_2\ c)\ \mathsf{else}\ \mathsf{inj}_2\ (\pi_2\ c)\ \mathsf{in} \\
& \mathsf{let}\ \mathit{setL} = \lambda\ \mathit{dep}, v.\ \mathit{dep} \leftarrow (\mathsf{false}, v)\ \mathsf{in} \\
& \mathsf{let}\ \mathit{setH} = \lambda\ \mathit{dep}, v.\ \mathit{dep} \leftarrow (\mathsf{true}, v)\ \mathsf{in} \\
& \mathsf{pack}\ (\mathsf{ref}(\mathsf{true}, \mathit{secret}), \mathit{get}, \mathit{setL}, \mathit{setH})
\end{aligned}$$

Using our semantic model, we can prove that this program inhabits an existential type.

Proposition 6.4.3.

$$\cdot \mid \cdot \mid \cdot \vDash \mathit{valDepPack} \approx_{\zeta} \mathit{valDepPack} : \exists \alpha.\ \left(\alpha^{\bot} \times \left(\alpha^{\bot} \xrightarrow{\top} \mathbb{N}^{\top} + \mathbb{N}^{\bot}\right) \times \left(\alpha^{\bot} \xrightarrow{\top} \mathbb{N}^{\bot} \xrightarrow{\bot} 1\right) \times \left(\alpha^{\bot} \xrightarrow{\top} \mathbb{N}^{\top} \xrightarrow{\bot} 1\right)\right)$$

This allows statically typed clients to store both secret and public information in the reference, but they must do so through the mutators *setL* and *setH*. When clients want to read the reference, they can do so with the *get* function which gives a value of type $\mathbb{N}^{\top} + \mathbb{N}^{\bot}$.

6.4.3 Computing with memoization

The following example shows an implementation of a service for computing a function with memoization. The service takes a function f as input and then allows clients to compute f on client-provided inputs; when doing so, the service remembers the last input and corresponding result and returns this directly if the client asks for the same input again. The idea, of course, would be that the function f is very expensive to compute, so the client would therefore like to memoize the already computed values in case they are needed again. This behavior is

implemented with a single reference that points to a tuple consisting of the last input value and the corresponding result. The code for this service is shown below.

$$
\begin{array}{l}
\mathit{memoize} \triangleq \lambda\, f, \mathit{init}. \\
\quad \mathsf{let}\ \mathit{cache} = \mathsf{ref}(\mathit{init}, f\, \mathit{init})\ \mathsf{in} \\
\quad \mathsf{let}\ \mathit{recompute} = \lambda\, v.\ \mathsf{let}\ \mathit{result} = f\, v\ \mathsf{in}\ \mathit{cache} \leftarrow (v, \mathit{result}); \mathit{result}\ \mathsf{in} \\
\quad \lambda\, v.\ \mathsf{let}\ (w, \mathit{result}) = !\, \mathit{cache}\ \mathsf{in} \\
\qquad \mathsf{if}\ v = w\ \mathsf{then}\ \mathit{result}\ \mathsf{else}\ \mathit{recompute}\ v
\end{array}
$$

First, let us see why we cannot give a static type to this program. Suppose we have a function f of type $\mathbb{N}^{\bot} \xrightarrow{\top} \mathbb{N}^{\bot}$. The issue then is giving a reasonable type to the reference for the cache. If we type it at $\bot$, then the returned function will necessarily have a latent effect label at $\bot$, and it is therefore not interchangeable with the input function. If we instead type the reference at $\top$, then we must label the output of the resulting function to $\top$ as well.

Clearly, we cannot hope to give a reasonable type to this program using our static type system, so we will instead try to define a security condition for it. For a suitable function f from $\mathbb{N}^{\iota}$ to $\mathbb{N}^{\iota}$, we would like to show that $\mathit{memoize}\, f\, 0$ has the semantic type $[\![\mathbb{N}^{\iota} \xrightarrow{\top} \mathbb{N}^{\iota}]\!]$, so any well-typed client can use this to compute f with caching.

Note that the latent effect label of the returned function is $\top$ even though the function writes to the cache. The secrecy of the cache itself is independent of the secrecy of the outputs of the function f, but instead varies based on the secrecy of the context the last call that updated the cache happened in.

For this to be secure, *memoize* relies on the input function f to "act" purely. Intuitively, the function must behave as if it was a pure function on all terminating inputs. This rules out programs such as the following that tries to exploit the memoization by counting the number of times f has been called:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{counter} = \mathsf{ref}(0)\ \mathsf{in} \\
\mathsf{let}\ f' = \mathit{memoize}\ (\lambda\, \_.\ \mathit{counter} \leftarrow (!\, \mathit{counter} + 1); !\, \mathit{counter})\ 0\ \mathsf{in} \\
\mathsf{if}\ \mathit{secret}\ \mathsf{then}\ f'\, 0\ \mathsf{else}\ (); \\
f'\, 0
\end{array}
$$

This allows us to prove that $\mathit{memoize}\, f\, 0$ is semantically secure and we can therefore link this with any piece of statically typed code that makes use of this function with memoization, while maintaining security of the whole program.

Proposition 6.4.4. *For any purely acting function f from $\mathbb{N}^{\iota}$ to $\mathbb{N}^{\iota}$, we have that*

$$\cdot \mid \cdot \mid \cdot \vDash \mathit{memoize}\, f\, 0 \approx_{\zeta} \mathit{memoize}\, f\, 0 : \mathbb{N}^{\iota} \xrightarrow{\top} \mathbb{N}^{\iota}$$

6.4.4 Higher-order functions and dynamicly allocated references

Consider the following variation by Frumin et al. [FKB21a] of the "awkward" example, originally given by Pitts and Stark [PS98] when studying the challenges of proving contextual equivalence about higher-order functions and state:

$$\mathit{awk} \triangleq \lambda\, v.\ \mathsf{let}\ x = \mathsf{ref}(v)\ \mathsf{in}\ \lambda\, f.\ x \leftarrow 1; f\,(); !\, x$$

When applied to a value v, the program returns a closure that, when invoked with a function f, always returns the constant value 1. From an information-flow control perspective this means that even if *awk* is invoked with a sensitive argument, it will always be safe to consider the output of the closure as public. This fact crucially relies on the reference x being local to the closure. The program is not well-typed using the syntactic type system as x has to contain both sensitive and public values. However, we can semantically type *awk*.

Proposition 6.4.5. $\cdot \mid \cdot \mid \cdot \vDash \mathit{awk} \approx_\zeta \mathit{awk} : \mathbb{N}^\top \xrightarrow{\bot} (1 \xrightarrow{\bot} 1) \xrightarrow{\bot} \mathbb{N}^\bot$

To prove that the contents of the reference is in fact public after invoking the function, we use an invariant with a two-state protocol (defined using Iris ghost state) on the contents of the reference; see the accompanying Coq code for more details.

6.4.5 *Parametricity and free theorems*

We can use our model to prove free theorems; here are two simple examples. As far as we know, such properties have not been shown for information-flow control type systems before.

Proposition 6.4.6. *If* $\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \forall_{\iota_1} \alpha.\, \alpha^{\iota_2} \xrightarrow{\iota_3} \alpha^{\iota_2}$ *and* $(\sigma, e\ \_\ v) \to^* (\sigma', v')$ *then* $v = v'$.

Proposition 6.4.7. *There does not exist a non-diverging* e *where*

$$\Xi \mid \Psi \mid \Gamma \vdash_{pc} e : \forall_\top \alpha.\, \alpha^\top \xrightarrow{\top} \alpha^\bot$$

6.5 *Related work*

Logical relations for information-flow security. Sabelfeld and Sands [SS99; SS01] present a model of information-flow security based on partial-equivalence relations; they establish various semantic properties about the model and use it to prove a termination-sensitive notion of noninterference for a calculus equipped with a simple security type system, first-order state, and probabilistic choice. Zdancewic [Zda02] proves a security-typed simply-typed lambda calculus sound using a logical-relations argument but uses a translation-based argument when considering mutable state. Abadi et al. [Aba+99] introduce *the dependency core calculus* (DCC), a pure calculus designed to capture the central notion of dependency arising in a setting like information-flow security. They prove noninterference using a denotational semantics based on partial equivalence relations. Heintze and Riecke [HR98] also prove noninterference of the pure fragment of the SLam calculus using logical relations. Pottier and Conchon [PC00] conjecture that the noninterference proofs of Abadi et al. [Aba+99] and Heintze and Riecke [HR98] cannot easily deal with recursive or polymorphic types. Compared to our work, all of the above consider simpler settings with respect to language features and type systems.

Using Iris, Frumin et al. [FKB21a] present a separation logic for proving a timing-sensitive notion of noninterference for concurrent programs. On top of this logic, they build a logical-relations model of a simple type system that allows them to compositionally verify and integrate syntactically well-typed and ill-typed parts. In contrast to Frumin et al. we focus on termination-insensitive noninterference and (in part for this reason) our type system is more permissible.

Our models are directly inspired by Rajani and Garg [RG20] that describe a step-indexed Kripke-style logical-relations model for two information-flow control type systems for a sequential language with higher-order state similar to ours. However, their type system does not support impredicative polymorphism and their semantic model cannot easily be extended to support this due to their use of syntactic worlds. Our semantic handling of label polymorphism is also different due to our use of semantic worlds. Rajani and Garg use their relation to prove that the fine-grained and coarse-grained static IFC systems are equivalent; Vassena et al. [Vas+19] show a similar result for dynamic information-flow control systems.

Noninterference and polymorphism. Abadi [Aba06] introduces a polymorphic DCC in the style of System F for access control in distributed systems. Inspired by the polymorphic DCC, Arden and Myers [AM16b] study a pure authorization calculus with polymorphic type-abstraction. Pottier and Simonet [PS03] study an ML-like language with let-polymorphism, recursion, references, and exceptions. In contrast to our work, these works consider less expressive notions of polymorphism than us or study pure calculi and prove noninterference using a syntactic approach which does not scale to relational reasoning for impredicative polymorphism in the presence of higher-order state. Morever, they do not benefit from the semantic approach with compositional integration of syntactically well-typed and syntactically ill-typed components.

The proof technique for noninterference of DCC by Abadi et al. [Aba+99] suggests that it is possible to use the parametric polymorphism in System F to model the dependency of DCC. Based on previous work of Tse and Zdancewic [TZ04], Bowman and Ahmed [BA15] provide a translation from the recursion-free fragment of DCC to F_ω, translating noninterference into parametricity. Algehed and Bernardy [AB19] leverage parametricity of the Calculus of Constructions to prove noninterference for a polyvariant variation of DCC and Algehed et al. [ABH20] show noninterference of a dynamic information-flow control library using a parametricity theorem. All these works *model* information-flow properties using parametricity whereas we add impredicative type polymorphism to a security-typed language.

6.6 Conclusion

We present the first semantic model of an information-flow control type system with impredicative polymorphism (universal and existential types), recursive types, and general reference types, and show how we can use our model to reason about syntactically ill-typed but semantically sound code. We showcase our methodology on multiple interesting examples and how our approach allows for compositional integration. Our semantic model guarantees termination-insensitive noninterference and we formalize it using logical relations on top of the Iris program logic framework. To do so, we introduce a novel re-usable program logic construct and theory of Modal Weakest Preconditions.

7 Asynchronous Probabilistic Couplings in Higher-Order Separation Logic

Abstract

Probabilistic couplings are the foundation for many probabilistic relational program logics and arise when relating random sampling statements across two programs. In relational program logics, this manifests as dedicated coupling rules that, *e.g.*, say we may reason as if two sampling statements return the same value. However, this approach fundamentally requires aligning or "synchronizing" the sampling statements of the two programs which is not always possible.

In this paper we develop Clutch, a higher-order probabilistic relational separation logic that addresses this issue by supporting asynchronous probabilistic couplings. We use Clutch to develop a logical step-indexed logical relational to reason about contextual refinement and equivalence of higher-order programs written in a rich language with higher-order local state and impredicative polymorphism. Finally, we demonstrate the usefulness of our approach on a number of case studies.

All the results that appear in the paper have been formalized in the Coq proof assistant using the Coquelicot library and the Iris separation logic framework.

7.1 Introduction

Relational reasoning is a useful technique for proving properties of probabilistic programs. By relating a complex probabilistic program to a simpler one, we can often reduce a challenging verification task to an easier one. In addition, certain important properties of probabilistic programs are naturally expressed in a relational form, such as stability of machine learning algorithms [BE02], differential privacy [DR13], and provable security [GM84]. Consequently, a number of relational program logics and models have been developed for probabilistic programs, *e.g.*, pRHL [Bar+15], approximate pRHL [Bar+16a; Bar+16b; Bar+12], EpRHL [Bar+18], HO-RPL [Agu+21], Polaris [TH19], logical relations [BB15; JSV10; Wan+18], and differential logical relations [LG22].

Many probabilistic relational program logics make use of *probabilistic couplings* [Lin02; Tho00; Vil08], a mathematical tool for reasoning about pairs of probabilistic processes. Informally, couplings correlate outputs of two processes by specifying how corresponding sampling statements are correlated.

To understand how couplings work in such logics, let us consider a pRHL-like logic. In pRHL and its variants, we prove Hoare *quadruples* of the form $\{P\}\ e_1 \sim e_2\ \{Q\}$, where e_1 and e_2 are two probabilistic programs, and P and Q are pre and post-*relations* on states of the two programs. Couplings arise when reasoning about random sampling statements in the two

programs, such as in the following rule:

$$\frac{}{\{P[v/x_1, v/x_2]\}\ x_1 \xleftarrow{\$} d \sim x_2 \xleftarrow{\$} d\ \{P\}}\ \textsc{prhl-couple}$$

Here, the two programs both sample from the same distribution d and store the result in variable x_1 and x_2, respectively. The rule says that we may reason *as if* the two sampling statements return the same value v in both programs, and one says that the sample statements have been "coupled". This is a powerful method that integrates well with existing reasoning principles from relational program logics .

However, this coupling rule requires aligning or "synchronizing" the sampling statements of the two programs: both programs have to be executing the sample statements we want to couple for their next step. To enable this alignment, pRHL has various rules that enable taking steps on one side of the quadruple at a time or commuting statements in a program. Nevertheless, with the rules from existing logics, it is not always possible to synchronize sampling statements.

For example, consider the following program written in an ML-like language that *eagerly* performs a probabilistic coin flip and returns the result in a thunk:

$$\mathit{eager} \triangleq \mathsf{let}\ b = \mathsf{flip}()\ \mathsf{in}\ \lambda\_.\ b$$

An indistinguishable—but *lazy*—version of the program does the probabilistic coin flip only when the thunk is invoked for the first time, and then stores the result in a reference that is read from in future invocations:

$$\begin{array}{l}\mathit{lazy} \triangleq \mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in}\\ \quad \lambda\_.\ \mathsf{match}\ !r\ \mathsf{with}\\ \qquad \mathsf{Some}\ (b) \Rightarrow b\\ \qquad \mid \mathsf{None} \quad \Rightarrow \mathsf{let}\ b = \mathsf{flip}()\ \mathsf{in}\\ \qquad\qquad\qquad\quad r \leftarrow \mathsf{Some}\ (b);\\ \qquad\qquad\qquad\quad b\\ \quad \mathsf{end}\end{array}$$

The usual symbolic execution rules of relational program logics will allow us to progress the two sides independently according to the operational semantics, but they will *not* allow us to line up the $\mathsf{flip}()$ expression in eager with that in lazy so that a coupling rule like PRHL-COUPLE can be applied. Intuitively, the $\mathsf{flip}()$ expression in eager is evaluated immediately but the $\mathsf{flip}()$ expression in lazy only gets evaluated when the thunk is invoked—to relate the two thunks one is forced to first symbolically evaluate the eager sampling, but this makes it impossible to couple it with the lazy sampling.

While the example may seem contrived, these kinds of transformations of eager and lazy sampling are widely used, *e.g.*, in proofs in the Random Oracle Model [BR93] and in game playing proofs [BR04; BR06]. For this reason, systems like EasyCrypt [Bar+13] and CertiCrypt [BGB09; BGB10] support reasoning about lazy/eager sampling through special-purpose rules for swapping statements that allows alignment of samplings; the approach is shown to work for a first-order language with global state and relies on syntactic criteria and assertions on memory disjointness. However, in rich enough languages (*e.g.* with general references and

closures) these kinds of swapping-equivalences are themselves non-trivial, even in the non-probabilistic case [DNB10; PS98].

In this paper we develop *Clutch*, a higher-order probabilistic relational separation logic that addresses this issue by supporting *asynchronous* probabilistic couplings. To do so, Clutch introduces a novel kind of ghost state, called *presampling tapes*. Presampling tapes let us reason about sampling statements as if they executed ahead of time and stored their results for later use. This converts the usual alignment problem of coupling rules into the task of reasoning about this special form of state. Fortunately, reasoning about state is well-addressed with the tools of modern separation logics.

Clutch provides a logical step-indexed logical relation [DAB09] to reason about *contextual refinement and equivalence* of probabilistic higher-order programs written in $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$, a rich language with a probabilistic choice operator, higher-order local state, recursive types, and impredicative polymorphism. Intuitively, expressions e_1 and e_2 of type τ are contextually equivalent if no well-typed context $\mathcal{C}$ can distinguish them, *i.e.*, if the expression $\mathcal{C}[\,e_1\,]$ has the same observable behaviors as $\mathcal{C}[\,e_2\,]$. Contextual equivalence can be decomposed into contextual refinement: we say e_1 refines e_2 at type τ, written $e_1 \precsim_{\text{ctx}} e_2 : \tau$, if, for all contexts $\mathcal{C}$, if $\mathcal{C}[\,e_1\,]$ has some observable behavior, then so does $\mathcal{C}[\,e_2\,]$. As our language is probabilistic, here 'observable behavior' means the *probability* of observing an outcome, such as termination. Using the *logical approach* [Tim+22], in Clutch, types are interpreted as relations expressed in separation logic. The resulting model allows us to prove, among other examples, that the eager program above is contextually equivalent to the lazy program.

The work presented in this paper is *foundational* [App01] in the sense that all results, including the semantics, the logic, the analysis results, the relational model, and all the examples are formalized in the Coq proof assistant [Coq22] using the Coquelicot library [BLM15] and the Iris separation logic framework [Jun+16; Jun+18b; Jun+15; Kre+17].

In summary, we make the following contributions:

- A higher-order probabilistic relational separation logic, Clutch, for reasoning about probabilistic programs written in $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$, an ML-like programming language with local state, recursive types, and impredicative polymorphism.
- A proof method for relating asynchronous probabilistic samplings in a program logic; a methodology that allows us to reason about sampling as if it was state and to exploit existing separation logic mechanisms such as *ghost state* and *invariants* to reason about probabilistic programs. We demonstrate the usefulness of the approach with a number of case studies.
- The first coupling-based relational program logic to reason about contextual refinement and equivalence of programs in a higher-order language with local state, recursive types, and impredicative polymorphism.
- Novel technical ideas, namely, *refinement couplings*, a *coupling modality*, and a *tape erasure* argument, that allow us to prove soundness of the relational logic.
- Full mechanization in the Coq proof assistant using Coquelicot and the Iris separation logic framework.

7.2 Key ideas

The key conceptual novelties of the Clutch logic are twofold: a *logical refinement judgment* and a novel kind of ghost resource, called *presampling tapes*. The refinement judgment $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ should be read as "the expression e_1 refines the expression e_2 at type τ" and it satisfies a range of structural and symbolic execution rules as showcased in Figure 7.2 and further explained in Section 7.4. Just like contextual refinement, the judgment is indexed by a type τ—the environment Δ assigns semantic interpretations to type variables in τ and $\mathcal{E}$ is an *invariant mask* as elaborated on in Section 7.4. Both are safely ignored in this section. The intuitive meaning of the judgment is formally reflected by the following soundness theorem.

Theorem 7.2.1 (Soundness). *If* $\emptyset \vDash e_1 \precsim e_2 : \tau$ *is derivable in Clutch then* $e_1 \precsim_{ctx} e_2 : \tau$.

The refinement judgment is *internal* to the ambient Clutch separation logic. This means that we can combine the judgment in arbitrary ways with other logical connectives: *e.g.*, the *separating conjunction* $P * Q$ and its adjoint *separating implication* (magic wand) $P \mathrel{-\!\!*} Q$. All inference rules that we present can be internalized as propositions in the logic and we will use an inference rule with premises $P_1, \ldots, P_n$ and conclusion Q as notation for $(P_1 * \ldots * P_n) \vdash Q$.

The language $\mathbf{F}^{\mathsf{flip}}_{\mu,\mathrm{ref}}$ contains a single probabilistic primitive flip that reduces uniformly to either true or false:

$$\mathsf{flip}(), \sigma \to^{1/2} b, \sigma \qquad\qquad b \in \{\mathsf{true}, \mathsf{false}\}$$

where σ is the current program state and $\to \subseteq \mathit{Cfg} \times [0,1] \times \mathit{Cfg}$ is a small-step transition relation, annotated with a probability that a transition occurs. To reason relationally about probabilistic choices that *can* be synchronized, Clutch admits a classical coupling rule that allows us to continue reasoning as if the two sampled values are related by a bijection $f : \mathsf{bool} \to \mathsf{bool}$.

$$\textsc{rel-couple-flips}\ \frac{f \text{ bijection} \qquad \forall b.\ \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim K'[\,f(b)\,] : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}()\,] \precsim K'[\,\mathsf{flip}()\,] : \tau}$$

where K and K' are arbitrary evaluation contexts.

To support *asynchronous* couplings we introduce *presampling tapes*. Reminiscent of how *prophecy variables* [AL88; AL91; Jun+20] allow us to talk about the future, presampling tapes give us the means to talk about the outcome of probabilistic choices *in the future*. Operationally, a tape is a finite sequence of Booleans, representing future outcomes of flip commands. Each tape is labeled with an identifier ι, and a program's state is extended with a finite map from labels to tapes. A tape can be allocated using a tape primitive:

$$\mathsf{tape}, \sigma \to^{1} \iota, \sigma[\iota \mapsto \epsilon] \qquad\qquad \text{if } \iota = \mathrm{fresh}(\sigma)$$

which extends the mapping with an empty tape and returns its fresh label ι. The flip primitive can then be annotated with a tape label ι. If the corresponding tape is empty, $\mathsf{flip}(\iota)$ reduces to true or false with equal probability:

$$\mathsf{flip}(\iota), \sigma \to^{1/2} b, \sigma \qquad\qquad \text{if } \sigma(\iota) = \epsilon \text{ and } b \in \{\mathsf{true}, \mathsf{false}\}$$

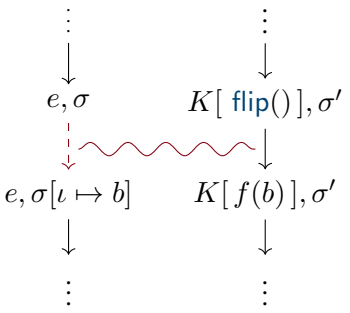


Figure 7.1: Illustration of an asynchronous coupling established through the rule REL-COUPLE-TAPE-L.

but if the tape is *not* empty, $\mathsf{flip}(\iota)$ reduces deterministically by taking off the first element of the tape and returning it:

$$\mathsf{flip}(\iota), \sigma[\iota \mapsto b \cdot \vec{b}] \to^1 b, \sigma[\iota \mapsto \vec{b}]$$

However, *no* primitives in the language operationally add values to the tapes. Instead, values are added to tapes as part of presampling steps that are *ghost operations* used in the logic. This is purely a proof-device that has no operational effect: in the end, tapes can in fact be erased through refinement.

