RELOC RELOADED: A MECHANIZED RELATIONAL LOGIC FOR FINE-GRAINED CONCURRENCY AND LOGICAL ATOMICITY

DAN FRUMIN, ROBBERT KREBBERS, AND LARS BIRKEDAL

University of Groningen and Radboud University
e-mail address: d.frumin@rug.nl

Radboud University and Delft University of Technology
e-mail address: mail@robbertkrebbers.nl

Aarhus University
e-mail address: birkedal@cs.au.dk

Abstract. We present a new version of ReLoC: a relational separation logic for proving refinements of programs with higher-order state, fine-grained concurrency, polymorphism and recursive types. The core of ReLoC is its *refinement judgment* $e \precsim e' : \tau$, which states that a program e refines a program e' at type τ. ReLoC provides type-directed structural rules and symbolic execution rules in separation-logic style for manipulating the judgment, whereas in prior work on refinements for languages with higher-order state and concurrency, such proofs were carried out by unfolding the judgment into its definition in the model. ReLoC's abstract proof rules make it simpler to carry out refinement proofs, and enable us to generalize the notion of logically atomic specifications to the relational case, which we call *logically atomic relational specifications*.

We build ReLoC on top of the Iris framework for separation logic in Coq, allowing us to leverage features of Iris to prove soundness of ReLoC, and to carry out refinement proofs in ReLoC. We implement tactics for interactive proofs in ReLoC, allowing us to mechanize several case studies in Coq, and thereby demonstrate the practicality of ReLoC.

ReLoC Reloaded extends ReLoC (LICS'18) with various technical improvements, a new Coq mechanization, and support for Iris's prophecy variables. The latter allows us to carry out refinement proofs that involve reasoning about the program's *future*. We also expand ReLoC's notion of logically atomic relational specifications with a new flavor based on the HOCAP pattern by Svendsen *et al*.

1. Introduction

A fundamental question in computer science is *when two programs are equivalent*? The "golden standard" of program equivalence is *contextual equivalence*, stated directly in terms of the operational semantics. Intuitively, expressions e and e' are contextually equivalent if no well-typed client can distinguish them, which formally means that for all well-typed contexts $\mathcal{C}$, the expression $\mathcal{C}[\,e\,]$ has same observable behaviors as $\mathcal{C}[\,e'\,]$. Contextual equivalence can be further decomposed into *contextual refinement*. An expression e *contextually refines* e' if,

Key words and phrases: separation logic, concurrency, program refinement, Iris, Coq.

Preprint submitted to
Logical Methods in Computer Science

© Dan Frumin, Robbert Krebbers, and Lars Birkedal
Creative Commons

for all contexts $\mathcal{C}$, if $\mathcal{C}[\,e\,]$ has some observable behavior, then so does $\mathcal{C}[\,e'\,]$. Expressions e and e' are contextually equivalent iff e contextually refines e' and *vice versa*.

Contextual refinement and contextual equivalence have many applications in computer science. One such application is to specify programs in terms of other programs. For example, one can specify an implementation of a program module (say, a map) that internally uses an efficient but complicated data structure (say, a balanced search tree) by stating that it refines an implementation that internally uses an inefficient but easy to understand data structure (say, an unordered list). In the context of a typed language that supports data abstraction, a specification of a program module in terms of refinement shows that clients of the program module cannot depend on the internal representation of data. This can be seen as an instance of the *representation independence* principle [Rey74, Mit86].

In the context of concurrency, contextual refinement is often used to specify a fine-grained concurrent program module by stating that it contextually refines a coarse-grained version. This is similar to showing that a fine-grained program module is *linearizable* [HW90, FORY10], *i.e.,* each fine-grained operation appears to take place instantaneously. A simple example is the specification of a fine-grained concurrent counter by a coarse-grained one, see Figure 1 for the code. The increment operation of the fine-grained version, $\mathsf{counter}_i$, takes an "optimistic" lock-free approach to incrementing the value using a compare-and-set operation inside a loop. If the value of the counter has been changed (for instance, by some other thread), then the fine-grained counter reattempts the increment from the beginning. The increment operation of the coarse-grained version, $\mathsf{counter}_s$, is performed inside a critical section guarded by a lock. We can state the desired refinement as follows:

$$\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Due to the instrumentation of the coarse-grained version with locks, this refinement expresses that each operation of the fine-grained version takes place instantaneously. We will use the counter as a simple running example throughout the paper.

Another application of contextual refinement and contextual equivalence is to state algebraic properties of program constructs. For example, let us consider the non-deterministic choice operator $e_1 \oplus e_2$, which non-deterministically executes the expression e_1 or e_2. Using contextual equivalence, we can state that this operator is commutative ($e_1 \oplus e_2 \simeq_{ctx} e_2 \oplus e_1$), associative ($e_1 \oplus (e_2 \oplus e_3) \simeq_{ctx} (e_1 \oplus e_2) \oplus e_3$), and that sequential composition distributes over the operator ($(e_1 \oplus e_2); e_3 \simeq_{ctx} (e_1; e_3) \oplus (e_2; e_3)$).

Proving contextual refinement and contextual equivalence. Contextual refinement $e \precsim_{ctx} e' : \tau$ (and contextual equivalence $e \simeq_{ctx} e' : \tau$) are very strong notions because they relate the expressions e and e' in *any* well-typed context $\mathcal{C}$ with a hole of type τ. As a consequence, proving contextual refinement and equivalence directly is challenging—one has to consider arbitrary contexts $\mathcal{C}$, which are only known to be well-typed. Contextual refinement and equivalence are therefore typically proved indirectly using approaches based on bisimulations (*e.g.,* [Gor99, Pit00, KW06, SP07]) or logical relations (*e.g.,* [Pit05, Ahm06, DAB09, BST12, TTA+13]). In the present paper we focus on approaches based on logical relations because they scale well to increasingly rich programming languages with features such as impredicative polymorphism, recursive types, higher-order state, and fine-grained concurrency.

In the approaches based on logical relations, the key is a notion of *logical refinement*, notation $e \precsim e' : \tau$. Logical refinement is defined by structural recursion over the type τ,

Fine-grained grained version (*i.e.*, the implementation):

$$\mathsf{read} \triangleq \lambda c.\ !\,c$$

$$\mathsf{inc}_i \triangleq \mathsf{rec}\ \mathit{inc}\ c = \mathsf{let}\ n = !\,c\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(c, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ \mathit{inc}\ c$$

$$\mathsf{counter}_i \triangleq \mathsf{let}\ c = \mathsf{ref}(0)\ \mathsf{in}\ ((\lambda().\ \mathsf{read}\ c), (\lambda().\ \mathsf{inc}_i\ c))$$

Coarse-grained version (*i.e.*, the specification):

$$\mathsf{inc}_s \triangleq \lambda c\, l.\ \mathsf{acquire}\ l;\ \mathsf{let}\ n = !\,c\ \mathsf{in}\ c \leftarrow (1+n);\ \mathsf{release}\ l;\ n$$

$$\mathsf{counter}_s \triangleq \mathsf{let}\ l = \mathsf{newlock}\ ()\ \mathsf{in}\ \mathsf{let}\ c = \mathsf{ref}(0)\ \mathsf{in}\ ((\lambda().\ \mathsf{read}\ c), (\lambda().\ \mathsf{inc}_s\ c\ l))$$

FIGURE 1. A fine-grained and coarse-grained concurrent counter. (Note that the read operation is shared by both.)

rather than by quantification over all contexts. The soundness theorem of logical relations states that logical refinement implies contextual refinement, *i.e.*, that $e \precsim e' : \tau$ implies $e \precsim_{ctx} e' : \tau$. As a result, proving contextual refinement can be reduced to proving logical refinement, which is generally much easier.

Unfortunately, it is difficult to construct a suitable notion of logical refinement when considering language features like recursive types and higher-order state. In the presence of (general) recursive types, no structurally-recursive definition over the type exists, and in the presence of higher-order references, one needs some notion of *recursively-defined worlds* [BRS+11]. The technique of *step-indexing* [AAV02, Ahm04] has been used to stratify the definitions using recursion over a natural number, called the *step-index*, which corresponds to the number of computation steps performed by the program.

Step-indexing has shown to be very effective by a large body of work on step-indexed logical relations, *e.g.,* [NDR11, HD11, BST12, ÇPG16, RG18]. However, definitions and proofs are intricate because step-indices appear practically everywhere—they even appear in definitions and proofs related to language features (say, products or sums) for which step-indexing is orthogonal. Dreyer *et al.* thus proposed the "logical approach" to logical relations [DAB09, DNRB10] to hide step-indices by abstracting and internalizing them in a modal logic using the *later* modality ($\triangleright$) [AMRV07]. Turon *et al.* [TTA+13, TDB13] further developed the logical approach by using separation logic [ORY01, O'H07, Bro07] to abstract over program states and to handle (fine-grained) concurrency.

More recently, Krebbers *et al.* [KTB17] and Timany [Tim18] defined a logical relation for program refinement based on the work of Turon *et al.* in the state-of-the-art higher-order concurrent separation logic Iris [JKJ+18, JSS+15, JKBD16, KJB+17]. Iris supports impredicative invariants [SB14] and used-defined ghost state, which can be used to streamline the definition of the logical relation, and to carry out proofs of challenging program refinements. The meta theory of Iris is mechanized in the Coq proof assistant, and Iris comes equipped with a *proof mode* [KTB17, KJJ+18]—an extensive set of Coq tactics for separation logic proofs—which allowed them to mechanize all their results in Coq.

Problem statement and key idea. To prove refinements of complicated program modules in a scalable fashion, it is important to decompose refinement proofs into smaller refinements that can be proved in isolation. As a simple example, let us consider the refinement of the fine-grained and coarse-grained concurrent counter from Figure 1:

$$\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

We wish to decompose the proof of this refinement into refinements for the read and increment operations. Naively, one might consider proving contextual refinements for these operations. Unfortunately, such contextual refinements do not hold—they only hold *conditionally* under the assumption that the internal state in both of the implementations is related (including the state of the lock used by the coarse-grained version).

Instead of performing composition at the level of contextual refinement, our key idea is to perform composition at the level of logical refinement. By generalizing logical refinement to become an internal (*i.e.,* first-class) notion in (the Iris) separation logic, we can use the connectives of separation logic to express conditional refinements. Logical refinements for the operations of the concurrent counter are as follows:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, \mathsf{read}\ c_i) \precsim (\lambda().\, \mathsf{read}\ c_s) : \mathsf{unit} \to \mathsf{int}$$
$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, \mathsf{inc}_i\ c_i) \precsim (\lambda().\, \mathsf{inc}_s\ c_s\ lk) : \mathsf{unit} \to \mathsf{int}.$$

We use the *magic wand* ($-\!\!*$, also known as *separating implication*) to make these refinements conditional under the invariant I_{cnt} (expressed using Iris's invariant connective $\boxed{I}$), which is defined as $I_{\mathsf{cnt}} \triangleq \exists n \in \mathbb{N}.\, c_i \mapsto_{\mathsf{i}} n * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})$. The invariant I_{cnt} intuitively expresses that in between function calls, the values of both counters are equal, and the lock (used in the coarse-grained implementation) is in unlocked state. With logical refinements for the individual operations at hand, we can compose them into the logical refinement $\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int})$, which using soundness gives us the desired contextual refinement $\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int})$.

Treating logical refinement as an internal notion in separation logic succinctly distinguishes our work from prior work. In prior work on refinements for rich languages, *e.g.,* the aforementioned work by Turon *et al.* [TTA+13, TDB13], Krebbers *et al.* [KTB17], and Timany [Tim18], logical refinement is an external notion (*i.e.,* a proposition in ordinary mathematics, rather than in separation logic), which means that one cannot concisely state refinements that are conditional on the program state. To state and prove such refinements, one needs to unfold the definition of the logical refinement into the model.

Apart from being able to decompose refinement proofs, internalizing logical refinement gives us a number of other tangible benefits. First, it allows us to develop type-directed structural rules and symbolic execution rules for proving logical refinements. Our symbolic execution rules closely resemble the typical rules for symbolic execution in separation logic, but come in two forms: for the program on the left-hand side and right-hand side of the refinement, making it possible to write concise proofs.

Second, by internalizing logical refinement we can state logical refinements that apply to the situation when the expression on the one side of the refinement contains a program subject to specification, while the expression on the other side is arbitrary. We call such specifications *relational specifications*. Relational specifications take the ability to decompose refinement proofs one step further. As a simple example, let us consider the example from Figure 1, where we proved that a fine-grained concurrent counter refines a coarse-grained version. This refinement is insufficient if we want to prove that a program module that

uses internally the fine-grained counter (say, a ticket lock) refines another module that does not use the coarse-grained counter (say, a spin lock). However, we can instead formulate a *relational specification* for the program module that is proven just once, and derive different logical (and thus by soundness, contextual) refinements from it.

A key challenge in stating relational specifications for operations is to concisely capture that they behave as-if they were atomic, *i.e.,* they appear to take place instantaneously. There has been a long line of work on *logically atomic specifications* to reason about atomicity in the context of Hoare-style logics [JP11, SBP13, dRPDG14, JSS+15, JLP+20]. We show that such logically atomic specifications generalize to the relational case, and call them *logically atomic relational specifications*. Concretely, we introduce relational specification patterns based on da Rocha Pinto *et al.*'s TaDA-style [dRPDG14] and Svendsen *et al.*'s HOCAP-style [SBP13] logically atomic specifications.

The ReLoC logic. Based on the previously described key ideas, we develop a relational separation logic called **ReLoC**. ReLoC is built on top of Iris, allowing the user to leverage the features of Iris such as invariants, (higher-order) ghost state, and prophecy variables. Invariants and ghost state state are powerful mechanisms that support reasoning about concurrent programs through used-defined protocols. Prophecy variables [AL91, JLP+20] allow for speculative reasoning about the future state of concurrent programs. In Iris they come in the form of ghost variables whose value can be referenced before they are specified, thus allowing one to "prophesize" their potential value. We show how these features can be used in ReLoC to prove challenging refinements.

We have implemented ReLoC as a shallow embedding on top of Iris in Coq [KTB17, KJJ+18]. In addition to mechanizing all meta-theoretic results of ReLoC, like its soundness theorem, we have implemented new tactics that support mechanized interactive reasoning about program refinements in ReLoC in a practical and modular way. To our knowledge, ReLoC is the first fully mechanized relational logic enabling reasoning about contextual refinements of programs in a fine-grained concurrent higher-order imperative programming language. The mechanization can be found at [FKB21a].

Contributions and structure of the paper.

- We present a relational logic **ReLoC** for reasoning about contextual refinements of fine-grained concurrent higher-order imperative programs. We present our target programming language (Section 2), an overview of ReLoC (Section 3), and a detailed description of its type-directed structural rules and symbolic execution rules (Section 4).
- We introduce relational specification patterns based on TaDA [dRPDG14] and HOCAP-style [SBP13] logically atomic specifications (Section 5).
- We show how to integrate *prophecy variables* into ReLoC, thereby enabling speculative reasoning in proofs of program refinements (Section 6).
- We describe the logical relations model of ReLoC in Iris (Section 7).
- We describe the mechanization of ReLoC in Coq, and explain how we support mechanized interactive reasoning in ReLoC in a practical and modular way (Section 8).

We discuss further related work in Section 9 and conclude in Section 10.

In addition to the case studies presented in this paper, we have also verified a collection of refinements of concurrent programs from the literature. We give a brief overview of these examples in Section 10.1; and the proofs can be found in the accompanying Coq sources.

$$
\begin{aligned}
\tau \in \mathit{Type} &::= \alpha \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau \mid \mathsf{ref}\ \tau \\
v \in \mathit{Val} &::= i \mid \ell \mid \mathbf{true} \mid \mathbf{false} \mid (v_1, v_2) \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid \mathsf{rec}\ f\ x = e \qquad i \in \mathbb{Z}, \ell \in \mathit{Loc} \\
&\quad \mid \Lambda.e \mid \mathsf{pack}\ v \mid \mathsf{fold}\ v \\
e \in \mathit{Expr} &::= x \mid v \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2) \mid \pi_i(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \qquad i \in \{1,2\} \\
&\quad \mid (\mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x) \to e_1 \mid \mathsf{inr}(x) \to e_2) \mid e_1(e_2) \mid e\langle\rangle \\
&\quad \mid \mathsf{pack}(e) \mid \mathsf{unpack}\ e_1\ \mathsf{in}\ x.\, e_2 \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \\
&\quad \mid \mathsf{ref}(e) \mid\ !\, e \mid e_1 \leftarrow e_2 \mid \mathsf{CAS}(e_1, e_2, e_3) \mid \mathsf{fork}\ \{e\} \mid \ldots
\end{aligned}
$$

FIGURE 2. The syntax of the HeapLang language.

Differences with the conference version of this paper. In the conference version of this paper [FKB18] we described the first version of ReLoC. This paper extends the conference paper in two ways. First, we introduce ReLoC Reloaded (in this paper referred to as just ReLoC), which has several new features, especially in terms its Coq mechanization. Second, we have expanded the presentation of, as well as the material covered by, the paper significantly. Concretely, ReLoC Reloaded has the following new features compared to its original version:

- ReLoC Reloaded's primitive refinement judgment $e \precsim e' : \tau$ is defined for closed expressions (*i.e.,* without free variables), and the version for open expressions (*i.e.,* with free variables) is a derived notion (see Definition 4.4).
- ReLoC Reloaded's underlying programming language is HeapLang—the default language of Iris's Coq mechanization. By having a tight integration of ReLoC with Iris's Coq ecosystem we managed to reuse more Coq code and integrate novel Iris features.
- One such feature that we have integrated into ReLoC Reloaded is the support for prophecy variables (Section 6), which was recently added to Iris [JLP+20].

Compared to the conference paper, we have significantly expanded Sections 2, 4 and 8, and added Sections 6, 7 and 10.2, which are completely new. We have extended Section 5 with HOCAP-style specifications, which we put into action by verifying a refinement between a ticket lock and a spin lock in Section 5.5.

2. The programming language

We consider a typed version of HeapLang, the default language that is shipped with Iris's Coq development [Iri20]. HeapLang is a call-by-value λ-calculus, with higher-order references, fork-based unstructured concurrency, and atomic operations for fine-grained concurrency, equipped with System-F-style types. The syntax is shown in Figure 2. We let α range over a countably infinite set TVar of type variables, which can be bound by the universal type $\forall\alpha.\tau$, existential type $\exists\alpha.\tau$, and recursive type $\mu\alpha.\tau$. We omit the usual Boolean and arithmetic operations such as addition, multiplication, equality, negation.

Most of the operations are standard, so we only discuss some subtleties. Type abstraction $\Lambda.e$, type application $e\langle\rangle$, and the pack/unpack constructs for packing/unpacking existential types do not contain type annotations, following *e.g.,* [Ahm06]. The fold/unfold constructs are used to fold/unfold iso-recursive types. The language includes standard operation on references $\mathsf{ref}(e)$ for allocation, $!\, e$ for dereferencing, and $e_1 \leftarrow e_2$ for assignment. The *atomic*

compare-and-set operation $\mathsf{CAS}(e_1, e_2, e_3)$ checks if the value stored at the location e_1 is equal to e_2, and, if so, sets the value at e_1 to e_3. The $\mathsf{fork}\ \{e\}$ construct creates a new thread, which will execute the expression e.

Syntactic sugar. We use syntactic sugar to define non-recursive functions, let-bindings, and sequential composition. We let $(\lambda x.\, e) \triangleq (\mathsf{rec}\ \_\ x = e)$ and $(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \triangleq ((\lambda x.\, e_2)\ e_1)$ and $(e_1; e_2) \triangleq (\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2)$. The underscore $\_$ denotes an anonymous binder, *i.e.,* a fresh variable that is unused in the body of the binding expression.

Type system. Typing judgments take the form $\Xi \mid \Gamma \vdash e : \tau$, where Γ is a context assigning types to program variables, and Ξ is a context of type variables. The inference rules for the typing judgments are standard; a selection of representative rules is given in Figure 3. The typing rule for the compare-and-set (CAS) operation has a side-condition $\mathrm{EqType}(\tau)$, which ensures that a compare-and-set can only be performed on word-sized data types, *i.e.,* the unit, Boolean, integer, and reference type.

Operational semantics. The operational semantics involves three reduction relations: pure head reduction $\to_{\mathsf{pure}}$, thread-local head reduction $\to_{\mathsf{h}}$, and thread-pool reduction $\to_{\mathsf{tp}}$, see Figure 3 for the rules. Head reduction $\to_{\mathsf{h}}$ is lifted to thread-pool reduction $\to_{\mathsf{tp}}$ using standard *call-by-value evaluation contexts* (in the style of Felleisen and Hieb [FH92]):

$$K \in \mathit{ECtx} ::= [\bullet] \mid e_1(K) \mid K(v_2) \mid e_1 \leftarrow K \mid K \leftarrow v_2 \mid \ldots$$

Thread-pool reduction $\to_{\mathsf{tp}}$ is defined on configurations $\rho = (\vec{e}, \sigma)$ consisting of a state σ (a finite partial map from locations to values) and a thread-pool $\vec{e}$ (a list of expressions corresponding to the threads) by interleaving, *i.e.,* by picking a thread and executing it, thread-locally, for one step. The only special case is $\mathsf{fork}\ \{e\}$, which spawns a thread e, and reduces itself to the unit value $()$.