The Clutch logic comes with a $\iota \hookrightarrow \vec{b}$ assertion that denotes *ownership* of the tape ι and its contents $\vec{b}$, analogously to how the traditional points-to-connective $\ell \mapsto v$ of separation logic denotes ownership of the location ℓ and its contents on the heap. When a tape is allocated, ownership of a fresh empty tape is acquired, *i.e.*,

$$\textsc{rel-alloc-tape-l}\ \frac{\forall \iota.\, \iota \hookrightarrow \epsilon \twoheadrightarrow \Delta \vDash_{\mathcal{E}} K[\,\iota\,] \precsim e : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{tape}\,] \precsim e : \tau}$$

Asynchronous couplings between probabilistic choices can be established in the refinement logic by coupling ghost presamplings with program steps. For example, the rule below allows us to couple an (unlabeled) probabilistic choice on the right with a presampling on the ι tape on the left:

$$\textsc{rel-couple-tape-l}\ \frac{f \text{ bijection} \qquad e \notin \mathit{Val} \qquad \iota \hookrightarrow \vec{b} \qquad \forall b.\, \iota \hookrightarrow \vec{b} \cdot b \twoheadrightarrow \Delta \vDash_{\mathcal{E}} e \precsim K'[\,f(b)\,] : \tau}{\Delta \vDash_{\mathcal{E}} e \precsim K'[\,\mathsf{flip}()\,] : \tau}$$

Intuitively, as illustrated in Figure 7.1, the rule allows us to couple a logical *ghost* presampling step on the left (illustrated using a red dashed arrow) with a *physical* sampling on the right. A symmetric rule holds for the opposite direction and two ghost presamplings can be coupled as well. When we—in the future—reach a presampled $\mathsf{flip}(\iota)$, we simply read off the presampled value from the ι tape, *i.e.*,

$$\textsc{rel-flip-tape-l}\ \frac{\iota \hookrightarrow b \cdot \vec{b} \qquad \iota \hookrightarrow \vec{b} \twoheadrightarrow \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}(\iota)\,] \precsim e_2 : \tau}$$

If we do not perform any presamplings, tapes and labels can be ignored and we can couple labeled sampling commands as if they were unlabeled:

$$\textsc{rel-flip-erase-r}\quad \frac{\iota \hookrightarrow_{\mathsf{s}} \epsilon \qquad \forall b.\, \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim K'[\,b\,] : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}()\,] \precsim K'[\,\mathsf{flip}(\iota)\,] : \tau}$$

To show that $lazy$ is a contextual refinement of $eager$ from Section 7.1, that is, $lazy \precsim_{\mathrm{ctx}} eager : \mathsf{unit} \to \mathsf{bool}$, we first define an intermediate labeled version of $lazy$:

$$\begin{aligned} lazy' \triangleq\ & \mathsf{let}\ \iota = \mathsf{tape}\ \mathsf{in} \\ & \mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\ & \lambda\_.\ \mathsf{match}\ !\,r\ \mathsf{with} \\ & \quad \mathsf{Some}\ (b) \Rightarrow b \\ & \quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in} \\ & \qquad\qquad r \leftarrow \mathsf{Some}\ (b); \\ & \qquad\qquad b \\ & \mathsf{end} \end{aligned}$$

By transitivity of contextual refinement and Theorem 7.2.1 it suffices to show $\vDash lazy \precsim lazy' : \mathsf{unit} \to \mathsf{bool}$ and $\vDash lazy' \precsim eager : \mathsf{unit} \to \mathsf{bool}$. The former follows straightforwardly using symbolic execution rules and REL-FLIP-ERASE-R. To show the latter we allocate a tape ι and a reference ℓ on the left by symbolic execution and couple the presampling of a Boolean b on the ι tape with the $\mathsf{flip}()$ on the right using REL-COUPLE-TAPE-L. This establishes an *invariant*

$$(\iota \hookrightarrow b * \ell \mapsto \mathsf{None}) \vee \ell \mapsto \mathsf{Some}\ (b)$$

that expresses how either b is on the ι tape and the location ℓ is empty *or* ℓ contains b. Invariants are particular kinds of propositions in Clutch that, in this particular case, are guaranteed to always hold at the beginning and at the end of the function evaluation. Under this invariant, we show that the two thunks are related by symbolic execution and rules for accessing invariants that we detail in Section 7.4. Similar arguments allow us to show the refinement in the other direction and consequently the contextual equivalence.

This example illustrates how presampling tapes are simple and powerful, yet merely a proof-device: the final equivalence does in fact hold for programs without any mention of tapes. One might be tempted to believe, though, that as soon as the idea of presampling arises, the high-level proof rules as supported by Clutch are straightforward to state and prove. This is *not* the case: As we will show throughout the paper, a great deal of care goes into defining a system that supports presampling while being sound. In Appendix C we discuss a counterexample that illustrates some of the subtleties.

Because the only probabilistic sampling primitive in $\mathbf{F}^{\mathsf{flip}}_{\mu,\mathrm{ref}}$ is flip, tapes in Clutch only store Boolean values. However, we believe the idea can be generalized to languages with sampling primitives for arbitrary discrete distributions by having tapes indexed with the distribution whose samples they draw from.

7.3 *Preliminaries and the language* $\mathbf{F}^{\mathsf{flip}}_{\mu,\mathrm{ref}}$

To account for non-terminating behavior, we will define our operational semantics using probability sub-distributions.

Definition 7.3.1 (Sub-distribution)**.** A (discrete) *sub-distribution* over a countable set A is a function $\mu : A \to [0,1]$ such that $\sum_{a \in A} \mu(a) \leq 1$. We write $\mathcal{D}(A)$ for the set of all sub-distributions over A.

Definition 7.3.2 (Support)**.** The *support* of $\mu \in \mathcal{D}(A)$ is the set of elements $\operatorname{supp}(\mu) = \{a \in A \mid \mu(a) > 0\}$.

Discrete probability subdistributions form a monad.

Lemma 7.3.3 (Probability Monad)**.** *Let $\mu \in \mathcal{D}(A)$, $a \in A$, and $f : A \to \mathcal{D}(B)$. Then*

1. $\operatorname{bind}(f,\mu)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b)$
2. $\operatorname{ret}(a)(a') \triangleq \begin{cases} 1 & \text{if } a = a' \\ 0 & \text{otherwise} \end{cases}$

gives $\mathcal{D}$ monadic structure. We write $\mu \gg\!= f$ as notation for $\operatorname{bind}(f,\mu)$.

The syntax of $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$ is defined by the grammar below.

$$
\begin{aligned}
v, w \in \mathit{Val} ::= {} & z \in \mathbb{Z} \mid \mathsf{true} \mid \mathsf{false} \mid () \mid \ell \in \mathit{Loc} \mid \iota \in \mathit{Label} \mid \\
& \mathsf{rec}\, f\ x = e \mid (v, w) \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \\
e \in \mathit{Expr} ::= {} & v \mid x \mid e_1(e_2) \mid \circledcirc_1\, e \mid e_1 \circledcirc_2 e_2 \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \\
& \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(v) \Rightarrow e_1 \mid \mathsf{inr}(w) \Rightarrow e_2\ \mathsf{end} \mid \mathsf{ref}(e) \mid\ !\, e \mid e_1 \leftarrow e_2 \mid \\
& \Lambda e \mid e\ \_ \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e \mid \mathsf{tape} \mid \mathsf{flip}(e) \\
\circledcirc_1 ::= {} & - \mid \neg \\
\circledcirc_2 ::= {} & + \mid - \mid = \mid \ldots \\
\sigma \in \mathit{State} \triangleq {} & (\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}) \times (\mathit{Label} \xrightarrow{\text{fin}} \mathbb{B}^*) \\
\rho \in \mathit{Cfg} \triangleq {} & \mathit{Expr} \times \mathit{State} \\
\tau \in \mathit{Type} ::= {} & \alpha \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \exists \alpha.\, \tau \mid \mu\, \alpha.\, \tau \mid \mathsf{ref}\ \tau \mid \mathsf{tape}
\end{aligned}
$$

The term language is mostly standard but note that there are no types in terms; we write Λe for type abstraction and $e\ \_$ for type application. $\mathsf{fold}\ e$ and $\mathsf{unfold}\ e$ are the special term constructs for iso-recursive types. $\mathsf{ref}(e)$ allocates a new reference, $!\, e$ dereferences the location e evaluates to, and $e_1 \leftarrow e_2$ assigns the result of evaluating e_2 to the location that e_1 evaluates to. We introduce syntactic sugar for lambda abstractions $\lambda x.\ e$ defined as $\mathsf{rec}\ \_\ x = e$, let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ defined as $(\lambda x.\ e_2)(e_1)$, and sequencing $e_1; e_2$ defined as $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$.

We implicitly coerce from $\sigma \in \mathit{State}$ to heaps and tapes, *e.g.*, $\sigma(\iota) = \pi_1(\sigma)(\iota)$ and $\sigma(\ell) = \pi_2(\sigma)(\ell)$. We define a call-by-value single step reduction relation $\to \subseteq \mathit{Cfg} \times [0,1] \times \mathit{Cfg}$ using evaluation contexts. The reduction relation is standard: all the non-probabilistic constructs reduce as usual with weight 1 and flip reduces uniformly as discussed in Section 7.2.

Let $\operatorname{step}(\rho) \in \mathcal{D}(\mathit{Cfg})$ denote the induced distribution of the single step reduction of configuration ρ. We define a stratified execution probability exec_n by induction on n:

$$
\operatorname{exec}_n(e,\sigma) \triangleq \begin{cases} \operatorname{ret}(e) & \text{if } e \in \mathit{Val} \\ \mathbf{0} & \text{if } e \notin \mathit{Val}, n = 0 \\ \operatorname{step}(e,\sigma) \gg\!= \operatorname{exec}_{(n-1)} & \text{otherwise} \end{cases}
$$

where $\mathbf{0}$ denotes the zero distribution. That is, $\mathrm{exec}_n(e,\sigma)(v)$ denotes the probability of stepping from the configuration (e,σ) to a value v in less than n steps. The probability that a full execution, starting from configuration ρ, reaches a value v is the limit of its stratified approximations, which exists by monotonicity and boundedness:

$$\mathrm{exec}(\rho)(v) \triangleq \mathsf{lim}_{n\to\infty}\, \mathrm{exec}_n(\rho)(v)$$

The probability that a full execution from a starting configuration ρ terminates then becomes $\mathrm{exec}_{\Downarrow}(\rho) \triangleq \sum_v \mathrm{exec}(\rho)(v)$.

Typing judgments have the form $\Theta \mid \Gamma \vdash e : \tau$ where Γ is a context assigning types to program variables, and Θ is a context of type variables that may occur in Γ and τ. The inference rules for the typing judgments are standard (see, *e.g.*, [FKB21b]) and omitted, except for the straightforward rules for typing tapes and samplings shown below:

$$\textsc{t-tape}\ \frac{}{\Theta \mid \Gamma \vdash \mathsf{tape} : \mathsf{tape}} \qquad \textsc{t-flip}\ \frac{\Theta \mid \Gamma \vdash e : \tau \quad \tau = \mathsf{unit} \vee \tau = \mathsf{tape}}{\Theta \mid \Gamma \vdash \mathsf{flip}(e) : \mathsf{bool}}$$

The notion of contextual refinement that we use is also mostly standard (see, *e.g.*, [BB15]) and uses the termination probability $\mathrm{exec}_{\Downarrow}$ as observation predicate. Since we are in a typed setting, we consider only typed contexts. A program context is well-typed, written $\mathcal{C} : (\Theta \mid \Gamma \vdash \tau) \Rightarrow (\Theta' \mid \Gamma' \vdash \tau')$, if for any term e such that $\Theta \mid \Gamma \vdash e : \tau$ we have $\Theta' \mid \Gamma' \vdash \mathcal{C}[\,e\,] : \tau'$. We say expression e_1 *contextually refines* expression e_2 if for all well-typed program contexts $\mathcal{C}$ resulting in a closed program then the termination probability of $\mathcal{C}[\,e_1\,]$ is bounded by the termination probability of $\mathcal{C}[\,e_2\,]$:

$$\Theta \mid \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau \triangleq \forall \tau', (\mathcal{C} : (\Theta \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau')), \sigma.\\ \mathrm{exec}_{\Downarrow}(\mathcal{C}[\,e_1\,], \sigma) \leq \mathrm{exec}_{\Downarrow}(\mathcal{C}[\,e_2\,], \sigma)$$

Note that contextual refinement is a precongruence, and that the statement itself is in the meta-logic (*e.g.*, Coq) and makes no mention of Clutch or Iris. Contextual equivalence $\Theta \mid \Gamma \vdash e_1 \simeq_{\mathrm{ctx}} e_2 : \tau$ is defined as the symmetric interior of refinement: $(\Theta \mid \Gamma \vdash e_1 \precsim_{\mathrm{ctx}} e_2 : \tau) \wedge (\Theta \mid \Gamma \vdash e_2 \precsim_{\mathrm{ctx}} e_1 : \tau)$.

7.4 The Clutch relational logic

In the style of ReLoC [FKB21b], we define a relational *logical refinement judgment* $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ as an internal notion in the Clutch separation logic by structural recursion over the type τ rather than by quantification over all contexts. The fundamental theorem of logical relations will then show that logical refinement implies contextual refinement; this means proving contextual refinement can be reduced to proving logical refinement, which is generally much easier. When defining and proving logical refinement we can leverage the features of modern separation logic, *e.g.*, (impredicative) invariants and (higher-order) ghost state as inherited from Iris, to model and reason about complex programs and language features.

Clutch is based on higher-order intuitionistic separation logic and the most important propositions are shown below.

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.\, P \mid \exists x.\, P \mid \\ & P * Q \mid P \mathrel{-\!\!*} Q \mid \Box P \mid \triangleright P \mid \mu x.\, P \mid \boxed{P}^{\mathcal{N}} \mid \boxed{P}^{\mathcal{N}} \mid \\ & \ell \mapsto v \mid \ell \mapsto_{\mathsf{s}} v \mid \iota \hookrightarrow \vec{b} \mid \iota \hookrightarrow_{\mathsf{s}} \vec{b} \mid \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \mid [\![\tau]\!]_{\Delta}(v_1, v_2) \mid \ldots \end{aligned}$$

As Clutch is an extension of Iris, it includes all the connectives of Iris such as the *later* modality $\triangleright$, the *persistence* modality $\Box$, *invariants* $\boxed{P}^{\mathcal{N}}$, and *non-atomic invariants* [The22], written $\boxed{P}^{\mathcal{N}}$, which we will introduce as needed throughout the paper. Some propositions are annotated by *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ and invariant names $\mathcal{N} \in \mathit{InvName}$ which are needed for bookkeeping of Iris's invariant mechanism in order to avoid reentrancy issues, where invariants are opened in a nested (and unsound) fashion. Ordinary Iris invariants can only be opened around atomic expressions (that evaluate to a value in a single step); this is crucial for soundness in a concurrent setting. Since $\mathbf{F}^{\mathrm{flip}}_{\mu,\mathrm{ref}}$ is a sequential language, we will generally favor the more flexible *non-atomic* invariants $\boxed{P}^{\mathcal{N}}$, which can be kept open during multiple execution steps—our use of non-atomic invariants will only require invariants to be closed at the end of proofs, as will become clear in Section 7.4.2.

Like ordinary separation logic, Clutch has heap assertions but since the logic is relational, these come in two forms: $\ell \mapsto v$ for the left-hand side and $\ell \mapsto_{\mathsf{s}} v$ for the right-hand side. For the same reason, tape assertions come in two forms as well, $\iota \hookrightarrow \vec{b}$ and $\iota \hookrightarrow_{\mathsf{s}} \vec{b}$ respectively.

7.4.1 Refinement judgments

The refinement judgment $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ should be read as "the expression e_1 refines the expression e_2 at type τ under the non-atomic invariants in the mask $\mathcal{E}$". The environment Δ assigns interpretations to type variables occurring in τ given by Clutch relations of type $\mathit{Val} \times \mathit{Val} \to \mathit{iProp}$. One such relation is the value interpretation $[\![\tau]\!]_\Delta(-,-)$ of a syntactic type $\tau \in \mathit{Type}$ which is used to define the refinement judgment, as discussed in Section 7.5. We discuss the use of invariants and the invariant mask further in Section 7.4.2. We refer to e_1 as the *implementation* and to e_2 as the *specification*.

In Figure 7.2, we present a selection of the type-directed structural and computational rules for proving logical refinement for deterministic reductions. Our computational rules resemble the typical backwards-style rules for symbolic execution from, *e.g.*, the weakest precondition calculus in Iris [Jun+18b], but come in forms for both the left-hand side and the right-hand side. For example, REL-PURE-L and REL-PURE-R allow us to symbolically execute "pure" reductions, *i.e.* reductions that do not depend on the state, such as β-reductions and projections. REL-STORE-L and REL-STORE-R on the other hand depends on state and require ownership of a location to store values to it.

The rules in Figure 7.3 showcase the computational rules for non-coupled probabilistic reductions and for interactions with presampling tapes. For example, the rules REL-FLIP-TAPE-L and REL-FLIP-TAPE-R allows us to read off values from a presampled tape as explained in Section 7.2; if the tapes are empty, REL-FLIP-TAPE-EMPTY-L and REL-FLIP-TAPE-EMPTY-R continue with a fresh sampling just like for unlabeled flips in REL-FLIP-L and REL-FLIP-R. Notice how the rules resemble the rules for interacting with the heap.

The main novelty of Clutch is the support for both synchronous and asynchronous couplings for which rules are shown in Figure 7.4. REL-COUPLE-FLIPS is a classical coupling rule that relates two samplings that can be aligned. REL-COUPLE-TAPE-L and REL-COUPLE-TAPE-R, on the other hand, are asynchronous coupling rules; they both couple a sampling reduction with an arbitrary expression on the opposite side by presampling a coupled value to a tape, as discussed in Section 7.2. Finally, REL-COUPLE-TAPES couples two ghost presamplings to two tapes, and hence offers full asynchrony.

REL-PURE-L
$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \triangleright(\Delta \vDash_{\mathcal{E}} K[\,e_1'\,] \precsim e_2 : \tau)}{\Delta \vDash_{\mathcal{E}} K[\,e_1\,] \precsim e_2 : \tau}$$

REL-PURE-R
$$\frac{e_2 \overset{\text{pure}}{\rightsquigarrow} e_2' \qquad \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,e_2'\,] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

REL-ALLOC-L
$$\frac{\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[\,\ell\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{ref}(v)\,] \precsim e_2 : \tau}$$

REL-ALLOC-R
$$\frac{\forall \ell.\, \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\ell\,] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{ref}(v)\,] : \tau}$$

REL-LOAD-L
$$\frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,!\ell\,] \precsim e_2 : \tau}$$

REL-LOAD-R
$$\frac{\ell \mapsto_{\mathsf{s}} v \qquad \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,!\ell\,] : \tau}$$

REL-STORE-L
$$\frac{\ell \mapsto v \qquad \ell \mapsto w \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\ell \leftarrow w\,] \precsim e_2 : \tau}$$

REL-STORE-R
$$\frac{\ell \mapsto_{\mathsf{s}} v \qquad \ell \mapsto_{\mathsf{s}} w \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\ell \leftarrow w\,] : \tau}$$

REL-PACK
$$\frac{\forall v_1, v_2.\, \mathsf{persistent}(R(v_1, v_2)) \qquad \Delta, \alpha \mapsto R \vDash_{\top} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\top} \mathsf{pack}\, e_1 \precsim \mathsf{pack}\, e_2 : \exists \alpha.\, \tau}$$

REL-REC
$$\frac{\Box\,(\forall v_1, v_2.\, [\![\tau]\!]_{\Delta}(v_1, v_2) \mathrel{-\!\!*} \Delta \vDash_{\top} (\mathsf{rec}\, f_1\, x_1 = e_1)\, v_1 \precsim (\mathsf{rec}\, f_2\, x_2 = e_2)\, v_2 : \tau \to \sigma)}{\Delta \vDash_{\top} \mathsf{rec}\, f_1\, x_1 = e_1 \precsim \mathsf{rec}\, f_2\, x_2 = e_2 : \tau \to \sigma}$$

REL-RETURN
$$\frac{[\![\tau]\!]_{\Delta}(v_1, v_2)}{\Delta \vDash_{\top} v_1 \precsim v_2 : \tau}$$

REL-BIND
$$\frac{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \qquad \forall v_1, v_2.\, [\![\tau]\!]_{\Delta}(v_1, v_2) \mathrel{-\!\!*} \Delta \vDash_{\top} K[\,v_1\,] \precsim K'[\,v_2\,] : \sigma}{\Delta \vDash_{\mathcal{E}} K[\,e_1\,] \precsim K'[\,e_2\,] : \sigma}$$

Figure 7.2: Selected structural and symbolic execution rules for the refinement judgment.

7.4.2 Invariants

Because the type system of $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$ is *not* substructural, types denote *knowledge*, not ownership. For example, closures can be invoked arbitrarily many times and hence ephemeral resources that may change over time cannot always be guaranteed to be available. For this reason, the *persistence modality* $\Box$ plays a crucial role in, for example, the rule REL-REC to guarantee that only *persistent* resources are used to verify the closure's body. We say P is persistent, written $\mathsf{persistent}(P)$ if $P \vdash \Box P$; otherwise, we say that P is *ephemeral*. Persistent resources can freely be duplicated ($\Box P \dashv\vdash \Box P * \Box P$) and eliminated ($\Box P \vdash P$). For example, invariants $\boxed{P}^{\mathcal{N}}$ and non-atomic invariants $\boxed{P}^{\mathcal{N}}$ are persistent: once established, they will remain true

REL-FLIP-L
$$\frac{\forall b.\, \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}()\,] \precsim e_2 : \tau}$$

REL-FLIP-R
$$\frac{\forall b.\, \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,b\,] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{flip}()\,] : \tau}$$

REL-ALLOC-TAPE-L
$$\frac{\forall \iota.\, \iota \hookrightarrow \epsilon \mathrel{-\!\!*} \Delta \vDash K[\,\iota\,] \precsim e : \tau}{\Delta \vDash K[\,\mathsf{tape}\,] \precsim e : \tau}$$

REL-ALLOC-TAPE-R
$$\frac{\forall \iota.\, \iota \hookrightarrow_{\mathsf{s}} \epsilon \mathrel{-\!\!*} \Delta \vDash e \precsim K[\,\iota\,] : \tau}{\Delta \vDash e \precsim K[\,\mathsf{tape}\,] : \tau}$$

REL-FLIP-TAPE-L
$$\frac{\iota \hookrightarrow b \cdot \vec{b} \qquad \iota \hookrightarrow \vec{b} \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}(\iota)\,] \precsim e_2 : \tau}$$

REL-FLIP-TAPE-R
$$\frac{\iota \hookrightarrow_{\mathsf{s}} b \cdot \vec{b} \qquad \iota \hookrightarrow_{\mathsf{s}} \vec{b} \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,b\,] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{flip}(\iota)\,] : \tau}$$

REL-FLIP-TAPE-EMPTY-L
$$\frac{\iota \hookrightarrow \epsilon \qquad \forall b.\, \iota \hookrightarrow \epsilon \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}(\iota)\,] \precsim e_2 : \tau}$$

REL-FLIP-TAPE-EMPTY-R
$$\frac{\iota \hookrightarrow_{\mathsf{s}} \epsilon \qquad \forall b.\, \iota \hookrightarrow_{\mathsf{s}} \epsilon \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,b\,] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[\,\mathsf{flip}(\iota)\,] : \tau}$$

Figure 7.3: Symbolic execution rules for tapes for the refinement judgment.

forever. On the contrary, ephemeral propositions like the points-to connective $\ell \mapsto v$ for the heap may be invalidated in the future when the location is updated. For exactly this reason, the rule REL-PACK also requires the interpretation of the type variable to be persistent, to guarantee that it does not depend on ephemeral resources.

To reason about, *e.g.*, functions that make use of ephemeral resources, a common pattern is to "put them in an invariant" to make them persistent, as sketched in Section 7.2 for the lazy/eager example. Since our language is sequential, when a function is invoked, no other code can execute before the function returns. This means that we can soundly keep invariants "open" and temporarily invalidate them for the entire duration of a function invocation—as long as the invariants are reestablished before returning. Non-atomic invariants allow us to capture exactly this intuition.

Figure 7.5 shows structural rules for the refinement judgment's interaction with non-atomic invariants. An invariant $\boxed{P}^{\mathcal{N}}$ can be allocated (REL-NA-INV-ALLOC) by giving up ownership of P. When opening an invariant (REL-NA-INV-OPEN) one obtains the resources P together with a resource $\mathsf{closeNaInv}_{\mathcal{N}}(P)$ that allows you to close the invariant again (REL-NA-INV-CLOSE) by reestablishing P. The mask annotation $\mathcal{E}$ on the refinement judgment keeps track of which invariants have been opened to avoid opening invariants in an (unsound) nested fashion. We guarantee that all invariants are closed by the end of evaluation by requiring $\top$, the set of all invariant names, as mask annotation on the judgment in all value cases (see, *e.g.*, REL-REC, REL-PACK, and REL-RETURN in Figure 7.2).