Contextual refinement. The notion of contextual refinement that we use is standard (see, *e.g.,* [Pit05] or [Har16, Chapters 46 & 47]). It formalizes the situation when the set of observations that can be made about the first program is a subset of observations that can be made about the second program. An observation about a program are made using a *program context* $\mathcal{C}$, which is a program with a hole:

$$\mathcal{C} \in \mathit{Ctx} ::= \square \mid \mathsf{rec}\ f\ x = \mathcal{C} \mid \mathcal{C}(e_2) \mid e_1(\mathcal{C}) \mid \Lambda.\mathcal{C} \mid \mathcal{C}\langle\rangle \mid \ldots$$

Since we are in a typed setting, we consider only *typed contexts*. A program context is well-typed, denoted as $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\Xi' \mid \Gamma' \vdash \tau')$, if for any term $\Xi \mid \Gamma \vdash t : \tau$ we have $\Xi' \mid \Gamma' \vdash \mathcal{C}[\,t\,] : \tau'$. The typing relation on contexts is standard, and can be derived from the typing rules in Figure 3.

We then define contextual refinement as follows. An expression e_1 *contextually refines* an expression e_2 at type τ, denoted as $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$, if no well-typed *program context* $\mathcal{C}$ resulting in a closed program can distinguish the two:

$$\begin{aligned}\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau \triangleq \forall \tau'\, (\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau'))\ v\ \vec{e_f}\ \sigma.\\ (\mathcal{C}[\,e_1\,], \emptyset) \to^*_{\mathsf{tp}} (v :: \vec{e_f}, \sigma) \implies\\ \exists v'\ \vec{e'_f}\ \sigma'.\, (\mathcal{C}[\,e_2\,], \emptyset) \to^*_{\mathsf{tp}} (v' :: \vec{e'_f}, \sigma').\end{aligned}$$

Contextual equivalence $\Xi \mid \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$ is defined as the symmetric closure of contextual refinement, *i.e.,* $(\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau) \wedge (\Xi \mid \Gamma \vdash e_2 \precsim_{ctx} e_1 : \tau)$.

Selected typing rules:

VAR-TYPED
$$\frac{\Gamma(x) = \tau}{\Xi \mid \Gamma \vdash x : \tau}$$

PROJ-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \pi_i(e) : \tau_i}$$

REC-TYPED
$$\frac{\Xi \mid x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash e : \tau_2}{\Xi \mid \Gamma \vdash \mathsf{rec}\ f\ x = e : \tau_1 \to \tau_2}$$

TLAM-TYPED
$$\frac{\Xi, \alpha \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda.e : \forall \alpha.\tau}$$

TAPP-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \forall \alpha.\tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \vdash e\langle\rangle : \tau[\tau'/\alpha]}$$

TPACK-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \tau[\tau'/\alpha]}{\Xi \mid \Gamma \vdash \mathsf{pack}\ e : \exists \alpha.\tau}$$

TUNPACK-TYPED
$$\frac{\Xi \mid \Gamma \vdash e_1 : \exists \alpha.\tau_1 \qquad \alpha, \Xi \mid x : \tau_1, \Gamma \vdash e_2 : \tau_2 \qquad \alpha \text{ is not free in } \Gamma \text{ or } \tau_2}{\Xi \mid \Gamma \vdash \mathsf{unpack}\ e_1\ \mathsf{in}\ x.\, e_2 : \tau_2}$$

FOLD-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \tau[\mu\tau/\alpha]}{\Xi \mid \Gamma \vdash \mathsf{fold}\ e : \mu\alpha.\tau}$$

UNFOLD-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \mu\alpha.\tau}{\Xi \mid \Gamma \vdash \mathsf{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

ALLOC-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{ref}(e) : \mathsf{ref}\ \tau}$$

LOAD-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \mathsf{ref}\ \tau}{\Xi \mid \Gamma \vdash\, !\, e : \tau}$$

STORE-TYPED
$$\frac{\Xi \mid \Gamma \vdash e_1 : \mathsf{ref}\ \tau \qquad \Xi \mid \Gamma \vdash e_2 : \tau}{\Xi \mid \Gamma \vdash e_1 \leftarrow e_2 : \mathsf{unit}}$$

CAS-TYPED
$$\frac{\Xi \mid \Gamma \vdash e_1 : \mathsf{ref}\ \tau \quad \Xi \mid \Gamma \vdash e_2 : \tau \quad \Xi \mid \Gamma \vdash e_3 : \tau \quad \mathrm{EqType}(\tau)}{\Xi \mid \Gamma \vdash \mathsf{CAS}(e_1, e_2, e_3) : \mathsf{bool}}$$

FORK-TYPED
$$\frac{\Xi \mid \Gamma \vdash e : \mathsf{unit}}{\Xi \mid \Gamma \vdash \mathsf{fork}\ \{e\} : \mathsf{unit}}$$

Selected rules of pure reduction $e_1 \to_{\mathsf{pure}} e_2$ and thread-local call-by-value head-reduction $(e, \sigma) \to_{\mathsf{h}} (e', \sigma')$:

PROJ
$$\pi_i\ (v_1, v_2) \to_{\mathsf{pure}} v_i$$

BETA
$$(\mathsf{rec}\ f\ x = e)\ v \to_{\mathsf{pure}} e[v/x][\mathsf{rec}\ f\ x = e/f]$$

TBETA
$$(\Lambda.e)\langle\rangle \to_{\mathsf{pure}} e$$

UNPACK
$$\mathsf{unpack}\ (\mathsf{pack}\ v)\ \mathsf{in}\ x.\, e \to_{\mathsf{pure}} e[v/x]$$

UNFOLD
$$\mathsf{unfold}\ (\mathsf{fold}\ v) \to_{\mathsf{pure}} v$$

PURE
$$\frac{e_1 \to_{\mathsf{pure}} e_2}{(e_1, \sigma) \to_{\mathsf{h}} (e_2, \sigma)}$$

ALLOC
$$\frac{\sigma(\ell) = \bot}{(\mathsf{ref}(v), \sigma) \to_{\mathsf{h}} (\ell, \sigma[\ell \leftarrow v])}$$

DEREF
$$\frac{\sigma(\ell) = v}{(!\,\ell, \sigma) \to_{\mathsf{h}} (v, \sigma)}$$

STORE
$$\frac{\sigma(\ell) = v}{(\ell \leftarrow v', \sigma) \to_{\mathsf{h}} ((), \sigma[\ell \leftarrow v'])}$$

CAS-FAIL
$$\frac{\sigma(\ell) \neq v_1}{(\mathsf{CAS}(\ell, v_1, v_2), \sigma) \to_{\mathsf{h}} (\mathbf{false}, \sigma)}$$

CAS-SUC
$$\frac{\sigma(\ell) = v_1}{(\mathsf{CAS}(\ell, v_1, v_2), \sigma) \to_{\mathsf{h}} (\mathbf{true}, \sigma[\ell \leftarrow v_2])}$$

Thread-pool reduction $(\vec{e}, \sigma) \to_{\mathsf{tp}} (\vec{e'}, \sigma')$:

$$\frac{(e, \sigma) \to_{\mathsf{h}} (e', \sigma')}{(\vec{e_1}\ K[\, e\,]\ \vec{e_2}, \sigma) \to_{\mathsf{tp}} (\vec{e_1}\ K[\, e'\,]\ \vec{e_2}, \sigma')} \qquad (\vec{e_1}\ K[\,\mathsf{fork}\ \{e\}\,]\ \vec{e_2}, \sigma) \to_{\mathsf{tp}} (\vec{e_1}\ K[\,()\,]\ \vec{e_2}\ e, \sigma)$$

FIGURE 3. The type system and operational semantics of HeapLang.

Note that contextual refinement only takes termination into account, and does not require the resulting values v and v' to be equal. Demanding the equality on the resulting values would make contextual refinement too strong. For example, the terms $(\lambda x.\, x + 1)$ and $(\lambda x.\, 1 + x)$ of function type would not be deemed contextually equivalent, because they terminate to syntactically different values in the empty program context.

There are, however, equivalent formulations of contextual refinement which equate the resulting values v and v'. In order to do that, it is necessary to restrict the typed context $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau')$ to those for which τ' is a directly observable type, like Booleans or integers. For example, we could have used the following equivalent[1] definition (a variation of **true**-adequate contextual equivalence from [Pit05, Exercise 7.5.10]):

$$\forall(\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \mathsf{bool}))\ \vec{e_f}\ \sigma.$$
$$(\mathcal{C}[\,e_1\,], \emptyset) \to^*_{\mathsf{tp}} (\mathbf{true} :: \vec{e_f}, \sigma) \implies \exists \vec{e'_f}\ \sigma'.\, (\mathcal{C}[\,e_2\,], \emptyset) \to^*_{\mathsf{tp}} (\mathbf{true} :: \vec{e'_f}, \sigma').$$

3. A tour of ReLoC

This section gives a tour of ReLoC by demonstrating its key logical connectives and proof rules. We first describe ReLoC's grammar, soundness statement, and rule format (Section 3.1). After that, we put ReLoC to action by proving contextual refinements of two program modules. The first is a bit module, which demonstrates ReLoC's type-directed structural rules and symbolic execution rules for reasoning about pure programs (Section 3.3). The second is the concurrent counter module from Section 1, which involves reasoning about internal state and concurrency. Specifically we demonstrate how ReLoC is used to reason about stateful programs using symbolic execution (Section 3.4.1), concurrency using invariants (Section 3.4.2), and recursive functions and loops using Löb induction (Section 3.4.3).

3.1. Grammar and soundness.

ReLoC is based on higher-order intuitionistic separation logic, and the grammar of its propositions is:

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid P \wedge Q \mid P \vee Q \mid P \implies Q \\ & \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_{\mathsf{i}} v \mid \ell \mapsto_{\mathsf{s}} v \mid (\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau) \\ & \mid [\![\tau]\!]_\Delta(v_1, v_2) \mid \boxed{P}^{\mathcal{N}} \mid \triangleright P \mid \Box P \mid {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \mid \ldots \end{aligned}$$

ReLoC is an extension of Iris and therefore includes all connectives of Iris, in particular, the *later* modality $\triangleright$, *persistence* modality $\Box$, *update* modality ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$, and *invariant assertion* $\boxed{P}^{\mathcal{N}}$. We introduce these connectives in passing throughout this section. Some of these connectives are annotated by *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ and *invariant names* $\mathcal{N} \in \mathit{InvName}$, which are needed for bookkeeping related to Iris's invariant mechanism. Until we introduce invariants in Section 3.4.2, we will omit these annotations. Similarly, we will ignore the later modality $\triangleright$ until we explain it in Section 3.4.3.

An essential difference to vanilla Iris is that ReLoC has internal (or first-class) *refinement judgments* $\Delta \models e_1 \precsim e_2 : \tau$, which should be read as "the expression e_1 refines the expression e_2 at type τ". Just like contextual refinement, the refinement judgment in ReLoC is indexed by a type τ. The judgment contains an environment Δ which assigns *interpretations* to type

[1] Proving that this definition is equivalent to the one presented earlier is not a very complicated, albeit laborious, task. See the Coq mechanization for the formal proof.

variables. These interpretations are given by an Iris relation of type $\mathit{Val} \times \mathit{Val} \to \mathit{iProp}$. One such kind of relation, the *value interpretation* relation $[\![\tau]\!]_\Delta(-,-) : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ (for each syntactic type τ of $\mathsf{HeapLang}$) will be discussed in Section 4. We elide the contexts Δ in refinement judgments whenever they are empty.

The intuitive meaning of $\Delta \models e_1 \precsim e_2 : \tau$ is that e_1 is safe, and all of its behaviors can be simulated by e_2. It is a simulation in the sense that any execution step of e_1 can be matched by a (possibly empty) sequence of execution steps of e_2. Borrowing the terminology from languages with non-determinism, we think of e_1 as being *demonic* and e_2 as being *angelic*. That is, the non-deterministic choices of e_1 (*e.g.,* scheduling of forked-off threads) are selected by an external demon; whereas for the non-deterministic choices of e_2, an angle blesses the person proving the refinement with an ability to select a choice themselves.

Since we often use refinement judgments to specify programs, we refer to the left-hand side e_1 as the *implementation*, and to right-hand side e_2 as the *specification*. The intuitive meaning is formally reflected by the soundness theorem w.r.t. contextual refinement.

Theorem 3.1 (Soundness)**.** *If the refinement judgment $\emptyset \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC, then $\emptyset \mid \emptyset \vdash e_1 \precsim_{ctx} e_2 : \tau$.*

In this section we only consider closed programs e_1 and e_2; we will see how ReLoC (and its soundness theorem) generalize to open terms in Section 4.4.

Like ordinary separation logic, ReLoC has *heap assertions*. Since ReLoC is relational, these come in two forms: $\ell \mapsto_{\mathsf{i}} v$ and $\ell \mapsto_{\mathsf{s}} v$, which signify ownership of a location ℓ with value v on the implementation and specification side, respectively.

Contrary to earlier work on logical refinements in Iris, *e.g.,* [KTB17, Tim18], refinement judgments $\Delta \models e_1 \precsim e_2 : \tau$ in ReLoC are first-class propositions. As such, we can combine them in arbitrary ways with the other logical connectives, and state conditional refinements. For example, the proposition

$$(\ell_1 \mapsto_{\mathsf{i}} v_1 * \ell_2 \mapsto_{\mathsf{s}} v_2 * \Delta \models e_1' \precsim e_2' : \sigma) \mathrel{-\!\!*} \Delta \models e_1 \precsim e_2 : \tau, \tag{3.1}$$

states that the e_1 refines e_2, under the assumption of another refinement and that certain locations have specified values in the heap. Having conditional refinements is crucial for modularity, as it allows us to formulate and prove refinements of individual methods of a data structure under the assumptions provided by the internal invariant of the data structure. The fact that refinement judgments are first class also plays an important role in the presentation of ReLoC's proof rules.

3.2. **Derivability and inference rules.** As standard in logic, Iris/ReLoC has a derivability relation $P \vdash Q$. We say that Q is derivable if $\mathsf{True} \vdash Q$. In many situations, we use magic wand $\mathrel{-\!\!*}$ instead of the derivability relation $\vdash$, because we have the standard deduction property:

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

Most of the inference rules we present can be internalized as ReLoC propositions by a magic wand or a derivability relation between the separating conjunction of the antecedents and

the consequent. We thus use the following notations:

$$\frac{P_1 \qquad \cdots \qquad P_n}{Q} \quad \text{is notation for} \quad (P_1 * \cdots * P_n) \mathrel{-\!\!*} Q,$$

$$\frac{P}{\overline{\overline{Q}}} \quad \text{is notation for} \quad (P \mathrel{-\!\!*} Q) \wedge (Q \mathrel{-\!\!*} P).$$

For instance, the conditional refinement in Formula (3.1) is presented as the following inference rule:

$$\frac{\ell_1 \mapsto_{\mathsf{i}} v_1 \qquad \ell_2 \mapsto_{\mathsf{s}} v_2 \qquad \Delta \models e_1' \precsim e_2' : \sigma}{\Delta \models e_1 \precsim e_2 : \tau}$$

In rules like this, it is useful to think of premises $\ell_1 \mapsto_{\mathsf{i}} v_1$ and $\ell_2 \mapsto_{\mathsf{s}} v_2$ as side conditions, and of the premise $\Delta \models e_1' \precsim e_2' : \sigma$ as the new goal that you get when you apply the rule. This *backwards-style* reasoning integrates well in the Coq proof assistant; we discuss it more in detail in Section 8.

We use the derivability relation $\vdash$ explicitly to state rules that cannot be internalized, *e.g.,* $\dfrac{\vdash P}{\vdash Q}$ states that if P is derivable, then Q is derivable. This is weaker than $\dfrac{P}{Q}$, which denotes that Q can be derived from P, *i.e.,* $P \vdash Q$.

3.3. **Example: Contextual equivalance of a bit module.** We demonstrate the basic usage of ReLoC by using its *type-directed structural* and *symbolic execution* rules to prove contextual equivalence of two implementations of a simple program module (representation independence). The module we consider represents a single bit data structure—it contains an initial value for the bit, an operation for flipping the bit, and an operation for converting the values of the abstract type to Booleans. We use an existential type (*i.e.,* abstract type) to hide the representation type and thus the type of the module:

$$\mathsf{TBit} \triangleq \exists \alpha.\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool}).$$

Perhaps the simplest implementation of the bit interface is the one that uses Booleans for the internal state:

$$\mathsf{bitbool} : \mathsf{TBit} \triangleq \mathsf{pack}(\mathbf{true}, (\lambda b.\, \neg b), (\lambda b.\, b)).$$

The second implementation models a bit by a number from the set $\{0, 1\}$:

$$\mathsf{flipnat} : \mathsf{int} \to \mathsf{int} \triangleq \lambda n.\, \mathsf{if}\ (n = 0)\ \mathsf{then}\ 1\ \mathsf{else}\ 0$$

$$\mathsf{bitnat} : \mathsf{TBit} \triangleq \mathsf{pack}(1, \mathsf{flipnat}, (\lambda n.\, n = 1)).$$

Before we explain how the contextual equivalence of these two implementation is formally proved in ReLoC, let us informally discuss why these implementations are equivalent. Note that the underlying types (int and bool) are not isomorphic. This, however, is not going to be a problem, because the underlying types are hidden/existentially abstracted in the module signature. As a consequence of that, a (well-typed) client has to be polymorphic in the type α, and can thus only create and modify values of α through the functions provided by the module. A client that uses the bitnat module can only construct integers 0 and 1 (using the initial value and applying the flip function a number of times). Thus, requiring an isomorphism between the underlying types is too strict—for example, we do not care

Value interpretation rules:

VAL-VAR
$$\frac{\Delta(\alpha)(v_1, v_2)}{[\![\alpha]\!]_\Delta(v_1, v_2)}$$

VAL-UNIT
$$\frac{v_1 = v_2 = ()}{[\![\mathsf{unit}]\!]_\Delta(v_1, v_2)}$$

VAL-BOOL
$$\frac{\exists b \in \mathbb{B}.\, v_1 = v_2 = b}{[\![\mathsf{bool}]\!]_\Delta(v_1, v_2)}$$

VAL-INT
$$\frac{\exists n \in \mathbb{Z}.\, v_1 = v_2 = n}{[\![\mathsf{int}]\!]_\Delta(v_1, v_2)}$$

Type-directed structural rules:

REL-RETURN
$$\frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \models v_1 \precsim v_2 : \tau}$$

REL-PAIR
$$\frac{\Delta \models e_1 \precsim e_2 : \tau \qquad \Delta \models e_1' \precsim e_2' : \sigma}{\Delta \models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \sigma}$$

REL-PACK
$$\frac{\forall v_1, v_2.\, \mathsf{persistent}(R(v_1, v_2)) \qquad [\alpha := R], \Delta \models e_1 \precsim e_2 : \tau}{\Delta \models \mathsf{pack}\ e_1 \precsim \mathsf{pack}\ e_2 : \exists \alpha.\tau}$$

REL-REC
$$\frac{\square\left(\forall v_1, v_2.\, [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} (\Delta \models (\mathsf{rec}\ f_1\ x_1 = e_1)\ v_1 \precsim (\mathsf{rec}\ f_2\ x_2 = e_2)\ v_2 : \sigma)\right)}{\Delta \models (\mathsf{rec}\ f_1\ x_1 = e_1) \precsim (\mathsf{rec}\ f_2\ x_2 = e_2) : \tau \to \sigma}$$

Symbolic execution rules:

REL-PURE-L
$$\frac{e_1 \to_{\mathsf{pure}} e_1' \qquad \triangleright(\Delta \models K[\,e_1'\,] \precsim e_2 : \tau)}{\Delta \models K[\,e_1\,] \precsim e_2 : \tau}$$

REL-PURE-R
$$\frac{e_2 \to_{\mathsf{pure}} e_2' \qquad \Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2'\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

REL-ALLOC-L'
$$\frac{\forall \ell.\, \ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models K[\,\ell\,] \precsim e_2 : \tau}{\Delta \models K[\,\mathsf{ref}(v)\,] \precsim e_2 : \tau}$$

REL-ALLOC-R
$$\frac{\forall \ell.\, \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,\ell\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{ref}(v)\,] : \tau}$$

REL-LOAD-L-INV
$$\frac{\boxed{P}^{\mathcal{N}} \qquad \left(\triangleright P * \mathsf{closeInv}_{\mathcal{N}}(P)\right) \mathrel{-\!\!*} \exists v.\, \ell \mapsto_{\mathsf{i}} v * \triangleright\left(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\top \setminus \mathcal{N}} K[\,v\,] \precsim e_2 : \tau\right)}{\Delta \models K[\,!\,\ell\,] \precsim e_2 : \tau}$$

REL-LOAD-R
$$\frac{\ell \mapsto_{\mathsf{s}} v \qquad \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,!\,\ell\,] : \tau}$$

REL-STORE-R
$$\frac{\ell \mapsto_{\mathsf{s}} - \qquad \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\ell \leftarrow v\,] : \tau}$$

REL-CAS-L-INV
$$\frac{\boxed{P}^{\mathcal{N}} \qquad \begin{array}{l}\triangleright P * \mathsf{closeInv}_{\mathcal{N}}(P) \mathrel{-\!\!*} \\ \exists v.\, \ell \mapsto_{\mathsf{i}} v * \triangleright \begin{pmatrix}(v = v_1 * \ell \mapsto_{\mathsf{i}} v_2 \mathrel{-\!\!*} \Delta \models_{\top \setminus \mathcal{N}} K[\,\mathbf{true}\,] \precsim e_2 : \tau) \wedge \\ (v \neq v_1 * \ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\top \setminus \mathcal{N}} K[\,\mathbf{false}\,] \precsim e_2 : \tau)\end{pmatrix}\end{array}}{\Delta \models K[\,\mathsf{CAS}(\ell, v_1, v_2)\,] \precsim e_2 : \tau}$$

Invariants rules:

REL-INV-ALLOC
$$\frac{\triangleright P \qquad \boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \Delta \models e_1 \precsim e_2 : \tau}{\Delta \models e_1 \precsim e_2 : \tau}$$

REL-INV-RESTORE
$$\frac{\mathsf{closeInv}_{\mathcal{N}}(P) \qquad \triangleright P \qquad \Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau}{\Delta \models_{\mathcal{E} \setminus \mathcal{N}} e_1 \precsim e_2 : \tau}$$

FIGURE 4. Selected rules of ReLoC.

what Boolean value an integer 7 might correspond to, because the number 7 can never be constructed using the functions provided by bitnat.