REL-COUPLE-FLIPS
$$\frac{f \text{ bijection} \qquad \forall b.\, \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim K'[\,f(b)\,] : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}()\,] \precsim K'[\,\mathsf{flip}()\,] : \tau}$$

REL-COUPLE-TAPE-L
$$\frac{f \text{ bijection} \qquad e \notin \mathit{Val} \qquad \iota \hookrightarrow \vec{b} \qquad \forall b.\, \iota \hookrightarrow (\vec{b} \cdot b) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e \precsim K[\,f(b)\,] : \tau}{\Delta \vDash_{\mathcal{E}} e \precsim K[\,\mathsf{flip}()\,] : \tau}$$

REL-COUPLE-TAPE-R
$$\frac{f \text{ bijection} \qquad \iota \hookrightarrow_{\mathsf{s}} \vec{b} \qquad \forall b.\, \iota \hookrightarrow_{\mathsf{s}} (\vec{b} \cdot f(b)) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim e : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}()\,] \precsim e : \tau}$$

REL-COUPLE-TAPES
$$\frac{\begin{array}{c} f \text{ bijection} \\ e_1 \notin \mathit{Val} \qquad \iota \hookrightarrow \vec{b} \qquad \iota' \hookrightarrow_{\mathsf{s}} \vec{b}' \qquad \forall b.\, \iota \hookrightarrow (\vec{b} \cdot b) * \iota' \hookrightarrow_{\mathsf{s}} (\vec{b}' \cdot f(b)) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \end{array}}{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}$$

Figure 7.4: Coupling rules for the Clutch refinement judgment.

REL-NA-INV-ALLOC
$$\frac{\triangleright P \qquad \boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}$$

REL-NA-INV-OPEN
$$\frac{\mathcal{N} \in \mathcal{E} \qquad \boxed{P}^{\mathcal{N}} \qquad \triangleright P * \mathsf{closeNaInv}_{\mathcal{N}}(P) \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}\setminus\mathcal{N}} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}$$

REL-NA-INV-CLOSE
$$\frac{\triangleright P \qquad \mathsf{closeNaInv}_{\mathcal{N}}(P) \qquad \Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}\setminus\mathcal{N}} e_1 \precsim e_2 : \tau}$$

Figure 7.5: Non-atomic invariant access rules for the Clutch refinement judgment.

Clutch invariants are inherited from Iris and hence they are *impredicative* [SB14] which means that the proposition P in $\boxed{P}^{\mathcal{N}}$ is *arbitrary* and can, *e.g.*, contain other invariant assertions. To ensure soundness of the logic and avoid self-referential paradoxes, invariant access guards P by the later modality $\triangleright$. When invariants are not used impredicatively, the later modality can mostly be ignored. The later modality is essential for the soundness of the logical relation and taking guarded fixpoints $\mu x.\, P$ that require the recursive occurrence x to appear under the later modality, but our use is entirely standard. We refer to [Jun+18b] for more details on the later modality and how it is used in Iris.

7.5 Model of Clutch

In this section we show how the connectives of Clutch are modeled through a shallow embedding in the *base logic* of the Iris separation logic [Jun+18b]. We describe how the refinement judgment $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ is modeled (Section 7.5.1) using a new *unary* weakest precondition theory (Section 7.5.3 and Section 7.5.4) and a *specification resource* (Section 7.5.5) construction in combination with a binary value interpretation $[\![\tau]\!]_\Delta : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ that describes how values are related (Section 7.5.2). Finally, we summarize how the final soundness theorem is proven (Section 7.5.6).

We note that the general skeleton of our model mimics prior work [FKB21b; KTB17; TH19; TDB13; Tur+13] but that the soundness theorem, how we prove it, and multiple technical facets are novel as we will highlight throughout this section. There are many details and layers to keep track of and for this reason, we will present the model using a top-down approach to not lose track of the bigger picture when working through the model.

7.5.1 Refinement judgment

Recall how the intuitive reading of the refinement judgment $\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau$ is that the expression e_1 refines the expression e_2 at type τ under the invariants in the mask $\mathcal{E}$ with interpretations of type variables in τ taken from Δ. This intuition is formally captured in Clutch as follows:

$$\Delta \vDash_{\mathcal{E}} e_1 \precsim e_2 : \tau \triangleq \forall K.\, \mathsf{specCtx} \mathrel{-\!\!*} \mathsf{spec}(K[\,e_2\,]) \mathrel{-\!\!*} \mathsf{naTok}(\mathcal{E}) \mathrel{-\!\!*} \mathsf{wp}\, e_1 \,\{v_1.\exists v_2.\, \mathsf{spec}(K[\,v_2\,]) * \mathsf{naTok}(\top) * [\![\tau]\!]_\Delta(v_1, v_2)\}$$

This definition has quite a few components and we will go over them bit by bit. First, note that we encode a relational specification into a unary specification by proving a unary weakest precondition about e_1 (the *implementation*), in which e_2 (the *specification*) is treated as a ghost resource $\mathsf{spec}(e)$ that can be updated to reflect execution steps. The weakest precondition connective $\mathsf{wp}\, e\, \{v.\Phi\}$ is a new probabilistic weakest precondition that we formally define and discuss in Section 7.5.3. In isolation it simply means that the execution of e is safe (*i.e.*, the probability of crashing is zero), and for every possible return value v of e, the postcondition $\Phi(v)$ holds. Note however, that it does not imply that the probability of termination is itself one. Next, the definition involves the resource $\mathsf{naTok}(\mathcal{E})$ that keeps track of the set of non-atomic invariants that are currently *closed*. Hence the definition gives the prover access to open the invariants in $\mathcal{E}$ but it requires *all* invariants ($\top$) to have been closed whenever e_1 returns. Finally, it involves the ghost specification connective $\mathsf{spec}(e)$ that states that the right-hand side program is currently executing the program e.

Putting everything together (ignoring specCtx for now), the full definition assumes that the right-hand side program is executing e_2 and that we have access to the invariants in $\mathcal{E}$, and concludes that the two executions can be coupled so that if e_1 reduces to some value v_1 then there exists a corresponding execution of e_2 to a value v_2 and all invariants have been closed. Moreover, the values v_1 and v_2 are related via the binary value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$ that we discuss in Section 7.5.2. By quantifying over K we close the definition under evaluation contexts on the right-hand side. For the left-hand side this is not needed as the weakest precondition satisfies the bind rule $\mathsf{wp}\, e\, \{v.\mathsf{wp}\, K[\,v\,]\, \{\Phi\}\} \vdash \mathsf{wp}\, K[\,e\,]\, \{\Phi\}$ for all evaluation contexts K.

The ghost specification connective spec(e), together with the specCtx proposition, satisfies a number of symbolic execution rules following the operational semantics. For brevity, we elide these rules—they correspond directly to all the right-hand side rules (ending in -R) shown in Figure 7.2 and Figure 7.3. The specCtx proposition is an Iris invariant that ties together the ghost specification resource spec(e) and the heap and tape assertions, $\ell \mapsto_{\mathsf{s}} v$ and $\iota \hookrightarrow_{\mathsf{s}} \vec{b}$, with an execution on the right-hand side as we discuss further in Section 7.5.5.

7.5.2 Binary value interpretation

The binary value interpretation $[\![\tau]\!]_\Delta$ characterizes the set of pairs of closed values (v_1, v_2) of type τ such that v_1 contextually refines v_2. The definition follows the usual structure of ("logical") logical relations, see, *e.g.*, [FKB21b], by structural recursion on τ and uses corresponding logical connectives. Functions are interpreted via (separating) implication, universal types are interpreted through universal quantification, *etc.*, as shown in Figure E.1 found in the appendix. The only novelty is the interpretation of the new type of tapes shown below:

$$[\![\mathsf{tape}]\!]_\Delta(v_1, v_2) \triangleq \exists \iota_1, \iota_2.\ (v_1 = \iota_1) * (v_2 = \iota_2) * \boxed{\iota_1 \hookrightarrow \epsilon * \iota_2 \hookrightarrow_{\mathsf{s}} \epsilon}^{\mathcal{N}.\iota_1.\iota_2}$$

The interpretation requires that the values are tape labels, *i.e.*, references to tapes, and that they are always empty as captured by the invariant. Intuitively, this guarantees through coupling rules and the symbolic execution rules from Figure 7.3 that we always can couple samplings on these tapes as needed in the compatibility lemma for T-FLIP as discussed in Section 7.5.6. Pointwise equality of the two tapes would also have been sufficient for the compatibility lemma but by requiring them to be empty we can prove general equivalences about label erasure such as $\iota : \mathsf{tape} \vdash \mathsf{flip}() \simeq_{\mathrm{ctx}} \mathsf{flip}(\iota) : \mathsf{bool}$.

7.5.3 A unary coupling weakest precondition

In most Iris-style program logics, the weakest precondition wp $e\ \{\Phi\}$ is a predicate stating that either the program e is a value satisfying Φ or it is reducible such that for any other term e' that it reduces to, then wp $e'\ \{\Phi\}$ must hold as well. The weakest precondition connective that we define in this section has—in isolation—the same intuition but it is fundamentally different. It is still a *unary* predicate used to interpret the left-hand side program, but in order to do relational reasoning, our definition of the weakest precondition pairs up the probability distribution of individual program steps of the left-hand side with the probability distribution of individual steps of *some* other program in such a way that there exists a *probabilistic coupling* among them. Through ghost resources and the specCtx invariant we will guarantee that this "other" program is connected to the specification program tracked by the spec(e) resource.

We recall that probabilistic couplings are used to prove relations between distributions. They are based on sampling from a joint distribution so that we can force the samples to be related in a particular manner:

Definition 7.5.1 (Coupling)**.** Let $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$. A sub-distribution $\mu \in \mathcal{D}(A \times B)$ is a *coupling* of μ_1 and μ_2 if

1. $\forall a.\ \sum_{b \in B} \mu(a, b) = \mu_1(a)$
2. $\forall b.\ \sum_{a \in A} \mu(a, b) = \mu_2(b)$

Given relation $R : A \times B$ we say μ is an R-coupling if furthermore $\mathrm{supp}(\mu) \subseteq R$. We write $\mu_1 \sim \mu_2 : R$ if there exists an R-coupling of μ_1 and μ_2.

Couplings can be constructed and composed along the monadic structure of the sub-distribution monad.

Lemma 7.5.2 (Composition of couplings)**.** *Let* $R : A \times B$, $S : A' \times B'$, $\mu_1 \in \mathcal{D}(A)$, $\mu_2 \in \mathcal{D}(B)$, $f_1 : A \to \mathcal{D}(A')$, *and* $f_2 : B \to \mathcal{D}(B')$.

1. *If* $(a, b) \in R$ *then* $\mathrm{ret}(a) \sim \mathrm{ret}(b) : R$.
2. *If* $\forall (a, b) \in R.\ f_1(a) \sim f_2(b) : S$ *and* $\mu_1 \sim \mu_2 : R$ *then* $\mu_1 \gg\!= f_1 \sim \mu_2 \gg\!= f_2 : S$

Once a coupling has been established, we can extract a concrete relation from it between the probability distributions. In particular, for $(=)$-couplings, we have the following result.

Lemma 7.5.3. *If* $\mu_1 \sim \mu_2 : (=)$ *then* $\mu_1 = \mu_2$.

The weakest precondition connective that we define is given by a guarded fixpoint of the equation below—the fixpoint exists because the recursive occurrence of the connective appears under the later modality.[1]

$$\begin{aligned}\mathsf{wp}\ e_1\ \{\Phi\} \triangleq\ & (e_1 \in \mathit{Val} \wedge \Phi(e_1)) \vee \\ & (e_1 \notin \mathit{Val} \wedge \forall \sigma_1, \rho_1'.\ S(\sigma_1) * G(\rho_1') \mathrel{-\!\!*} \\ & \quad \mathsf{execCoupl}((e_1, \sigma_1), \rho_1')(\lambda (e_2, \sigma_2), \rho_2'. \\ & \quad\quad \triangleright\, S(\sigma_2) * G(\rho_2') * \mathsf{wp}_{\mathcal{E}}\ e_2\ \{\Phi\}))\end{aligned}$$

The base case says that if the expression e_1 is a value then the postcondition $\Phi(e_1)$ must hold. On the other hand, if e_1 is *not* a value, we get to assume two propositions $S(\sigma_1)$ and $G(\rho_1')$ for any $\sigma_1 \in \mathit{State}$, $\rho_1' \in \mathit{Cfg}$, and then we must prove $\mathsf{execCoupl}((e_1, \sigma_1), \rho_1')(\ldots)$. The $S(\sigma_1)$ proposition is a *state interpretation* that interprets the state (the heap and the tapes) of the language as resources in Clutch and gives meaning to the $\ell \mapsto v$ and $\iota \hookrightarrow \vec{b}$ connectives. The $G(\rho_1)$ proposition is a *specification interpretation* that allows us to interpret and track the "other" program that we are constructing a coupling with—we return to the contents of the predicate in Section 7.5.5.

The key technical novelty and the essence of our solution is the *coupling modality*: Intuitively, the proposition $\mathsf{execCoupl}(\rho_1, \rho_1')(\lambda \rho_2, \rho_2'.\ P)$ says that there exists a series of (composable) couplings starting from configurations ρ_1 and ρ_1' that ends up in configurations ρ_2 and ρ_2' such that the proposition P holds. With this intuition in mind, the last clause of the weakest precondition says that the execution of (e_1, σ_1) can be coupled with the execution of ρ_1' such that the state and specification interpretations still hold for the end configurations, and the weakest precondition holds recursively for the continuation e_2.

[1] We omit from the definition occurrences of the *fancy update modality* needed for resource updates and necessary book-keeping related to (regular) Iris invariants—these matters are essential but our use is standard. For the Iris expert we refer to the appendix for the full definition.

7.5.4 The coupling modality

The coupling modality is defined as the least fixpoint of a fairly large, daunting looking equation that we reserve for the appendix. It has been carefully designed to support both synchronous and asynchronous couplings on both sides while still ensuring that the left program takes at least one step. The structure is simple: the equation is a large disjunction between all the different kinds of couplings we support. The fixpoint allows us to chain together multiple couplings but it always ends in base cases that couple a single step of the left-hand side program—this aligns with the usual intuition that each unfolding of the recursively defined weakest precondition corresponds to one physical program step. For instance, we can couple two physical program steps through the following rule:

$$\frac{\mathsf{red}(\rho_1) \qquad \mathrm{step}(\rho_1) \sim \mathrm{step}(\rho_1') : R \qquad \forall \rho_2, \rho_2'.\, R(\rho_2, \rho_2') \mathrel{-\!\!*} Z(\rho_2, \rho_2')}{\mathsf{execCoupl}(\rho_1, \rho_1')(Z)}$$

Intuitively, this says that to show $\mathsf{execCoupl}(\rho_1, \rho_1')(Z)$ we (1) have to show that the configuration ρ_1 is *red*ucible which means that the program *can* take a step (this is to guarantee safety of the left-hand side program), (2) pick a relation R and show that there exists an R-coupling of the two program steps, and (3) for all configurations ρ_2, ρ_2' in the support of the coupling, the predicate $Z(\rho_2, \rho_2')$ holds. This part of the definition of the coupling modality is key to the proof of the rule REL-COUPLE-FLIPS that couples two program samplings.

The coupling modality also allows to construct a coupling between a program step and a trivial (Dirac) distribution; this is used to validate proof rules that symbolically execute just one of the two sides. Indeed, the rule below allows us to progress the right-hand side independently from the left-hand side, but notice the occurrence of the coupling modality in the premise—this allows us to chain multiple couplings together.

$$\frac{\mathrm{ret}(\rho_1) \sim \mathrm{step}(\rho_1') : R \qquad \forall \rho_2'.\, R(\rho_1, \rho_2') \mathrel{-\!\!*} \mathsf{execCoupl}(\rho_1, \rho_2')(Z)}{\mathsf{execCoupl}(\rho_1, \rho_1')(Z)}$$

To support *asynchronous* couplings, we introduce a *state step* reduction relation $\rightarrow_\iota \subseteq \mathit{State} \times [0,1] \times \mathit{State}$ that uniformly at random samples a Boolean b to the end of the tape ι:

$$\sigma \rightarrow_\iota^{1/2} \sigma[\iota \rightarrow \vec{b} \cdot b] \qquad\qquad \text{if } \sigma(\iota) = \vec{b} \text{ and } b \in \{\mathsf{true}, \mathsf{false}\}$$

Let $\mathrm{step}_\iota(\sigma)$ denote the induced distribution of a single state step reduction of σ. The coupling modality allows us to introduce couplings between $\mathrm{step}_\iota(\sigma)$ and a sampling step:

$$\frac{\mathrm{step}_\iota(\sigma_1) \sim \mathrm{step}(\rho_1') : R \qquad \forall \sigma_2, \rho_2'.\, R(\sigma_2, \rho_2') \mathrel{-\!\!*} \mathsf{execCoupl}((e_1, \sigma_2), \rho_2')(Z)}{\mathsf{execCoupl}((e_1, \sigma_1), \rho_1')(Z)}$$

This particular rule is key to the soundness of the asynchronous coupling rule REL-COUPLE-TAPE-L that couples a sampling to a tape on the left with a program sampling on the right. Similarly looking consequence of the definition used to prove REL-COUPLE-TAPE-R and REL-COUPLE-TAPES exist as well. The crux is, however, that the extra state steps that we inject in the coupling modality to prove the asynchronous coupling rules *do not matter* (!) in the sense that they can be entirely erased as part of the adequacy theorem for the weakest precondition (Theorem 7.5.7).

7.5.5 A specification context with run ahead

The purpose of the specification context specCtx is to connect the $\mathsf{spec}(e)$ resource to the program e' that we are constructing a coupling with in the weakest precondition. We keep track of the expression e' with the specification interpretation G. When constructing a final closed proof we will want e to be equal to e', however, during proofs they are not always going to be the same—we will allow e to *run ahead* of e'. As a consequence, it will be possible to reason *independently* about the right-hand side without consideration of the left-hand side as exemplified by the rule below.

$$\frac{\textsc{spec-pure} \qquad \mathsf{specCtx} \qquad e \overset{\text{pure}}{\rightsquigarrow} e'}{\mathsf{spec}(K[\,e\,]) \twoheadrightarrow \mathsf{spec}(K[\,e'\,])}$$

To define specCtx we will use *two* instances of the *authoritative resource algebra* [Jun+15] from the Iris ghost theory. Setting up the theory is out of scope for this paper; it suffices to know that the instances satisfy $F_\bullet(a) * F_\circ(b) \vdash a = b$. To connect the two parts we will keep $\mathsf{specInterp}_\bullet(\rho)$ in the specification interpretation (that "lives" in the weakest precondition), and the corresponding $\mathsf{specInterp}_\circ(\rho)$ in specCtx:

$$\begin{aligned} G(\rho) &\triangleq \mathsf{specInterp}_\bullet(\rho) \\ \mathsf{specInv} &\triangleq \exists \rho, e, \sigma, n.\ \mathsf{specInterp}_\circ(\rho) * \mathsf{spec}_\bullet(e) * \mathsf{heaps}(\sigma) * \mathrm{execConf}_n(\rho)(e,\sigma) = 1 \\ \mathsf{specCtx} &\triangleq \boxed{\mathsf{specInv}}^{\mathcal{N}.\mathsf{spec}} \end{aligned}$$

This ensures that the configuration ρ tracked in the weakest precondition is the same as the configuration ρ tracked in specCtx. On top of this, specCtx contains resources $\mathsf{spec}_\bullet(e)$ and $\mathsf{heaps}(\sigma)$ while guaranteeing that the configuration (e, σ) can be reached in n deterministic program steps from ρ. The $\mathsf{heaps}(\sigma)$ resource gives meaning—using standard Iris ghost theory—to the heap and tape assertions, $\ell \mapsto_\mathsf{s} v$ and $\iota \hookrightarrow_\mathsf{s} \vec{b}$, just like the state interpretation in the weakest precondition. $\mathrm{execConf}_n(\rho) \in \mathcal{D}(\mathit{Cfg})$ denotes the distribution of n-step partial execution of ρ. By letting $\mathsf{spec}(e) = \mathsf{spec}_\circ(e)$ this construction permits the right-hand side program to progress (with deterministic reduction steps) without consideration of the left-hand side as exemplified by SPEC-PURE. However, when applying coupling rules that actually need to relate the two sides, the proof first "catches up" with $\mathsf{spec}(e)$ using the execCoupl rule that only progresses the right-hand side, before constructing the actual coupling.

7.5.6 Soundness

The soundness of the refinement judgment hinges on the soundness of the weakest precondition in combination with the right-hand side specification connective $\mathsf{spec}(e)$. The intermediate goal is to show a *coupling* between the two programs. However, contextual refinement is not defined as an equality between distributions, but rather as a pointwise inequality. For this reason we introduce a new notion of *refinement coupling*.

Definition 7.5.4 (Refinement Coupling)**.** Let $\mu_1 \in \mathcal{D}(A), \mu_2 \in \mathcal{D}(B)$. A sub-distribution $\mu \in \mathcal{D}(A \times B)$ is a *refinement coupling* of μ_1 and μ_2 if

1. $\forall a.\ \sum_{b \in B} \mu(a,b) = \mu_1(a)$

2. $\forall b.\ \sum_{a \in A} \mu(a, b) \leq \mu_2(b)$

Given relation $R : A \times B$ we say μ is an R-refinement-coupling if furthermore $\mathrm{supp}(\mu) \subseteq R$. We write $\mu_1 \lesssim \mu_2 : R$ if there exists an R-refinement-coupling of μ_1 and μ_2.

Note that this means that, for any $\mu \in \mathcal{D}(B)$ and any $R \subseteq A \times B$, the zero distribution $\mathbf{0}$ trivially satisfies $\mathbf{0} \lesssim \mu : R$. This reflects the asymmetry of both contextual refinement and our weakest precondition—it allows us to show that a diverging program refines any other program of appropriate type.

Refinement couplings can also be constructed and composed along the monadic structure of the sub-distribution monad and are implied by regular couplings:

Lemma 7.5.5. *If $\mu_1 \sim \mu_2 : R$ then $\mu_1 \lesssim \mu_2 : R$.*

Additionally, proving a $(=)$-refinement-coupling coincides with the relation between distributions that will allow us to reason about contextual refinement.

Lemma 7.5.6. *If $\mu_1 \lesssim \mu_2 : (=)$ then $\forall a.\ \mu_1(a) \leq \mu_2(a)$.*

The *adequacy* theorem of the weakest precondition is stated using refinement couplings.

Theorem 7.5.7 (Adequacy)**.** *Let $\varphi : Val \times Val \to Prop$ be a predicate in the meta-logic. If*

$$\mathsf{specCtx} * \mathsf{spec}(e') \vdash \mathsf{wp}\ e\ \{v. \exists v'.\ \mathsf{spec}(v') * \varphi(v, v')\}$$

is provable in Clutch then $\forall n.\ \mathrm{exec}_n(e, \sigma) \lesssim \mathrm{exec}(e', \sigma') : \varphi$.

As a simple corollary it follows from continuity of exec_n that $\mathrm{exec}(e, \sigma) \lesssim \mathrm{exec}(e', \sigma') : \varphi$.

The proof of the adequacy theorem goes by induction in both n and the execCoupl fixpoint, followed by a case distinction on the big disjunction in the definition of execCoupl. Most cases are simple coupling compositions along the monadic structure except the cases where we introduce state step couplings that rely on *erasure* in the following sense:

Lemma 7.5.8 (Erasure)**.** *If $\sigma_1(\iota) \in \mathsf{dom}(\sigma_1)$ then*

$$\mathrm{exec}_n(e_1, \sigma_1) \sim (\mathrm{step}_\iota(\sigma_1) \gg\!= \lambda \sigma_2.\ \mathrm{exec}_n(e_1, \sigma_2)) : (=)$$

Intuitively, this lemma tells us that we can prepend any program execution with a state step reduction and it will not have an effect on the final result. The idea behind the proof is that if we append a sampled bit b at the end of a tape, and if we eventually consume this bit, then we obtain the same distribution as if we never appended b in the first place. This is a property that one should *not* take for granted: the operational semantics has been carefully defined such that reading from an empty tape reduces to a bit as well, and none of the other program operations can alter or observe the contents of the tape. This ensures that presampled bits are untouched until consumed and that the proof and the execution is independent.

To show the soundness theorem of the refinement judgment, we extend the interpretation of types to typing contexts—$[\![\Gamma]\!]_\Delta(\vec{v}, \vec{w})$ iff for every $x_i : \sigma_i$ in Γ then $[\![\sigma_i]\!]_\Delta(v_i, w_i)$ holds—and the refinement judgment to open terms by closing substitutions as usual:

$$\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \tau \triangleq \forall \vec{v}, \vec{w}.\ [\![\Gamma]\!]_\Delta(\vec{v}, \vec{w}) \mathrel{-\!\!*} \Delta \vDash e_1[\vec{v}/\Gamma] \precsim e_2[\vec{w}/\Gamma] : \tau$$

where $e_1[\vec{v}/\Gamma]$ denotes simultaneous substitution of every x_i from Γ in e_1 by the value v_i.