This intuitive reasoning signals the key idea behind the *representation independence* principle [Mit86], which states that in order to prove that two modules are equivalent, it suffices to pick a *relation* between the underlying types and demonstrate that all the methods preserve this relation. For this example, a sensible candidate for such a relation is $\{(\textbf{true}, 1), (\textbf{false}, 0)\}$. Note that our relation does not include any integers other than 0 or 1, because as we previously explained, a well-typed client of bitnat cannot construct other integers. With the relation at hand, the informal proof is as follows. The initial values offered by the modules are related. The flip function preserves this relation. The function that converts "bits" to Booleans sends related values to the same Boolean.

We will now demonstrate how to carry out this argument formally in ReLoC. Specifically, we prove the following refinement using the rules in Figure 4:

$$\mathsf{bitbool} \precsim \mathsf{bitnat} : \mathsf{TBit}$$

The other direction can be proved in a similar way, which using soundness (Theorem 3.1), gives us the contextual equivalence $\mathsf{bitbool} \simeq_{ctx} \mathsf{bitnat} : \mathsf{TBit}$.

From a high-level point of view, the proof of this example involves applying ReLoC's type-directed structural rules following the structure of TBit. At the leaves of the proof, we continue with ReLoC's symbolic execution rules to perform computation steps.

Since TBit is an existential type, and both bitbool and bitnat are pack's, we start off by applying the type-directed structural rule REL-PACK. For that we need to pick a relation R, which will be the interpretation for the type variable α, and should link together the underlying representations of bits in bitbool and bitnat. We define the relation R as follows:

$$R(b, n) \triangleq (b = \textbf{true} \land n = 1) \lor (b = \textbf{false} \land n = 0).$$

Starting with the initial goal $\mathsf{bitbool} \precsim \mathsf{bitnat} : \mathsf{TBit}$, we apply REL-PACK. As a side-condition, we have to prove that R is *persistent* for any v_1, v_2, written as $\mathsf{persistent}(R(v_1, v_2))$, intuitively meaning that the proposition $R(v_1, v_2)$ does not assert ownership of any resources. We discuss persistent propositions in more detail in Sections 3.4.2 and 4.2, and for now we just note that the relation R is indeed persistent. After application of the REL-PACK rule the goal becomes:

$$[\alpha := R] \models (\textbf{true}, (\lambda b.\, \neg b), (\lambda b.\, b)) \precsim (1, \mathsf{flipnat}, (\lambda n.\, n = 1)) : \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool}).$$

By repeatedly applying the type-directed structural rule REL-PAIR we get three new goals:
(1) $[\alpha := R] \models \textbf{true} \precsim 1 : \alpha$;
(2) $[\alpha := R] \models (\lambda b.\, \neg b) \precsim \mathsf{flipnat} : \alpha \to \alpha$;
(3) $[\alpha := R] \models (\lambda b.\, b) \precsim (\lambda n.\, n = 1) : \alpha \to \mathsf{bool}$.

For the first goal, we can use the rules REL-RETURN and VAL-VAR, leaving us with the obligation $R(\textbf{true}, 1)$, which holds by the definition of R.

For the second and the third goal we need to prove refinements of two closures, for which we use the type-directed structural rule REL-REC. Let us look at the third goal in detail. After the application of REL-REC we have to show:

$$\square \left(\forall v_1, v_2.\, [\![\alpha]\!]_{[\alpha:=R]}(v_1, v_2) \mathrel{-\!\!*} [\alpha := R] \models (\lambda b.\, b)\ v_1 \precsim (\lambda n.\, n = 1)\ v_2 : \mathsf{bool}\right).$$

The goal is wrapped in Iris's *persistence modality* $\square$, which turns any proposition into a persistent one. Once again, we postpone the details about the persistence modality until

Sections 3.4.2 and 4.2, and only remark that here we are allowed to prove the goal without the $\Box$ modality. Using this information, and the rule VAL-VAR we reduce our goal to show:

$$R(v_1, v_2) \mathrel{-\!\!*} [\alpha := R] \models (\lambda b.\, b)\ v_1 \precsim (\lambda n.\, n = 1)\ v_2 : \mathsf{bool},$$

for arbitrary v_1, v_2. We then unfold the definition of R and observe that we need to distinguish two cases: (1) $v_1 = \mathbf{true}$ and $v_2 = 1$; (2) $v_1 = \mathbf{false}$ and $v_2 = 0$. Suppose we are in the first case (the second case is similar). We have to show:

$$[\alpha := R] \models (\lambda b.\, b)\ \mathbf{true} \precsim (\lambda n.\, n = 1)\ 1 : \mathsf{bool}.$$

At this point we apply ReLoC's *symbolic execution* rules: we symbolically reduce both the left-hand and the right-hand side of the refinement. For this we use the rules REL-PURE-L and REL-PURE-R (the later modalities ($\triangleright$) in these rules can be ignored for now, they will be explained in Section 3.4.3). These rules perform *pure reductions*, *i.e.,* reductions that do not depend on the heaps. In our case we have a β-reduction on the left-hand side, and a β-reduction and an evaluation of the binary operation (equality testing) on the right-hand side:

$$(\lambda b.\, b)\ \mathbf{true} \to_{\mathsf{pure}} \mathbf{true} \qquad (\lambda n.\, n = 1)\ 1 \to_{\mathsf{pure}} (1 = 1) \to_{\mathsf{pure}} \mathbf{true}.$$

After the repeated application of the said rules we arrive at a goal

$$[\alpha := R] \models \mathbf{true} \precsim \mathbf{true} : \mathsf{bool},$$

which we discharge by REL-RETURN and VAL-BOOL. This completes the proof of the refinement.

3.4. **Example: Contextual refinement of a concurrent counter.** The previous example showcased how ReLoC can be used to show contextual refinement and equivalence of pure program modules. In this subsection we prove contextual refinement of the fine-grained concurrent counter in Figure 1 from Section 1 by showing that it refines the coarse-grained counter. Specifically, we prove the following refinement:

$$\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Using soundness (Theorem 3.1), this contextual refinement can be reduced to proving the refinement judgment $\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int})$ in ReLoC.

The previous example demonstrated the basic usage of symbolic execution rules of ReLoC. Those symbolic execution rules were confined to the pure fragment of the programming language. In this example we show how to use ReLoC's symbolic execution rules for stateful computations and concurrency primitives. In addition to the type-directed structural rules and symbolic execution rules, the proof will require the usage of *invariants* for linking together the values of the two counters. We will use selected ReLoC rules from Figure 4. To symbolically execute the operations on locks that appear in $\mathsf{counter}_s$, we will also make use of the *relational specification* for locks in Figure 5. The lock specification is stated in terms of an abstract predicate $\mathsf{isLock}_\mathsf{s}(lk, \mathbf{false})$ (resp., $\mathsf{isLock}_\mathsf{s}(lk, \mathbf{true})$) stating that lk is a lock which is unlocked (resp., locked). The relational specification for locks can then be seen as consisting of symbolic execution rules that manipulate that abstract predicate.[2] We will see in Section 5.1.1 that these specifications can be proven for a simple spin lock.

[2]Because this specification is for the "angelic" right-hand side, it does not express mutual exclusion as it is common for separation logic specifications. We explain this by contrasting the specification with the one for the left-hand side in Section 5.1.2.

$$\textsc{newlock-r}\ \frac{\forall lk.\ \mathsf{isLock_s}(lk, \mathbf{false}) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,lk\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{newlock}\ ()\,] : \tau}$$

$$\textsc{acquire-r}\ \frac{\mathsf{isLock_s}(lk, \mathbf{false}) \qquad \mathsf{isLock_s}(lk, \mathbf{true}) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{acquire}\ lk\,] : \tau}$$

$$\textsc{release-r}\ \frac{\mathsf{isLock_s}(lk, b) \qquad \mathsf{isLock_s}(lk, \mathbf{false}) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{release}\ lk\,] : \tau}$$

FIGURE 5. Right-hand side relational specification for locks.

3.4.1. *Symbolic execution.* Recall that performing symbolic execution means reducing the left-hand or right-hand side of the refinement according to the computational rules. We have already seen the usage of REL-PURE-L, which allows us to perform pure computations. For this example we also use stateful symbolic execution rules in Figure 4. To start with the refinement proof, we apply the stateful symbolic execution rule REL-ALLOC-L' to the left-hand side to obtain:

$$c_i \mapsto_{\mathsf{i}} 0 \mathrel{-\!\!*} ((\lambda().\,\mathsf{read}\ c_i), (\lambda().\,\mathsf{inc}_i\ c_i)) \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Note that after the application of the rule we gain access to the resource $c_i \mapsto_{\mathsf{i}} 0$ representing the value of the counter on the left-hand side. Subsequently, using the symbolic execution rules REL-PURE-R, REL-ALLOC-R and NEWLOCK-R on the right-hand side the goal becomes:

$$c_i \mapsto_{\mathsf{i}} 0 * c_s \mapsto_{\mathsf{s}} 0 * \mathsf{isLock_s}(lk, \mathbf{false}) \mathrel{-\!\!*}$$
$$((\lambda().\,\mathsf{read}\ c_i), (\lambda().\,\mathsf{inc}_i\ c_i)) \precsim (\lambda().\,\mathsf{read}\ c_s), (\lambda().\,\mathsf{inc}_s\ c_s\ lk)) : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

In addition to gaining the resource $c_s \mapsto_{\mathsf{s}} 0$, representing the value of the right-hand side counter, we get access to the abstract predicate $\mathsf{isLock_s}(lk, \mathbf{false})$, which keeps track of the state of the lock lk on the right-hand side.

ReLoC's symbolic execution rules are inspired by the "backwards"-style Hoare rules of [IO01] and the weakest-precondition rules in Iris [KJB+17, JKJ+18].

3.4.2. *Invariants and persistent propositions.* At this point we wish to prove a refinement of two closures. By the rule REL-PAIR it would suffice to prove that both closures refine each other. However, if we were to apply REL-PAIR, we would be forced to split our resources in two: the resources needed for the refinement proof of the read function, and the resources needed for the refinement proof of the increment function. But both of those operations require access to the counter locations $c_i \mapsto_{\mathsf{i}} -$ and $c_s \mapsto_{\mathsf{s}} -$. To circumvent this issue we put said resources in a global *invariant* $\boxed{P}$, which allows P to be shared between different parts of the program (and between different threads). In our running example, we establish the invariant $\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}}$ (using REL-INV-ALLOC), where:

$$I_{\mathsf{cnt}} \triangleq \exists n \in \mathbb{N}.\ c_i \mapsto_{\mathsf{i}} n * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock_s}(lk, \mathbf{false}).$$

The invariant $\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}}$ not only allows us to share access to c_i and c_s, but also ensures that the values of the respective counters match up. For our invariant we pick any fresh invariant name $\mathcal{N} \in \mathit{InvName}$ (more on the invariant names below).

Invariants $\boxed{P}$ are *persistent*: once established, they will remain valid for the rest of the verification. This differentiates them from *ephemeral* propositions like $\ell \mapsto_{\mathsf{i}} v$ and $\ell \mapsto_{\mathsf{s}} v$, which could be invalidated in the future by actions of the program or proof.

The notion of being persistent is expressed in ReLoC (and Iris) by means of the *persistence* modality $\Box$. The purpose of $\Box\, P$ is to say that P holds without asserting any ephemeral propositions. The most important rules for the $\Box$ modality are $\Box\, P = \Box\, P * \Box\, P$ and $\Box\, P \mathrel{-\!\!*} P$, which allow to freely duplicate $\Box\, P$ and finally get P out. We say that P is *persistent*, written as $\mathsf{persistent}(P)$, if $P \vdash \Box\, P$; otherwise, we say that P is *ephemeral*. To prove $\Box\, P$, one can only use persistent resources like $\boxed{P}$, and not ephemeral resources like $\ell \mapsto_{\mathsf{i}} v$. We refer to stripping off the persistence modality in the context of persistent hypotheses as *introducing the* $\Box$ *modality*. We make that precise and give rules for the $\Box$ modality in Section 4.3.

Once the invariant $\boxed{I_{\mathsf{cnt}}}$ for our running example has been established, we can duplicate it, and apply REL-PAIR to obtain two goals:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, \mathsf{read}\ c_i) \precsim (\lambda().\, \mathsf{read}\ c_s) : \mathsf{unit} \to \mathsf{int}$$

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, \mathsf{inc}_i\ c_i) \precsim (\lambda().\, \mathsf{inc}_s\ c_s\ \mathit{lk}) : \mathsf{unit} \to \mathsf{int}.$$

We first describe how to prove the refinement of read. As $\lambda x.\, e$ is syntactic sugar for $\mathsf{rec}\ \_\ x = e$, we can apply REL-REC at the function type $\mathsf{unit} \to \mathsf{int}$ and obtain the new goal:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} \Box \big(\forall v\, v'.\, [\![\mathsf{unit}]\!]_\Delta(v, v') \mathrel{-\!\!*} (\lambda().\, !\, c_i)\ v \precsim (\lambda().\, !\, c_s)\ v' : \mathsf{int}\big).$$

By VAL-UNIT, we obtain that $[\![\mathsf{unit}]\!]_\Delta(v, v')$ implies $v = v' = ()$. Moreover, since $\boxed{I_{\mathsf{cnt}}}$ is our only hypothesis, and it is persistent, we can strip off the $\Box$ modality, arriving at the following goal:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, !\, c_i)\ () \precsim (\lambda().\, !\, c_s)\ () : \mathsf{int}.$$

Accessing invariants. The fact that invariants are persistent (and thus can be duplicated, *i.e.*, $\boxed{P} = \boxed{P} * \boxed{P}$) comes with a cost—once a proposition P has been turned into an invariant $\boxed{P}$, one is only allowed to access P during a single *atomic* execution step on the left-hand side. This restriction is crucial as the scheduling of threads on the left-hand side is demonic. When proving a refinement, we have to consider all possible interleavings of threads. If we were to be able to access an invariant for the duration of multiple steps, another thread could be scheduled in between, and observe that the invariant was temporarily broken.

Scheduling on the right-hand side, however, is angelic. That is, when proving a refinement, we have the ability to select the choice of scheduling. As a consequence, ReLoC allows us to execute multiple steps on the right-hand side while accessing an invariant.

Let us take a look at the way accessing invariants in ReLoC works. We do so by continuing the proof of our running example (after introducing $\Box$ and performing pure symbolic execution steps):

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} !\, c_i \precsim !\, c_s : \mathsf{int}.$$

At this point we would like to access the locations c_i and c_s stored in the invariant $\boxed{I_{\mathsf{cnt}}}$. For this we use the rule REL-LOAD-L-INV in Figure 4.

This rule is quite a mouthful, so let us first take a look at its shape before going into detail about the mask annotations and later modalities $\triangleright$. The essence of REL-LOAD-L-INV is that it provides temporary access to the resources P guarded by the invariant. In addition, it provides the *invariant closing resource* $\mathsf{closeInv}_{\mathcal{N}}(P)$, which can restore the invariant (using the rule REL-INV-RESTORE). The resources P can be used to prove $\ell \mapsto_{\mathsf{i}} v$, which is needed to justify the symbolic execution step on the left. Afterwards, we are left with the goal $\Delta \models_{\top\setminus\mathcal{N}} K[\,v\,] \precsim e_2 : \tau$. We typically do not immediately restore the invariant (using REL-INV-RESTORE), but first use the resources P to perform matching symbolic execution steps on the right.

In our example, by applying REL-LOAD-L-INV, we obtain $c_i \mapsto_{\mathsf{i}} n$ and $c_s \mapsto_{\mathsf{s}} n$ and $\mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})$, for some $n \in \mathbb{N}$, reducing our goal to $\models_{\top\setminus\mathcal{N}} n \precsim\, !\, c_s : \mathsf{int}$. We then use REL-LOAD-R to reduce our goal to $\models_{\top\setminus\mathcal{N}} n \precsim n : \mathsf{int}$. Because these steps did not change the heap, REL-INV-RESTORE's premises for closing the invariant are trivially met. The refinement proof is then concluded by applying the structural rules REL-RETURN and VAL-INT.

Let us take a look at the rules REL-LOAD-L-INV and REL-INV-RESTORE in more detail. A crucial aspect of these rules is that they ensure that access to the invariant $\boxed{P}^{\mathcal{N}}$ is *temporary*, *i.e.,* that P is only used during a single symbolic execution step on the left-hand side (but possibly several steps on the right), and that the same invariant cannot be opened twice. This is achieved by tagging each invariant $\boxed{P}^{\mathcal{N}}$ with a name $\mathcal{N} \in \mathit{InvName}$, and by keeping track of which invariants have been accessed. The latter is done in a way similar to Iris—like Iris's Hoare triples $\{P\}\, e\, \{Q\}_{\mathcal{E}}$, our refinement judgments $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ are annotated with a *mask* $\mathcal{E} \subseteq \mathit{InvName}$ of accessible invariants. By default all invariants are accessible, so we write $\Delta \models e_1 \precsim e_2 : \tau$ for $\Delta \models_{\top} e_1 \precsim e_2 : \tau$, where $\top$ is the set of all invariant names.

An invariant namespace is a (non-empty) list of strings or values: $\mathit{InvName} = \mathit{List}(\mathit{String} + \mathit{Val})$. When opening an invariant and removing it from a mask, we coerce an invariant namespace $\mathcal{N}$ into a mask by taking its upwards extension $\mathcal{N}^{\uparrow} = \{\mathcal{N}.x_1.\ldots.x_n \mid n \in \mathbb{N},\, x_i \in \mathit{String} + \mathit{Val}\}$. Abusing the notation, we write $\mathcal{E} \setminus \mathcal{N}$ for $\mathcal{E} \setminus \mathcal{N}^{\uparrow}$.

When accessing an invariant, *e.g.,* using REL-LOAD-L-INV or REL-CAS-L-INV, its namespace is removed from the mask annotation of the judgment. The removal of the namespace from the mask guarantees that invariants are only used for a single execution step on the left-hand side. After all, all rules for symbolic execution on the left-hand side require a $\top$ mask, whereas those for the right-hand side allow for an arbitrary mask. The only way of performing a subsequent step on the left-hand side is thus by first restoring the mask to $\top$, which can only be done by restoring the invariants that have been accessed (using the rule REL-INV-RESTORE).

One may wonder why refinement judgments are annotated with a mask instead of a Boolean that indicates if an invariant has been opened. As we will show in Section 4, ReLoC allows one to access multiple invariants simultaneously. To avoid *reentrancy*—which means accessing the same invariant twice in a nested fashion—we need to know exactly which invariants are opened.

An additional aspect to note is that invariants $\boxed{P}^{\mathcal{N}}$ in ReLoC (and Iris) are *impredicative* [SB14, JKJ+18]. This means that P is allowed to contain other invariant assertions $\boxed{Q}^{\mathcal{N}'}$ or even refinement judgments $e \precsim t : \tau$. As a consequence, to ensure soundness of the logic, all rules for invariants only provide access to $\triangleright P$, *i.e., P* "guarded" by the *later* modality $\triangleright$. When invariants are not used impredicatively (*i.e.,* invariants over so called

timeless propositions, which include connectives of first-order logic and heap assertions), these modalities can be soundly omitted.

3.4.3. *Later modality and Löb induction.* The later modality $\triangleright$ is not only used for resolving the impredicativity issues, but also for handling general recursion. As is custom in logics based on step-indexing [AM01], such as Iris, the later modality $\triangleright$ and Löb induction are used to reason about recursive functions. Specifically, Iris provides the following rules for $\triangleright$:

$$\frac{P}{\triangleright P}\ \triangleright\text{-intro} \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}\ \triangleright\text{-mono} \qquad \frac{\triangleright P \vdash P}{\vdash P}\ \text{Löb}$$

In our example, this means that by Löb induction (rule Löb), we may prove $\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$, under the assumption of the induction hypothesis $\triangleright(\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int})$. The induction hypothesis is 'guarded' by a $\triangleright$, and can only be used after we have performed a step of symbolic execution on the left-hand side. That is why the symbolic execution rules for the left-hand side contain the later modality in the premises. Let us see how it works in the example. We use rel-pure-l to arrive at:

$$\triangleright(\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}) \mathrel{-\!\!*}$$
$$\triangleright(\mathsf{let}\ c = !\,c_i\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(c_i, c, 1+c)\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}).$$

By monotonicity (rule $\triangleright$-mono), we can now remove $\triangleright$ both from the induction hypothesis and from the goal. Subsequently, we symbolically execute the load operation using the invariant, just like in the previous section, reaching the goal

$$\mathsf{if}\ \mathsf{CAS}(c_i, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$$

for some $n \in \mathbb{N}$. To symbolically execute the compare-and-set (CAS), we use rel-cas-l-inv. By this rule, we have to consider two outcomes, depending on whether the original value of the counter has changed between the load and compare-and-set operations or not.

(1) Suppose that the value of the counter c_i has changed. In that case the compare-and-set operation fails and we are left with

$$c_i \mapsto_{\mathsf{i}} m * c_s \mapsto_{\mathsf{s}} m * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false}) \mathrel{-\!\!*}$$
$$\models_{\top\setminus\mathcal{N}} \mathsf{if}\ \mathbf{false}\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$$

for some $m \neq n$. Because the symbolic heap has not been changed, we can easily restore the invariant and execute the $\mathsf{if}\ \mathbf{false}\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$ to obtain $\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$, which is exactly our induction hypothesis.

(2) If the value has not changed, then the compare-and-set succeeds and we are left with the new goal:

$$c_i \mapsto_{\mathsf{i}} (1+n) * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false}) \mathrel{-\!\!*}$$
$$\models_{\top\setminus\mathcal{N}} \mathsf{if}\ \mathbf{true}\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}.$$

At this point we use the symbolic execution rules rel-store-r, rel-load-r and the lock specifications from Figure 5 to symbolically execute the right-hand side of the refinement and update the resources to match:

$$c_i \mapsto_{\mathsf{i}} (1+n) * c_s \mapsto_{\mathsf{s}} (1+n) * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false}) \mathrel{-\!\!*}$$
$$\models_{\top\setminus\mathcal{N}} \mathsf{if}\ \mathbf{true}\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim n : \mathsf{int}.$$

We can then restore the invariant and symbolically execute the left-hand side to finish the proof.

Note that the point in the proof when we symbolically execute $\mathsf{inc}_s\ c_s\ lk$ on the right-hand side corresponds to the linearization point of inc_i.

This concludes the proof of the counter refinement. For the purposes of the proof, we have used some derived rules and principles in ReLoC. In the next section we will present an overview of primitive rules—the very core of ReLoC—and show how they can be used to recover the kind of intuitive reasoning we employed in this section.

4. A closer look at ReLoC

We now explain some of the more technical details of ReLoC, and show how the principles that we have used in Section 3 can be obtained from ReLoC's primitive proof rules. First, we describe how to work with invariants using Iris's update modality $\Rightarrow$ (Section 4.1). Then we explain the role and rules of persistent propositions (Section 4.2), and go through a selection of ReLoC's primitive proof rules and explain how the symbolic execution and structural rules can be derived from them (Section 4.3). Finally, we demonstrate how ReLoC's rules can be used to prove the *fundamental property*: if we can derive a typing judgment $\vdash e : \tau$, then e refines itself, *i.e.,* $e \precsim e : \tau$. To prove the fundamental property, we need to generalize the relational judgment to open terms, and prove the structural rules for open terms as well (Section 4.4).

A selection of ReLoC's primitive proof rules are shown in Figure 6.

4.1. Invariants and the update modality.

The rules for invariants in Figure 4 in Section 3.4.2 are fairly restrictive, *e.g.,* they allow us to open at most one invariant at the same time. Moreover, several of those rules, *e.g.,* REL-LOAD-L-INV and REL-CAS-L-INV, mix together symbolic execution and invariant manipulation. We now present ReLoC's more primitive proof rules, which integrate Iris's flexible mechanism for invariants and ghost state, and which can be used to derive rules such as like REL-LOAD-L-INV and REL-CAS-L-INV.

Invariants and ghost state in Iris are controlled via the *update modality* ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$. The intuition behind ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$ is to express that under the assumption that the invariants in $\mathcal{E}_1$ are accessible initially, one can obtain P, and end up in the situation where the invariants in $\mathcal{E}_2$ are accessible. Thus, for showing P we can open the invariants from $\mathcal{E}_1$ and have to restore the invariants from $\mathcal{E}_2$ (the invariants from $\mathcal{E}_1 \setminus \mathcal{E}_2$ may remain open). Furthermore, this modality allows one to perform changes to Iris's ghost state via *frame preserving updates*; for a description of those we refer the reader to [JKJ+18].

The key rules of the update modality are:

$$\frac{P}{{}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P}\ \Rightarrow\text{-INTRO} \qquad \frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q}\ \Rightarrow\text{-MONO}$$

$$\frac{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}\,{}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_3} P}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_3} P}\ \Rightarrow\text{-IDEMP} \qquad \frac{P * {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} (P * Q)}\ \Rightarrow\text{-SEP}$$

These rules say that the update modality is a monad, which is indexed (due to the masks), and strong (due to rule $\Rightarrow$-SEP). In ReLoC (and Iris) proofs, we often need to eliminate update modalities in the proof context, which is allowed if the goal is an update modality

Value interpretation rules:

$$\text{VAL-VAR}\ \frac{\Delta(\alpha)(v_1, v_2)}{[\![\alpha]\!]_\Delta(v_1, v_2)} \qquad \text{VAL-UNIT}\ \frac{v_1 = v_2 = ()}{[\![\mathsf{unit}]\!]_\Delta(v_1, v_2)} \qquad \text{VAL-BOOL}\ \frac{\exists b \in \mathbb{B}.\, v_1 = v_2 = b}{[\![\mathsf{bool}]\!]_\Delta(v_1, v_2)} \qquad \text{VAL-INT}\ \frac{\exists n \in \mathbb{Z}.\, v_1 = v_2 = n}{[\![\mathsf{int}]\!]_\Delta(v_1, v_2)}$$

$$\text{VAL-PROD}\ \frac{\exists v_1, v_2, w_1, w_2.\, v = (v_1, v_2) * w = (w_1, w_2) * [\![\tau]\!]_\Delta(v_1, w_1) * [\![\sigma]\!]_\Delta(v_2, w_2)}{[\![\tau \times \sigma]\!]_\Delta(v, w)}$$

$$\text{VAL-ARR}\ \frac{\square(\forall w_1\, w_2.\, [\![\tau]\!]_\Delta(w_1, w_2) \mathrel{-\!\!*} \Delta \models v_1\ w_1 \precsim v_2\ w_2 : \sigma)}{[\![\tau \to \sigma]\!]_\Delta(v_1, v_2)}$$

Monadic rules:

$$\text{REL-RETURN}\ \frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \models v_1 \precsim v_2 : \tau} \qquad \text{REL-BIND}\ \frac{\Delta \models e_1 \precsim e_2 : \tau \qquad \forall v_1\, v_2.\, [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} \Delta \models K_1[\,v_1\,] \precsim K_2[\,v_2\,] : \sigma}{\Delta \models K_1[\,e_1\,] \precsim K_2[\,e_2\,] : \sigma}$$

Type-directed structural rules:

$$\text{REL-FORK}\ \frac{\Delta \models e_1 \precsim e_2 : \mathsf{unit}}{\Delta \models \mathsf{fork}\ \{e_1\} \precsim \mathsf{fork}\ \{e_2\} : \mathsf{unit}}$$

Symbolic execution rules:

$$\text{REL-LOAD-L}\ \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\exists v.\, \ell \mapsto_{\mathsf{i}} v * \triangleright \left(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau\right)\right)}{\Delta \models K[\,!\,\ell\,] \precsim e_2 : \tau}$$

$$\text{REL-STORE-L}\ \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\ell \mapsto_{\mathsf{i}} - \mathrel{-\!\!*} \triangleright \left(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau\right)\right)}{\Delta \models K[\,\ell \leftarrow v\,] \precsim e_2 : \tau}$$

$$\text{REL-CAS-L}\ \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\begin{array}{r}\exists v'.\, \ell \mapsto_{\mathsf{i}} v' * \triangleright \left(v' \neq v_1 \mathrel{-\!\!*} \triangleright(\ell \mapsto_{\mathsf{i}} v' \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{false}\,] \precsim e_2 : \tau)\right) \wedge \\ \triangleright \left(v' = v_1 \mathrel{-\!\!*} \triangleright(\ell \mapsto_{\mathsf{i}} v_2 \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{true}\,] \precsim e_2 : \tau)\right)\end{array}\right)}{\Delta \models K[\,\mathsf{CAS}(\ell, v_1, v_2)\,] \precsim e_2 : \tau}$$

Invariants rules (INV-ALLOC and INV-ACCESS are inherited from Iris):

$$\text{REL-UPD}\ \frac{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \Delta \models_{\mathcal{E}_2} e_1 \precsim e_2 : \tau}{\Delta \models_{\mathcal{E}_1} e_2 \precsim e_2 : \tau} \qquad \text{INV-ALLOC}\ \frac{\triangleright P}{\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}} \qquad \text{INV-ACCESS}\ \frac{\mathcal{N} \subseteq \mathcal{E} \qquad \boxed{P}^{\mathcal{N}}}{{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \mathrel{{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Longrightarrow\!\!\!*^{\mathcal{E}}} \mathsf{True})}$$

FIGURE 6. Selected primitive rules of ReLoC.

with corresponding source mask. This is expressed by the following derived rule:

$$\frac{\Rightarrow\text{-ELIM}\quad {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \qquad P \mathrel{-\!\!*} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_3} Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_3} Q}$$

This rule is derivable from $\Rightarrow$-MONO, and $\Rightarrow$-IDEMP.

Before we will describe the rules of the update modality related to invariants, let us describe some syntactic sugar that we inherit from Iris. We write $\Rightarrow_{\mathcal{E}} P$ for ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P$, and $\Rightarrow P$ for $\Rightarrow_{\top} P$, where $\top$ is the set of all invariant names. Moreover, since the update modality is often combined with the magic wand, we write $P \mathrel{{}^{\mathcal{E}_1}\!\Rrightarrow\!\!\!\!*^{\mathcal{E}_2}} Q$ for $P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q$, and follow the same conventions for omitting masks on $\Rrightarrow\!\!\!*$ as used for $\Rightarrow$.

ReLoC's main rule for interacting with the update modality is REL-UPD. It allows to eliminate an update modality around a refinement judgment. To get an idea of how this rule is used, let us take a look at the primitive rule INV-ALLOC for allocating an invariant. The derived rule REL-INV-ALLOC in Figure 4 is a composition of REL-UPD with Iris's rules $\Rightarrow$-ELIM and INV-ALLOC.

By combining REL-UPD with Iris's rules $\Rightarrow$-ELIM and INV-ACCESS for accessing invariants, one can turn an invariant $\boxed{P}^{\mathcal{N}}$ into its content P, together with a way of restoring the invariant $\triangleright P \mathrel{{}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!\!\!\!*^{\mathcal{E}}} \mathsf{True}$. It is important to notice that by using the combination of these rules, the mask on the refinement judgment changes from $\mathcal{E}$ into $\mathcal{E}\setminus\mathcal{N}$. This prohibits access to the invariant $\mathcal{N}$ until it has been restored—thus preventing reentrancy. Restoring the invariant is done by using the rule REL-UPD with the premise $\triangleright P \mathrel{{}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!\!\!\!*^{\mathcal{E}}} \mathsf{True}$. This requires one to give up P, and in turn transforms the mask of the judgment back into $\mathcal{E}$. Note that one can use INV-ACCESS multiple times to open multiple invariants.

Invariants and symbolic execution. Opening invariants through REL-UPD and INV-ACCESS as described above is fairly limited. Once we open an invariant, the mask at the refinement judgment changes from $\top$ into $\top\setminus\mathcal{N}$, which prevents any symbolic execution on the left-hand side. The rules for symbolic execution on that side require the mask to be $\top$. As we discussed in Section 3.4.2 already, this restriction to the $\top$ mask on left-hand side rules is crucial. It is unsound to perform multiple symbolic execution steps on the left while an invariant is open. To see why this is the case, consider the following refinement:

$$(\lambda x.\, \mathsf{let}\ n = !\,x\ \mathsf{in}\ x \leftarrow n+1; n) \precsim \mathsf{inc}_i : \mathsf{ref}\ \mathsf{int} \to \mathsf{int}$$

This refinement does not hold because the two programs can be distinguished by the context:

$$\mathsf{let}\ c = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{let}\ f = [\bullet]\ \mathsf{in}\ \mathsf{fork}\ \{f\ c\}\,; f\,c.$$

The left-hand side is basically the coarse-grained increment operation inc_s without the lock protection. Thus, the function on the left-hand side does not guarantee thread-safety: the value of the passed reference can change unpredictably if the function is invoked in parallel with itself. By contrast, the inc_i always increments the counter monotonically.

If we were allowed to perform multiple symbolic execution rules on the left-hand side, then we could have proven the above refinement, using an invariant of $\boxed{\exists n.\, c_s \mapsto_{\mathsf{s}} n * c_i \mapsto_{\mathsf{i}} n}$.

In order to support symbolic execution with invariants, ReLoC provides additional rules to simultaneously access an invariant and perform a single atomic symbolic execution step on the left-hand side. Examples of such rules are REL-LOAD-L, REL-STORE-L and REL-CAS-L.

We can now explain the derived rule REL-LOAD-L-INV in terms of the primitive rules. The proposition $\triangleright P \, {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$ is used for closing the invariant $\mathcal{N}$ because it changes the mask from $\mathcal{E}\setminus\mathcal{N}$ to $\mathcal{E}$. Thus $\mathsf{closeInv}_{\mathcal{N}}(P) \triangleq (\triangleright P \, {}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow^{\top} \mathsf{True})$. To prove REL-LOAD-L-INV from Figure 4, we apply REL-LOAD-L to obtain the goal:

$$ {}^{\top}\!\Rrightarrow^{\top\setminus\mathcal{N}} \big(\exists v.\, \ell \mapsto_{\mathsf{i}} v * \triangleright \big(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\top\setminus\mathcal{N}} K[\,v\,] \precsim e : \tau\big)\big).$$

We then use INV-ACCESS and $\Rrightarrow$-ELIM to get the premise of REL-LOAD-L-INV. In the same way REL-CAS-L-INV can be derived from REL-CAS-L. Finally, the closing rule REL-INV-RESTORE is a consequence of the definition of $\mathsf{closeInv}_{\mathcal{N}}(P)$ and REL-UPD.

Using ReLoC's primitive symbolic execution rules such as REL-LOAD-L, REL-STORE-L and REL-CAS-L one can also derive the following weaker, but perhaps more intuitive, symbolic execution rule:

$$\text{REL-STORE-L'}\quad \frac{\ell \mapsto_{\mathsf{i}} v \qquad \triangleright \big(\ell \mapsto_{\mathsf{i}} w \mathrel{-\!\!*} \Delta \models K[\,()\,] \precsim e_2 : \tau\big)}{\Delta \models K[\,\ell \leftarrow w\,] \precsim e_2 : \tau}$$

Since these rules have a $\top$ mask, they can only be used when no invariants have been opened. Recall that by contrast, the symbolic execution rules for the right-hand side, such as REL-LOAD-R, REL-STORE-R in Figure 4, which are of a similar shape, can be performed even with invariants open because they allow the mask to be arbitrary.

4.2. **The persistence modality.** Recall from Section 3.4.2 that a proposition P is persistent, written as $\mathsf{persistent}(P)$, if $P \vdash \Box P$, where $\Box$ is Iris's *persistence* modality. The $\Box$ modality plays an important role in ReLoC because it makes it possible to express that if two expressions are related, they remain related forever. For example, the persistence modality plays a crucial role in the rule REL-REC in Figure 4—it ensures that ephemeral resources (such as heap assertions) are not used for the verification of the closure's body. After all, closures can be invoked arbitrarily many times at different points in time (possibly concurrently), and hence it is impossible to guarantee that ephemeral resources will still be available when the closure is called. For example, without the $\Box$ modality in the premise of REL-REC one would be able to prove the following unsound refinement:

$$\mathsf{let}\ \ell = \mathsf{ref}(0)\ \mathsf{in}\ \lambda().\, \ell \leftarrow 1 + !\,\ell;\, !\,\ell \ \precsim\ \lambda().\, 1 \ :\ \mathsf{unit} \to \mathsf{int}.$$

One would use REL-ALLOC-L' to obtain the heap assertion $\ell \mapsto_{\mathsf{i}} 0$, and subsequently use that assertion to verify the body of the closure. Fortunately, the $\Box$ modality in REL-REC prevails—$\ell \mapsto_{\mathsf{i}} 0$ is ephemeral, not persistent, so cannot be moved under a $\Box$.

In Section 3.4.2 we gave an idea of the core rules of the persistence modality. Let us now take a look at the rules in more detail:

$$\Box\text{-DUP}\ \frac{\Box P * \Box P}{\Box P} \qquad \Box\text{-ELIM}\ \frac{\Box P}{P} \qquad \Box\text{-MONO}\ \frac{P \vdash Q}{\Box P \vdash \Box Q} \qquad \Box\text{-IDEMP}\ \frac{\Box P}{\Box\Box P} \qquad \Box\text{-SEP}\ \frac{\Box P * \Box Q}{\Box(P * Q)}$$

The rules $\Box$-DUP and $\Box$-ELIM say that the $\Box P$ is duplicable, and one can get P out. The rule $\Box$-IDEMP says that $\Box P$ itself is persistent. The rules $\Box$-ELIM, $\Box$-MONO and $\Box$-IDEMP say that $\Box$ is in fact a co-monad. Finally, $\Box$ commutes with most logical connectives, for example, the separating conjunction, as expressed by $\Box$-SEP.

If we wish to prove $\Box Q$ under the assumptions $P_1, \ldots, P_n$, where each P_i is persistent, then we can introduce the $\Box$ modality and prove Q from $P_1, \ldots, P_n$:

$$\frac{\mathsf{persistent}(P_1) \qquad \ldots \qquad \mathsf{persistent}(P_n) \qquad P_1 * \cdots * P_n \vdash Q}{P_1 * \cdots * P_n \vdash \Box Q}\ \Box\text{-INTRO}$$

This rule is derivable from the definition of $\mathsf{persistent}(-)$, $\Box$-SEP, and $\Box$-MONO.

Note that $\mathsf{persistent}(P)$ is defined through the validity relation $P \vdash \Box P$; *i.e.,* it is a meta-logical notion (in terms of the mechanization, $\mathsf{persistent}(P)$ is a Coq-level predicate, not an Iris-level predicate). As such, the rule above does not fit the description we have given to the inference rules in Section 3.1. Rather, it should be seen as a family of inference rules indexed by meta-level propositions $\mathsf{persistent}(P_1), \ldots, \mathsf{persistent}(P_n)$.

4.3. **Value interpretation and monadic rules.** In addition to the refinement judgment $\Delta \models e_1 \precsim e_2 : \tau$, which relates expressions e_1 and e_2, ReLoC provides the value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$, which relates values v_1 and v_2. The rule REL-RETURN expresses that $[\![\tau]\!]_\Delta(v_1, v_2)$ implies $\Delta \models v_2 \precsim v_2 : \tau$. However, the inverse direction does not hold, $[\![\tau]\!]_\Delta(v_1, v_2)$ is strictly stronger than $\Delta \models v_1 \precsim v_2 : \tau$ as its rules (in Figure 6) are bidirectional, whereas those for the expression judgment are unidirectional. The bidirectionality is crucial for the rule REL-REC in Figure 4, as it contains $[\![\tau]\!]_\Delta(v_1, v_2)$ in negative position—*i.e.,* as a client of REL-REC one gets $[\![\tau]\!]_\Delta(v_1, v_2)$ as an assumption and hence needs to eliminate it.

We want the value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$ to be persistent, because our type system is not substructural, *i.e.,* types denote knowledge, but not ownership of data. For example, in typing the expression $e_1 \leftarrow e_2$ with STORE-TYPED, we use the same context Γ for type checking both e_1 and e_2. In order to semantically validate such rules, we want the propositions $[\![\tau]\!]_\Delta(v_1, v_2)$ to be duplicable. To that end, we require all the interpretations in the context Δ to be persistent. That is why the rule REL-PACK in Figure 4 has a side-condition $\forall v_1, v_2.\, \mathsf{persistent}(R(v_1, v_2))$.

The value interpretation also appears in the monadic rules REL-RETURN and REL-BIND in Figure 6. These rules are used to derive all type-directed structural rules of ReLoC, with the exception of REL-FORK, which is the sole primitive type-directed structural rule. As an example, consider the type-directed structural rule for the first projection π_1.

Lemma 4.1. *The following rule is derivable:*

$$\frac{\Delta \models e_1 \precsim e_2 : \tau \times \sigma}{\Delta \models \pi_1(e_1) \precsim \pi_1(e_2) : \tau}$$

Proof. By REL-BIND it suffices to show:

- $\Delta \models e_1 \precsim e_2 : \tau \times \sigma$, but this is exactly our assumption;
- for any v, w: $[\![\tau \times \sigma]\!]_\Delta(v, w) \mathrel{-\!\!*} \Delta \models \pi_1(v) \precsim \pi_1(w) : \tau$.

By VAL-PROD we have values v_i, w_i for $i \in \{1, 2\}$ such that $v = (v_1, v_2)$ and $w = (w_1, w_2)$ and $[\![\tau]\!]_\Delta(v_1, w_1) * [\![\sigma]\!]_\Delta(v_2, w_2)$. Using REL-PURE-L and REL-PURE-R we reduce the goal $\Delta \models \pi_1(v_1, v_2) \precsim \pi_1(w_1, w_2) : \tau$ to $\Delta \models v_1 \precsim w_1 : \tau$. At this point we apply REL-RETURN. $\square$

4.4. **Fundamental theorem and refinements of open terms.** The type-directed structural rules[3] are also used for proving the following theorem, which is a standard result for logical relation models of type systems:

Theorem 4.2 (Fundamental theorem for closed terms)**.** *If expression e is well typed,* i.e., *$\vdash e : \tau$, then e refines itself,* i.e., *the judgment $e \precsim e : \tau$ is derivable in ReLoC.*

We wish to prove this theorem by induction on the typing derivation. But in order to make it work, we need to generalize the theorem to open terms (*e.g.,* in order to deal with the REC-TYPED case). Consequently, we need to generalize ReLoC's refinement judgment $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ to open terms e_1 and e_2 whose free variables are bound by the typing context Γ. To define the refinement judgment for open terms, we first define a standard notion of a *closing substitution*.

Definition 4.3. A mapping $\gamma : \mathit{Var} \to \mathit{Val} \times \mathit{Val}$ is a *closing substitution w.r.t. the typing environment* Γ, notation $[\![\Gamma]\!]^*_\Delta(\gamma)$, if

$$\forall (x, \tau) \in \Gamma.\, [\![\tau]\!]_\Delta(\gamma_1(x), \gamma_2(x)),$$

where $\gamma_i(x) = \pi_i(\gamma(x))$ is the i-th projection of $\gamma(x)$.