We then show, using the structural and symbolic execution rules of the refinement judgment, that the typing rules are *compatible* with the relational interpretation: for every typing rule, if we have a pair of related terms for every premise, then we also have a pair of related terms for the conclusion. See for instance the compatibility rule for T-FLIP below in the case $\tau = \mathsf{tape}$ that follows using REL-BIND and REL-COUPLE-TAPES.

$$\textsc{flip-compat}\ \frac{\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \mathsf{tape}}{\Delta \mid \Gamma \vDash \mathsf{flip}(e_1) \precsim \mathsf{flip}(e_2) : \mathsf{bool}}$$

As a consequence of the compatibility rules, we obtain the fundamental theorem of logical relations.

Theorem 7.5.9 (Fundamental theorem). *Let $\Xi \mid \Gamma \vdash e : \tau$ be a well-typed term, and let Δ assign a relational interpretation to every type variable $\alpha \in \Xi$. Then $\Delta \mid \Gamma \vDash e \precsim e : \tau$.*

The compatibility rules, moreover, yield that the refinement judgment is a congruence, and together with Theorem 7.5.7 we can then recover contextual refinement:

Theorem 7.5.10 (Soundness). *Let Ξ be a type variable context, and assume that, for all Δ assigning a relational interpretation $Val \times Val \to iProp$ to the type variables in Ξ we can derive $\Delta \mid \Gamma \vDash e_1 \precsim e_2 : \tau$. Then, $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$.*

7.6 Case studies

In this section, we give an overview of some of the example equivalences we have proven with Clutch. Further details are found in Appendix F and our mechanized Coq development. In particular, in Appendix F we discuss an example from Sangiorgi and Vignudelli [SV16], which previous logical relations for probabilistic programs without asynchronous couplings could not prove [Biz16, Sec. 1.5].

7.6.1 Hash functions

When analyzing data structures that use hash functions, one commonly models the hash function under the *uniform hash assumption* or the *random oracle model* [BR93]. That is, a hash function h from a set of keys K to values V behaves as if, for each key k, the hash $h(k)$ is randomly sampled from a uniform distribution over V, independently of all the other keys. Of course, hash functions are not known to satisfy this assumption perfectly, but it can nevertheless be a useful modeling assumption for analyzing programs that use hashes.

The function $eager\_hash$ in Figure 7.6 encodes such a model of hash functions in $\mathbf{F}^{\mathsf{flip}}_{\mu,\mathsf{ref}}$. (We explain the reason for the "eager" name later.) Given a non-negative integer n, executing $eager\_hash\ n$ returns a hash function with $K = \{0, \ldots, n\}$ and $V = \mathbb{B}$. To do so, it initializes a mutable map m and then calls $sample\_all$, which samples a Boolean b with flip for each key k and stores the results in m. These Booleans serve as the hash values. On input k, the hash function returned by $eager\_hash$ looks up k in the map m and returns the result, with a default value of false if $k \notin K$.

```
eager_hash ≜
  λn. let m = init_map () in
      sample_all m (n + 1);
      (λk. match get m k with
             Some (b) ⇒ b
           | None     ⇒ false
           end)
```

```
lazy_hash ≜
  λn. let vm = init_map () in
      let tm = init_map () in
      alloc_tapes tm (n + 1);
      (λk. match get vm k with
             Some (b) ⇒ b
           | None     ⇒ match get tm k with
                          Some (ι) ⇒
                            let b = flip(ι) in
                            set vm b;
                            b
                        | None ⇒ false
                        end
           end)
```

Figure 7.6: Eager and lazy models of hash functions.

However, this model of uniform hash functions can be inconvenient for proofs because all of the random hash values are sampled *eagerly* when the function is initialized. To overcome this, an important technique in pencil-and-paper proofs is to show that the hash values can be sampled *lazily* [MF21]. That is, we only sample a key k's hash value when it is hashed for the first time. This lets us more conveniently couple that sampling step with some step in another program.

Motivated by applications to proofs in cryptography, Almeida et al. [Alm+19] formalized in EasyCrypt a proof of equivalence between an eager and lazy random oracle. Although sufficient for their intended application, this proof was done in the context of a language that uses syntactic restrictions to model the hash function's private state. To the best of our knowledge, no such equivalence proof between lazy and eager sampling has previously been given for a language with higher-order state and general references.

As an application of Clutch, we prove such an equivalence in $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$. The function $lazy\_hash$ shown in Figure 7.6 encodes the lazy sampling version of the random hash generator. For its internal state, the lazy hash uses two mutable maps: the tape map tm stores tapes to be used for random sampling, and the value map vm stores the previously sampled values for keys that have been hashed. After initializing these maps, it calls $alloc\_tapes$, which allocates a tape for each key $k \in K$ and stores the associated tape in tm, but does not yet sample hashes for any keys. The hash function returned by $lazy\_hash$ determines the hash for a key k in two stages. It first looks up k in vm to see if k already has a sampled hash value, and if so, returns the found value. Otherwise, it looks up k in the tape map tm. If no tape is found, then k must not be in K, so the function returns false. If a tape ι is found, then the code samples a Boolean b from this tape with flip, stores b for the key k in vm, and then returns b.

We prove that the eager and lazy versions are contextually equivalent, that is,

$$\vdash eager\_hash\ n \simeq_{\text{ctx}} lazy\_hash\ n : \mathsf{int} \to \mathsf{bool}$$

The core idea behind this contextual equivalence proof is to maintain an invariant between the internal state of the two hash functions. Let m be the internal map used by the eager hash

and let tm and vm be the tape and value maps, respectively, for the lazy hash. Then, at a high level, the invariant maintains the following properties:

1. $\mathsf{dom}(m) = \mathsf{dom}(tm) = \{0, \ldots, n\}$.
2. For all $k \in \{0, \ldots, n\}$, if $m[k] = b$ then either
   a) $vm[k] = b$, or
   b) $vm[k] = \bot$ and $tm[k] = \iota$ for some tape label ι such that $\iota \hookrightarrow [b]$.

Case (a) and (b) of the second part of this invariant capture the two possible states each key k can be in. Either the hash of k has been looked up before (case a), and so the sampled value stored in vm must match that of m, or it has not been looked up (case b) and the tape for the key must contain the same value as $m[k]$ for its next value.

To establish this invariant when the hashes are initialized, we asynchronously couple the eager hash function's flip for key k with a tape step for the tape ι associated with k in the lazy table. The invariant ensures that the values returned by the two hash functions will be the same when a key k is queried. The cases of the invariant correspond to the branches of the lazy function's match statements: if the key k is in K and has been queried before, the maps will return the same values found in m and vm. If it has not been queried before, then the flip statement in the lazy version will sample the value on the tape for the key, which matches $m[k]$. Moreover, the update that writes this sampled value to vm preserves the invariant, switching from case (b) to case (a) for the queried key.

We have used this more convenient lazy encoding to verify examples that use hash functions. For instance, one scheme to implement random number generators is to use a cryptographic hash function [BK15]. The following implements a simplified version of a scheme:

$$
\begin{array}{l}
\mathit{init\_hash\_rng} \triangleq \lambda\_.\ \mathsf{let}\ f = \mathit{lazy\_hash}\ \mathtt{MAX}\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in} \\
\qquad\qquad (\lambda\_.\ \mathsf{let}\ n = !\, c\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{let}\ b = f\ n\ \mathsf{in} \\
\qquad\qquad\quad c \leftarrow n + 1; \\
\qquad\qquad\quad b)
\end{array}
$$

When run, $\mathit{init\_hash\_rng}$ generates a lazy hash function f for the key space $K = \{0, \ldots, \mathtt{MAX}\}$ for some fixed constant MAX. It also allocates a counter c as a reference initialized to 0. It returns a sampling function, let us call it h, that uses f and c to generate random Booleans. Each time h is called, it loads the current value n from c and hashes n with f to get a Boolean b. It then increments c and returns the Boolean b. Repeated calls to h return independent, uniformly sampled Booleans, so long as we make no more than MAX calls.

We prove that $\mathit{init\_hash\_rng}$ is contextually equivalent to the following "bounded" random number generator that directly calls flip:

$$
\begin{array}{l}
\mathit{init\_bounded\_rng} \triangleq \lambda\_.\ \mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in} \\
\qquad\qquad\qquad (\lambda\_.\ \mathsf{let}\ n = !\, c\ \mathsf{in} \\
\qquad\qquad\qquad\quad \mathsf{let}\ b = \mathsf{if}\ n \leq \mathtt{MAX}\ \mathsf{then}\ \mathsf{flip}()\ \mathsf{else}\ \mathsf{false}\ \mathsf{in} \\
\qquad\qquad\qquad\quad c \leftarrow n + 1; \\
\qquad\qquad\qquad\quad b)
\end{array}
$$

The proof works by showing that, so long as $n \leq \texttt{MAX}$, then each time a sample is generated, the value of n will not have been hashed before. Thus, we may couple the random hash value with the flip call in $init\_bounded\_rng$. This argument relies on the fact that the counter c is private, encapsulated state, which is easy to reason about using the relational judgment since Clutch is a separation logic.

7.6.2 Lazily sampled big integers

Certain randomized data structures, such as treaps [SA96], need to generate random *priorities* as operations are performed on the structure. One can view these priorities as an abstract data type equipped with a total order supporting two operations: (1) a *sample* function that randomly generates a new priority according to some distribution, and (2) a *comparison* operation that takes a pair of priorities (p_1, p_2) and returns -1 (if $p_1 < p_2$), 0 (if $p_1 = p_2$), or 1 (if $p_2 < p_1$). The full details of how priorities are used in such data structures are not relevant here. Instead, what is important to know is that it is ideal to avoid *collisions*, that is, sampling the same priority multiple times.

A simple way to implement priorities is to represent them as integers sampled from some fixed set $\{0, \ldots, n\}$. However, to minimize collisions, we may need to make n very large. But making n large has a cost, because then priorities requires more random bits to generate and more space to store. An alternative is to *lazily* sample the integer that represents the priority. Because we only need to compare priorities, we can delay sampling bits of the integer until they are needed. A lazily-sampled integer can be encoded as a pair of a tape label ι and a linked list of length at most N, where each node in the list represents a *digit* of the integer in base B, with the head of the list being the most significant digit.

In the appendix, we describe such an implementation of lazily-sampled integers, with $N = 8$ and $B = 2^{32}$. Our Coq development contains a proof that this implementation is contextually equivalent to code that eagerly samples a 256-bit integer by bit-shifting and adding 8 32-bit integers. Crucially, this contextual equivalence is at an *abstract* existential type τ. Specifically, we define the type of abstract priorities,

$$\tau \triangleq \exists \alpha.\, (\mathsf{unit} \to \alpha) \times ((\alpha \times \alpha) \to \mathsf{int})$$

Then we have the following equivalence:

$$\vdash (sample\_lazy\_int, cmp\_lazy) \simeq_{\text{ctx}} (sample256, cmp) : \tau$$

where cmp is just primitive integer comparison. The proof uses tapes to presample the bits that make up the lazy integer and couples these with the eager version. The cmp_lazy function involves traversal and mutation of the linked lists representing the integers being compared, which separation logic is well-suited for reasoning about.

7.7 Coq formalization

All the results presented in the paper, including the background on probability theory, the formalization of the logic, and the case studies have been formalized in the Coq proof assistant [Coq22]. The results about probability theory are built on top of the Coquelicot library [BLM15], extending their results to real series indexed over arbitrary countable types.

Although we build our logic on top of Iris [Jun+18b], significant work is involved in formalizing the operational semantics of probabilistic languages, our new notion of weakest precondition that internalizes the coupling-based reasoning, and the erasure theorem that allows us to conclude the existence of a coupling. Our logic integrates smoothly with the Iris Proof Mode [KTB17] and we have adapted much of the tactical support from ReLoC [FKB21b] to reason about the relational judgment.

7.8 Related work

Separation logic. Relational separation logics have been developed on top of Iris for a range of properties, such as contextual refinement [FKB21b; KTB17; TB19; Tim+18], simulation [Cha+19; Gäh+22; Tim+21], and security [FKB21a; GTB22; Gre+21a]. The representation of the right-hand side program as a resource is a recurring idea, but our technical construction with run ahead is novel. With the exception of Tassarotti and Harper [TH19], probabilistic languages have not been considered in Iris. In Tassarotti and Harper [TH19], the authors develop a logic to show refinement between a probabilistic program and a semantic model, not a program. The logic relies on couplings, but it requires synchronization of sampling.

Batz et al. [Bat+19] presents a framework in which logical assertions are functions ranging over the non-negative reals. The connectives of separation logic are given an interpretation as maps from pairs of non-negative reals to the positive reals. This work focuses on proving quantitative properties of a single program, *e.g.*, bounding the probability that certain events happen. A variety of works have developed separation logics in which the separating conjunction models various forms of probabilistic independence [Bao+21; Bao+22; BHL20]. For example, the statement $P * Q$ is taken to mean "the distribution of P is independent from the distribution of Q".

Prophecy variables [AL88; AL91] have been integrated into separation logic in both a unary setting [Jun+20] and a relational setting [FKB21b]. The technical solution uses program annotations and physical state reminiscent of our construction with presampling tapes, but prophecy resolution is a physical program step, whereas presampling in our work is a logical operation. Prophecies can also be erased through refinement [FKB21b].

Probabilistic couplings. Probabilistic couplings are a technique from probability theory that can, *e.g.*, be used to prove equivalences between distributions or mixing times of Markov chains [Ald83]. In computer science, they have been used to reason about relational properties of programs such as equivalences [Bar+15] and differential privacy [Bar+16a]. These techniques have also been extended to higher-order programs [Agu+21]. However, these logics requires the sampling points on both programs to be synchronized in order to construct couplings. In a higher-order setting, the logic in Aguirre et al. [Agu+18] allows to establish so-called "shift couplings" between probabilistic streams that evolve at different rates, but these rules are ad-hoc and limited to the stream type.

Logical relations. Step-indexed logical relations have been applied to reason about contextual equivalence of probabilistic programs in a variety of settings. In Bizjak and Birkedal [BB15], logical relations are developed for a language similar to ours, although only first-order state is considered. This work has since been extended to a language with continuous probabilistic choice (but without state and impredicative polymorphism) [Wan+18], for which equivalence

is shown by establishing a measure preserving transformation between the sources of randomness for both programs. Recently, this was further extended to support nested inference queries [ZA22].

Another line of work [LG20; LG21; LG22] focuses on using so called differential logical relations to reason about contextual distance rather than equivalence. Here programs are related using metrics rather than equivalence relations, which allows to quantify how similar programs are.

Cryptographic frameworks. CertiCrypt [BGB09; BGB10] is a framework for cryptographic game-playing proofs written in a simple probabilistic first-order while-language ("pWhile"). CertiCrypt formalizes a denotational semantics for pWhile in Coq and supports reasoning about the induced notion of program equivalence via a pRHL, and provides dedicated tactics for lazy/eager sampling transformations. These kind of transformations are non-trivial for expressive languages like ours. CertiCrypt also provides a quantitative unary logic.

EasyCrypt [Bar+13] is a standalone prover for higher-order logic building on CertiCrypt's ideas. It leverages the first-order nature of pWhile for proof automation via SMT solvers. EasyCrypt extends pWhile with a module system [Bar+21] to support reasoning about abstract code as module parameters. It integrates a quantitative unary logic with pRHL, and supports reasoning about complexity in terms of oracle calls [Bar+21]. Both automation and these kind of properties are out of scope for our work but would be interesting future directions.

In FCF [PM15], programs are written as Coq expressions in the free sub-distribution monad. Proofs are conducted in a pRHL-like logic, where successive sampling statements can straightforwardly be swapped thanks to the commutativity of the monad.

SSProve [Aba+21; Has+21] supports modular crypto proofs by composing "packages" of programs written in the free monad for state and probabilities. The swap rule in SSProve allows exchanging commands which maintain a state invariant. Reasoning about dynamically allocated local state is not supported.

IPDL [Gan+23] is a process calculus for stating and proving cryptographic observational equivalences. IPDL is mechanized in Coq and targeted at equational reasoning about interactive message-passing in high-level cryptographic protocol models, and hence considers a different set of language features.

7.9 Conclusion

We have presented Clutch, a novel higher-order probabilistic relational separation logic with support for asynchronous probabilistic coupling-based proofs of contextual refinement and equivalence of probabilistic higher-order programs with local state and impredicative polymorphism. We have proved the soundness of Clutch formally in Coq using a range of new technical concepts and ideas such as *refinement couplings*, *presampling tapes*, and a *coupling modality*. We have demonstrated the usefulness of our approach through several example program equivalences that, to the best of our knowledge, were not possible to establish with previous methods. Future work includes extending the ideas of Clutch to concurrency and other probabilistic properties.

Bibliography

[Aba06] Martín Abadi. "Access control in a core calculus of dependency". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*. 2006, pp. 263–273. DOI: 10.1145/1159803.1159839 (cit. on p. 147).

[Aba+99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 1999, pp. 147–160. DOI: 10.1145/292540.292555 (cit. on pp. 10, 117, 146, 147).

[AL88] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. 1988, pp. 165–175. DOI: 10.1109/LICS.1988.5115 (cit. on pp. 151, 170).

[AL91] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Theor. Comput. Sci.* 82.2 (1991), pp. 253–284. DOI: 10.1016/0304-3975(91)90224-P (cit. on pp. 151, 170).

[Aba+21] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq". In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. 2021, pp. 1–15. DOI: 10.1109/CSF51468.2021.00048 (cit. on p. 171).

[AM16a] Tatsuya Abe and Toshiyuki Maeda. "Observation-Based Concurrent Program Logic for Relaxed Memory Consistency Models". In: *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*. 2016, pp. 63–84. DOI: 10.1007/978-3-319-47958-3_4 (cit. on p. 80).

[ALO00] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. "Generalized Isolation Level Definitions". In: *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. 2000, pp. 67–78. DOI: 10.1109/ICDE.2000.839388 (cit. on p. 80).

[Agu+18] Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. "Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European*

Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. 2018, pp. 214–241. DOI: 10.1007/978-3-319-89884-1_8 (cit. on p. 170).

[Agu+21] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, Shin-ya Katsumata, and Tetsuya Sato. "Higher-order probabilistic adversarial computations: categorical semantics and program logics". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473598 (cit. on pp. 13, 15, 148, 170).

[Aha+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. "Causal Memory: Definitions, Implementation, and Programming". In: *Distributed Comput.* 9.1 (1995), pp. 37–49. DOI: 10.1007/BF01784241 (cit. on pp. 53, 56, 58, 80, 81).

[Ahm04] Amal Ahmed. "Semantics of Types for Mutable State". PhD thesis. Princeton University, 2004 (cit. on pp. 10, 118).

[Ahm+10] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. "Semantic foundations for typed assembly languages". In: *ACM Trans. Program. Lang. Syst.* 32.3 (2010), 7:1–7:67. DOI: 10.1145/1709093.1709094 (cit. on pp. 3, 10).

[AAV02] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. "A Stratified Semantics of General References A Stratified Semantics of General References". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 2002, p. 75. DOI: 10.1109/LICS.2002.1029818 (cit. on p. 118).

[Ald83] David J. Aldous. "Random walks on finite groups and rapidly mixing Markov chains". In: *Séminaire de probabilités de Strasbourg* 17 (1983), pp. 243–297 (cit. on p. 170).

[AB19] Maximilian Algehed and Jean-Philippe Bernardy. "Simple noninterference from parametricity". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 89:1–89:22. DOI: 10.1145/3341693 (cit. on p. 147).

[ABH20] Maximilian Algehed, Jean-Philippe Bernardy, and Catalin Hritcu. "Dynamic IFC Theorems for Free!" In: *CoRR* abs/2005.04722 (2020). arXiv: 2005.04722 (cit. on p. 147).

[AR17] Maximilian Algehed and Alejandro Russo. "Encoding DCC in Haskell". In: *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017*. 2017, pp. 77–89. DOI: 10.1145/3139337.3139338 (cit. on pp. 10, 117).

[Alg+10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. "Fences in Weak Memory Models". In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 2010, pp. 258–272. DOI: 10.1007/978-3-642-14295-6_25 (cit. on p. 80).

[Alm+19] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. "Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pp. 1607–1622. DOI: 10.1145/3319535.3363211 (cit. on p. 167).

[AM01] Andrew Appel and David McAllester. "An Indexed Model of Recursive Types for Foundational Proof-Carrying Code". In: *TOPLAS* 23.5 (2001), pp. 657–683 (cit. on pp. 3, 10).

[App01] Andrew W. Appel. "Foundational Proof-Carrying Code". In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. 2001, pp. 247–256. DOI: 10.1109/LICS.2001.932501 (cit. on pp. 10, 20, 83, 150).

[App11] Andrew W. Appel. "Verified Software Toolchain - (Invited Talk)". In: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 2011, pp. 1–17. DOI: 10.1007/978-3-642-19718-5_1 (cit. on p. 3).

[AM16b] Owen Arden and Andrew C. Myers. "A Calculus for Flow-Limited Authorization". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, pp. 135–149. DOI: 10.1109/CSF.2016.17 (cit. on pp. 10, 117, 147).

[Arm+19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 71:1–71:31. DOI: 10.1145/3290384 (cit. on p. 80).

[Bai+13] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. "Bolt-on causal consistency". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 2013, pp. 761–772. DOI: 10.1145/2463676.2465279 (cit. on p. 52).

[Bao+21] Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. "A Bunched Logic for Conditional Independence". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 2021, pp. 1–14. DOI: 10.1109/LICS52264.2021.9470712 (cit. on p. 170).

[Bao+22] Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. "A separation logic for negative dependence". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–29. DOI: 10.1145/3498719 (cit. on p. 170).

[Bar+21] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. "Mechanized Proofs of Adversarial Complexity and Application to Universal Composability". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 2541–2563. DOI: 10.1145/3460120.3484548 (cit. on p. 171).

[BK15] Elaine B. Barker and John M. Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. National Institute of Standards and Technology, June 2015. DOI: 10.6028/nist.sp.800-90ar1 (cit. on pp. 168, 213).

[Bar+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. "EasyCrypt: A Tutorial". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 2013, pp. 146–166. DOI: 10.1007/978-3-319-10082-1_6 (cit. on pp. 149, 171).

[Bar+15] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. "Relational Reasoning via Probabilistic Coupling". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. 2015, pp. 387–401. DOI: 10.1007/978-3-662-48899-7_27 (cit. on pp. 13, 15, 148, 170).

[Bar+18] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. "Proving expected sensitivity of probabilistic programs". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 57:1–57:29. DOI: 10.1145/3158145 (cit. on pp. 15, 148).

[Bar+16a] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. "Advanced Probabilistic Couplings for Differential Privacy". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM, 2016, pp. 55–67. DOI: 10.1145/2976749.2978391 (cit. on pp. 15, 148, 170).

[Bar+16b] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. "Proving Differential Privacy via Probabilistic Couplings". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. 2016, pp. 749–758. DOI: 10.1145/2933575.2934554 (cit. on pp. 15, 148).

[BGB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. "Formal certification of code-based cryptographic proofs". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 2009, pp. 90–101. DOI: 10.1145/1480881.1480894 (cit. on pp. 149, 171).

[BGB10] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. "Programming Language Techniques for Cryptographic Proofs". In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. 2010, pp. 115–130. DOI: 10.1007/978-3-642-14052-5_10 (cit. on pp. 149, 171).

[BHL20] Gilles Barthe, Justin Hsu, and Kevin Liao. "A probabilistic separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 55:1–55:30. DOI: 10.1145/3371123 (cit. on p. 170).

[Bar+12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. "Probabilistic relational reasoning for differential privacy". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 2012, pp. 97–110. DOI: 10.1145/2103656.2103670 (cit. on pp. 15, 148).

[Bat+19] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Quantitative separation logic: a logic for reasoning about probabilistic pointer programs". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 34:1–34:29. DOI: 10.1145/3290347 (cit. on p. 170).

[Bee08] Robert Beers. "Pre-RTL formal verification: an intel experience". In: *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*. 2008, pp. 806–811. DOI: 10.1145/1391469.1391675 (cit. on p. 82).

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*. 1993, pp. 62–73. DOI: 10.1145/168588.168596 (cit. on pp. 149, 166).

[BR04] Mihir Bellare and Phillip Rogaway. *Code-Based Game-Playing Proofs and the Security of Triple Encryption*. Cryptology ePrint Archive, Paper 2004/331. https://eprint.iacr.org/2004/331. 2004 (cit. on p. 149).

[BR06] Mihir Bellare and Phillip Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *Advances in Cryptology - EUROCRYPT 2006*. Ed. by Serge Vaudenay. 2006, pp. 409–426 (cit. on p. 149).

[BB17] Lars Birkedal and Ales Bizjak. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*. 2017 (cit. on pp. 31, 55, 77, 78, 91, 132).

[Bir+11] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. "Step-indexed kripke models over recursive worlds". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 2011, pp. 119–132. DOI: 10.1145/1926385.1926401 (cit. on pp. 10, 118).

[BST12] Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. "A Concurrent Logical Relation". In: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*. Ed. by Patrick Cégielski and Arnaud Durand. Vol. 16. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 107–121. ISBN: 978-3-939897-42-2. DOI: 10.4230/LIPIcs.CSL.2012.107 (cit. on p. 85).

[Biz16] Aleš Bizjak. "On Semantics and Applications of Guarded Recursion". PhD thesis. Aarhus University, Jan. 2016 (cit. on pp. 166, 218).

[BB15] Ales Bizjak and Lars Birkedal. "Step-Indexed Logical Relations for Probability". In: *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 279–294. DOI: 10.1007/978-3-662-46678-0_18 (cit. on pp. 148, 155, 170).

[BA12] Hans-Juergen Boehm and Sarita V. Adve. "You don't know jack about shared variables or memory models". In: *Commun. ACM* 55.2 (2012), pp. 48–54. DOI: 10.1145/2076450.2076465 (cit. on p. 53).

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. "Coquelicot: A User-Friendly Library of Real Analysis for Coq". In: *Math. Comput. Sci.* 9.1 (2015), pp. 41–62 (cit. on pp. 150, 169).

[BAP15] Richard Bornat, Jade Alglave, and Matthew J. Parkinson. "New Lace and Arsenic: adventures in weak memory with a program logic". In: *CoRR* abs/1512.01416 (2015). arXiv: 1512.01416 (cit. on p. 80).

[Bou+17] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. "On verifying causal consistency". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 2017, pp. 626–638 (cit. on p. 81).

[BE02] Olivier Bousquet and André Elisseeff. "Stability and Generalization". In: *J. Mach. Learn. Res.* 2 (2002), pp. 499–526. ISSN: 1532-4435. DOI: 10.1162/153244302760200704 (cit. on p. 148).

[BA15] William J. Bowman and Amal Ahmed. "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 2015, pp. 101–113. DOI: 10.1145/2784731.2784733 (cit. on p. 147).

[Bre00] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. Ed. by Gil Neiger. ACM, 2000, p. 7. DOI: 10.1145/343477.343502 (cit. on p. 106).

[BSW04] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. "From Session Causality to Causal Consistency". In: *PDP*. IEEE Computer Society, 2004, pp. 152–158 (cit. on p. 71).

[Bur+12] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. "Eventually Consistent Transactions". In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 67–86. DOI: 10.1007/978-3-642-28869-2_4 (cit. on p. 80).

[Car+22] Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. "Applying formal verification to microkernel IPC at meta". In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. 2022, pp. 116–129. DOI: 10.1145/3497775.3503681 (cit. on pp. 3, 82).

[CBG15] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. "A Framework for Transactional Consistency Models with Atomic Visibility". In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. 2015, pp. 58–71. DOI: 10.4230/LIPIcs.CONCUR.2015.58 (cit. on p. 80).

[CGY17] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. "Algebraic Laws for Weak Consistency". In: *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*. Ed. by Roland Meyer and Uwe Nestmann. Vol. 85. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 26:1–26:18. DOI: 10.4230/LIPIcs.CONCUR.2017.26 (cit. on p. 80).

[Cha+19] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying concurrent, crash-safe systems with Perennial". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. 2019, pp. 243–258. DOI: 10.1145/3341301.3359632 (cit. on pp. 10, 115, 170).

[Cha+21] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. "GoJournal: a verified, concurrent, crash-safe journaling system". In: *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. 2021, pp. 423–439 (cit. on p. 82).

[CLS16] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. "Formal Verification of Multi-Paxos for Distributed Consensus". In: *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. Ed. by John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou. Vol. 9995. Lecture Notes in Computer Science. 2016, pp. 119–136. DOI: 10.1007/978-3-319-48989-6_8 (cit. on p. 115).

[Cha+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (2008), 4:1–4:26. DOI: 10.1145/1365815.1365816 (cit. on p. 52).

[CD10] Kristina Chodorow and Michael Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010 (cit. on p. 70).

[CP13] Koen Claessen and Michal H. Palka. "Splittable pseudorandom number generators using cryptographic hashing". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. Ed. by Chung-chieh Shan. ACM, 2013, pp. 47–58. DOI: 10.1145/2503778.2503784 (cit. on p. 214).

[Coo+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1277–1288. DOI: 10.14778/1454159.1454167 (cit. on p. 80).

[Coq22] Coq Development Team. *The Coq Proof Assistant*. Version 8.16. Sept. 2022. DOI: 10.5281/zenodo.7313584 (cit. on pp. 20, 150, 169).

[CS15] Karl Crary and Michael J. Sullivan. "A Calculus for Relaxed Memory". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 623–636. DOI: 10.1145/2676726.2676984 (cit. on p. 80).

[Cro+17] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. "Seeing is Believing: A Client-Centric Specification of Database Isolation". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*. 2017, pp. 73–82. DOI: 10.1145/3087801.3087802 (cit. on p. 80).

[Dan+22] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. "Compass: strong and compositional library specifications in relaxed memory separation logic". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. 2022, pp. 792–808. DOI: 10.1145/3519939.3523451 (cit. on p. 3).

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. "Dynamic multirole session types". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 435–446. DOI: 10.1145/1926385.1926435 (cit. on p. 50).

[Din+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. "Views: compositional reasoning for concurrent programs". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 287–300. DOI: 10.1145/2429069.2429104 (cit. on pp. 3, 23).

[Din+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. "Concurrent Abstract Predicates". In: *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. Ed. by Theo D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 504–528. DOI: 10.1007/978-3-642-14107-2_24 (cit. on pp. 3, 23, 55).

[DRG18] Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Philippa Gardner. "A perspective on specifying and verifying concurrent modules". In: *J. Log. Algebraic Methods Program.* 98 (2018), pp. 1–25. DOI: 10.1016/j.jlamp.2018.03.003 (cit. on p. 55).

[DV16] Marko Doko and Viktor Vafeiadis. "A Program Logic for C11 Memory Fences". In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. 2016, pp. 413–430. DOI: 10.1007/978-3-662-49122-5_20 (cit. on p. 80).

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. "Logical Step-Indexed Logical Relations". In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. 2009, pp. 71–80. DOI: 10.1109/LICS.2009.34 (cit. on pp. 10, 118, 119, 150).

[DNB10] Derek Dreyer, Georg Neis, and Lars Birkedal. "The impact of higher-order state and control effects on local relational reasoning". In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 2010, pp. 143–156. DOI: 10.1145/1863543.1863566 (cit. on p. 150).

[DR13] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy". In: *Foundations and Trends® in Theoretical Computer Science* 9.3-4 (2013), pp. 211–407. ISSN: 1551-305X, 1551-3068. DOI: 10.1561/0400000042 (cit. on p. 148).

[EHN20] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. "Verified Analysis of Random Binary Tree Structures". In: *J. Autom. Reason.* 64.5 (2020), pp. 879–910. DOI: 10.1007/s10817-020-09545-0 (cit. on p. 215).

[FT13] Luminous Fennell and Peter Thiemann. "Gradual Security Typing with References". In: *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*. 2013, pp. 224–239. DOI: 10.1109/CSF.2013.22 (cit. on pp. 142, 143).

[Flo67] Robert W Floyd. "Assigning meanings to programs". In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1 (cit. on pp. 2, 22).

[FKB18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. 2018, pp. 442–451. DOI: 10.1145/3209108.3209174 (cit. on p. 115).

[FKB21a] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "Compositional Non-Interference for Fine-Grained Concurrent Programs". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. 2021, pp. 1416–1433. DOI: 10.1109/SP40001.2021.00003 (cit. on pp. 10, 145, 146, 170).

[FKB21b] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity". In: *Log. Methods Comput. Sci.* 17.3 (2021). DOI: 10.46298/lmcs-17(3:9)2021 (cit. on pp. 10, 155, 160, 161, 170).

[Gäh+22] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. "Simuliris: a separation logic framework for verifying concurrent program optimizations". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498689 (cit. on pp. 10, 115, 170).

[Gan+23] Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. "A Core Calculus for Equational Proofs of Cryptographic Protocols". In: *Proc. ACM Program. Lang.* 7.POPL (2023). DOI: 10.1145/3571223 (cit. on p. 171).

[Gar+18] Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. "Paxos Consensus, Deconstructed and Abstracted". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 912–939. DOI: 10.1007/978-3-319-89884-1_32 (cit. on p. 115).

[Geo+21] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. "Efficient and provable local capability revocation using uninitialized capabilities". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. DOI: 10.1145/3434287 (cit. on p. 10).

[GTB22] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. "Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–30. DOI: 10.1145/3527318 (cit. on pp. 10, 170).

[GL02] Seth Gilbert and Nancy A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601 (cit. on p. 52).

[GM82] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy* (1982). DOI: 10.1109/sp.1982.10014 (cit. on p. 117).

[GM84] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption". In: *J. Comput. Syst. Sci.* 28.2 (1984), pp. 270–299. DOI: 10.1016/0022-0000(84)90070-9 (cit. on p. 148).

[Gom+17] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. "Verifying strong eventual consistency in distributed systems". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 109:1–109:28. DOI: 10.1145/3133933 (cit. on p. 115).

[Gon+21a] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. "Distributed causal memory: Modular specification and verification in higher-order distributed separation logic". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434323 (cit. on pp. 18, 19, 82).

[Gon+21b] Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. *Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic: Technical Appendix*. [Online]. 2021 (cit. on p. 19).

[Gon+22] Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. "Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols". [Online]. 2022 (cit. on p. 6).

[Got+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. "'Cause I'm strong enough: reasoning about consistency choices in distributed systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 371–384. DOI: 10.1145/2837614.2837625 (cit. on pp. 53, 80).

[Gra78] Jim Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. Ed. by Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle. Vol. 60. Lecture Notes in Computer Science. Springer, 1978, pp. 393–481. DOI: 10.1007/3-540-08755-9_9 (cit. on pp. 43, 83, 96).

[Gre+23] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. "Asynchronous Probabilistic Couplings in Higher-Order Separation Logic". In: *CoRR* abs/2301.10061 (2023). arXiv: 2301.10061 (cit. on pp. 18, 19).

[Gre+21a] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. "Mechanized logical relations for termination-insensitive noninterference". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434291 (cit. on pp. 18, 19, 170).

[Gre+21b] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. *Mechanized logical relations for termination-insensitive noninterference: Technical Appendix*. [Online]. 2021 (cit. on p. 19).

[GTA19] Simon Oddershede Gregersen, Søren Eller Thomsen, and Aslan Askarov. "A Dependently Typed Library for Static Information-Flow Control in Idris". In: *Principles of Security and Trust - 8th International Conference, POST 2019, Proceedings*. 2019, pp. 51–75. DOI: 10.1007/978-3-030-17138-4_3 (cit. on pp. 10, 18, 117, 143).

[Guo+13] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. "Failure Recovery: When the Cure Is Worse Than the Disease". In: *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. 2013 (cit. on p. 53).

[Has+21] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. *SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq*. Cryptology ePrint Archive, Paper 2021/397. https://eprint.iacr.org/2021/397. 2021 (cit. on p. 171).

[HLP01] Klaus Havelund, Michael R. Lowry, and John Penix. "Formal Analysis of a Space-Craft Controller Using SPIN". In: *IEEE Trans. Software Eng.* 27.8 (2001), pp. 749–765. DOI: 10.1109/32.940728 (cit. on p. 82).

[Haw+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving practical distributed systems correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 1–17. DOI: 10.1145/2815400.2815428 (cit. on pp. 22, 47, 50, 80, 114).

[HR98] Nevin Heintze and Jon G. Riecke. "The SLam Calculus: Programming with Secrecy and Integrity". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 1998, pp. 365–377. DOI: 10.1145/268946.268976 (cit. on pp. 10, 117, 146).

[HBK20] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris: session-type based reasoning in separation logic". In: *PACMPL* 4 (2020), 6:1–6:30. DOI: 10.1145/3371074 (cit. on p. 50).

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cit. on p. 2).

[Hol97] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521 (cit. on pp. 7, 22, 49, 80, 82).

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Chris Hankin. Vol. 1381. Lecture Notes in Computer Science. Springer, 1998, pp. 122–138. DOI: 10.1007/BFb0053567 (cit. on p. 50).

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty asynchronous session types". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 273–284. DOI: 10.1145/1328438.1328472 (cit. on p. 50).

[JTD21] Koen Jacobs, Amin Timany, and Dominique Devriese. "Fully abstract from static to gradual". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. DOI: 10.1145/3434288 (cit. on p. 3).

[JM05] Mauro Jaskelioff and Stephan Merz. "Proving the Correctness of Disk Paxos". In: *Arch. Formal Proofs* 2005 (2005) (cit. on p. 115).

[JB12] Jonas Braband Jensen and Lars Birkedal. "Fictional Separation Logic". In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 377–396. DOI: 10.1007/978-3-642-28869-2_19 (cit. on p. 3).

[JSV10] Patricia Johann, Alex Simpson, and Janis Voigtländer. "A Generic Operational Metatheory for Algebraic Effects". In: *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*. 2010, pp. 209–218. DOI: 10.1109/LICS.2010.29 (cit. on p. 148).

[Jun+18a] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the rust programming language". In: *PACMPL* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154 (cit. on pp. 3, 26, 119).

[Jun+16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. "Higher-order ghost state". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 2016, pp. 256–269. DOI: 10.1145/2951913.2951943 (cit. on pp. 3, 23, 54, 76, 82, 118, 150, 198).

[Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cit. on pp. 3, 34, 54, 64, 76, 82, 90, 91, 94, 98, 118, 132, 150, 156, 159, 160, 170).

[Jun+20] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. "The future is ours: prophecy variables in separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113 (cit. on pp. 151, 170, 204).

[Jun+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 637–650. DOI: 10.1145/2676726.2676980 (cit. on pp. 3, 23, 54, 82, 118, 150, 164).

[Kai+17a] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris". In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 2017, 17:1–17:29. DOI: 10.4230/LIPIcs.ECOOP.2017.17 (cit. on pp. 3, 68, 80, 81).

[Kai+17b] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris". In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Ed. by Peter Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 17:1–17:29. DOI: 10.4230/LIPIcs.ECOOP.2017.17 (cit. on p. 26).

[Kak+18] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. "Alone together: compositional reasoning and inference for weak isolation". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 27:1–27:34. DOI: 10.1145/3158115 (cit. on pp. 53, 80).

[Kel04] Pertti Kellomäki. *An Annotated Specification of the Consensus Protocol of Paxos Using Superposition in PVS*. Tech. rep. Tampere University of Technology. Institute of Software Systems., 2004 (cit. on p. 115).

[Kil+07] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. "Mace: language support for building distributed systems". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 179–188. DOI: 10.1145/1250734.1250755 (cit. on pp. 22, 49, 80).

[Kra+20] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. "Inductive sequentialization of asynchronous programs". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 2020, pp. 227–242. DOI: 10.1145/3385412.3385980 (cit. on p. 115).

[Kre+18] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: a general, extensible modal framework for interactive proofs in separation logic". In: *PACMPL* 2.ICFP (2018), 77:1–77:30. DOI: 10.1145/3236772 (cit. on pp. 3, 54, 118, 121, 140).

[Kre+17] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic". In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26 (cit. on pp. 3, 23, 82, 118, 134, 150).

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "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*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 205–217 (cit. on pp. 3, 8, 10, 25, 54, 83, 85, 86, 115, 119, 160, 170).

[Kri+20] Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. "Verifying concurrent search structure templates". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 2020, pp. 181–196. DOI: 10.1145/3385412.3386029 (cit. on p. 82).

[KSB17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. "A relational model of types-and-effects 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*. 2017, pp. 218–231. DOI: 10.1145/3009837.3009877 (cit. on p. 115).

[Kro+20a] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. "Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Proceedings*. 2020, pp. 336–365. DOI: 10.1007/978-3-030-44914-8_13 (cit. on pp. 18, 19, 53, 54, 70, 82, 86, 88).

[Kro+20b] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. *Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems - Technical Appendix*. [Online]. 2020 (cit. on p. 19).

[LG20] Ugo Dal Lago and Francesco Gavazzo. "Differential Logical Relations Part II: Increments and Derivatives". In: *Proceedings of the 21st Italian Conference on Theoretical Computer Science, Ischia, Italy, September 14-16, 2020*. Ed. by Gennaro Cordasco, Luisa Gargano, and Adele A. Rescigno. Vol. 2756. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 101–114 (cit. on p. 171).

[LG21] Ugo Dal Lago and Francesco Gavazzo. "Differential logical relations, part II increments and derivatives". In: *Theor. Comput. Sci.* 895 (2021), pp. 34–47. DOI: 10.1016/j.tcs.2021.09.027 (cit. on p. 171).

[LG22] Ugo Dal Lago and Francesco Gavazzo. "Effectful program distancing". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–30. DOI: 10.1145/3498680 (cit. on pp. 148, 171).

[Lah19] Ori Lahav. "Verification under causally consistent shared memory". In: *ACM SIGLOG News* 6.2 (2019), pp. 43–56. DOI: 10.1145/3326938.3326942 (cit. on p. 81).

[LB20] Ori Lahav and Udi Boker. "Decidable verification under a causally consistent shared memory". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 2020, pp. 211–226. DOI: 10.1145/3385412.3385966 (cit. on p. 80).

[Lam77] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904 (cit. on p. 22).

[Lam78] Leslie Lamport. "The Implementation of Reliable Distributed Multiprocess Systems". In: *Computer Networks* 2 (1978), pp. 95–114. DOI: 10.1016/0376-5075(78)90045-4 (cit. on p. 22).

[Lam92] Leslie Lamport. "Hybrid Systems in TLA$^+$". In: *Hybrid Systems*. Ed. by Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel. Vol. 736. Lecture Notes in Computer Science. Springer, 1992, pp. 77–102. DOI: 10.1007/3-540-57318-6_25 (cit. on pp. 7, 22, 49, 80, 82, 85).

[Lam98] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229 (cit. on p. 83).

[Lam01] Leslie Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58 (cit. on pp. 83, 100).

[Lam19] Leslie Lamport. *Paxos: How to Win a Turing Award*. https://github.com/tlaplus/Examples/tree/master/specifications/PaxosHowToWinATuringAward. Accessed: 2022-12-05. 2019 (cit. on p. 85).

[Lar17] Frederic Lardinois. *With Cosmos DB, Microsoft wants to build one database to rule them all*. (Accessed on 23/06/2021). 2017 (cit. on p. 82).

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cit. on pp. 50, 80).

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814 (cit. on p. 2).

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. "Chapar: certified causally consistent distributed key-value stores". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 357–370. DOI: 10.1145/2837614.2837622 (cit. on pp. 22, 50, 53, 80).

[LN13] Ruy Ley-Wild and Aleksandar Nanevski. "Subjective auxiliary state for coarse-grained concurrency". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 561–574. DOI: 10.1145/2429069.2429134 (cit. on pp. 3, 23).

[LZ06] Peng Li and Steve Zdancewic. "Encoding Information Flow in Haskell". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. 2006, p. 16. DOI: 10.1109/CSFW.2006.13 (cit. on pp. 10, 117).

[LF16] Hongjin Liang and Xinyu Feng. "A program logic for concurrent objects under fair scheduling". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 385–399. DOI: 10.1145/2837614.2837635 (cit. on p. 116).

[LF18] Hongjin Liang and Xinyu Feng. "Progress of concurrent objects with partial methods". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 20:1–20:31. DOI: 10.1145/3158108 (cit. on p. 116).

[LF21] Hongjin Liang and Xinyu Feng. "Abstraction for conflict-free replicated data types". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 636–650. DOI: 10.1145/3453483.3454067 (cit. on pp. 115, 116).

[Lin02] T. Lindvall. *Lectures on the Coupling Method*. Dover Books on Mathematics Series. Dover Publications, Incorporated, 2002. ISBN: 978-0-486-42145-2 (cit. on pp. 13, 148).

[Llo+11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. 2011, pp. 401–416. DOI: 10.1145/2043556.2043593 (cit. on pp. 52, 53, 198).

[LC15] Luísa Lourenço and Luís Caires. "Dependent Information Flow Types". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 317–328. DOI: 10.1145/2676726.2676994 (cit. on pp. 10, 117, 143).

[MP22] Jean-Marie Madiot and François Pottier. "A separation logic for heap space under garbage collection". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–28. DOI: 10.1145/3498672 (cit. on p. 3).

[Mad+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. "An Axiomatic Memory Model for POWER Multiprocessors". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 2012, pp. 495–512. DOI: 10.1007/978-3-642-31424-7_36 (cit. on p. 80).

[MSB17] Ognjen Maric, Christoph Sprenger, and David A. Basin. "Cutoff Bounds for Consensus Algorithms". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 217–237. DOI: 10.1007/978-3-319-63390-9_12 (cit. on p. 115).

[MJ21] Glen Mével and Jacques-Henri Jourdan. "Formal verification of a concurrent bounded queue in a weak memory model". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473571 (cit. on pp. 3, 82).

[MF21] Arno Mittelbach and Marc Fischlin. *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography*. Information Security and Cryptography. Springer, 2021. ISBN: 978-3-030-63286-1. DOI: 10.1007/978-3-030-63287-8 (cit. on p. 167).

[Mur+16] Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. "Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, pp. 417–431. DOI: 10.1109/CSF.2016.36 (cit. on p. 143).

[Mye99] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 1999, pp. 228–241. DOI: 10.1145/292540.292561 (cit. on pp. 10, 117).

[NPS20] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. "Proving the Safety of Highly-Available Distributed Objects". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 544–571. DOI: 10.1007/978-3-030-44914-8_20 (cit. on p. 115).

[NBG11] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. "Verification of Information Flow and Access Control Policies with Dependent Types". In: *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. 2011, pp. 165–179. DOI: 10.1109/SP.2011.12 (cit. on p. 143).

[Nan+14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. "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*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 290–310. DOI: 10.1007/978-3-642-54833-8_16 (cit. on pp. 3, 23).

[New14] Chris Newcombe. "Why Amazon Chose TLA +". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. 2014, pp. 25–39. DOI: 10.1007/978-3-662-43652-3_3 (cit. on p. 82).

[New+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon web services uses formal methods". In: *Commun. ACM* 58.4 (2015), pp. 66–73. DOI: 10.1145/2699417 (cit. on p. 82).

[Nie+22] Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. "Modular verification of op-based CRDTs in separation logic". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 1788–1816. DOI: 10.1145/3563351 (cit. on p. 6).

[OHe07] Peter W. O'Hearn. "Resources, concurrency, and local reasoning". In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 271–307. DOI: 10.1016/j.tcs.2006.12.035 (cit. on pp. 3, 4, 23, 26, 82).

[OP99] Peter W. O'Hearn and David J. Pym. "The logic of bunched implications". In: *Bull. Symb. Log.* 5.2 (1999), pp. 215–244. DOI: 10.2307/421090 (cit. on p. 4).

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1 (cit. on pp. 2, 4).

[OO14] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319 (cit. on p. 115).

[OG76] Susan S. Owicki and David Gries. "An Axiomatic Proof Technique for Parallel Programs I". In: *Acta Informatica* 6 (1976), pp. 319–340. DOI: 10.1007/BF00268134 (cit. on p. 2).

[Pad+17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. "Paxos made EPR: decidable reasoning about distributed protocols". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 108:1–108:31. DOI: 10.1145/3140568 (cit. on p. 115).

[Par10] Matthew J. Parkinson. "The Next 700 Separation Logics - (Invited Paper)". In: *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*. 2010, pp. 169–182. DOI: 10.1007/978-3-642-15057-9_12 (cit. on p. 3).

[PJP15] Willem Penninckx, Bart Jacobs, and Frank Piessens. "Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 158–182. DOI: 10.1007/978-3-662-46669-8_7 (cit. on p. 114).

[PM15] Adam Petcher and Greg Morrisett. "The Foundational Cryptography Framework". In: *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. 2015, pp. 53–72. DOI: 10.1007/978-3-662-46666-7_4 (cit. on p. 171).

[PS98] Andrew M. Pitts and Ian D. B. Stark. "Operational Reasoning for Functions with Local State". In: *Higher Order Operational Techniques in Semantics*. Ed. by A. D. Gordon and A. M. Pitts. Publications of the Newton Institute. Cambridge University Press, 1998, pp. 227–273 (cit. on pp. 145, 150).

[Plo04] Gordon D. Plotkin. "A structural approach to operational semantics". In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 17–139 (cit. on p. 2).