Definition 4.4. The *refinement judgment $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ for open terms* is defined as:

$$\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau \;\triangleq\; \forall \gamma.\, [\![\Gamma]\!]^*_\Delta(\gamma) \mathrel{-\!\!*} \Delta \models \gamma_1(e_1) \precsim \gamma_2(e_2) : \tau.$$

Using the refinement judgment for open terms we can now state versions of the type-directed structural rules for open terms. For example:

$$\frac{\Delta \mid x : \tau, \Gamma \models e_1 \precsim e_2 : \sigma}{\Delta \mid \Gamma \models \lambda x.\, e_1 \precsim \lambda x.\, e_2 : \tau \to \sigma}$$

This rule can be proven by unfolding the definition of the refinement judgment for open terms and proceeding as in Lemma 4.1. Finally, we can state and prove the generalized version of the fundamental theorem for open terms:

Theorem 4.5 (Fundamental theorem for open terms)**.** *If $\Xi \mid \Gamma \vdash e : \tau$, then $\Delta \mid \Gamma \models e \precsim e : \tau$ is derivable in ReLoC, for all Δ which contain the variables from Ξ.*

Proof. By induction on the typing derivation, using the versions of the type-directed structural rules for open terms. □

With the refinement judgments generalized to open terms, we can state and the generalized version of Theorem 3.1 for open terms, which we prove in Section 7.5.

Theorem 4.6 (Soundness for open terms)**.** *Let Ξ be a type environment. Suppose that refinement judgment $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC for all Δ which contain the variables from Ξ. Then $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$.*

[3]Our *type-directed structural rules* are often called *compatibility lemmas* in the logical relation literature.

5. Relational specifications in ReLoC

Due to its first-class refinement judgments, ReLoC can be used to give *relational specifications* to programs. Similar to Hoare triples, relational specifications abstract away from a program's implementation by expressing its behavior in terms of a pre- and postcondition. Relational specifications apply to the situation when the expression on the one side of the refinement contains a program subject to specification, while the expression on the other side is arbitrary. In Figure 5 in Section 3 we saw an example of a right-hand side relational specifications for locks, which we then used to verify a counter module.

We start this section by describing the general format of non-atomic relational specifications (Section 5.1). Non-atomic relational specifications are sufficient to give strong specifications for the right-hand left, but due to the demonic nature of the left-hand side, we often need stronger specifications for the left-hand side (Section 5.2). We therefore introduce *logically atomic relation specifications*, which generalize da Rocha Pinto *et al.*'s TaDA-style specifications [dRPDG14] (Section 5.3) and Svendsen *et al.*'s HOCAP-style specifications [SBP13] (Section 5.4) from the Hoare-logic setting to the relational setting. Finally, we show how to use logically atomic specifications to verify a ticket lock (Section 5.5).

5.1. Non-atomic relational specifications.

5.1.1. *Right-hand side relational specifications.* Consider the following implementation of a lock, which we refer to as a *spin lock*:

$$\begin{aligned}\mathsf{newlock} &\triangleq \lambda().\ \mathsf{ref}(\mathbf{false})\\ \mathsf{acquire} &\triangleq \lambda\ell.\ \mathsf{if}\ \mathsf{CAS}(\ell, \mathbf{false}, \mathbf{true})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{acquire}\ \ell\\ \mathsf{release} &\triangleq \lambda\ell.\ \ell \leftarrow \mathbf{false}\end{aligned}$$

For this specific implementation, we can prove the rules in Figure 5 in Section 3.4 by defining the lock predicates as follows:

$$\mathsf{isLock_s}(lk, b) \triangleq lk \in \mathit{Loc} * lk \mapsto_{\mathsf{s}} b.$$

The rules for locks in Figure 5 follow a certain pattern. For an expression e_2 that, under precondition P, reduces to v, with postcondition $Q(x, v)$, we have the following rule:

$$\frac{P \qquad \forall x, v.\ Q(x, v) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

The postcondition $Q : X \times \mathit{Val} \to \mathit{iProp}$ also depends on a type X, provided by the provider of the rule. This rule pattern can be considered a relational version of a Hoare triple for a program on the right-hand side of the refinement judgment.

5.1.2. *Left-hand side relational specifications.* We can formulate a similar pattern for programs on the left-hand side of the refinement judgment:

$$\frac{P \qquad \forall x, v.\ Q(x, v) \mathrel{-\!\!*} \Delta \models K[\,v\,] \precsim e_2 : \tau}{\Delta \models K[\,e_1\,] \precsim e_2 : \tau}$$

Using this pattern we can state and prove a relational version of the standard separation logic specification for locks, which is shown in Figure 7. This specification makes use of the lock predicate $\mathsf{isLock_i}(\gamma, lk, R)$, which states that lk protects the resources described by the

$$\textsc{newlock-l}\ \frac{R \qquad \forall lk,\, \gamma.\, \mathsf{isLock_i}(\gamma, lk, R) \mathrel{-\!\!*} K[\,lk\,] \precsim e_2 : \tau}{K[\,\mathsf{newlock}\ ()\,] \precsim e_2 : \tau} \qquad \textsc{is-lock-pers}\ \frac{\mathsf{isLock_i}(\gamma, lk, R)}{\Box\, \mathsf{isLock_i}(\gamma, lk, R)}$$

$$\textsc{acquire-l}\ \frac{\mathsf{isLock_i}(\gamma, lk, R) \qquad (\mathsf{locked_i}(\gamma) \mathrel{-\!\!*} R \mathrel{-\!\!*} K[\,()\,] \precsim e_2 : \tau)}{K[\,\mathsf{acquire}\ lk\,] \precsim e_2 : \tau}$$

$$\textsc{release-l}\ \frac{\mathsf{isLock_i}(\gamma, lk, R) \qquad \mathsf{locked_i}(\gamma) \qquad R \qquad K[\,()\,] \precsim e_2 : \tau}{K[\,\mathsf{release}\ lk\,] \precsim e_2 : \tau}$$

FIGURE 7. Left-hand side relational specification for locks.

proposition R. When creating a lock using newlock, the resources R have to be given up, and the persistent lock predicate $\mathsf{isLock_i}(\gamma, lk, R)$ is given in return. A thread that acquires a lock by calling acquire gets access to R for the duration of the critical section, and has to give R back when calling release. The token $\mathsf{locked_i}(\gamma)$, where γ is a *ghost name* associated to the lock, makes sure that a lock can only be released when it has been acquired. To prove the left-hand side specification for the spin lock, we define the lock predicate $\mathsf{isLock_i}(\gamma, lk, R)$ following the usual definition in Iris:

$$\mathsf{isLock_i}(\gamma, lk, R) \triangleq lk \in Loc * \boxed{(lk \mapsto_{\mathsf{i}} \mathbf{false} * \mathsf{locked_i}(\gamma) * R) \vee (lk \mapsto_{\mathsf{i}} \mathbf{true})}^{\mathcal{N}_{\mathsf{lock}}}$$

This definition uses an Iris invariant to express that the lock is either unlocked ($lk \mapsto_{\mathsf{i}} \mathbf{false}$), in which case it holds the token $\mathsf{locked_i}(\gamma)$ and the resources R, or locked ($lk \mapsto_{\mathsf{i}} \mathbf{true}$), in which case it holds no resources, since those are held by the thread that acquired the lock. The token $\mathsf{locked_i}(\gamma)$ is an exclusive resource that is obtained from Iris's ghost theory.

5.1.3. *Left- versus right-hand side relational specifications.* As we have seen in the specifications for the lock, there is an asymmetry between the left- and right-hand side specifications. The left-hand side specification of acquire (ACQUIRE-L) can be used regardless of whether the lock is unlocked, whereas the right-hand specification (ACQUIRE-R) can be used solely if the lock is unlocked (*i.e.,* if $\mathsf{isLock_s}(lk, \mathbf{false})$). This is due to the demonic nature of the left-hand side and the angelic nature of the right-hand side. For acquire on the left-hand side, we have to consider an arbitrary execution, whereas for acquire on the right-hand side we have to provide an execution ourselves. That is, for acquire on the right-hand side we have to show that it actually acquires the lock and reduces to (), which is only possible when the lock is unlocked. For this reason we use the predicate $\mathsf{isLock_s}(lk, b)$, which tracks the state b of the lock.

5.2. **The need for logically atomic specifications.** Recall from Section 4.1 that for any primitive (stateful) operation we have a symbolic execution rule that allows the client to access shared resourced stored in an invariant. For example, the rule REL-STORE-L for the store operation is as follows:

$$\frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \big(\ell \mapsto_{\mathsf{i}} - * \triangleright \big(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau\big)\big)}{\Delta \models K[\,\ell \leftarrow v\,] \precsim e_2 : \tau}$$

Concretely, the update modality ${}^{\top}\!\Rightarrow^{\mathcal{E}}$ in the premise of this rule allows users to use INV-ACCESS to access an invariant for the duration of the operation. The mask $\mathcal{E}$ in the refinement judgment $\Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau$ (that appears in the premise of the rule) forces the user to close the invariant at the end of the duration of the operation. The ability to open an invariant is sound because operations such as store are *physically atomic—i.e.,* they reduce in one step. As a consequence of being physically atomic, other threads cannot observe that the invariant has been broken during the execution of the operation.

In contrast, methods of a concurrent program module are typically composed of several operations and hence they are not physically atomic. For example, consider the increment function inc_i of the fine-grained counter module from Figure 1:

$$\mathsf{inc}_i \triangleq \mathsf{rec}\ \mathit{inc}\ c = \mathsf{let}\ n = !\,c\ \mathsf{in} \\ \mathsf{if}\ \mathsf{CAS}(c, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ \mathit{inc}\ c$$

This function is a compound expression that does not reduce to a value in a single step. Nevertheless, during the execution of this function there is a single instant at which the whole operation actually appears to take the effect—namely the successful reduction of the compare-and-set operation (CAS). This instant is called the *linearization point.* What it means is that, for an outside observer, the method inc_i behaves *as if* it was atomic, and we wish to express that in this function's relational specification.

This phenomenon is called *logical atomicity* in the literature, and has been studied extensively in the context of Hoare-style logics [JP11, dRPDG14, SBP13, JSS$^+$15, JLP$^+$20]. In the upcoming subsections we will how to generalize the concept of logical atomicity to the relational setting, and how that gives rise to *logically atomic relational specifications.* Concretely, we generalize da Rocha Pinto *et al.*'s TaDA-style specifications [dRPDG14] (Section 5.3) and Svendsen *et al.*'s HOCAP-style specifications [SBP13] (Section 5.4) from the Hoare-logic setting to the relational setting. Establishing the formal comparison between the two styles is out of the scope of this paper. Rather, we demonstrate that both approaches can be applied to the context of relational specifications.

5.3. **TaDA-style relational specifications.**

5.3.1. *Formulating TaDA-style specifications.* We take inspiration from the encoding of TaDA-style logically atomic Hoare triples in Iris [JSS$^+$15] and assign the following logically atomic relational specification to inc_i:

$$\text{INC-I-L-TADA}\quad \frac{\square\ {}^{\top}\!\Rightarrow^{\mathcal{E}} \exists n.\ c \mapsto_{\mathsf{i}} n * \left(\begin{array}{l}\left(c \mapsto_{\mathsf{i}} n\ {}^{\mathcal{E}}\!\Rightarrow\!\!\!\ast^{\top}\ \mathsf{True}\right) \wedge \\ \left(c \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau\right)\end{array}\right)}{K[\,\mathsf{inc}_i\ c\,] \precsim e : \tau}$$

Contrary to the non-atomic specification, we do not have $c \mapsto_{\mathsf{i}} n$ as a premise of the rule directly, but instead the premise contains a way of obtaining $c \mapsto_{\mathsf{i}} n$. The typical way of obtaining $c \mapsto_{\mathsf{i}} n$ is by accessing an invariant, which is formally done by using the update modality ${}^{\top}\!\Rightarrow^{\mathcal{E}}$ in the premise combined with INV-ACCESS from Figure 6.

To justify the remaining part of the premise of the rule we need to take a closer took at the behavior of $\mathsf{inc}_i\ c$, whose implementation (Figure 1) we recall to be as follows:

$$\mathsf{inc}_i \triangleq \mathsf{rec}\ inc\ c = \mathsf{let}\ n = !\,c\ \mathsf{in}\\ \mathsf{if}\ \mathsf{CAS}(c, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ inc\ c$$

The compare-and-set operation (CAS) can either succeed or fail. If it succeeds, then we have managed to update our resources to $c \mapsto_{\mathsf{i}} (n+1)$, and we can proceed with proving $\models_{\mathcal{E}} K[\,n\,] \precsim e : \tau$ under that premise. This explains the $(c \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau)$ clause. If, however, the compare-and-set fails, then we need to be able to restart the whole computation of $\mathsf{inc}_i\ c$. For that we must be able to return $c \mapsto_{\mathsf{i}} n$ to the invariant. Hence the $(c \mapsto_{\mathsf{i}} n\ {}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True})$ clause. (The same clause is used for performing operations that do not modify the state, such as dereferencing.)

Finally, we know that the computation can either succeed or be restarted—but not both. We have to accommodate for both situations, just not at the same time. Hence the last two clauses described here are connected by an intuitionistic conjunction ($\wedge$), instead of the separating conjunction ($*$).

5.3.2. *Using TaDA-style specifications.* We use the logically atomic relational specification INC-I-L-TADA to prove the refinement that we have seen in Section 3.4.2. The new proof is more modular since it does not appeal to the definition of inc_i. The refinement that we want to prove is as follows:

$$\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}} \mathrel{-\!\!*} \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ l : \mathsf{int}$$

Recall that $I_{\mathsf{cnt}} \triangleq \exists n \in \mathbb{N}.\, c_i \mapsto_{\mathsf{i}} n * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})$. To prove this goal, we use INC-I-L-TADA. After introducing the persistence modality (using $\Box$-INTRO, which is allowed, because there are no ephemeral assumptions in our context), this gives the following new goal (under the assumption $\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}}$):

$${}^{\top}\!\Rrightarrow^{\top\setminus\mathcal{N}} \exists n.\, c_i \mapsto_{\mathsf{i}} n * \left(\begin{array}{l}(c_i \mapsto_{\mathsf{i}} n\ {}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}) \wedge\\ (c_i \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\top\setminus\mathcal{N}} n \precsim \mathsf{inc}_s\ c_s\ l : \mathsf{int})\end{array}\right)$$

At this point we can open up the invariant I_{cnt} (using INV-ACCESS), and thereby introduce the update modality. The contents of the invariant provides us with a witness for the existential quantifier and allows us to discharge $c \mapsto_{\mathsf{i}} n$. We are left with proving the conjunction:

$$(c_i \mapsto_{\mathsf{i}} n\ {}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}) \quad \wedge \quad (c_i \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\top\setminus\mathcal{N}} n \precsim \mathsf{inc}_s\ c_s\ l : \mathsf{int})$$

under the assumption of the unused resources $\mathsf{isLock}_{\mathsf{s}}(l, \mathbf{false})$ and $c_s \mapsto_{\mathsf{s}} n$ from the invariant, and the invariant closing resource $\triangleright\, I_{\mathsf{cnt}}\ {}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}$.

The first conjunct corresponds to the case in which we close the invariant without modifying anything in our current context (*i.e.,* the compare-and-set has failed). It follows directly from the invariant closing resource. It thus remains to prove the second conjunct (*i.e.,* the compare-and-set has succeeded), which means we should prove $\models_{\top\setminus\mathcal{N}} n \precsim \mathsf{inc}_s\ c_s\ l : \mathsf{int}$ under the assumptions $c_i \mapsto_{\mathsf{i}} (n+1)$ and $\mathsf{isLock}_{\mathsf{s}}(l, \mathbf{false})$ and $c_s \mapsto_{\mathsf{s}} n$ and $\triangleright\, I_{\mathsf{cnt}}\ {}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}$. At this point we finish the proof by symbolically executing $\mathsf{inc}_s\ c_s\ l$ on the right-hand side before closing the invariant using invariant closing resource.

5.3.3. *General format of TaDA-style specifications.* The general format of logically atomic rules for logical refinements is the following:

$$\frac{R \qquad \Box\, {}^{\top}\!\Rrightarrow^{\mathcal{E}} \exists x.\, P(x) * \left(\begin{array}{l}\left(P(x)\ {}^{\mathcal{E}}\!\Longrightarrow\!\!\!\!\!\!\ast^{\top}\ \mathsf{True}\right) \wedge \\ \left(\forall v.\, Q(x,v) * R \mathrel{-\!\!*}\ \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau\right)\end{array}\right)}{K[\,e_1\,] \precsim e_2 : \tau}$$

Here, $P : X \to \mathit{iProp}$ is a predicate describing consumed resources, and $Q : X \times \mathit{Val} \to \mathit{iProp}$ is a predicate describing produced resources, both dependent on a type X supplied by the provider of the rule (*e.g.,* a library that exports the program e_1). This parameter X is selected on per-specification basis. For example, for the counter module X is going to be the type of natural numbers.

We include a frame R, which can be chosen by the client, for the following reason. The second premise of the rule resides below a persistence modality. Whenever we prove a goal of the form $\Box\, P$ we must prove P using only persistent resources, and thus have to throw all the ephemeral resources away (see $\Box$-INTRO in Section 4.2). However, we do not want to throw away all the ephemeral resources that we have for eternity (as they might be needed to close invariants afterwards or to proceed otherwise with the proof), so we give them up only temporarily, by collecting them in R.

5.3.4. *Proving TaDA-style specifications.* Following the general scheme, we now state and prove the TaDA-style specification of our increment function:

INC-I-L-TADA-GEN

$$\frac{R \qquad {}^{\top}\!\Rrightarrow^{\mathcal{E}} \Box\, \exists n.\, c \mapsto_{\mathsf{i}} n * \left(\begin{array}{l}\left(c \mapsto_{\mathsf{i}} n\ {}^{\mathcal{E}}\!\Longrightarrow\!\!\!\!\!\!\ast^{\top}\ \mathsf{True}\right) \wedge \\ \left(c \mapsto_{\mathsf{i}} (n+1) * R \mathrel{-\!\!*}\ \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau\right)\end{array}\right)}{K[\,\mathsf{inc}_i\ c\,] \precsim e : \tau}$$

To prove this specification, we proceed by Löb induction and symbolic execution. At the point when we need to symbolically dereference c we apply REL-LOAD-L. We then use the update that we have as a premise of the specification to obtain $c \mapsto_{\mathsf{i}} n$ for some n. After providing $c \mapsto_{\mathsf{i}} n$ for the load operation, we use the first conjunct $c \mapsto_{\mathsf{i}} n * {}^{\mathcal{E}}\!\Longrightarrow\!\!\!\!\!\!\ast^{\top}\ \mathsf{True}$ to restore the mask on the refinement judgment.

After that we have to symbolically execute the compare-and-set operation; we apply REL-CAS-L and use the update that we have as a premise again to obtain $c \mapsto_{\mathsf{i}} m$ for some m, as needed for REL-CAS-L. If $m \neq n$, then the compare-and-set operation has failed, and we can restart the proof first by restoring the mask on the refinement judgment (using the closing update), and then using the Löb induction hypothesis. If $m = n$, then the compare-and-set operation has succeeded, and the points-to connective is updated to $c \mapsto_{\mathsf{i}} (n+1)$. Then we can use the second conjunct $c \mapsto_{\mathsf{i}} (n+1) * R \mathrel{-\!\!*}\ \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau$ to arrive at the exact conclusion that we need: $\models_{\mathcal{E}} K[\,n\,] \precsim e : \tau$.

5.4. **HOCAP-style relational specifications.** We now present another form of logically atomic relational specifications—*HOCAP-style logical atomic relational specifications*, which are based on the logically atomic specifications by Svendsen *et al.* [SBP13] in the eponymous logic. Contrary to TaDA-style specifications, which come in a precisely specified format (Section 5.3.3), HOCAP-style specifications do not have a precise format. This provides the flexibility that not only can they be used to represent linearization points, but they can

Rules for abstract predicates:

$$\textsc{cnt-agree}\ \frac{\mathsf{cntAuth}_\gamma(n) \qquad \mathsf{cnt}_\gamma(q,m)}{n = m} \qquad \textsc{cnt-agree'}\ \frac{\mathsf{cnt}_\gamma(q_1,n) \qquad \mathsf{cnt}_\gamma(q_2,m)}{n = m} \qquad \textsc{cnt-combine}\ \frac{\mathsf{cnt}_\gamma(q_1,n) * \mathsf{cnt}_\gamma(q_2,n)}{\mathsf{cnt}_\gamma(q_1+q_2,n)}$$

$$\textsc{cnt-update}\ \frac{\mathsf{cntAuth}_\gamma(n) \qquad \mathsf{cnt}_\gamma(1,m)}{\Rightarrow\!\!\!\!\Rightarrow \mathsf{cntAuth}_\gamma(k) * \mathsf{cnt}_\gamma(1,k)} \qquad \textsc{Cnt-persistent}\ \frac{\mathsf{isCnt}_\gamma(c,\mathcal{N})}{\Box\, \mathsf{isCnt}_\gamma(c,\mathcal{N})}$$

Relational specification:

$$\textsc{new-i-l-hocap}\ \frac{\forall c\, \gamma.\, \mathsf{isCnt}_\gamma(c,\mathcal{N}) * \mathsf{cnt}_\gamma(1,n) \mathrel{-\!\!*} K[\,c\,] \precsim e_2 : \tau}{K[\,\mathsf{ref}(n)\,] \precsim e_2 : \tau}$$

$$\textsc{inc-i-l-hocap}\ \frac{\mathcal{E} \cap \mathcal{N}^\uparrow = \emptyset \qquad \mathsf{isCnt}_\gamma(c,\mathcal{N}) \qquad (\forall n.\, \mathsf{cntAuth}_\gamma(n)\ {}^{\top\setminus\mathcal{N}}\!\!\Rightarrow\!\!\!\!\!\!*\,{}^{\top\setminus\mathcal{N}\setminus\mathcal{E}}\ \mathsf{cntAuth}_\gamma(n+1) * \models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau)}{K[\,\mathsf{inc}_i\ c\,] \precsim e_2 : \tau}$$

$$\textsc{read-i-l-hocap}\ \frac{\mathcal{E} \cap \mathcal{N}^\uparrow = \emptyset \qquad \mathsf{isCnt}_\gamma(c,\mathcal{N}) \qquad (\forall n.\, \mathsf{cntAuth}_\gamma(n)\ {}^{\top\setminus\mathcal{N}}\!\!\Rightarrow\!\!\!\!\!\!*\,{}^{\top\setminus\mathcal{N}\setminus\mathcal{E}}\ \mathsf{cntAuth}_\gamma(n) * \models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau)}{K[\,\mathsf{read}\ c\,] \precsim e_2 : \tau}$$

FIGURE 8. HOCAP-style logically atomic relational specification for a fine-grained concurrent counter.

also be used to represent arbitrary observable interactions with the abstract state. This flexibility allows us to give strong specifications to non-linearizable methods (Section 5.4.4), which we use in the ticket lock refinement proof (Section 5.5).

5.4.1. *Formulating HOCAP-style specifications.* Let us consider the HOCAP-style specification in Figure 8 for the fine-grained concurrent counter from Figure 1 in Section 1. Contrary to the TaDA-style specification, the HOCAP-style specification does not expose the underlying state of the counter (*i.e.,* $\ell \mapsto_{\mathsf{i}} n$) directly, but instead provides an abstract view of the state through abstract predicates.

The persistent predicate $\mathsf{isCnt}_\gamma(c,\mathcal{N})$ asserts that the value c represents a counter. The specification is parameterized by a namespace $\mathcal{N}$ for the internal invariants associated with the specification. The ghost name γ is used to link c with the predicates $\mathsf{cnt}_\gamma(q,n)$ and $\mathsf{cntAuth}_\gamma(n)$, which describe the abstract state of the counter. The predicate $\mathsf{cnt}_\gamma(q,n)$ provides the *client view* of the abstract state of the counter. It is similar to the fractional heap points-to connective from separation logic—it associates a value (a natural number n) to the counter, and can be split and combined according to the fractional component q (CNT-AGREE', CNT-COMBINE). The predicate $\mathsf{cntAuth}_\gamma(m)$ provides the *module view* of the

abstract state of the counter. It agrees with the client view (CNT-AGREE) and can be used together with $\mathsf{cnt}_\gamma(1, n)$ to update the value associated to the counter (CNT-UPDATE).

Ownership of the module view predicate $\mathsf{cntAuth}_\gamma(m)$ is given to the user only during the execution of the counter operations. Consider, for example, the specification INC-I-L-HOCAP for the atomic increment function inc_i. From the client's point of view, there is only one place where inc_i observably interacts with the abstract state of the counter, namely during its linearization point. For this point of access, the user has to provide the update:

$$\mathsf{cntAuth}_\gamma(n) \ {}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\!\!\ast^{\top\setminus\mathcal{N}\setminus\mathcal{E}} \ \mathsf{cntAuth}_\gamma(n+1) * \ \models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau.$$

The user is given the module view $\mathsf{cntAuth}_\gamma(n)$ of the counter, and has to update it to $\mathsf{cntAuth}_\gamma(n+1)$. For that, the user has to appeal to CNT-UPDATE and has to provide $\mathsf{cnt}_\gamma(q, n)$ themselves (either as an immediate resource or from an invariant in $\mathcal{E}$, which can be opened thanks to the update modality). After the abstract state is updated, the user has to prove the refinement judgment $\models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau$. Similar to the TaDA-style specifications, the mask on the refinement is set to $\top \setminus \mathcal{E}$, allowing the user to perform some reasoning on the right-hand side before closing all the invariants from $\mathcal{E}$.

5.4.2. *Using HOCAP-style specifications.* We use the HOCAP-style logically atomic relational specification from Figure 8 to prove the refinement that we have seen in Section 3.4.2:

$$\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Since the HOCAP-style specifications are stated in terms of abstract predicates, instead of the $\ell \mapsto_{\mathsf{i}} n$ connective, we need a slightly different invariant than the one we used for the proof using the TaDA-style specification in Section 5.3.2, namely:

$$\mathsf{isCnt}_\gamma(c_i, \mathcal{N}.\mathsf{cnt}) * \boxed{\exists n \in \mathbb{N}.\, \mathsf{cnt}_\gamma(1, n) * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})}^{\mathcal{N}.\mathsf{inv}}.$$

After having established this invariant, the refinement proofs for the increment and read methods proceed similar to the corresponding TaDA-style proofs (Section 5.3.2), except that in the increment case the user has to use CNT-UPDATE alongside INC-I-L-HOCAP in order to update the ghost state $\mathsf{cnt}_\gamma(1, n)$ accordingly. We do not give the proof here, and direct the reader to the accompanying Coq mechanization.

In Section 5.5 we will another example of a client using the HOCAP-style specification for the counter.

5.4.3. *Proving HOCAP-style specifications.* We discuss how to prove that the implementation of the fine-grained counter meets the HOCAP-style specifications. To do so, we first use Iris's ghost theory to define the predicates $\mathsf{cnt}_\gamma(q, n)$ and $\mathsf{cntAuth}_\gamma(n)$ (the details of this definition are omitted). We then define the predicate $\mathsf{isCnt}_\gamma(c, \mathcal{N})$, which provides the internal invariant of the module:

$$\mathsf{isCnt}_\gamma(c, \mathcal{N}) \triangleq c \in \mathit{Loc} * \boxed{\exists n \in \mathbb{N}.\, c \mapsto_{\mathsf{i}} n * \mathsf{cntAuth}_\gamma(n)}^{\mathcal{N}}$$

This invariant states that the physical value n of c corresponds to the logical value n of the predicate $\mathsf{cntAuth}_\gamma(n)$. To see how this invariant is used, let us consider the proof of INC-I-L-HOCAP for the inc_i operation. Since inc_i is defined recursively, we prove this rule by Löb induction. We proceed by symbolically executing the left-hand side, accessing the invariant $\mathcal{N}$ to dereference c for some value n. It then remains to show:

$$\mathsf{if}\ \mathsf{CAS}(c, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i \ c \precsim e_2 : \tau.$$

At this point we use the atomic symbolic execution rule for compare-and-set REL-CAS-L (with $\mathcal{E} = \mathcal{N}$). We introduce the update modality ${}^{\top}\!\Rrightarrow^{\top\setminus\mathcal{N}}$ we obtain by accessing the invariant $\mathcal{N}$ using Iris's strong invariant access rule INV-ACCESS-STRONG, which is needed because we need to access invariants in a non-well-bracketed way:

$$\frac{\text{INV-ACCESS-STRONG}\qquad \mathcal{N}^{\uparrow} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}}\ \mathcal{E} \Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\forall \mathcal{E}'.\ \triangleright P\ \mathcal{E}' \Rrightarrow^{\mathcal{E}'\cup\mathcal{N}} \mathsf{True})}$$

It remains to consider two cases. If the compare-and-set (CAS) has failed, we close the invariant (by setting $\mathcal{E}' = \top \setminus \mathcal{N}$) and appeal to the induction hypothesis. If the compare-and-set has succeeded, we are left to show the following:

$$\cdots * c \mapsto_{\mathsf{i}} (n+1) * \mathsf{cntAuth}_\gamma(n) \mathrel{-\!\!*}\ \models_{\top\setminus\mathcal{N}} n \precsim e_2 : \tau.$$

We first use the update ${}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top\setminus\mathcal{N}\setminus\mathcal{E}}$ that is provided by the premise of the rule to update $\mathsf{cntAuth}_\gamma(n)$ into $\mathsf{cntAuth}_\gamma(n+1)$. This moreover provides a proof of $\models_{\top\setminus\mathcal{E}} n \precsim e_2 : \tau$. At this point our goal becomes $\models_{\top\setminus\mathcal{N}\setminus\mathcal{E}} n \precsim e_2 : \tau$. We close the invariant $\mathcal{N}$ (by setting $\mathcal{E}' = \top \setminus \mathcal{N} \setminus \mathcal{E}$) and restore the mask on the refinement proposition in our goal, resulting in $\models_{\top\setminus\mathcal{E}} n \precsim e_2 : \tau$, which is exactly what we obtained from the update ${}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top\setminus\mathcal{N}\setminus\mathcal{E}}$.

5.4.4. *HOCAP-style specifications for non-linearizable operations.* Using HOCAP-style specifications we can also specify operations that are not linearizable. Consider the following "weak increment" function that we can add to the counter module:

$$\mathsf{wkincr} \triangleq \lambda c.\ c \leftarrow (!\, c + 1)$$

The function increments the value in the location c non-atomically. What kind of specification can we give to wkincr? To answer this we have to examine what the update $\Rrightarrow$ represents in the HOCAP-style specifications. In the previous examples with linearizable functions, the updates represented observations about the abstract state that the clients could make, and they corresponded to the linearization points. But there is no reason why we should pin them to linearization points only. Rather, we can let the update correspond to any operation that is observable through the abstract state. In wkincr there are two points where such operations happen, which we can represent through two nested updates:

$$\frac{\begin{array}{l}\text{CNT-WK-INCR-L}\qquad \mathcal{E} \cap \mathcal{N}^{\uparrow} = \emptyset \qquad \mathsf{isCnt}_\gamma(c, \mathcal{N}) \\ \forall n.\ \mathsf{cntAuth}_\gamma(n) \Rrightarrow^{\top\setminus\mathcal{N}} \mathsf{cntAuth}_\gamma(n) * (\forall m.\ \mathsf{cntAuth}_\gamma(m)\ {}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top\setminus\mathcal{N}\setminus\mathcal{E}} \\ \qquad\qquad \mathsf{cntAuth}_\gamma(n+1) * \models_{\top\setminus\mathcal{E}} K[\,()\,] \precsim e_2 : \tau)\end{array}}{K[\,\mathsf{wkincr}\ c\,] \precsim e_2 : \tau}$$

The first update binds the value n, which is obtained from the initial read operation $!\, c$. In the conclusion of this update the client needs to return the $\mathsf{cntAuth}_\gamma(n)$ predicate, as in the specification READ-I-L-HOCAP. In addition to that predicate, the client has to provide the second update, in which $\mathsf{cntAuth}_\gamma(m)$ has to be updated to $\mathsf{cntAuth}_\gamma(n+1)$, corresponding to the assignment $c \leftarrow n+1$. The value m corresponds to the intermediate state of the counter, which might have changed in between the dereferencing of c and the assignment to it. The presence of two updates differentiates methods that have a linearization point, such as inc_i, and non-linearizable methods, such as wkincr.

In the next section we will see how a client might use the specification CNT-WK-INCR-L.

$$\begin{aligned}
\mathsf{newlock}_{\mathsf{TL}} &\triangleq \lambda().\,(\mathsf{ref}(0), \mathsf{ref}(0))\\
\mathsf{acquire}_{\mathsf{TL}} &\triangleq \lambda(l_{\mathsf{o}}, l_{\mathsf{n}}).\,\mathsf{let}\ n = \mathsf{inc}_i\ l_{\mathsf{n}}\ \mathsf{in}\ \mathsf{wait_loop}\ n\ l_{\mathsf{o}}\\
\mathsf{wait_loop} &\triangleq \lambda n\ l_{\mathsf{o}}.\,\mathsf{if}\ (n = \mathsf{read}\ l_{\mathsf{o}})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{wait_loop}\ n\ l_{\mathsf{o}}\\
\mathsf{release}_{\mathsf{TL}} &\triangleq \lambda(l_{\mathsf{o}}, l_{\mathsf{n}}).\,\mathsf{wkincr}\ l_{\mathsf{o}}
\end{aligned}$$

FIGURE 9. Ticket lock implementation.

5.5. **Ticket lock refinement from HOCAP-style specs.** We show how our HOCAP-style relational specifications for the fine-grained concurrent counter (Figure 8) is used to prove that a *ticket lock* refines a *spin lock* (or, rather, that a ticket lock refines any lock satisfying the specification in Figure 5). The proof in this section demonstrates several important features of ReLoC. First, it demonstrates compositionality of proofs in ReLoC both by employing relational specifications for the left- and right-hand sides, and by reducing the refinement proof of a program module into separate reusable refinement proofs of the module functions. Second, the proof highlights the integration of Iris ghost state to facilitate CAP-style [DDG+10] reasoning with abstract predicates.

A ticket lock [MS91, Section 2.2] is a ticket-based data structure for mutual exclusion, which is fair—threads racing to enter a critical section will gain access to it in the order of arrival at the critical section. The implementation of the ticket lock is given in Figure 9. The two locations associated with the lock, l_{o} and l_{n}, point to the identifiers of the current owner of the lock, and to the total number of issued tickets, respectively. When a thread wants to enter a critical section using the $\mathsf{acquire}_{\mathsf{TL}}$ function, it first requests a new ticket (by atomically increasing the value of l_{n} using the inc_i function), and then spins until the value of the current owner of the lock matches the ticket number (using $\mathsf{wait\_loop}$).

The function $\mathsf{release}_{\mathsf{TL}}$ uses the weak increment wkincr (Section 5.4.4) on the location l_{o}. It does not need to use an atomic increment (*i.e.,* inc_i), because, if the lock is used in a well-bracketed manner, only the owner of the lock will be calling the $\mathsf{release}_{\mathsf{TL}}$ function.

Concretely, the refinement that we wish to show is the following:

$$\begin{aligned}
&\mathsf{pack}(\mathsf{newlock}_{\mathsf{TL}}, \mathsf{acquire}_{\mathsf{TL}}, \mathsf{release}_{\mathsf{TL}}) \precsim\\
&\mathsf{pack}(\mathsf{newlock}, \mathsf{acquire}, \mathsf{release}) : \exists\alpha.\,(\mathsf{unit} \to \alpha) \times (\alpha \to \mathsf{unit}) \times (\alpha \to \mathsf{unit}).
\end{aligned}$$

Here, $\mathsf{newlock}$, $\mathsf{acquire}$ and $\mathsf{release}$ are any operations that satisfy the relational specification from Figure 5 (for example, the spin lock from Section 5.1.1).

Proof outline. To prove the refinement above we employ our general strategy for proving refinements for stateful program modules in ReLoC:

(1) We define an invariant $\mathsf{lockInv}$ linking together the underlying representations of each individual pair of locks, which we use to define a witness for the existential type α.
(2) We prove the refinements for each method in the signature.
(3) Finally, we combine those proofs together into a module refinement proof. This is what we refer to as a *component-wise* refinement proof.

We stress that to carry out the proof we neither need to refer to the implementation of the fine-grained concurrent counter (on the left-hand side), nor to the implementation of the

$$\textsc{newIssuedTickets}\quad \Rightarrow\!\!\!\!\mid \exists\gamma.\, \mathsf{issuedTickets}_\gamma(0)$$

$$\textsc{issueNewTicket}\quad \frac{\mathsf{issuedTickets}_\gamma(m)}{\Rightarrow\!\!\!\!\mid \mathsf{issuedTickets}_\gamma(m+1) * \mathsf{ticket}_\gamma(m)}$$

$$\textsc{ticket-nondup}\quad \frac{\mathsf{ticket}_\gamma(n) \qquad \mathsf{ticket}_\gamma(n)}{\mathsf{False}}$$

FIGURE 10. The ticket ghost theory.

spin lock (on the right-hand side). Rather, we only refer to the HOCAP-style specification for the fine-grained counter and the relational specification for the spin lock.

Proof of the refinement. To match up the physical representation of tickets in the lock we use Iris's ghost theory to define abstract predicates tracking the tickets. We will use two ghost predicates: $\mathsf{issuedTickets}_\gamma(m)$ saying that m tickets have been issued in total, and $\mathsf{ticket}_\gamma(n)$ representing the n-th ticket. The predicates satisfy the rules in Figure 10.

To prove the refinement of lock modules, we need to pick a relation (serving as the interpretation for the witness α of the existential type) that links the two modules together. We use the the relation $\mathsf{lockInt}$ defined as follows:

$$\mathsf{lockInv}_\gamma(\gamma_\mathsf{o}, \gamma_\mathsf{n}, lk) \triangleq \exists (o\, n : \mathbb{N})\, (b : \mathbb{B}).\, \mathsf{cnt}_{\gamma_\mathsf{o}}(1, o) * \mathsf{cnt}_{\gamma_\mathsf{n}}(1, n) * \mathsf{isLock}_\mathsf{s}(lk, b) * \mathsf{issuedTickets}_\gamma(n) * (\mathsf{ticket}_\gamma(o) \vee b = \mathbf{false})$$

$$\mathsf{lockInt}((l_\mathsf{o}, l_\mathsf{n}), lk) \triangleq \exists \gamma_\mathsf{o}, \gamma_\mathsf{n}, \gamma.\, \mathsf{isCnt}_{\gamma_\mathsf{o}}(l_\mathsf{o}, \mathcal{N}.\mathsf{o}) * \mathsf{isCnt}_{\gamma_\mathsf{n}}(l_\mathsf{n}, \mathcal{N}.\mathsf{n}) * \boxed{\mathsf{lockInv}_\gamma(\gamma_\mathsf{o}, \gamma_\mathsf{n}, lk)}^{\mathcal{N}.\mathsf{inv}}$$

Here, $(l_\mathsf{o}, l_\mathsf{n})$ is the ticket lock on the left-hand side, and lk is the specification lock on the right-hand side. The $\mathsf{lockInt}$ relation states that l_o and l_n are concurrent counters with ghost names γ_o and γ_n, that satisfy the invariant $\mathsf{lockInv}$. This invariant describes the relation between the values representing two locks. It states that the values of the counters l_o and l_n are o and n, respectively, and that exactly n tickets have been issued. Furthermore, the right-hand side lock lk is locked iff the ticket $\mathsf{ticket}_\gamma(o)$ of the current owner of the lock is in the invariant; that is, $\mathsf{ticket}_\gamma(o)$ was given up by a thread that acquired the lock.

Using REL-PACK we subdivide the main refinement proof into three refinements for the functions that constitute the lock module:

(1) $[\alpha := \mathsf{lockInt}] \models \mathsf{newlock}_\mathsf{TL} \precsim \mathsf{newlock} : \mathsf{unit} \to \alpha$;
(2) $[\alpha := \mathsf{lockInt}] \models \mathsf{acquire}_\mathsf{TL} \precsim \mathsf{acquire} : \alpha \to \mathsf{unit}$;
(3) $[\alpha := \mathsf{lockInt}] \models \mathsf{release}_\mathsf{TL} \precsim \mathsf{release} : \alpha \to \mathsf{unit}$.

Proposition 5.1. $[\alpha := \mathsf{lockInt}] \models \mathsf{newlock}_\mathsf{TL} \precsim \mathsf{newlock} : \mathsf{unit} \to \alpha$.

Proof. By rule REL-REC it suffices to show $[\alpha := \mathsf{lockInt}] \models \mathsf{newlock}_\mathsf{TL}\ () \precsim \mathsf{newlock}\ () : \alpha$. By applying the symbolic execution rules and NEW-I-L-HOCAP we are left with the goal:

$$\mathsf{isCnt}_{\gamma_\mathsf{n}}(l_\mathsf{n}, \mathcal{N}.\mathsf{n}) * \mathsf{isCnt}_{\gamma_\mathsf{o}}(l_\mathsf{o}, \mathcal{N}.\mathsf{o}) * \mathsf{cnt}_{\gamma_\mathsf{o}}(1, 0) * \mathsf{cnt}_{\gamma_\mathsf{n}}(1, 0) * \mathsf{isLock}_\mathsf{s}(lk, \mathbf{false}) \mathrel{-\!\!*} [\alpha := \mathsf{lockInt}] \models (l_\mathsf{o}, l_\mathsf{n}) \precsim lk : \alpha.$$

From the premises we can allocate the invariant $\mathsf{lockInv}_\gamma(\gamma_\mathsf{o}, \gamma_\mathsf{n}, lk)$, and obtain $\mathsf{lockInt}((l_\mathsf{o}, l_\mathsf{n}), lk)$. We finish the proof with REL-RETURN. □

To prove the $\mathsf{acquire}_{\mathsf{TL}}$ refinement we need the following helper lemma.

Lemma 5.2. $\mathsf{ticket}_\gamma(m) \vdash [\alpha := \mathsf{lockInt}] \models \mathsf{wait\_loop}\ m\ l_\mathsf{o} \precsim \mathsf{acquire}\ lk : \mathsf{unit}$, *provided we have* $\boxed{\mathsf{lockInv}_\gamma(\gamma_\mathsf{o}, \gamma_\mathsf{n}, lk)}^{\mathcal{N}.\mathsf{inv}}$.