[PA93] Gordon D. Plotkin and Martín Abadi. "A Logic for Parametric Polymorphism". In: *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. 1993, pp. 361–375. DOI: 10.1007/BFb0037118 (cit. on p. 119).

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cit. on pp. 22, 49, 80).

[PC00] François Pottier and Sylvain Conchon. "Information flow inference for free". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. 2000, pp. 46–57. DOI: 10.1145/351240.351245 (cit. on p. 146).

[PS03] François Pottier and Vincent Simonet. "Information flow inference for ML". In: *ACM Trans. Program. Lang. Syst.* 25.1 (2003), pp. 117–158. DOI: 10.1145/596980.596983 (cit. on pp. 10, 117, 147).

[Rah+15] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. "Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML". In: *ECEASST* 72 (2015). DOI: 10.14279/tuj.eceasst.72.1013 (cit. on pp. 22, 47, 50).

[Rah+17] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. "EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems". In: *Sci. Comput. Program.* 148 (2017), pp. 26–48. DOI: 10.1016/j.scico.2017.05.009 (cit. on p. 50).

[RG20] Vineet Rajani and Deepak Garg. "On the expressiveness and semantics of information flow types". In: *Journal of Computer Security* 28.1 (2020), pp. 129–156. DOI: 10.3233/JCS-191382 (cit. on pp. 12, 117, 119, 123, 138, 147).

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cit. on pp. 2, 4, 25).

[RDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A Logic for Time and Data Abstraction". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Ed. by Richard E. Jones. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 207–231. DOI: 10.1007/978-3-662-44202-9_9 (cit. on pp. 3, 23).

[Rus15] Alejandro Russo. "Functional pearl: two can keep a secret, if one of them uses Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 2015, pp. 280–288. DOI: 10.1145/2784731.2784756 (cit. on pp. 10, 117).

[RCH08] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. 2008, pp. 13–24. DOI: 10.1145/1411286.1411289 (cit. on pp. 10, 117).

[SS99] Andrei Sabelfeld and David Sands. "A PER Model of Secure Information Flow in Sequential Programs". In: *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. 1999, pp. 40–58. DOI: 10.1007/3-540-49099-X_4 (cit. on p. 146).

[SS01] Andrei Sabelfeld and David Sands. "A Per Model of Secure Information Flow in Sequential Programs". In: *High. Order Symb. Comput.* 14.1 (2001), pp. 59–91. DOI: 10.1023/A:1011553200337 (cit. on p. 146).

[SV16] Davide Sangiorgi and Valeria Vignudelli. "Environmental bisimulations for probabilistic higher-order languages". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 595–607. DOI: 10.1145/2837614.2837651 (cit. on pp. 166, 218).

[SZ18] Sven Schewe and Lijun Zhang, eds. *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*. Vol. 118. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. ISBN: 978-3-95977-087-3 (cit. on p. 80).

[SS71] Dana Scott and C. Strachey. "Towards a Mathematical Semantics for Computer Languages". In: *Proceedings of the Symposium on Computers and Automata* 21 (1971) (cit. on p. 2).

[Sco70] Dana S. Scott. *Outline of a Mathematical Theory of Computation*. Tech. rep. PRG–2. Oxford, England, 1970 (cit. on p. 2).

[SA96] Raimund Seidel and Cecilia R. Aragon. "Randomized Search Trees". In: *Algorithmica* 16.4/5 (1996), pp. 464–497 (cit. on pp. 169, 215).

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Mechanized verification of fine-grained concurrent programs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 77–87. DOI: 10.1145/2737924.2737964 (cit. on p. 23).

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and proving with distributed protocols". In: *PACMPL* 2.POPL (2018), 28:1–28:30. DOI: 10.1145/3158116 (cit. on pp. 22, 23, 47, 50, 51, 80, 115).

[Sev+11] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "Relaxed-memory concurrency and verified compilation". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 2011, pp. 43–54. DOI: 10.1145/1926385.1926393 (cit. on p. 80).

[Sha+11] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. "Convergent and Commutative Replicated Data Types". In: *Bull. EATCS* 104 (2011), pp. 67–88 (cit. on pp. 83, 106, 110).

[Sim03a] Vincent Simonet. "Flow Caml in a Nutshell". In: *Proc. of the first APPSEM-II workshop*. Ed. by Graham Hutton. Nottingham, United Kingdom, 2003 (cit. on pp. 117, 123).

[Sim03b] Vincnet Simonet. *The Flow Caml system*. 2003 (cit. on pp. 10, 117).

[Siv12] Swaminathan Sivasubramanian. "Amazon dynamoDB: a seamlessly scalable non-relational database service". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 2012, pp. 729–730. DOI: 10.1145/2213836.2213945 (cit. on p. 70).

[Spi+21] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. "Transfinite Iris: resolving an existential dilemma of step-indexed separation logic". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*. 2021, pp. 80–95. DOI: 10.1145/3453483.3454031 (cit. on pp. 3, 92, 112, 115).

[Spr+20] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. "Igloo: soundly linking compositional refinement and separation logic for distributed system verification". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 152:1–152:31. DOI: 10.1145/3428220 (cit. on p. 114).

[Ste21] Léo Stefanesco. "Asynchronous and Relational Soundness Theorems for Concurrent Separation Logic. (Théorémes de correction asynchrone et relationnelle de la logique de séparation concurrente)". PhD thesis. Université de Paris, France, 2021 (cit. on pp. 9, 113, 114).

[SB14] Kasper Svendsen and Lars Birkedal. "Impredicative Concurrent Abstract Predicates". 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*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 149–168. DOI: 10.1007/978-3-642-54833-8_9 (cit. on pp. 3, 4, 23, 159).

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. "Modular Reasoning about Separation of Concurrent Data Structures". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 169–188. DOI: 10.1007/978-3-642-37036-6_11 (cit. on pp. 4, 55).

[TS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN: 978-0-13-239227-3 (cit. on p. 70).

[TH19] Joseph Tassarotti and Robert Harper. "A separation logic for concurrent randomized programs". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 64:1–64:30. DOI: 10.1145/3290377 (cit. on pp. 13, 115, 148, 160, 170).

[TJH17] Joseph Tassarotti, Ralf Jung, and Robert Harper. "A Higher-Order Logic for Concurrent Termination-Preserving Refinement". In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 909–936. DOI: 10.1007/978-3-662-54434-1_34 (cit. on pp. 112, 114, 115).

[Ter+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994*. 1994, pp. 140–149. DOI: 10.1109/PDIS.1994.331722 (cit. on pp. 56, 70).

[The22] The Iris Development Team. *The Iris 4.0 Reference*. Version August 20. 2022 (cit. on p. 156).

[Tho00] Hermann Thorisson. *Coupling, stationarity, and regeneration*. Probability and its Applications (New York). New York: Springer-Verlag, 2000, pp. xiv+517. ISBN: 0-387-98779-7 (cit. on pp. 13, 148).

[TB19] Amin Timany and Lars Birkedal. "Mechanized relational verification of concurrent programs with continuations". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 105:1–105:28. DOI: 10.1145/3341709 (cit. on pp. 3, 10, 119, 170).

[TB21] Amin Timany and Lars Birkedal. "Reasoning about monotonicity in separation logic". In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. 2021, pp. 91–104. DOI: 10.1145/3437992.3439931 (cit. on p. 98).

[Tim+21] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. "Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic". In: *CoRR* abs/2109.07863 (2021). arXiv: 2109.07863 (cit. on pp. 10, 18, 19, 170).

[Tim+22] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. "A Logical Approach to Type Soundness". Unpublished manuscript. 2022 (cit. on p. 150).

[Tim+18] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. "A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 64:1–64:28. DOI: 10.1145/3158152 (cit. on pp. 3, 10, 26, 115, 119, 137, 170).

[TCP11] Bernardo Toninho, Luís Caires, and Frank Pfenning. "Dependent session types via intuitionistic linear type theory". In: *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*. Ed. by Peter Schneider-Kamp and Michael Hanus. ACM, 2011, pp. 161–172. DOI: 10.1145/2003476.2003499 (cit. on p. 50).

[TZ04] Stephen Tse and Steve Zdancewic. "Translating dependency into parametricity". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. 2004, pp. 115–125. DOI: 10.1145/1016850.1016868 (cit. on p. 147).

[Tur49] Alan M. Turing. "Checking a Large Routine". In: *Report on a Conference on High Speed Automatic Computation, June 1949*. University Mathematical Laboratory, Cambridge University, 1949, pp. 67–69 (cit. on p. 2).

[TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. "Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 2013, pp. 377–390. DOI: 10.1145/2500365.2500600 (cit. on pp. 3, 23, 85, 119, 160).

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. "GPS: navigating weak memory with ghosts, protocols, and separation". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 2014, pp. 691–707. DOI: 10.1145/2660193.2660243 (cit. on pp. 3, 80).

[Tur+13] Aaron J Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. "Logical relations for fine-grained concurrency". In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 343–356 (cit. on pp. 3, 85, 160).

[Tyu+19] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. "Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 2019, pp. 636–650. DOI: 10.1145/3299869.3314049 (cit. on p. 52).

[VN13] Viktor Vafeiadis and Chinmay Narayan. "Relaxed separation logic: a program logic for C11 concurrency". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 2013, pp. 867–884. DOI: 10.1145/2509136.2509532 (cit. on p. 80).

[Vas+18] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95 (2018), pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003 (cit. on pp. 10, 117).

[Vas+19] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. "From fine- to coarse-grained dynamic information flow control and back". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 76:1–76:31. DOI: 10.1145/3290389 (cit. on p. 147).

[VP21] Paulo Emílio de Vilhena and François Pottier. "A separation logic for effect handlers". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–28. DOI: 10.1145/3434314 (cit. on p. 3).

[VPJ20] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. "Spy game: verifying a local generic solver in Iris". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 33:1–33:28. DOI: 10.1145/3371101 (cit. on p. 82).

[Vil08] C. Villani. *Optimal Transport: Old and New*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2008. ISBN: 9783540710509 (cit. on pp. 13, 148).

[VB21] Simon Friis Vindum and Lars Birkedal. "Contextual refinement of the Michael-Scott queue (proof pearl)". In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. 2021, pp. 76–90. DOI: 10.1145/3437992.3439930 (cit. on pp. 82, 98).

[VFB22] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. "Mechanized verification of a fine-grained concurrent queue from meta's folly library". In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. 2022, pp. 100–115. DOI: 10.1145/3497775.3503689 (cit. on p. 82).

[Vog09] Werner Vogels. "Eventually consistent". In: *Commun. ACM* 52.1 (2009), pp. 40–44. DOI: 10.1145/1435417.1435432 (cit. on p. 106).

[Wan+18] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. "Contextual equivalence for a probabilistic language with continuous random variables and recursion". In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 87:1–87:30. DOI: 10.1145/3236782 (cit. on pp. 148, 170).

[Wil+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 357–368. DOI: 10.1145/2737924.2737958 (cit. on pp. 22, 47, 50, 80, 115).

[Woo+16] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. "Planning for change in a formal verification of the raft consensus protocol". In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*. Ed. by Jeremy Avigad and Adam Chlipala. ACM, 2016, pp. 154–165. DOI: 10.1145/2854065.2854081 (cit. on p. 115).

[Xio+19] Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner. "Data Consistency in Transactional Storage Systems: a Centralised Approach". In: *CoRR* abs/1901.10615 (2019). arXiv: 1901.10615 (cit. on pp. 53, 80).

[Zda02] Stephan Arthur Zdancewic. "Programming Languages for Information Security". PhD thesis. USA: Cornell University, 2002. ISBN: 0493830499 (cit. on p. 146).

[ZBP14] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. “Formal Specification and Verification of CRDTs”. In: *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*. Ed. by Erika Ábrahám and Catuscia Palamidessi. Vol. 8461. Lecture Notes in Computer Science. Springer, 2014, pp. 33–48. DOI: 10.1007/978-3-662-43613-4_3 (cit. on p. 115).

[ZA22] Yizhou Zhang and Nada Amin. “Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion”. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–28. DOI: 10.1145/3498677 (cit. on p. 171).

[ZM07] Lantian Zheng and Andrew C. Myers. “Dynamic security labels and static information flow control”. In: *Int. J. Inf. Sec.* 6.2-3 (2007), pp. 67–84. DOI: 10.1007/s10207-007-0019-9 (cit. on p. 143).

A Indirect Causal Dependency

The proof of the indirect causal dependency example [Llo+11] makes use of predicates

$$\begin{aligned} \mathsf{pending} &: \mathit{iProp} \\ \mathsf{shot} &: \mathit{WriteEvent} \to \mathit{iProp} \\ \mathsf{hist} &: \wp^{\mathsf{fin}}(\mathit{WriteEvent}) \xrightarrow{\mathsf{fin}} \mathit{iProp} \end{aligned}$$

satisfying the laws below. The predicates pending and shot are defined using the *oneshot* resource algebra [Jun+16] whereas the hist predicate is defined using a fractional agreement resource algebra. We refer to the Coq formalization for all the details and the full proof.

Laws for ghost resources

$$\begin{aligned} &\mathsf{pending} \Rrightarrow \mathsf{shot}(w) \\ &\mathsf{shot}(w) * \mathsf{shot}w' \vdash w = w' \\ &\mathsf{pending} * \mathsf{shot}(w) \vdash \mathsf{False} \\ &\boxed{\diamond} * \boxed{\diamond} \vdash \mathsf{False} \\ &\mathsf{hist}(h_1) * \mathsf{hist}(h_2) \vdash h_1 = h_2 \\ &\mathsf{hist}(h_1) * \mathsf{hist}(h_1) \vdash \mathsf{hist}(h_2) * \mathsf{hist}(h_2) \end{aligned}$$

Invariants

$$\begin{aligned} \mathsf{Inv}_x \triangleq\ & \exists h.\, x \rightharpoonup_u h * \mathsf{hist}(h) * \big((\mathsf{pending} * \forall w \in h.\, w.v \neq 1) \vee \\ & \big(\boxed{\diamond} * \exists w.\, \mathsf{shot}(w) * w.v = 37 * \mathrm{Maximum}(h) = \mathsf{Some}\ w * \\ & \quad \forall w' \in h.\, w'.v = 37 \Rightarrow w = w'\big)\big) \\ \mathsf{Inv}_y \triangleq\ & \exists h.\, y \rightharpoonup_u h * \forall w \in h.\, w.v = 1 \Rightarrow \exists w'.\, vc(w') < vc(w) * \mathsf{shot}(w') \end{aligned}$$

Proof sketch

Node i, proof outline

$$
\begin{array}{l}
\{\mathsf{Seen}(i,s) * \mathsf{hist}(h) * \boxed{\diamond}^{\text{dashed}} * h \subseteq \lfloor s \rfloor * \boxed{\mathsf{Inv}_x}\} \\
\quad \left|\begin{array}{l}
\{\mathsf{hist}(h) * \mathsf{hist}(h) * x \rightharpoonup_u h * \mathsf{pending} * \boxed{\diamond}^{\text{dashed}}\} \\
\quad \mathsf{write}(x,0) \\
\left\{\begin{array}{l}\exists a, s' \supseteq s.\, \mathsf{Seen}(i, s' \uplus \{a\}) * \mathsf{hist}(h \uplus \{\lfloor a \rfloor\}) \\ * \mathsf{hist}(h \uplus \{\lfloor a \rfloor\}) * x \rightharpoonup_u h \uplus \{\lfloor a \rfloor\} * \mathsf{pending} * \boxed{\diamond}^{\text{dashed}}\end{array}\right\}
\end{array}\right. \text{open } \mathsf{Inv}_x \\
\{\exists h', s'.\, \mathsf{Seen}(i,s') * \mathsf{hist}(h) * \boxed{\diamond}^{\text{dashed}} * h' \subseteq s' * \boxed{\mathsf{Inv}_x}\} \\
\{\mathsf{Seen}(i,s') * \mathsf{hist}(h') * \boxed{\diamond}^{\text{dashed}} * h' \subseteq \lfloor s' \rfloor * \boxed{\mathsf{Inv}_x}\} \\
\quad \left|\begin{array}{l}
\{\mathsf{hist}(h) * \mathsf{hist}(h) * x \rightharpoonup_u h * \mathsf{pending} * \boxed{\diamond}^{\text{dashed}}\} \\
\quad \mathsf{write}(x,37) \\
\left\{\begin{array}{l}\exists s'' \supseteq s', a'.\, \mathsf{Seen}(i, s'' \uplus \{a'\}) * \mathsf{hist}(h \uplus \{\lfloor a' \rfloor\}) * \mathsf{hist}(h \uplus \{\lfloor a' \rfloor\}) * x \rightharpoonup_u h \uplus \{\lfloor a' \rfloor\} \\ * \boxed{\diamond}^{\text{dashed}} * \mathrm{Maximum}(h \uplus \{\lfloor a' \rfloor\}) = \mathsf{Some}\ \lfloor a' \rfloor * \mathsf{shot}(\lfloor a' \rfloor)\end{array}\right\}
\end{array}\right. \text{open } \mathsf{Inv}_x \\
\{\exists h', s''.\, \mathsf{Seen}(i,s'') * \mathsf{hist}(h') * \boxed{\mathsf{Inv}_x}\}
\end{array}
$$

Node j, proof outline

$$
\begin{array}{l}
\{\mathsf{Seen}(j,s) * \boxed{\mathsf{Inv}_x} * \boxed{\mathsf{Inv}_y}\} \\
\quad \mathsf{wait}(x = 37) \\
\{\exists s' \supseteq s, a_x.\, \mathsf{Seen}(j,s) * \mathsf{shot}(\lfloor a_x \rfloor) * w_x \in s' * \ldots\} \\
\{\mathsf{Seen}(j,s') * \mathsf{shot}(w_x)\} \\
\quad \left|\begin{array}{l}
\{\exists h_y, s'.\, x \rightharpoonup_u h_y * \ldots\} \\
\quad \mathsf{write}(y,1) \\
\left\{\begin{array}{l}\exists a_y, s'' \supseteq s'.\, \mathsf{Seen}(j, s'' \uplus \{a_y\}) * x \rightharpoonup_u h_y \uplus \{\lfloor a_y \rfloor\} * \mathsf{shot}(w_x) * \\ a_y.v = 1 * a_y \in s'' * vc(\lfloor a_x \rfloor) < vc(\lfloor a_y \rfloor)\end{array}\right\}
\end{array}\right. \text{open } \mathsf{Inv}_y \\
\{\mathsf{Seen}(j,s'') * \boxed{\mathsf{Inv}_y}\}
\end{array}
$$

Node k, proof outline

$$
\begin{array}{l}
\{\mathsf{Seen}(k,s) * \boxed{\mathsf{Inv}_x} * \boxed{\mathsf{Inv}_y} *\} \\
\quad \mathsf{wait}(y = 1) \\
\{\exists s' \subseteq s, a_y \in s', w_x.\, \mathsf{Seen}(k,s') * \mathsf{shot}(w_x) * vc(w_x) < vc(\lfloor a_y \rfloor) * a_y.v = 1\} \\
\{\mathsf{Seen}(k,s') * \mathsf{shot}(w_x)\} \\
\quad \left|\begin{array}{l}
\{\exists h_x.\, x \rightharpoonup_u h_x * \mathsf{shot}(w_x) * \mathsf{shot}(w_x) * \mathrm{Maximum}(h_x) = \mathsf{Some}\ w_x * w_x.v = 37\} \\
\quad \mathsf{read}(x) \\
\{v.\, \exists s''.\, \mathsf{Seen}(k,s'') * v = \mathsf{Some}\ 37\}
\end{array}\right. \text{open } \mathsf{Inv}_x \\
\{v.\, \exists s''.\, \mathsf{Seen}(k,s'') * v = \mathsf{Some}\ 37\}
\end{array}
$$

B Guarantees for Client-Centric Consistency

Session manager library implementation

```
(* Client stub *)
type db_key
type db_value

type sm_req =
  IR
| RR of db_key
| WR of db_key * db_value

type sm_res =
  InitRes
| ReadRes of db_value
| WriteRes

let rec listen_wait_seqid skt seq_id =
  let res_raw = listen_wait skt in
  let res = deser_res (fst res_raw) in
  let tag = fst res in
  let vl = snd res in
  if (tag = !seq_id) then
    seq_id := !seq_id + 1;
    vl
  else
    listen_wait_seqid skt seq_id

let ses_exec skt seq_id lock A req =
  acquire lock;
  let msg = ser_req req in
  sendTo skt msg A;
  let res = listen_wait_seqid skt seq_id in
  release lock;
  res
```

```
(* Request handler *)
let rec request_handler skt rd_fn wr_fn =
  let req_raw = listen_wait skt in
  let sender = snd req_raw in
  let req = deser_req (fst req_raw) in
  let seq_id = fst req in
  let res =
    match (snd req) with
    | Some IR -> InitRes
    | Some RR k -> ReadRes (rd_fn k)
    | Some WR (k, v) -> wr_fn k v; WriteRes
    | None -> assert false
  in
  sendTo skt (ser_res (seq_id, res)) sender;
  request_handler skt rd_fn wr_fn

let server dbs db_id req_addr =
  let fns = init dbs db_id in
  let rd_fn = fst fns in
  let wr_fn = snd fns in
  let skt = socket () in
  socketbind skt req_addr;
  request_handler skt rd_fn wr_fn

let sm_setup client_addr =
  let skt = socket () in
  socketbind skt client_addr;
  let seq_id = ref 0 in
  let l = newlock () in
  let connect_fn A =
    ses_exec skt seq_id l A IR; () in
  let read_fn A key =
    ses_exec skt seq_id l A (RR key) in
  let write_fn A key vl =
    ses_exec skt seq_id l A (WR (key, vl)) in
  (connect_fn, read_fn, write_fn)
```

Session manager specifications

SM-INIT

$$\{\top\}\ \langle ip_{client};\ \mathsf{sconnect}(ip_i)\rangle \left\{ \exists s.\mathsf{Seen}(i,s) * \mathop{\Large *}_{k\in\mathsf{Keys}} \exists h_k.\mathsf{Snap}(k,h_k) * \boxed{\mathsf{GlobalInv}}^{\mathcal{N}_{\mathsf{GI}}} \right\}$$

SM-READ

$$\{ip_i \Rrightarrow \Phi_i * \mathsf{Seen}(i,s) * \mathsf{Snap}(k,h)\}$$
$$\langle ip_{client};\ \mathsf{sread}(ip_i,k)\rangle$$
$$\left\{ \begin{array}{l} \exists s' \supseteq s, h' \supseteq h.\ * \mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') * \\ v.\big(\ (v = \mathsf{None} * s'_{|k} = \emptyset) \\ \quad \vee \big(\exists a, w.\ v = \mathsf{Some}\ w * a.v = w * a.k = k * a \in \mathrm{Maximals}(s'_{|k}) * \lfloor a \rfloor \in h'\big)\ \big) \end{array} \right\}$$

SM-WRITE

$$\{ip_i \Rrightarrow \Phi_i * \mathsf{Seen}(i,s) * \mathsf{Snap}(k,h)\}$$
$$\langle ip_{client};\ \mathsf{swrite}(ip_i,k,v)\rangle$$
$$\left\{ \begin{array}{l} \exists a, s' \subseteq s, h' \subseteq h.\ a.k = k * a.v = v * \mathsf{Seen}(i,s') * \mathsf{Snap}(k,h') \\ * a \notin s * a' \in s' * \lfloor a \rfloor \notin h * \lfloor a' \rfloor \in h' * \lfloor a \rfloor \in \mathrm{Maximals}(h') * \mathrm{Maximum}(s') = \mathsf{Some}\ a \end{array} \right\}$$

Session guarantees specifications

SM-READ-YOUR-WRITES

$$\{ip_i \Rrightarrow \Phi_i\}$$
$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{swrite}(ip_i,k,v_w); \mathsf{sread}(ip_i,k)\rangle$$
$$\left\{ v_o.\ \begin{array}{l} \exists s, a_w, a_r, v_r.\ v_o = \mathsf{Some}\ v_r * a_w.k = k * a_w.v = v_w * a_r.k = k * a_r.v = v_r \\ * \mathsf{Seen}(a,s) * a_w, a_r \in s * \neg(a_r.t < a_w.t) \end{array} \right\}$$