Proof. We prove the refinement by Löb induction and symbolic execution. After some pure symbolic executions steps we are left with the goal:

$$[\alpha := \mathsf{lockInt}] \models \mathsf{if}\ (m = \mathsf{read}\ l_\mathsf{o})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{wait\_loop}\ m\ l_\mathsf{o} \precsim \mathsf{acquire}\ lk : \mathsf{unit}.$$

We then apply READ-I-L-HOCAP, after which it remains to prove

$$\mathsf{cntAuth}_{\gamma_\mathsf{o}}(o) \ {}^{\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}'\setminus\mathcal{N}} \mathsf{cntAuth}_{\gamma_\mathsf{o}}(o) * [\alpha := \mathsf{lockInt}] \models \mathsf{if}\ (m = o)\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{wait\_loop}\ m\ l_\mathsf{o} \precsim \mathsf{acquire}\ lk : \mathsf{unit}.$$

for any number o. In case $m \neq o$, we symbolically execute the left-hand side and appeal to the induction hypothesis. In case $m = o$, we proceed by accessing the invariant $\boxed{\mathsf{lockInv}_\gamma(\gamma_\mathsf{o}, \gamma_\mathsf{n}, lk)}^{\mathcal{N}.\mathsf{inv}}$. Note that it cannot be the case that $b = \mathbf{true}$, because then we would have two copies of $\mathsf{ticket}_\gamma(o)$: one from the assumption of the lemma and one from the invariant. Thus, the case $b = \mathbf{true}$ can be eliminated by TICKET-NONDUP. Then it must be the case that $b = \mathbf{false}$. In that case we have $\mathsf{isLock}_\mathsf{s}(lk, \mathbf{false})$ and we can apply ACQUIRE-R to update it to $\mathsf{isLock}_\mathsf{s}(lk, \mathbf{true})$, changing the right-hand side to $()$.

We finish by closing the invariant, picking this time $b = \mathbf{true}$ and storing the $\mathsf{ticket}_\gamma(o)$ from the assumption of the lemma in the invariant. □

Proposition 5.3. $[\alpha := \mathsf{lockInt}] \models \mathsf{acquire}_{\mathsf{TL}} \precsim \mathsf{acquire} : \alpha \to \mathsf{unit}$.

Proof. We use REL-REC and symbolic execution rules, and then INC-I-L-HOCAP and Lemma 5.2. When we apply INC-I-L-HOCAP, we use the update to issue a new ticket using ISSUENEWTICKET. This ticket will be used for the assumption of Lemma 5.2. □

Proposition 5.4. $[\alpha := \mathsf{lockInt}] \models \mathsf{release}_{\mathsf{TL}} \precsim \mathsf{release} : \alpha \to \mathsf{unit}$.

Proof. We use REL-REC, and then symbolic execution and CNT-WK-INCR-L, after which the new goal becomes

$$\mathsf{cntAuth}_{\gamma_\mathsf{o}}(n) \Rrightarrow_{\mathcal{E}'} \mathsf{cntAuth}_{\gamma_\mathsf{o}}(n) * (\forall m.\, \mathsf{cntAuth}_{\gamma_\mathsf{o}}(m) \ {}^{\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}'\setminus\mathcal{N}} \mathsf{cntAuth}_{\gamma_\mathsf{o}}(n+1) * \models_{\top\setminus\mathcal{N}} () \precsim \mathsf{release}\ lk : \tau)$$

for an arbitrary n. By framing $\mathsf{cntAuth}_{\gamma_\mathsf{o}}(n)$, it suffices to show

$$\mathsf{cntAuth}_{\gamma_\mathsf{o}}(m) \ {}^{\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}'\setminus\mathcal{N}} \mathsf{cntAuth}_{\gamma_\mathsf{o}}(n+1) * \models_{\top\setminus\mathcal{N}} () \precsim \mathsf{release}\ lk : \tau$$

for an arbitrary m. We utilize this update by accessing the invariant and getting access to $\mathsf{cnt}_{\gamma_\mathsf{o}}(1, o)$. Using this proposition and $\mathsf{cntAuth}_{\gamma_\mathsf{o}}(m)$ we apply CNT-UPDATE to get

$$\mathsf{cnt}_{\gamma_\mathsf{o}}(1, n+1) * \mathsf{cntAuth}_{\gamma_\mathsf{o}}(n+1).$$

We frame the second separating conjunct, and use RELEASE-R to reduce the right-hand side to $()$. Finally we close the invariant and finish the proof with REL-RETURN. □

6. Speculative reasoning using prophecy variables

In addition to Iris's ordinary ghost state mechanism, which allows to reason about the history of a program, Iris has recently been extended with a mechanism for speculative reasoning based on *prophecy variables*, which allows to reason about the future of a program [AL91, JLP+20]. A prophecy variable is a ghost variable that can reference a value that is determined in the future of a program's execution. While the program that is subject to verification cannot make use of the value of a prophecy variable itself—they are ghost variables—the value can be used in proofs, *e.g.,* to speculatively choose a reduction step in the right-hand program. In this section we show how Iris's mechanism for prophecy variables is integrated into ReLoC and can be put to action to prove challenging refinements.

We start this section by illustrating the need for prophecy variables with a motivational example (Section 6.1). We then introduce the proof rules for prophecy variables in ReLoC (Section 6.2), and use them to verify the motivational example (Section 6.3). We finish with another example demonstrating the applicability of prophecy variables to algebraic reasoning for concurrent programs (Section 6.4).

6.1. Motivational example.

The ghost state mechanism of Iris that we have seen so far has allowed us to reason about the history of the program's execution. However, in some cases that is not enough, and it is required to take the future of the program's execution into account. As a simple motivating example let, us consider the implementations $\mathsf{new\_coin}$ and $\mathsf{new\_coin\_lazy}$ of a coin module in Figure 11. They both implement a virtual coin that can be flipped (using the first closure) and whose value can be read (using the second closure), but there is an important difference. The eager version ($\mathsf{new\_coin}$) calculates the flipped value in the flip function immediately—using the rand function in Figure 12, which gives a non-deterministic Boolean value—and the read function just reads that value. In contrast, the lazy version ($\mathsf{new\_coin\_lazy}$) does not perform any non-deterministic calculations in its flip operation $\mathsf{flip\_lazy}$. Instead, it sets the value of the coin to "undetermined" (*i.e.,* $\mathbf{None}$), and postpones the actual calculation to the $\mathsf{read\_lazy}$ function.

While these two implementations are rather different, they are contextually equivalent—for clients of the module it is not observable if the coin is flipped eagerly or lazily. To prove that, we wish to establish the following refinements in ReLoC:

$$\mathsf{new\_coin} \precsim \mathsf{new\_coin\_lazy} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$$

$$\mathsf{new\_coin\_lazy} \precsim \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$$

The first refinement can be proved with the tools that we have already described. We start by symbolically executing both implementations, obtaining references c and c_l to the internal state of the eager coin and lazy coin, respectively. We then establish the following invariant linking together the two internal states:

$$\boxed{\exists(b : \mathbb{B}).\, c \mapsto_{\mathsf{i}} b * (c_l \mapsto_{\mathsf{s}} \mathbf{None} \vee c_l \mapsto_{\mathsf{s}} \mathbf{Some}(b))}.$$

This invariant can be easily shown to be preserved during the $\mathsf{flip} \precsim \mathsf{flip\_lazy}$ refinement proof. During the $\mathsf{read} \precsim \mathsf{read\_lazy}$ refinement proof we can choose which value $\mathsf{rand}\ ()$ reduces to based on the current value of c, using REL-RAND-R.

However, we cannot prove the second refinement with the same strategy. The problem is that during the $\mathsf{flip\_lazy} \precsim \mathsf{flip}$ refinement we reset the value of c_l (on the left-hand side) to $\mathbf{None}$, and then we have to establish a simulation on the right-hand side by picking a

The eager and lazy implementation:

$$\begin{array}{ll}
\mathsf{new_coin} \triangleq \mathsf{let}\ c = \mathsf{ref}(\mathbf{false})\ \mathsf{in} & \mathsf{new_coin_lazy} \triangleq \mathsf{let}\ c = \mathsf{ref}(\mathbf{Some}(\mathbf{false}))\ \mathsf{in} \\
\quad ((\lambda().\,\mathsf{flip}\ c), (\lambda().\,\mathsf{read}\ c)) & \quad ((\lambda().\,\mathsf{flip_lazy}\ c), (\lambda().\,\mathsf{read_lazy}\ c)) \\
\mathsf{flip} \triangleq \lambda c.\, c \leftarrow \mathsf{rand}\ () & \mathsf{flip_lazy} \triangleq \lambda c.\, c \leftarrow \mathbf{None} \\
\mathsf{read} \triangleq \lambda c.\, !\,c & \mathsf{read_lazy} \triangleq \lambda c.\, \mathsf{match}\ !\,c\ \mathsf{with} \\
 & \quad \mid \mathbf{Some}(v) \rightarrow v \\
 & \quad \mid \mathbf{None} \rightarrow \\
 & \quad\quad \mathsf{let}\ x = \mathsf{rand}\ ()\ \mathsf{in} \\
 & \quad\quad \mathsf{if}\ \mathsf{CAS}(c, \mathbf{None}, x) \\
 & \quad\quad \mathsf{then}\ x\ \mathsf{else}\ \mathsf{read_lazy}\ c\ ()
\end{array}$$

The instrumented lazy implementation with prophecy variables:

$$\begin{array}{l}
\widehat{\mathsf{new_coin_lazy}} \triangleq \mathsf{let}\ c = \mathsf{ref}(\mathbf{Some}(\mathbf{false}))\ \mathsf{in} \\
\quad \mathsf{let}\ p = \mathsf{newproph}\ \mathsf{in} \\
\quad \mathsf{let}\ lk = \mathsf{newlock}\ ()\ \mathsf{in} \\
\quad ((\lambda().\,\widehat{\mathsf{flip_lazy}}\ c\ lk), (\lambda().\,\widehat{\mathsf{read_lazy}}\ c\ lk\ p)) \\
\widehat{\mathsf{flip_lazy}} \triangleq \lambda c\ lk.\,\mathsf{acquire}\ lk;\ c \leftarrow \mathbf{None};\ \mathsf{release}\ lk \\
\widehat{\mathsf{read_lazy}} \triangleq \lambda c\ lk\ p.\,\mathsf{acquire}\ lk; \\
\quad\quad \mathsf{let}\ r = \mathsf{match}\ !\,c\ \mathsf{with} \\
\quad\quad\quad \mid \mathbf{Some}(v) \rightarrow v \\
\quad\quad\quad \mid \mathbf{None} \rightarrow \\
\quad\quad\quad\quad \mathsf{let}\ x = \mathsf{rand}\ ()\ \mathsf{in} \\
\quad\quad\quad\quad c \leftarrow \mathbf{Some}(x);\ \mathsf{resolve}\ p\ \mathsf{to}\ x;\ x \\
\quad\quad \mathsf{in}\ \mathsf{release}\ lk;\ r
\end{array}$$

FIGURE 11. The implementations of the coin module.

$$\mathsf{rand} \triangleq \lambda().\,\mathsf{let}\ y = \mathsf{ref}(\mathbf{false})\ \mathsf{in}\ \mathsf{fork}\ \{y \leftarrow \mathbf{true}\};\ !\,y$$

$$\textsc{rel-rand-l}\ \frac{\forall b \in \mathbb{B}.\ (K[\,b\,] \precsim t : \tau)}{K[\,\mathsf{rand}\ ()\,] \precsim t : \tau} \qquad \textsc{rel-rand-r}\ \frac{b \in \mathbb{B} \qquad e \precsim K[\,b\,] : \tau}{e \precsim K[\,\mathsf{rand}\ ()\,] : \tau}$$

FIGURE 12. The implementation and relational specification of the rand function.

value for rand () that will be assigned to c. But this value has to be the same value that is picked by rand () in read_lazy. Thus we have to pick a value "from the future".

To facilitate this style of reasoning, prophecy variables have been introduced into Iris [JLP+20]. Originally, prophecy variables were used to prove refinements between state machines [AL91]. Lately they have been used in Iris for establishing linearizability of concurrent data structures without a fixed linearization point. In the rest of this section we show how we integrated prophecy variables into ReLoC.

$$\textsc{rel-newproph-l}\ \frac{\forall \vec{v}\ p.\ \mathsf{proph}(p, \vec{v}) \mathrel{-\!\!*} \Delta \models K[\,p\,] \precsim e_2 : \tau}{\Delta \models K[\,\mathsf{newproph}\,] \precsim e_2 : \tau}$$

$$\textsc{rel-resolveproph-l}\ \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \exists \vec{v}.\ \mathsf{proph}(p, \vec{v}) * \big(\forall \vec{w}.\ (\vec{v} = w :: \vec{w}) * \mathsf{proph}(p, \vec{w}) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau\big)}{\Delta \models K[\,\mathsf{resolve}\, p\, \mathsf{to}\ w\,] \precsim e_2 : \tau}$$

$$\textsc{rel-newproph-r}\ \frac{\forall p.\ (\Delta \models_{\mathcal{E}} e_1 \precsim K[\,p\,] : \tau)}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{newproph}\,] : \tau} \qquad \textsc{rel-resolveproph-r}\ \frac{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{resolve}\, p\, \mathsf{to}\ w\,] : \tau}$$

FIGURE 13. The ReLoC proof rules for prophecy variables.

6.2. **Prophecy instructions and proof rules.** While Iris's ordinary ghost state mechanism only appears at the level of the logic, prophecy variables appear as instrumented instructions in the source program.[4] The instruction newproph creates a new prophecy variable. The instruction $\mathsf{resolve}\, p\, \mathsf{to}\ v$ resolves a prophecy variable p to a value v.

The symbolic execution rules for the prophecy instructions are given in Figure 13. In the right-hand side, the prophecy instructions are no-ops and therefore do not have any pre- or post-conditions. Prophecy instructions that appear on the left-hand side, however, operate on additional ghost state, and thus have pre- and postconditions. The ghost predicate $\mathsf{proph}(p, \vec{v})$ says that the prophecy variable p will be resolved, in the future, with values from the vector $\vec{v}$. Initially, a prophecy variable created with newproph has an arbitrary vector $\vec{v}$ associated with it. Only after symbolically executing $\mathsf{resolve}\, p\, \mathsf{to}\ w$ we learn that this vector $\vec{v}$ contains w at the head position. The trick behind the prophecy variables ghost state is that we can already refer to the head element of $\vec{v}$ before resolving it to some w. We will see how to use this in establishing the refinement between the lazy coin and the eager coin in the next section. Note that the rules for the left-hand side are written in logically atomic style: compare, for example, REL-RESOLVEPROPH-L and REL-STORE-L.

To see how the instrumented instructions for prophecy variables are used, suppose we want to prove a contextual refinement $e_1 \precsim_{ctx} e_2 : \tau$ that involves speculative reasoning. We first prove a refinement $\hat{e}_1 \precsim e_2 : \tau$, where $\hat{e}_1$ is a version of e_1 instrumented with prophecy variables, and then prove $e_1 \precsim \hat{e}_1 : \tau$ to show that the prophecy variables can be erased. By soundness of ReLoC and transitivity of contextual refinement, this gives a contextual refinement $e_1 \precsim_{ctx} e_2 : \tau$ that refers only to the original programs.

6.3. **Proving the coin refinement.** To prove the refinement $\mathsf{new\_coin\_lazy} \precsim \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$ from Section 6.1 we instrument the lazy implementation new_coin_lazy with prophecy variables so we can speculate on the outcome of rand in read_lazy. The instrumented implementation $\widehat{\mathsf{new\_coin\_lazy}}$ is shown in Figure 12. In addition to prophecy variables, we also instrumented the implementation with locks to ensure that there is no interference between updating the reference c and resolving the prophecy variable p.

[4]The semantics of HeapLang has to be instrumented to support prophecy variables, we refer the reader to [JLP+20, Section 3] for details.

With the instrumented program at hand, we will prove the chain of refinements:

$$\mathsf{new\_coin\_lazy} \precsim \widehat{\mathsf{new\_coin\_lazy}} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$$

$$\widehat{\mathsf{new\_coin\_lazy}} \precsim \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool}).$$

Via ReLoC's soundness theorem, we can compose these refinements at the level of contextual refinement to obtain:

$$\mathsf{new\_coin\_lazy} \precsim_{ctx} \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool}).$$

Note that that we only use the instrumented implementation $\widehat{\mathsf{new\_coin\_lazy}}$ for the intermediate step, which means that prophecy variables and locks do not appear at all in the final statement above. The approach of using prophecies as an intermediate step works not just for closed programs, but also for open programs, as it does not rely on an erasure theorem [JLP+20, Section 3.5]. Moreover, as the example demonstrates, it allows us to make use of locks in the instrumented program.[5].

The first refinement ($\mathsf{new\_coin\_lazy} \precsim \widehat{\mathsf{new\_coin\_lazy}}$) is easy to prove, we simply use the no-op symbolic execution rules for prophecies on the right-hand side (Figure 13). The second refinement ($\widehat{\mathsf{new\_coin\_lazy}} \precsim \mathsf{new\_coin}$) is where the mechanism of prophecy variables comes to help. We symbolically execute the allocation parts of $\widehat{\mathsf{new\_coin\_lazy}}$ and $\mathsf{new\_coin}$. We then use the relational specification for locks (Section 5.1.2) with the following lock invariant:

$$\exists \vec{v}.\, \mathsf{proph}(p, \vec{v}) * \big((c_l \mapsto_{\mathsf{i}} \mathbf{None} * c \mapsto_{\mathsf{s}} (hd\ \vec{v})) \vee\ (\exists (b : \mathbb{B}).\, c_l \mapsto_{\mathsf{i}} \mathbf{Some}(b) * c \mapsto_{\mathsf{s}} b)\big).$$

This invariant says that if the value of the lazy coin is **None**, then the value of the eager coin is determined by the prophecy variable p. There are two main implications of this:

(1) In the refinement between $\widehat{\mathsf{flip\_lazy}}$ and flip, the invariant can be (re)established, because we can pick the value of $\mathsf{rand}\ ()$ on the right-hand side to be the head element of $\vec{v}$—the future value of the lazy coin is already bound at this point.
(2) In the refinement between $\widehat{\mathsf{read\_lazy}}$ and read (specifically, in the **None** branch), we obtain a non-deterministic Boolean x from symbolically executing $\mathsf{rand}\ ()$ on the left-hand side, and we update the value of c_l to be x. Moreover, we resolve the prophecy variable p to x, which gives us much desired information: the head element of $\vec{v}$ was x all along! This information allows us to transition from the left disjunct to the right disjunct in the invariant and complete the proof.

6.4. **Algebraic reasoning about non-deterministic choice.** In this section we give another example of the use of prophecy variables: we verify several algebraic properties of non-deterministic choice modulo contextual equivalence. (In)equational theories of the non-deterministic choice operator were previously considered in the context of domain theory, where non-determinism is usually modeled using power domains [Plo76, Smy76], and in the context of algebraic effects [SV20, JSV10]. Power domains and the denotational semantics approach does not seem to scale easily to languages with concurrency and higher-order store. An operational approach to equational theory of a programming language with non-determinism was considered in [BBS13] using step-indexed logical relations. There the authors show several contextual equivalences involving non-determinism, both finite

[5] *Atomic prophecy resolution* was introduced in [JLP+20] as an alternative to locks to deal with atomicity of prophecy resolution.

(*e.g.,* picking a Boolean) and countable (*e.g.,* picking a natural number). In this subsection, we provide conceptually simple proofs for contextual equivalences involving finite non-determinism only. However, we were also able to prove that non-deterministic choice and sequential composition distribute over each other. Proving this crucially relies on speculative reasoning which we formalize using prophecy variables.

We do not have a non-deterministic choice operation built-in the language, but we can define it using the rand function from Section 6.1. The operation or non-deterministically executes one of its thunked arguments:

$$\mathsf{or}\ t_1\ t_2 \triangleq \mathsf{if}\ \mathsf{rand}\ ()\ \mathsf{then}\ t_1\ ()\ \mathsf{else}\ t_2\ ()$$

We write $e_1 \oplus e_2$ for $\mathsf{or}\ (\lambda().\, e_1)\ (\lambda().\, e_2)$. The expression $e_1 \oplus e_2$ thus non-deterministically reduces to either e_1 or e_2. From the rules for the rand function (Figure 12), we derive the following symbolic execution rules for $\oplus$:

$$\textsc{rel-or-l}\ \frac{(\Delta \models K[\,e_1\,] \precsim t : \tau) \land (\Delta \models K[\,e_2\,] \precsim t : \tau)}{\Delta \models K[\,e_1 \oplus e_2\,] \precsim t : \tau}$$

$$\textsc{rel-or-r-1}\ \frac{\Delta \models_{\mathcal{E}} t \precsim K[\,e_1\,] : \tau}{\Delta \models_{\mathcal{E}} t \precsim K[\,e_1 \oplus e_2\,] : \tau} \qquad \textsc{rel-or-r-2}\ \frac{\Delta \models_{\mathcal{E}} t \precsim K[\,e_2\,] : \tau}{\Delta \models_{\mathcal{E}} t \precsim K[\,e_1 \oplus e_2\,] : \tau}$$

The rules for $\oplus$ are reminiscent of the rules for disjunction ($\lor$) in sequent calculus. To symbolically execute $\oplus$ on the left-hand side (*c.f.* to eliminate $\lor$) it is necessary to establish refinements for both operands (*c.f.* to consider both disjuncts), and to symbolically execute $\oplus$ on the right-hand side (*c.f.* to introduce $\lor$) it suffices to establish a refinement for one of the operands (*c.f.* prove one of the disjuncts).