SM-MONOTONIC-READS

$$\{ip_i \Rrightarrow \Phi_i\}$$
$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{let}\ v_1 = \mathsf{sread}(ip_i,k)\ \mathsf{in}\ \mathsf{let}\ v_2 = \mathsf{sread}(ip_i,k)\ \mathsf{in}\ (v_1,v_2)\rangle$$
$$\left\{ \begin{array}{l} \exists v_{o1}, v_{o2}.\ v_o = (v_{o1}, v_{o2}) \\ \quad v_o.\ * \left(\begin{array}{l} (\exists s.v_{o1} = \mathsf{None} * v_{o2} = \mathsf{None} * \mathsf{Seen}(a_1,s) * s_{|k} = \emptyset) \vee \\ (\exists s, v_2, a_2.v_{o1} = \mathsf{None} * v_{o2} = \mathsf{Some}\ v_2 * \mathsf{Seen}(a_1,s) \\ * a_2.k = k * a_2.v = v_2 * a_2 \in \mathrm{Maximals}(s_{|k})) \vee \\ (\exists s, v_1, v_2, a_1, a_2.v_{o1} = \mathsf{Some}\ v_1 * v_{o2} = \mathsf{Some}\ v_2 * \mathsf{Seen}(a_1,s) \\ * a_1.k = k * a_1.v = v_1 * a_2.k = k * a_2.v = v_2 * a_2 \in \mathrm{Maximals}(s_{|k}) \\ * \neg(a_2.t < a_1.t)) \end{array} \right) \end{array} \right\}$$

SM-MONOTONIC-WRITES

$$\{ip_i \Rightarrow \Phi_i\}$$

$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{swrite}(ip_i, k_1, v_1); \mathsf{swrite}(ip_i, k_2, v_2)\rangle$$

$$\left\{\begin{array}{l}\exists s_1, a_1, a_2.\ a_1.k = k_1 * a_1.v = v_1 * a_2.k = k_2 * a_2.v = v_2 \\ * \mathsf{Seen}(a_1, s_1) * a_1, a_2 \in s_1 * a_1.t < a_2.t \\ * (\forall a, s_2, a_2.\mathsf{Seen}(a_2, s_2) * a \in s_2 * a_2.t \leq e.t \\ \qquad \Longrightarrow\!\!\!\!\!\!\ast_{\top}\ \exists a_1', a_2'.\ \lfloor a_1' \rfloor = \lfloor a_1 \rfloor * \lfloor a_2' \rfloor = \lfloor a_2 \rfloor * a_1', a_2' \in s_2 * a_1'.t < a_2'.t)\end{array}\right\}$$

SM-WRITES-FOLLOW-READS

$$\{ip_i \Rightarrow \Phi_i\}$$

$$\langle ip_{client};\ \mathsf{sconnect}(ip_i); \mathsf{let}\ v = \mathsf{sread}(ip_i, k_r)\ \mathsf{in}\ \mathsf{swrite}(ip_i, k_w, v_w); v\rangle$$

$$\left\{\begin{array}{l}\exists s_1, a_w.\ a_w.k = k_w * a_w.v = v_w * \mathsf{Seen}(a_1, s_1) * a_w \in s_1 \\ v_o.\ * \left(\begin{array}{l} v_o = \mathsf{None} \vee \\ \exists a_r, v_r.\ v_o = \mathsf{Some}\ v_r * a_r.k = k_r * a_r.v = v_r * a_r \in s_1 * a_r.t < a_w.t \\ * (\forall a, s_2, a_2.\mathsf{Seen}(a_2, s_2) * a \in s_2 * a_w.t \leq e.t \\ \qquad \Longrightarrow\!\!\!\!\!\!\ast_{\top}\ \exists a_r', a_w'.\ \lfloor a_r' \rfloor = \lfloor a_r \rfloor * \lfloor a_w' \rfloor = \lfloor a_w \rfloor * a_r', a_w' \in s_2 * a_r'.t < a_w'.t)\end{array}\right)\end{array}\right\}$$

C Counterexample

At a first glance, the high-level relational rules for reasoning about *presampling tapes* may cause one to believe that once the idea of presampling arises, the rest is mostly straightforward and obvious to state and define. But this is *not* the case: a great deal of care goes into phrasing the rules of the operational semantics and the relational logic if you do not want to end up with an unsound system! One may also wonder whether it is in fact necessary for tapes and labels to appear in the program and program state, but as we will illustrate below, they do in fact play a subtle yet crucial role.

Consider the following program *flip_or* that applies a logical disjunction to two freshly sampled bits

$$\begin{aligned} \mathit{flip\_or} \triangleq\ & \mathsf{let}\ x = \mathsf{flip}()\ \mathsf{in} \\ & \mathsf{let}\ y = \mathsf{flip}()\ \mathsf{in} \\ & x \mid\mid y \end{aligned}$$

and compare it to the program *flip* that just samples a bit

$$\mathit{flip} \triangleq \mathsf{flip}()$$

These two programs are obviously *not* contextually equivalent: with probability $3/4$ the program *flip_or* will return true whereas the program *flip* only does so with probability $1/2$. Yet, if we introduce a rule for flip that could draw from any pre-sampled tape (*i.e.*, without requiring sampling statements to be annotated with the tape they will draw from), we would be able to prove that they are equivalent.

Let us introduce the following (unsound!) rule:

$$\textsc{rel-tape-unsound}\quad \frac{\iota \hookrightarrow b \cdot \vec{b} \qquad \iota \hookrightarrow \vec{b} \mathrel{-\!\!*} \Delta \vDash_{\mathcal{E}} K[\,b\,] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[\,\mathsf{flip}()\,] \precsim e_2 : \tau}$$

The rule says that when sampling on the left-hand side, we may instead draw a bit b from *some* presampling tape ι. To see why this rule cannot be sound, we will show

$$\vDash \mathit{flip} \precsim \mathit{flip\_or} : \mathsf{bool}$$

First, we introduce two tapes with resources $\iota_1 \hookrightarrow \epsilon$ and $\iota_2 \hookrightarrow \epsilon$ on the left-hand side (either explicitly in code as in Clutch or as pure ghost resources, if that is possible in our hypothetical logic). Second, we couple the tape ι_1 with the x-sampling and ι_2 with the y-sampling using REL-COUPLE-TAPE-L such that we end up with $\iota_1 \hookrightarrow b_1$ and $\iota_2 \hookrightarrow b_2$ and the goal

$$\vDash \mathsf{flip}() \precsim b_1 \mid\mid b_2 : \mathsf{bool}$$

Finally, we do a case distinction on both b_1 and b_2: if both of them are true, or both are false, it does not matter which tape we use when applying REL-TAPE-UNSOUND. If, on the other hand, only b_i is true, we choose ι_i and apply REL-TAPE-UNSOUND which finishes the proof.

The crucial observation is that by labeling tapes in the program syntax, however, we prevent *the prover* from doing case analysis on presampled values to decide which tape to read—the syntax will dictate which tape to use and hence which value to read. Concretely, in $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$, unlabeled flips always reduce uniformly at random and only labeled sampling statements will read from presampling tapes which prevents us from proving the unsound REL-TAPE-UNSOUND.

Besides motivating why soundly allowing presampling is subtle, this counterexample also emphasizes why the fact that labels appear in the program and in the program syntax is important. We do not claim that it is a necessity, but like for *prophecy variables* [Jun+20] where similar "ghost information" is needed in the actual program code, it is not at all obvious how to do without it.

D Lazy/eager coin

In this section we give a more detailed proof of the lazy-eager coin example from Section 7.1. We will go through the proof step by step but omit the use of REL-PURE-L and REL-PURE-R which should be interleaved with the application of most of the mentioned proof rules.

Recall the definitions

$$
\begin{aligned}
\mathit{eager} &\triangleq \mathsf{let}\, b = \mathsf{flip}() \,\mathsf{in}\, \lambda\_.\, b \\
\mathit{lazy} &\triangleq \mathsf{let}\, r = \mathsf{ref}(\mathsf{None}) \,\mathsf{in} \\
&\quad \lambda\_.\, \mathsf{match}\, !r \,\mathsf{with} \\
&\qquad \mathsf{Some}\ (b) \Rightarrow b \\
&\qquad \mid \mathsf{None} \quad \Rightarrow \mathsf{let}\, b = \mathsf{flip}() \,\mathsf{in} \\
&\qquad\qquad\qquad\quad r \leftarrow \mathsf{Some}\ (b); \\
&\qquad\qquad\qquad\quad b \\
&\quad \mathsf{end}
\end{aligned}
$$

The goal is to show $\vdash \mathit{lazy} \simeq_{\mathrm{ctx}} \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$ which we do by first showing $\mathit{lazy} \precsim_{\mathrm{ctx}} \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$ and then $\mathit{eager} \precsim_{\mathrm{ctx}} \mathit{lazy} : \mathsf{unit} \to \mathsf{bool}$.

To show $\mathit{lazy} \precsim_{\mathrm{ctx}} \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$, we first define an intermediate labeled version of lazy:

$$
\begin{aligned}
\mathit{lazy}' &\triangleq \mathsf{let}\, \iota = \mathsf{tape} \,\mathsf{in} \\
&\quad \mathsf{let}\, r = \mathsf{ref}(\mathsf{None}) \,\mathsf{in} \\
&\quad \lambda\_.\, \mathsf{match}\, !r \,\mathsf{with} \\
&\qquad \mathsf{Some}\ (b) \Rightarrow b \\
&\qquad \mid \mathsf{None} \quad \Rightarrow \mathsf{let}\, b = \mathsf{flip}(\iota) \,\mathsf{in} \\
&\qquad\qquad\qquad\quad r \leftarrow \mathsf{Some}\ (b); \\
&\qquad\qquad\qquad\quad b \\
&\quad \mathsf{end}
\end{aligned}
$$

By transitivity of contextual refinement and Theorem 7.2.1 it is sufficient to show $\vDash \mathit{lazy} \precsim \mathit{lazy}' : \mathsf{unit} \to \mathsf{bool}$ and $\vDash \mathit{lazy}' \precsim \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$.

The first refinement $\vDash \mathit{lazy} \precsim \mathit{lazy}' : \mathsf{unit} \to \mathsf{bool}$ is mostly straightforward. By applying REL-ALLOC-L followed by REL-ALLOC-TAPE-R and REL-ALLOC-R we are left with the goal of proving that the two thunks are related, given $\iota \hookrightarrow_{\mathsf{s}} \epsilon$, $\ell \mapsto \mathsf{None}$ and $\ell' \mapsto_{\mathsf{s}} \mathsf{None}$ for some fresh label ι and fresh locations on the heap ℓ and ℓ'. Using REL-NA-INV-ALLOC we allocate the invariant

$$\iota \hookrightarrow_{\mathsf{s}} \epsilon * ((\ell \mapsto \mathsf{None} * \ell' \mapsto_{\mathsf{s}} \mathsf{None}) \vee (\exists b.\, \ell \mapsto \mathsf{Some}\ (b) * \ell' \mapsto_{\mathsf{s}} \mathsf{Some}\ (b)))$$

with some name $\mathcal{N}$ that expresses how the ι tape is always empty and that *either* both ℓ and ℓ' contain None *or* both contain Some (b) for some b. We continue by REL-REC after which

we open the invariant and do a case distinction on the disjunction in the invariant. If ℓ and ℓ' are empty, this is the first time we invoke the function. We continue using REL-LOAD-L and REL-LOAD-R:

REL-LOAD-L
$$\frac{\ell \mapsto v \qquad \ell \mapsto v \twoheadrightarrow \Delta \vDash_{\mathcal{E}} K[v] \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}} K[!\ell] \precsim e_2 : \tau}$$

REL-LOAD-R
$$\frac{\ell \mapsto_{\mathsf{s}} v \qquad \ell \mapsto_{\mathsf{s}} v \twoheadrightarrow \Delta \vDash_{\mathcal{E}} e_1 \precsim K[v] : \tau}{\Delta \vDash_{\mathcal{E}} e_1 \precsim K[!\ell] : \tau}$$

after which we are left with the goal

$$\vDash_{\top\setminus\mathcal{N}} \begin{array}{l}\mathsf{let}\ b = \mathsf{flip}()\ \mathsf{in}\\ r \leftarrow \mathsf{Some}\ (b); b\end{array} \precsim \begin{array}{l}\mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in}\\ r \leftarrow \mathsf{Some}\ (b); b\end{array} : \mathsf{unit} \to \mathsf{bool}$$

We continue using REL-FLIP-ERASE-R to couple the two flips, we follow by REL-STORE-L and REL-STORE-R to store the fresh bit on the heaps, we close the invariant (now showing the right disjunct as the locations have been updated) using REL-NA-INV-CLOSE, and we finish the case using REL-RETURN as the program returns the same Boolean b on both sides.

If ℓ and ℓ' were *not* empty, this is not the first time the function is invoked and we load the same Boolean on both sides using REL-LOAD-L and REL-LOAD-R and finish the proof using REL-NA-INV-CLOSE and REL-RETURN.

For the second refinement $\vDash \mathit{lazy}' \precsim \mathit{eager} : \mathsf{unit} \to \mathsf{bool}$ we start by allocating the tape on the left using REL-ALLOC-TAPE-L which gives us ownership of a fresh tape $\iota \hookrightarrow \epsilon$. We now couple the ι tape with the unlabeled $\mathsf{flip}()$ on the right using REL-COUPLE-TAPE-L. This gives us that for some b then $\iota \hookrightarrow b$ and the $\mathsf{flip}()$ on the right returned b as well. We continue by allocating the reference on the left using REL-ALLOC-L which gives us some location ℓ and $\ell \mapsto \mathsf{None}$. Now, we allocate the invariant

$$(\iota \hookrightarrow b * \ell \mapsto \mathsf{None}) \vee \ell \mapsto \mathsf{Some}\ (b)$$

for some name $\mathcal{N}$ which expresses that *either* the location ℓ is empty but b is on the ι tape, *or* b has been stored at ℓ. We are now left with proving that the two thunks are related under this invariant. We continue using REL-REC after which we open the invariant using REL-NA-INV-OPEN, do a case distinction on the disjunction, and continue using REL-LOAD-L. If the location ℓ is empty, we have to show

$$\vDash_{\top\setminus\mathcal{N}} \begin{array}{l}\mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in}\\ r \leftarrow \mathsf{Some}\ (b); b\end{array} \precsim b : \mathsf{unit} \to \mathsf{bool}$$

But as we own $\iota \hookrightarrow b$ we continue using REL-FLIP-TAPE-L, REL-STORE-L, REL-NA-INV-CLOSE (now establishing the right disjunct as ℓ has been updated), and REL-RETURN as the return value b is the same on both sides. If the location ℓ was *not* empty, we know $\ell \mapsto \mathsf{Some}\ (b)$ which means REL-LOAD-L reads b from ℓ and we finish the proof using REL-NA-INV-CLOSE (reestablishing the right disjunct) and REL-RETURN.

The proof of $\mathit{eager} \precsim_{\mathrm{ctx}} \mathit{lazy} : \mathsf{unit} \to \mathsf{bool}$ is analogous and we have shown the contextual equivalence of the programs eager and lazy.

E Model of Clutch

The value interpretation of types is shown in Figure E.1.

Weakest precondition

The full definition of the weakest precondition, including the fancy update modality and invariant masks, is the guarded fixpoint of the equation found below.

$$\begin{aligned}\mathsf{wp}_{\mathcal{E}}\, e_1\, \{\Phi\} \triangleq\ & (e_1 \in \mathit{Val} \wedge \Rrightarrow_{\mathcal{E}} \Phi(e_1)) \vee \\ & (e_1 \notin \mathit{Val} \wedge \forall \sigma_1, \rho_1. \\ & \quad S(\sigma_1) * G(\rho_1) \mathrel{-\!\!*} {}_{\mathcal{E}}\Rrightarrow_{\emptyset} \\ & \quad \mathsf{execCoupl}(e_1, \sigma_1, \rho_1)(\lambda e_2, \sigma_2, \rho_2. \\ & \qquad \triangleright\, {}_{\emptyset}\Rrightarrow_{\mathcal{E}} S(\sigma_2) * G(\rho_2) * \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\}))\end{aligned}$$

Coupling modality

To define the coupling modality, we define a stratified partial execution distribution $\mathrm{execConf}_n(e, \sigma) \in \mathcal{D}(\mathit{Cfg})$.

$$\mathrm{execConf}_n(e, \sigma) \triangleq \begin{cases} \mathrm{ret}(e, \sigma) & \text{if } e \in \mathit{Val} \text{ or } n = 0 \\ \mathrm{step}(e, \sigma) \gg\!= \mathrm{execConf}_{(n-1)} & \text{otherwise} \end{cases}$$

The coupling modality used in the definition of the weakest precondition is found in Figure E.2. Simple rules that follow by unfolding are found in Figure E.3.

$$
\begin{aligned}
[\![\alpha]\!]_\Delta(v_1, v_2) &\triangleq \Delta(\alpha)(v_1, v_2) \\
[\![\mathsf{unit}]\!]_\Delta(v_1, v_2) &\triangleq v_1 = v_2 = () \\
[\![\mathsf{int}]\!]_\Delta(v_1, v_2) &\triangleq \exists z \in \mathbb{Z}.\, v_1 = v_2 = z \\
[\![\mathsf{bool}]\!]_\Delta(v_1, v_2) &\triangleq \exists b \in \mathbb{B}.\, v_1 = v_2 = b \\
[\![\tau \to \sigma]\!]_\Delta(v_1, v_2) &\triangleq \Box\, (\forall w_1, w_2.\, [\![\tau]\!]_\Delta(w_1, w_2) \mathrel{-\!\!*} \Delta \vDash v_1\ w_1 \precsim v_2\ w_2 : \sigma) \\
[\![\tau \times \sigma]\!]_\Delta(v_1, v_2) &\triangleq \exists w_1, w_1', w_2, w_2'.\, (v_1 = (w_1, w_1')) * (v_2 = (w_2, w_2')) * \\
&\qquad\qquad [\![\tau]\!]_\Delta(w_1, w_2) * [\![\sigma]\!]_\Delta(w_1', w_2') \\
[\![\tau + \sigma]\!]_\Delta(v_1, v_2) &\triangleq \exists w_1, w_2.\, (v_1 = \mathsf{inl}(w_1) * v_2 = \mathsf{inl}(w_2) * [\![\tau]\!]_\Delta(w_1, w_2)) \ \vee \\
&\qquad\qquad (v_1 = \mathsf{inr}(w_1) * v_2 = \mathsf{inr}(w_2) * [\![\sigma]\!]_\Delta(w_1, w_2)) \\
[\![\mu\, \alpha.\, \tau]\!]_\Delta(v_1, v_2) &\triangleq (\mu R.\, \lambda(v_1, v_2).\, \exists w_1, w_2.\, (v_1 = \mathsf{fold}\, w_1) * (v' = \mathsf{fold}\, w_2) * \\
&\qquad\qquad \triangleright [\![\tau]\!]_{\Delta, \alpha \mapsto R}(w_1, w_2))(v_1, v_2) \\
[\![\forall \alpha.\, \tau]\!]_\Delta(v_1, v_2) &\triangleq \Box\ (\forall R.\, (\Delta, \alpha \mapsto R \vDash v_1\ \_ \precsim v_2\ \_ : \tau) \\
[\![\exists \alpha.\, \tau]\!]_\Delta(v_1, v_2) &\triangleq \exists R, w_1, w_2.\, (v_1 = \mathsf{pack}\, w_2) * (v_2 = \mathsf{pack}\, w_2) * [\![\tau]\!]_{\Delta, \alpha \mapsto R}(w_1, w_2) \\
[\![\mathsf{ref}\ \tau]\!]_\Delta(v_1, v_2) &\triangleq \exists \ell_1, \ell_2.\, (v_1 = \ell_1) * (v_2 = \ell_2) * \\
&\qquad\qquad \boxed{\exists w_1, w_2.\, \ell_1 \mapsto w_1 * \ell_2 \mapsto_{\mathsf{s}} w_2 * [\![\tau]\!]_\Delta(w_1, w_2)}^{\mathcal{N}.\ell_1.\ell_2} \\
[\![\mathsf{tape}]\!]_\Delta(v_1, v_2) &\triangleq \exists \iota_1, \iota_2.\, (v_1 = \iota_1) * (v_2 = \iota_2) * \boxed{\iota_1 \hookrightarrow \epsilon * \iota_2 \hookrightarrow_{\mathsf{s}} \epsilon}^{\mathcal{N}.\iota_1.\iota_2}
\end{aligned}
$$

Figure E.1: Relational interpretation of types

$$
\begin{aligned}
&\mathsf{execCoupl}(e_1, \sigma_1, e_1', \sigma_1')(Z) \triangleq \mu\Psi : \mathit{Cfg} \times \mathit{Cfg} \to \mathit{iProp}.\\
&\quad (\exists R.\, \mathsf{red}(e_1, \sigma_1) *\\
&\qquad \mathrm{step}(e_1, \sigma_1) \sim \mathrm{step}(e_1', \sigma_1') : R *\\
&\qquad \forall \rho_2, \rho_2'.\, R(\rho_2, \rho_2') \mathrel{-\!\!*} \Rrightarrow_\emptyset Z(\rho_2, \rho_2')) \lor\\
&\quad (\exists R.\, \mathsf{red}(e_1, \sigma_1) *\\
&\qquad \mathrm{step}(e_1, \sigma_1) \sim \mathrm{ret}(e_1', \sigma_1') : R *\\
&\qquad \forall \rho_2.\, R(\rho_2, (e_1', \sigma_1')) \mathrel{-\!\!*} \Rrightarrow_\emptyset Z(\rho_2, (e_1', \sigma_1'))) \lor\\
&\quad (\exists R, n.\ \mathrm{ret}(e_1, \sigma_1) \sim \mathrm{execConf}_n(e_1', \sigma_1') : R *\\
&\qquad\quad \forall \rho_2'.\, R((e_1, \sigma_1), \rho_2') \mathrel{-\!\!*} \Rrightarrow_\emptyset \Psi((e_1, \sigma_1), \rho_2')) \lor\\
&\quad \left(\bigvee_{\iota \in \sigma_1} \begin{array}{l} \exists R.\ \mathrm{step}_\iota(\sigma_1) \sim \mathrm{step}(e_1', \sigma_1') : R *\\ \quad \forall \sigma_2, \rho_2'.\, R(\sigma_2, \rho_2') \mathrel{-\!\!*} \Rrightarrow_\emptyset \Psi((e_1, \sigma_2), \rho_2') \end{array}\right) \lor\\
&\quad \left(\bigvee_{\iota' \in \sigma_2} \begin{array}{l} \exists R.\ \mathrm{step}(e_1, \sigma_1) \sim \mathrm{step}_{\iota'}(\sigma_1') : R *\\ \quad \forall \rho_2, \sigma_2'.\, R(\rho_2, \sigma_2') \mathrel{-\!\!*} \Rrightarrow_\emptyset Z(\rho_2, (e_1', \sigma_2')) \end{array}\right) \lor\\
&\quad \left(\bigvee_{(\iota, \iota') \in \sigma_1 \times \sigma_1'} \begin{array}{l} \exists R.\ \mathrm{step}_\iota(\sigma_1) \sim \mathrm{step}_{\iota'}(\sigma_1') : R *\\ \quad \forall \sigma_2, \sigma_2'.\, R(\sigma_2, \sigma_2') \mathrel{-\!\!*} \Rrightarrow_\emptyset\\ \qquad\qquad \Psi((e_1, \sigma_2), (e_1', \sigma_2')) \end{array}\right)
\end{aligned}
$$

Figure E.2: Full definition of execCoupl.

$$\frac{\mathsf{red}(\rho_1) \qquad \mathrm{step}(\rho_1) \sim \mathrm{step}(\rho_1') : R \qquad \forall \rho_2, \rho_2'.\, R(\rho_2, \rho_2') \mathrel{-\!\!*} Z(\rho_2, \rho_2')}{\mathsf{execCoupl}(\rho_1, \rho_1')(Z)}$$

$$\frac{\mathsf{red}(\rho_1) \qquad \mathrm{step}(\rho_1) \sim \mathrm{ret}(\rho_1') : R \qquad \forall \rho_2.\, R(\rho_2, \rho_1') \mathrel{-\!\!*} \mathsf{execCoupl}(\rho_2, \rho_1')(Z)}{\mathsf{execCoupl}(\rho_1, \rho_1')(Z)}$$

$$\frac{\mathrm{ret}(\rho_1) \sim \mathrm{execConf}_n(\rho_1') : R \qquad \forall \rho_2'.\, R(\rho_1, \rho_2') \mathrel{-\!\!*} \mathsf{execCoupl}(\rho_1, \rho_2')(Z)}{\mathsf{execCoupl}(\rho_1, \rho_1')(Z)}$$

$$\frac{\mathrm{ret}(\rho_1) \sim \mathrm{step}(\rho_1') : R \qquad \forall \rho_2'.\, R(\rho_1, \rho_2') \mathrel{-\!\!*} \mathsf{execCoupl}(\rho_1, \rho_2')(Z)}{\mathsf{execCoupl}(\rho_1, \rho_1')(Z)}$$

$$\frac{\mathrm{step}_\iota(\sigma_1) \sim \mathrm{step}(\rho_1') : R \qquad \forall \sigma_2, \rho_2'.\, R(\sigma_2, \rho_2') \mathrel{-\!\!*} \mathsf{execCoupl}((e_1, \sigma_2), \rho_2')(Z)}{\mathsf{execCoupl}((e_1, \sigma_1), \rho_1')(Z)}$$

$$\frac{\begin{array}{c}\mathsf{red}(\rho_1) \\ \mathrm{step}(\rho_1) \sim \mathrm{step}_\iota(\sigma_1') : R \end{array} \qquad \forall \rho_2, \sigma_2'.\, R(\rho_2, \sigma_2') \mathrel{-\!\!*} \mathsf{execCoupl}(\rho_2, (e_1', \sigma_2'))(Z)}{\mathsf{execCoupl}(\rho_1, (e_1', \sigma_1'))(Z)}$$

$$\frac{\mathrm{step}_\iota(\sigma_1) \sim \mathrm{step}_{\iota'}(\sigma_1') : R \qquad \forall \sigma_2, \sigma_2'.\, R(\sigma_2, \sigma_2') \mathrel{-\!\!*} \mathsf{execCoupl}((e_1, \sigma_2), (e_1', \sigma_2'))(Z)}{\mathsf{execCoupl}((e_1, \sigma_1, (e_1', \sigma_1'))(Z)}$$

Figure E.3: execCoupl unfolding rules.

F On Case Studies and Additional Examples

F.1 Eager/Lazy Hash Function

As explained in the body of the paper, it is common to model the hash function as if it satisfies the so-called *uniform hash assumption* or the *random oracle model*. That is, a hash function h from a set of keys K to values V behaves as if, for each key k, the hash $h(k)$ is randomly sampled from a uniform distribution over V, independently of all the other keys.

Figure F.1 gives the complete code for the eager hash function that was excerpted earlier. Given a non-negative integer n, executing $eager\_hash\ n$ returns a hash function with $K = \{0, \ldots, n\}$ and $V = \mathbb{B}$. To do so, it first initializes a mutable map m, and then calls $sample\_all$ on m. For each key $k \in K$, the function $sample\_all$ samples a boolean b with flip and stores the value b for the key k in the map m. This sampled boolean serves as the hash for b. The function returned by $eager\_hash$ uses this map to look up the hash values of keys. On input k, it looks up k in the map and returns the resulting value if one is found, and otherwise returns false. Since $sample\_all$ adds every key in K to the map, this latter scenario only happens if k is not in K.

Figure F.2 gives the full code for the lazy sampling version of the random hash generator. Given a non-negative integer n, executing $lazy\_hash\ n$ returns a hash function for the key space $K = \{0, \ldots, n\}$. For its internal state, it uses two physical maps, the tape map tm, stores tapes to be used for random sampling, and the value map vm, stores the previously sampled values for keys that have been hashed. After initializing these maps, it calls $alloc\_tapes$, which allocates a tape for each key $k \in K$ and stores the associated tape in tm. The hash function returned by $lazy\_hash$ determines the hash for a key k in two stages. It first looks up k in vm to see if k already has a previously sampled hash value, and if so, returns the found value. Otherwise, it looks up k in the tape map tm. If no tape is found, then k must not be in K, so

```
sample_all ≜ rec f m n =
               let n' = n − 1 in
               if n' < 0 then () else
                 let b = flip() in
                 set m n' b;
                 f m n'

eager_hash ≜ λn. let m = init_map () in
                 sample_all m (n + 1);
                 (λk. match get m k with
                        Some (b) ⇒ b
                      | None     ⇒ false
                      end)
```

Figure F.1: Eager hash function.

```
alloc_tapes ≜ rec f m n =
                let n' = n − 1 in
                if n' < 0 then () else
                  let ι = tape in
                  set m n' ι;
                  f m n'

lazy_hash ≜ λn.
              let vm = init_map () in
              let tm = init_map () in
              alloc_tapes tm (n + 1);
              (λk. match get vm k with
                     Some (b) ⇒
                       b
                   | None ⇒ match get tm k with
                              Some (ι) ⇒
                                let b = flip(ι) in
                                set vm b;
                                b
                            | None ⇒ false
                            end
                   end)
```

Figure F.2: Lazy hash function.

the function returns false. If a tape ι is found, then the code samples a boolean b from this tape with flip, stores b for the key k in vm, and then returns b.

We also prove that for any non-negative number n, the eager and lazy versions are contextually equivalent, that is, $\vdash$ *eager_hash* $n \simeq_{\text{ctx}}$ *lazy_hash* n : int $\to$ bool. The core idea behind this contextual equivalence proof is to maintain a particular invariant between the internal state of the two hash functions. Let m be the internal map used by the eager hash and let tm and vm be the tape and value maps, respectively, for the lazy hash. Then, at a high level, the invariant maintains the following properties:

1. $\text{dom}(m) = \text{dom}(tm) = \{0, \ldots, n\}$.
2. For all $k \in \{0, \ldots, n\}$, if $m[k] = b$ then either
   a) $vm[k] = b$, or
   b) $vm[k] = \bot$ and $tm[k] = \iota$ for some tape label ι such that $\iota \hookrightarrow [b]$.

Case (a) and (b) of the second part of this invariant capture the two possible states each key k can be in. Either hash of k has been looked up before (case a), and so the sampled value stored in vm must match that of m, or it has not been looked up (case b) and the tape for the key must contain the same value as $m[k]$ for its next bit.

To establish this invariant when the hashes are initialized, we asynchronously couple the eager hash function's flip for key k with a tape step for the tape ι associated with k in the lazy table. The invariant ensures that the values returned by the two hash functions will be the same when a key k is queried. The cases of the invariant corresponding to the branches of the lazy function's match statements: if the key k is in K and has been queried before, the maps will return the same values found in m and vm. If it has not been queried before, then the flip statement in the lazy version will be draw the value on the tape for the key, which matches

$m[k]$. Moreover, the update that writes this sampled value to vm preserves the invariant, switching from case (b) to case (a) for queried key.

F.2 Random Generators from Hashes

Here we provide further details on generating a random boolean sampler from a lazy hash function. The following function, $init\_hash\_rng$, returns a function that can be used to generate random booleans:

$$
\begin{aligned}
init\_hash\_rng \triangleq \lambda\_.\ &\mathsf{let}\ f = lazy\_hash\ \mathtt{MAX}\ \mathsf{in}\\
&\mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in}\\
&(\lambda\_.\ \mathsf{let}\ n = !\,c\ \mathsf{in}\\
&\quad \mathsf{let}\ b = f\ n\ \mathsf{in}\\
&\quad c \leftarrow n + 1;\\
&\quad b)
\end{aligned}
$$

When run, $init\_hash\_rng$ generates a lazy hash function f for the key space $K = \{0, \ldots, \mathtt{MAX}\}$ for some fixed constant MAX. It then allocates a counter c as a mutable reference initialized to 0. The returned function, let us call it h, uses f and c to generate random booleans. Each time h is called, it loads the current value n form c, hashes n with f to get a boolean b. It then increments c and returns the boolean b, which serves as a random boolean. Repeated calls to h return independent, uniformly sampled booleans, so long as we make no more than MAX calls. The reason this works is that we have assumed the hash function f is uniformly random, so the hashes of different keys are independently sampled. So long as we make fewer than MAX calls to h, each call will hash a distinct number n (the current counter value), so it will be independent of all previous and future calls. After MAX calls, c will exceed MAX and so we will hash a key outside of f's key space. Recall from the previous example that $lazy\_hash$ returns false on inputs outside its key space.

This example might at first seem artificial, but using cryptographic primitives such as hashes or block ciphers to generate pseudorandom numbers is in fact commonly done [BK15]. Although the example here is simplified compared to real implementations, it captures one of the core verification challenges common to real implementations. Namely, for correctness, one must show that the "key" or "counter" being hashed or encrypted (here the values of n obtained from c) are not re-used.

To capture the guarantees of $init\_hash\_rng$ more formally, we prove that $init\_hash\_rng$ is contextually equivalent to the following "bounded" random number generator that directly calls flip:

$$
\begin{aligned}
init\_bounded\_rng \triangleq \lambda\_.\ &\mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in}\\
&(\lambda\_.\ \mathsf{let}\ n = !\,c\ \mathsf{in}\\
&\quad \mathsf{let}\ b = \mathsf{if}\ n \leq \mathtt{MAX}\ \mathsf{then}\ \mathsf{flip}()\ \mathsf{else}\ \mathsf{false}\ \mathsf{in}\\
&\quad c \leftarrow n + 1;\\
&\quad b)
\end{aligned}
$$

With $init\_bounded\_rng$, the returned generator function again uses a counter c, however the value of this counter is just used to track the number of samples generated. If the number of

calls is less than or equal to MAX, it returns a boolean generated by a call to flip. Otherwise, it just returns false. (In $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$, integers are unbounded, so there is no issue with overflow.)

At a high level, the proof proceeds by maintaining the following invariant relating the generator functions returned by both $init\_hash\_rng$ and $init\_bounded\_rng$. Let f be the hash function in the hash-based generator, n_h be its counter value, and n_b be the value of the bounded generator's counter. Then:

1. $n_h = n_b$.
2. The key space of f is $\{0, \ldots, \texttt{MAX}\}$
3. For all $k \in \{0, \ldots, \texttt{MAX}\}$, if $k \geq n_h$ then f has not yet hashed k.

The first and second parts of the invariant guarantees that once n_h exceeds MAX, and thus falls outside f's key space, both generators will return the same value of false. In addition, based on the third part of the invariant, so long as $n_h \in \{0, \ldots, \texttt{MAX}\}$ when the hash-based generator evaluates $f\ n_h$, we will be able to couple the hash value it samples with the flip command in the bounded generator, ensuring that both generators return the same value.

The $init\_hash\_rng$ function above generates a single random generator from a hash function. However, in some scenarios, it is necessary to be able to generate multiple independent streams of random numbers. For example, in a language with parallelism or concurrency, using a single generator returned by $init\_hash\_rng$ in multiple threads would mean sharing mutable access to the counter c, so that synchronization primitives would be needed to prevent racy accesses. Related issues have motivated the need to "split" a random number generator into two streams in the context of a lazy language like Haskell [CP13].

While $\mathbf{F}^{\text{flip}}_{\mu,\text{ref}}$ is sequential, we can still explore the question of how to create multiple independent random generators. One approach would be to call $init\_hash\_rng$ multiple times. But that assumes that we have the ability to initialize multiple random oracle hash functions. In practice, if we instantiate the random oracle model with a particular concrete hash function, like SHA-256, we cannot feasibly use different hash functions each time $init\_hash\_rng$ is called. Instead, we would like a way to generate multiple independent random number generators from a single hash function.

The solution is to get the illusion of multiple independent hash functions out of a single hash function by partitioning the key space of the hash. Specifically, we use a wrapper around the hash function so that it now takes *two* integers as input, instead of one, to obtain a so-called *keyed* hash[1]:

$$
\begin{aligned}
lazy\_keyed\_hash \triangleq \lambda\_.\ & \\
& \mathsf{let}\ f = lazy\_hash\ (2^{(p_k+p_v)} - 1)\ \mathsf{in} \\
& (\lambda k\ v.\ f\ (k \cdot 2^{p_v} + v))
\end{aligned}
$$

The returned keyed hash function takes two inputs, a key k and a value v to be hashed, combines them into a single integer, and calls the lazy hash function f on that single integer. The values p_k and p_v are fixed constants that determine the range of the keys and values that can be hashed. If h is the returned hash function, we can treat the partially-applied functions $h\ k_1$

[1] In cryptographic settings, a similar construction is called a hash message authentication code (HMAC). However, HMACs typically use a different way of combining the two arguments to avoid certain vulnerabilities.

and $h\ k_2$ for $k_1 \neq k_2$ as if they were two independent hash functions, so long as we do not apply the functions to values v that are larger than $2^{p_v} - 1$.

Using this, we have verified a version of the hash-based random number generator that supports splitting multiple independent generators out of a single hash. Each generator has a distinct key k its own internal counter c. When a sample is requested, it reads the value n from the counter and computes the keyed hash of k and n to get a boolean. To ensure that the keys used by the generators are distinct, we assign keys using a shared counter that is incremented every time a new generator is initialized. The complete details can be found in the accompanying Coq development.

F.3 Lazily Sampled Big Integers

Our last example is motivated by a data structure called a treap [SA96]. A treap is a binary search tree structure that relies on randomization to ensure with high probability that the tree will be balanced. One of the key aspects of the treap is that every key that is inserted into the treap is first assigned a random numerical *priority*. During the insertion process, this priority value is compared with the priorities of keys already in the treap. The exact details of this comparison process are not relevant here; what is important to know is that, ideally, all of the assigned priority values are different (that is, there are no *collisions* of priorities). Thus, in analyzing the treap, it is common to treat these priorities as if they are sampled from a continuous distribution, such as the uniform distribution on the interval $[0, 1]$, to ensure that the probability of collisions is 0. Eberl et al. [EHN20] have previously mechanized such an analysis of treaps in Isabelle/HOL.

In actual implementations, the priorities are instead typically represented with some fixed precision, say as an approximate floating point number sampled from $[0, 1]$, or as an integer sampled uniformly from some set $\{0, \ldots, n\}$, so that there is some probability of collision. However, in the latter case, as long as n is big enough relative to the number of keys added to the tree, the probability of a collision can be kept low, and the performance properties of the treap are preserved. The probability of a collision is an instance of the well-known "birthday problem".

But in some scenarios, we may need to decide on n without knowing in advance how many keys will end up being added to the treap. If we err on the conservative side by making n very large, say $2^{256} - 1$, the probability of a collision will be very low, but we will need to use 256 bits to store the priorities, which is wasteful if we end up only storing a moderate number of nodes.

An alternative is to *lazily* sample the integer that represents the priority. The insight is that the actual numerical value of the priorities is not relevant: the only operation that they must support is comparing two priorities to determine if they are equal, and if not, which one is larger. Figure F.3 gives an implementation of a lazily-sampled integer A lazily-sampled integer is encoded as a pair of a tape label ι and a linked list of length at most N, where each node in the list represents a *digit* of the integer in base B, with the head of the list being the most significant digit. For concreteness, here we consider $N = 8$ and $B = 2^{32}$, so that the encoded numbers can be at most $2^{256} - 1$. Rather than sampling all digits up front, we instead only sample digits when needed as part of comparing a lazy integer to another.

The function *sample_lazy_int* samples a lazy integer by generating a tape and a reference to an empty linked list. At this point, the sampled integer is entirely indeterminate. Given a

```
sample_lazy_int ≜ λ_. (tape, ref None)

get_next ≜ λ ι r.
             match !r with
               Some v ⇒ v
             | None   ⇒ let z = sample32 ι in
                        let next = ref None in
                        r ← Some (z, next);
                        (z, next)
             end

cmp_list ≜ rec f n ι1 l1 ι2 l2 =
           if n = 0 then 0
           else
             let (z1, l1') = get_next ι1 l1 in
             let (z2, l2') = get_next ι2 l2 in
             let res = cmp z1 z2 in
             if res = 0 then
               f (n − 1) ι1 l1' ι2 l2'
             else res

cmp_lazy ≜ λ (x1, x2).
           let (ι1, l1) = x1 in
           let (ι2, l2) = x2 in
           if l1 = l2 then 0
           else cmp_list 8 ι1 l1 ι2 l2
```

Figure F.3: Implementation of lazily-sampled integers.

tape label ι and a reference r to a digit in a lazy integer's list, evaluating $get\_next\ \iota\ r$ returns the integer z for that digit and a reference $next$ to the digit after r in the list. There are two alternatives when getting the digit: either (1) the digit for r has already been sampled, so that $!\,r$ will be Some v, where v is a pair of the form $(z, next)$; or (2) the digit for r has not yet been sampled, so that $!\,r$ will be None. In case 1, $get\_next\ \iota\ r$ just returns v. In case 2, get_next will first sample the value z for the digit by calling $sample32\ \iota$, which generates a 32 bit integer by sampling it bit-by-bit with repeated calls to flip ι. It then allocates a new reference $next$ for the next digit in the list, initialized to a value None. Before returning $(z, next)$ it stores this pair in the reference r.

Evaluating $cmp\_list\ n\ \iota_1\ l_1\ \iota_2\ l_2$ compares the two lazy integers (ι_1, l_1) and (ι_2, l_2) by doing a digit-by-digit comparison. It returns -1 if the first integer is smaller, 0 if the integers are equal, and 1 if the first integer is larger. The first argument n tracks the number of remaining digits in the integers. Let us consider the case that $n > 0$ first (the else branch). In that case, cmp_list will call get_next on each integer to get the next digit. Let z_1 and z_2 be these digit values, respectively. These digits are compared using cmp, which returns -1 (if $z_1 < z_2$), 0 (if $z_1 = z_2$), or 1 ($z_1 > z_2$). Because the lazy integers are stored with most-significant digits earlier in the list, if $cmp\ z_1\ z_2$ is non-zero we already know which lazy integer is larger, and the result of $cmp\ z_1\ z_2$ gives the correct ordering of the whole lazy integer. On the other hand if cmp returns 0, then $z_1 = z_2$, in which case we cannot yet tell which lazy integer is larger. Thus, cmp_list recursively calls itself to compare the next digits in the lists, decrementing the n argument to track that there is one fewer digit remaining. In the base case of the recursion, when $n = 0$, that means all digits of the integers have been equal, hence the value of the integers are equal, so we return 0. Because get_next conveniently encapsulates the sampling of unsampled digits, cmp_list looks like a normal traversal of the two linked lists, as if they were eagerly sampled.

Note that if x is a lazy integer, then comparing x with itself using cmp_list unfortunately forces us to sample all of the unsampled digits of x. The routine cmp_lazy is a wrapper to cmp_list that implements a small optimization to avoid this. The function cmp_lazy takes as input a pair of lazy integers (x_1, x_2). Before calling cmp_list, it first checks whether the pointers to the heads of x_1 and x_2's lists are equal; if they are the two integers must be equal, so it returns 0 immediately without calling cmp_list.

We prove that this implementation of lazily-sampled integers is contextually equivalent to code that eagerly samples an entire 256-bit integer by bit-shifting and adding 8 32-bit integers. This contextual equivalence is at an *abstract* existential type τ. Specifically, we define

$$\tau \triangleq \exists\alpha.\, (\mathsf{unit} \to \alpha) \times ((\alpha \times \alpha) \to \mathsf{int})$$

Then we have the following equivalence:

$$\vdash (sample\_lazy\_int, cmp\_lazy) \simeq_{\text{ctx}} (sample256, cmp) : \tau$$

The starting point for the proof is that when $sample256$ samples the 8 32-bit integers needed to assemble the 256-bit integer, we couple these samples with identical samples on the tape ι generated by $sample\_lazy\_int$. Then, the key invariant used in the proof says that if we combine the digits of a lazy int that have already been sampled, plus the remaining digits on the tape ι, the result represents an integer that is equivalent to the corresponding one generated by $sample256$. This holds initially and is preserved by calls to get_next during cmp_list, since it moves digits from the tape to the linked list representing the integer.

$$L \triangleq (\lambda\_.\ \mathsf{true}) \oplus (\lambda\_.\ \mathsf{false})$$
$$I \triangleq \lambda\_.\ (\mathsf{true} \oplus \mathsf{false})$$

$$K \triangleq \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ (\lambda\_.\ M) \oplus (\lambda\_.\ N)$$
$$H \triangleq \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \lambda\_.\ (M \oplus N)$$
$$H_\iota \triangleq \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{let}\ \iota = \mathsf{tape}\ \mathsf{in}\ \lambda\_.\ (M \oplus_\iota N)$$

where

$$M \triangleq \mathsf{if}\ !x = 0\ \mathsf{then}\ x \leftarrow 1; \mathsf{true}\ \mathsf{else}\ \Omega$$
$$N \triangleq \mathsf{if}\ !x = 0\ \mathsf{then}\ x \leftarrow 1; \mathsf{false}\ \mathsf{else}\ \Omega$$
$$\Omega \triangleq (\mathsf{rec}\ f\ x = fx)()$$
$$e_1 \oplus\ e_2 \triangleq \mathsf{if}\ \mathsf{flip}()\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$
$$e_1 \oplus_\iota e_2 \triangleq \mathsf{if}\ \mathsf{flip}(\iota)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

Figure F.4: Sangiorgi and Vignudelli's example.

F.4 Sangiorgi and Vignudelli's "copying" example

Sangiorgi and Vignudelli prove a subtle contextual equivalence mixing probabilistic choice, local references, and state using environmental bisimulations [SV16]. Under call-by-value evaluation, λ-abstraction fails to distribute over probabilistic choice. This is contrary to call-by-name, and can easily be seen by considering the terms I and L in Figure F.4. When evaluated in context $(\lambda f.\ f() = f())[\cdot]$, I returns true (and false) with probability $\frac{1}{2}$, while L returns true with probability 1. The non-linear use of f in the context is characteristic of examples that behave differently under call-by-name and call-by-value. The equivalence of K and H is achieved by prohibiting such a "copying" use by exploiting local state.

The environmental bisimulation technique developed in [SV16] is sufficiently powerful to prove the equivalence as it works directly with the resulting distributions, but, to our knowledge, previous attempts at a proof working abstractly with programs via logical relations were not successful [Biz16, Sec. 1.5].

Intuitively, K and H should be equivalent despite the fact that abstraction does not distribute over probabilistic choice because the closures they return are protected by a counter that only allows them to be run once. On the first call, both have equal probability of returning true or false. On subsequent calls, the counter x ensures that they both diverge.

The key insight that allows us to prove K and H contextually equivalent in Clutch is to establish an asynchronous coupling between the two flip operations. Similarly to the proof of the lazy/eager coin example, we employ an intermediary version H_ι of the program H in which the sampling is delayed until the closure is run. The equivalence of H_ι and H follows from REL-FLIP-ERASE-R and standard symbolic execution rules.

The refinement $\emptyset \vDash_\top H_\iota \precsim K : \mathsf{unit} \to \mathsf{bool}$ is established by allocating an empty tape ι and coupling the (eager) flip() in K with the tape ι. Because allocation of ι is local to H_ι, we obtain exclusive ownership of the tape resource $\iota \hookrightarrow \epsilon$. In particular, other parts of the

program, *i.e.* the context in which H_ι is evaluated in, cannot sample to or consume bits from ι. By REL-COUPLE-TAPE-L, we resolve the flip() to b in K and obtain $\iota \hookrightarrow b$ for H_ι. We then allocate the non-atomic invariant:

$$(\iota \hookrightarrow b * x \mapsto_{\mathsf{s}} 0 * x \mapsto 0) \vee (\iota \hookrightarrow \epsilon * x \mapsto_{\mathsf{s}} 1 * x \mapsto 1)$$

The invariant describes the two possible states of the programs. Either the closures returned by K and H_ι have not been run yet, in which case the presampled bit b is still on tape ι and the counter x is 0 in both programs, or the bit has been consumed, and the counter is 1 in both programs. It is worth noting here that we will rely crucially on a form of local state encapsulation for tapes, which guarantees that once b has been read from ι, the tape remains empty. We only consider the case where $b = \mathsf{true}$; the other case is analogous.

With the invariant in hand, we apply the proof rule for functions to work on the bodies of the two closures. As a first step, we open our invariant, and are left to prove the equivalence in both cases of the disjunction. By virtue of the non-atomic nature of the invariant, we can keep it open for several steps of evaluation, involving flip, pure reductions, and state-manipulating operations, until it is finally reestablished.

In the first case, we read b from ι, yielding $\iota \hookrightarrow \epsilon$. We are left to prove the refinement of two structurally equal programs:

$$\mathsf{if}\ !x = 0\ \mathsf{then}\ x \leftarrow 1; \mathsf{true}\ \mathsf{else}\ \Omega$$

We take the first branch and set x to 1. We have now reproven the invariant, and both programs return true and conclude.

In the second case of the invariant, the closures have been invoked before, and we expect them to both diverge. However, before evaluation reaches Ω in H_ι, another $\mathsf{flip}(\iota)$ has to be resolved. Here we exploit the fact that the ι tape remains empty once we read b, as it is local to H_ι. Logically, this observation manifests in the fact that after the allocation of ι, its ownership has been transferred into the invariant, and is now reclaimed. We can thus use REL-FLIP-TAPE-EMPTY-L to resolve the flip on an empty tape to a new random bit b'. Irrespectively of the value of b', both programs diverge because we know that $x \mapsto_{\mathsf{s}} 1$ and $x \mapsto 1$. A diverging term refines any other term; in particular we appeal to REL-REC to conclude the proof.