Assume that e_1, e_2, e_3 are closed programs of type τ. Then using rel-or-r-1, rel-or-r-2, and rel-or-l, we prove the following equivalences:

$$e_1 \simeq_{ctx} e_1 \oplus e_1 : \tau \qquad e_1 \oplus e_2 \simeq_{ctx} e_2 \oplus e_1 : \tau \qquad e_1 \simeq_{ctx} e_1 \oplus \mathsf{diverge} : \tau$$

$$e_1 \oplus (e_2 \oplus e_3) \simeq_{ctx} (e_1 \oplus e_2) \oplus e_3 : \tau \qquad (e_1 \oplus e_2); e_3 \simeq_{ctx} (e_1; e_3) \oplus (e_2; e_3) : \tau$$

The equational theory that we obtain here is similar to the one obtained from the Hoare power domain, as $e_1 \oplus \mathsf{diverge}$ (where diverge is an infinite loop) is identified with e_1. The last equation states that non-deterministic choice distributes over sequential composition, and is standard in, *e.g.,* process calculi. What is less standard is the following equation, which is not validated by models based on bisimulation:

$$e_1; (e_2 \oplus e_3) \simeq_{ctx} (e_1; e_2) \oplus (e_1; e_3) : \tau.$$

This equation, however, holds in Kleene algebra-like models [HMSW11, Koz94]. If we think about proving this equation using the symbolic execution rules for $\oplus$, then we can observe that proving the refinement in right-to-left direction

$$(e_1; e_2) \oplus (e_1; e_3) \precsim e_1; (e_2 \oplus e_3) : \tau$$

is possible, and, by rel-or-l, it boils down to proving two refinements:

$$e_1; e_2 \precsim e_1; (e_2 \oplus e_3) : \tau \qquad e_1; e_3 \precsim e_1; (e_2 \oplus e_3) : \tau.$$

However, proving the refinement in left-to-right direction is harder:

$$e_1; (e_2 \oplus e_3) \precsim (e_1; e_2) \oplus (e_1; e_3) : \tau.$$

If we want to use the symbolic execution rules for $\oplus$, we have to "synchronize" both sides on e_1. To do that, we have to pick a branch for $\oplus$ on the right-hand side before we get to use REL-OR-L on the left-hand side, but we do not know ahead of time which branch to pick. To resolve this dependency, we use a prophecy variable to speculate on which branch $e_2 \oplus e_3$ will be taken on the left-hand side, and use the value of this prophecy variable to choose the appropriate branch of $(e_1; e_2) \oplus (e_1; e_3)$ on the right-hand side.

The intermediate program that is instrumented with prophecy variables is as follows:

$$\begin{array}{l}\mathsf{let}\ p = \mathsf{newproph}\ \mathsf{in}\\ e_1;\ \big((\mathsf{resolve}\, p\, \mathsf{to}\ 0; e_2) \oplus (\mathsf{resolve}\, p\, \mathsf{to}\ 1; e_3)\big)\end{array}$$

We can easily verify that the original program $e_1; (e_2 \oplus e_3)$ refines the instrumented one. To verify that the instrumented program refines $(e_1; e_2) \oplus (e_1; e_3)$ we symbolically execute newproph and obtain a predicate $\mathsf{proph}(p, \vec{v})$ associating a vector of future values $\vec{v}$ to the newly created prophecy variable p. Then we examine the head element w of the prophecy values $\vec{v}$. If w is 0, then we apply REL-OR-R-1, otherwise we apply REL-OR-R-2. Without loss of generality, suppose that w is 0; that is, $\vec{v} = 0 :: \vec{w}$ for some tail $\vec{w}$. First we "synchronize" the refinement proof on e_1 on both sides. Then we apply REL-OR-L. Because the premises of REL-OR-L are joined by intuitionistic conjunction $\wedge$, we can use the resource $\mathsf{proph}(p, \vec{v})$ for verifying both refinements:

$$\mathsf{proph}(p, 0 :: \vec{v'}) \mathrel{-\!\!*} \mathsf{resolve}\, p\, \mathsf{to}\ 0; e_2 \precsim e_2 : \tau$$
$$\mathsf{proph}(p, 0 :: \vec{v'}) \mathrel{-\!\!*} \mathsf{resolve}\, p\, \mathsf{to}\ 1; e_3 \precsim e_2 : \tau$$

The first refinement is reduced to $e_2 \precsim e_2 : \tau$, which follows from the fundamental property (Theorem 4.5) and the assumption that e_2 is well-typed. To prove the second refinement we symbolically execute $\mathsf{resolve}\, p\, \mathsf{to}\ 1$ on the left-hand side, at which point we reach a contradiction $0 = 1$.

7. The logical relations model of ReLoC

ReLoC extends Iris with logical connectives and corresponding proof rules for reasoning about refinements. In this section we show how this is achieved by modeling the connectives of ReLoC through a shallow embedding in Iris and proving the logical rules of ReLoC as mere lemmas in Iris. We describe how the refinement judgment $e_1 \precsim e_2 : \tau$ is modeled through Iris's weakest preconditions and a *ghost thread pool* construction (Section 7.1) combined with a *binary logical relation* $[\![\tau]\!]_\Delta$ that describes when values are related (Section 7.2). We then summarize how the ReLoC proof rules (Section 7.4) and soundness theorem (Section 7.5) are proved. The key definitions of the ReLoC model are shown in Figure 14.

The construction of our model generalizes prior work by Turon *et al.* [TTA+13, TDB13], which culminated in the CaReSL logic, and was subsequently mechanized in Iris by Krebbers *et al.* [KTB17] and Timany [Tim18]. We discuss the differences in Section 7.3.

7.1. The refinement judgment.

Recall from Section 3.1 that the intuitive meaning of the refinement proposition $e_1 \precsim e_2 : \tau$ is that *any* behavior of e_1 can be simulated by *some*

Refinement judgments:

$$\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau \triangleq \forall i, K.\, \mathsf{specCtx} \mathrel{-\!\!*} i \Mapsto K[\,e_2\,] \;{}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{wp}\, e_1 \{v_1.\, \exists v_2.\, i \Mapsto K[\,v_2\,] * [\![\tau]\!]_\Delta(v_1, v_2)\}$$

Interpretation of types:

$$\begin{aligned}
[\![\alpha]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \Delta(\alpha)(v_1, v_2)\\
[\![\mathsf{unit}]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, v_1 = v_2 = ()\\
[\![\mathsf{bool}]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, (v_1 = v_2 = \mathbf{true}) \vee (v_1 = v_2 = \mathbf{false})\\
[\![\mathsf{int}]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \exists n \in \mathbb{Z}.\, v_1 = v_2 = n\\
[\![\tau \times \sigma]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \exists w_1, w_2, w_1', w_2'.\, v_1 = (w_1, w_2) * v_2 = (w_1', w_2') * [\![\tau]\!]_\Delta(w_1, w_1') * [\![\sigma]\!]_\Delta(w_2, w_2')\\
[\![\tau + \sigma]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \exists w_1, w_2.\, (v_1 = \mathsf{inl}(w_1) * v_2 = \mathsf{inl}(w_2) * [\![\tau]\!]_\Delta(w_1, w_2)) \vee (v_1 = \mathsf{inr}(w_1) * v_2 = \mathsf{inr}(w_2) * [\![\sigma]\!]_\Delta(w_1, w_2))\\
[\![\tau \to \sigma]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \square\, (\forall w_1, w_2.\, [\![\tau]\!]_\Delta(w_1, w_2) \mathrel{-\!\!*} (\Delta \models v_1\ w_1 \precsim v_2\ w_2 : \sigma))\\
[\![\forall \alpha.\, \tau]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \square\, \big(\forall \Phi \in \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_\square.\, ([\alpha := \Phi], \Delta \models v_1\langle\rangle \precsim v_2\langle\rangle : \tau)\big)\\
[\![\exists \alpha.\, \tau]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \exists \Phi \in \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_\square.\, [\![\tau]\!]_{[\alpha:=\Phi],\Delta}(v_1, v_2)\\
[\![\mu \alpha.\, \tau]\!]_\Delta &\triangleq \mu \Phi.\, \lambda(v_1, v_2).\, \exists w_1, w_2.\, v_1 = \mathsf{fold}(w_1) * v_2 = \mathsf{fold}(w_2) * \triangleright [\![\tau]\!]_{[\alpha:=\Phi],\Delta}(w_1, w_2)\\
[\![\mathsf{ref}\ \tau]\!]_\Delta &\triangleq \lambda(v_1, v_2).\, \exists \ell_1, \ell_2 \in \mathit{Loc}.\, v_1 = \ell_1 * v_2 = \ell_2 * \boxed{\exists w_1, w_2.\, \ell_1 \mapsto_{\mathsf{i}} w_1 * \ell_2 \mapsto_{\mathsf{s}} w_2 * [\![\tau]\!]_\Delta(w_1, w_2)}^{(\ell_1, \ell_2)}
\end{aligned}$$

FIGURE 14. The model of ReLoC in Iris.

behavior of e_2. This intuitive idea is modeled in Iris as follows:

$$\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau \triangleq \forall i, K.\, \mathsf{specCtx} \mathrel{-\!\!*} i \Mapsto K[\,e_2\,] \;{}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{wp}\, e_1 \{v_1.\, \exists v_2.\, i \Mapsto K[\,v_2\,] * [\![\tau]\!]_\Delta(v_1, v_2)\}$$

This definition is quite a mouthful, so let us go over it piece by piece. First, it involves Iris's *weakest precondition* connective $\mathsf{wp}\, e\, \{\Phi\}$, which gives the weakest precondition under which execution of e is safe, and when e returns with value v, the postcondition $\Phi(v)$ holds. Second, it involves the *ghost thread pool* connective $i \Mapsto e$, which is defined through Iris's ghost theory, and states that the i-th ghost thread is executing a program e. Putting these pieces together (ignoring $\mathsf{specCtx}$ and ${}^{\mathcal{E}}\!\Rrightarrow^{\top}$ for now), this definition states that if a (ghost) thread i is executing right-hand side e_2, and left-hand side e_1 reduces to some value v_1, then a corresponding execution can be made so that (ghost) thread i is executing right-hand side v_2. The result values v_1 and v_2 of the left-hand and right-hand side should be related via the value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$, which we model in Section 7.2 via a logical relation. The quantification over K closes the definition under evaluation contexts. The expression e_1 on the left-hand side does not need to be closed under evaluation contexts because weakest preconditions enjoy the rule: $\mathsf{wp}\, e\, \{w.\, \mathsf{wp}\, K[\,w\,]\, \{\Phi\}\} \mathrel{-\!\!*} \mathsf{wp}\, K[\,e\,]\, \{\Phi\}$.

STEP-PURE
$$\frac{\mathsf{specCtx} \qquad i \Rrightarrow e \qquad e \to_{\mathsf{pure}} e'}{\Rrightarrow_{\mathcal{E}} i \Rrightarrow e'}$$

STEP-ALLOC
$$\frac{\mathsf{specCtx} \qquad i \Rrightarrow K[\,\mathsf{ref}(v)\,]}{\Rrightarrow_{\mathcal{E}} \exists \ell.\, i \Rrightarrow K[\,\ell\,] * \ell \mapsto_{\mathsf{s}} v}$$

STEP-STORE
$$\frac{\mathsf{specCtx} \qquad i \Rrightarrow K[\,\ell \leftarrow w\,] \qquad \ell \mapsto_{\mathsf{s}} v}{\Rrightarrow_{\mathcal{E}} i \Rrightarrow K[\,()\,] * \ell \mapsto_{\mathsf{s}} w}$$

STEP-FORK
$$\frac{\mathsf{specCtx} \qquad i \Rrightarrow K[\,\mathsf{fork}\ \{e\}\,]}{\Rrightarrow_{\mathcal{E}} \exists j.\, i \Rrightarrow K[\,()\,] * j \Rrightarrow e}$$

FIGURE 15. Selected rules for the ghost thread pool.

The ghost thread pool predicates satisfy a number of symbolic execution rules corresponding to executions in the operational semantics. A selection of these rules is given in Figure 15. The specCtx proposition is an Iris invariant that ties together the thread pool connectives $i \Rrightarrow e$ and the heap assertions $\ell \mapsto_{\mathsf{s}} v$ with a matching execution on the right-hand side. We will explain the role of specCtx in Section 7.5.

We should emphasize that the combination of the weakest precondition and the ghost thread pool in the definition of $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ model the demonic nature of e_1 and the angelic nature of e_2. To prove the weakest precondition $\mathsf{wp}\ e_1\ \{v_1.\ \exists v_2.\ i \Rrightarrow K[\,v_2\,] * [\![\tau]\!]_\Delta(v_1, v_2)\}$ one has to consider all behaviors of e_1, but has to establish only a single matching execution for e_2 by using the appropriate rules for the ghost thread pool.

7.2. **The logical relation.** The interpretation of types $[\![\tau]\!]_\Delta(v_1, v_2)$, as defined in Figure 14, expresses when two values v_1 and v_2 are related at type τ (in context Δ). The definition of $[\![\tau]\!]_\Delta(v_1, v_2)$ follows the usual structure of a logical relation, it is defined recursively on the structure of the type τ and uses the corresponding logical connectives via the Curry-Howard isomorphism. For example, products are defined via (separating) conjunction, sums are defined via disjunction, functions are defined via (separating) implication, universal types are defined via universal quantification, *etc.*

The interpretation of recursive types and reference types are somewhat more interesting, as they make use of Iris-specific connectives. The interpretation of the recursive type $\mu\alpha.\,\tau$ makes use of Iris's guarded fixed point operator $\mu x.\,t$, which is used to define recursive predicates without a restriction of the variance of the recursive occurrence x in t, but requires x to appear in *guarded* position, *i.e.,* under the later modality $\triangleright$ [JKJ+18, Section 5.6]. To define the interpretation of the reference type $\mathsf{ref}\ \tau$, we use the invariant

$$\boxed{\exists w_1, w_2.\, \ell_1 \mapsto_{\mathsf{i}} w_1 * \ell_2 \mapsto_{\mathsf{s}} w_2 * [\![\tau]\!]_\Delta(w_1, w_2)}^{(\ell_1, \ell_2)},$$

which states that whatever values are stored in ℓ_1 and ℓ_2 are always related at type $[\![\tau]\!]_\Delta$.

The persistence modality $\square$ in the interpretation for function types and universal types is used to ensure that the type interpretation is persistent and prevents the kind of issues described in Section 4.2. Similarly, in the interpretation of the universal and existential types we quantify over a *persistent* predicate $\Phi \in \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_\square$, where $\mathit{iProp}_\square$ is the subset of Iris propositions that is persistent.

7.3. **Differences with prior work.** The definition of the refinement $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ and value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$ generalize the versions by Krebbers *et al.* [KTB17] and Timany *et al.* [Tim18], which in turn adapted ghost thread pools by Turon *et al.* [TTA+13, TDB13] by modeling these in Iris. The main novelty is that our refinement judgment $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ is a first-class Iris proposition, instead of a meta-logical proposition. As we have demonstrated throughout this paper, this modification is simple, albeit crucial for writing conditional refinements and to obtain high-level proof rules for refinements.

Furthermore, to obtain high-level proof rules for invariants, we have equipped the refinement judgment with a mask $\mathcal{E}$, which keeps track of the invariants that may be opened. To give the appropriate semantics to the mask $\mathcal{E}$, our definition involves the update modality ${}^{\mathcal{E}}\!\Rrightarrow\!\!\!\!\!\!*{}^{\top}$. Note that the definition by Krebbers *et al.* [KTB17] and Timany [Tim18] is logically equivalent to $\vdash \Delta \models_\top e_1 \precsim e_2 : \tau$, where the derivability relation $\vdash$ of Iris is used to turn the judgment into a meta theoretical proposition, and the mask is set to $\top$.

7.4. **Deriving the primitive rules.** In Section 4.3 we have demonstrated that ReLoC's primitive monadic (REL-RETURN and REL-BIND) and symbolic execution rules can be used to derive ReLoC's high-level proof rules, such as its type-directed structural rules. In this section, we indicate how ReLoC's primitive rules are proved by unfolding the definition of the refinement judgment. We prove the symbolic execution rules through the following auxiliary rules, which allow us to lift Iris's rules for weakest preconditions and the ghost thread pool rules (Figure 15) to the refinement judgment:

$$\text{REL-WP-L}\quad \frac{\mathsf{wp}_\top\, e_1\, \{v_1.\, K[\,v_1\,] \precsim e_2 : \tau\}}{K[\,e_1\,] \precsim e_2 : \tau} \qquad \text{REL-WP-ATOMIC-L}\quad \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e_1\, \{v_1.\ \models_{\mathcal{E}} K[\,v_1\,] \precsim e_2 : \tau\} \qquad \mathsf{atomic}(e_1)}{K[\,e_1\,] \precsim e_2 : \tau}$$

$$\text{REL-STEP-R}\quad \frac{\forall j, K'.\, \mathsf{specCtx} * j \Rightarrow K'[\,K[\,e_2\,]\,] \Rrightarrow\!\!\!\!\!\!*_{\mathcal{E}}\ \exists v_2.\, j \Rightarrow K'[\,K[\,v_2\,]\,] * \models_{\mathcal{E}} e_1 \precsim K[\,v_2\,] : \tau}{\models_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

The rule REL-WP-L says that we can "take out" an expression e_1 in context K on the left-hand side, and reason about it using Iris's weakest precondition. The rule REL-WP-ATOMIC-L is similar, but it also allows for opening an invariant around e_1, in case e_1 is atomic.[6] The rule REL-STEP-R says that if we have an expression e_2 on the right-hand side in an evaluation context K, and we can reduce e_2 to a value v_2, using the ghost thread pool rules, then we can reduce the refinement proposition to $\models_{\mathcal{E}} e_1 \precsim K[\,v_2\,] : \tau$.

7.5. **Soundness.** Utilizing the definitions in this section, we outline the proof of the soundness theorem (Theorem 4.6), which says that ReLoC's refinement judgment is sound w.r.t. contextual refinement. Formally, if $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC for any Δ with $\Xi \subseteq \mathsf{dom}(\Delta)$, then $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$. To prove this theorem we make use of two key lemmas: adequacy of the refinement judgment (Theorem 7.1), and the fact that the refinement judgment is a precongruence (Lemma 7.2).

Theorem 7.1 (Adequacy of ReLoC)**.** *If* $\vdash \Delta \models e_1 \precsim e_2 : \tau$ *is derivable in ReLoC, and* $(e_1, \sigma) \to^*_{\mathsf{tp}} (v_1 :: \vec{e_f}_1, \sigma_1')$*, then there exists* $v_2, \vec{e_f}_2$*, and* σ_2' *such that* $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_f}_2, \sigma_2')$.

[6] Iris's weakest precondition connective $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ is also equipped with a mask to keep track of which invariants may be opened. This was the inspiration for the mask annotation at ReLoC's refinement judgment.

Lemma 7.2. *Let $\mathcal{C}$ be a well-typed context $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\Xi' \mid \Gamma' \vdash \tau')$, then we have*

$$\square(\forall \Delta.\, \Delta \mid \Gamma \models e_1 \precsim e_2 : \tau) \mathrel{-\!\!*} (\forall \Delta'.\, \Delta' \mid \Gamma' \models \mathcal{C}[\,e_1\,] \precsim \mathcal{C}[\,e_2\,] : \tau')$$

where Δ and Δ' contain at least the type variables in Ξ and Ξ', respectively.

Lemma 7.2 is proved by induction on $\mathcal{C}$ making using of ReLoC's type-directed structural rules (Section 4.4). The proof of Theorem 7.1 is rather involved, so before we discuss that, let us see how we prove the soundness theorem by putting these two lemmas together.

Proof of Theorem 4.6 (Soundness for open terms). Let Ξ be a type environment, and suppose that $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC for any Δ with $\Xi \subseteq \mathsf{dom}(\Delta)$. To prove $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$, suppose we have typed context $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau')$, and reduction $(\mathcal{C}[\,e_1\,], \emptyset) \to^*_{\mathsf{tp}} (v_1 :: \vec{e_{f1}}, \sigma_1)$. By Lemma 7.2, we have $\mathcal{C}[\,e_1\,] \precsim \mathcal{C}[\,e_2\,] : \tau'$. Then, by Theorem 7.1, we get that $(\mathcal{C}[\,e_2\,], \emptyset) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_{f2}}, \sigma_2)$ for some $v_2, \vec{e_{f2}}$ and σ_2, which concludes the proof. □

Proof of Theorem 7.1 (Adequacy of ReLoC). Suppose that $\Delta \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC, and we have $(e_1, \sigma) \to^*_{\mathsf{tp}} (v_1 :: \vec{e_{f1}}, \sigma_1')$. Now we should exhibit $v_2, \vec{e_{f2}}$, and σ_2' such that $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_{f2}}, \sigma_2')$. The high-level structure of the proof is as follows. First, we allocate the thread pool invariant $\mathsf{specCtx}$ and $0 \Mapsto e_2$ for the main-thread of the right-hand side. Second, by definition of the refinement judgment, we obtain a weakest precondition $\mathsf{wp}\ e_1\ \{v_1.\, \exists v_2.\, 0 \Mapsto v_2 * [\![\tau]\!]_\Delta(v_1, v_2)\}$. Third, by opening $\mathsf{specCtx}$ and using adequacy of Iris's weakest preconditions, we obtain $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_{f2}}, \sigma_2')$.

Carrying out these steps in detail—notably, setting up the required ghost theory for the ghost thread pool—involves some intricate reasoning using Iris features that are out of scope for this paper. We thus refer the interested reader to the Coq mechanization, and only highlight the key part—the definition of the thread pool invariant $\mathsf{specCtx}$:

$$\mathsf{specCtx} \triangleq \exists \vec{e_0}, \sigma_0.\, \boxed{\exists \vec{e}, \sigma.\, \mathsf{spec\_inv}(\vec{e}, \sigma) * (\vec{e_0}, \sigma_0) \to^*_{\mathsf{tp}} (\vec{e}, \sigma)}^{\mathcal{N}_{\mathrm{ReLoC}}}.$$

The invariant asserts that given an initial configuration $(\vec{e_0}, \sigma_0)$ for the right-hand side (which we set to be (e_2, σ) when allocating the invariant), the configuration $(\vec{e}, \sigma)$ can be reached via the reduction $(\vec{e_0}, \sigma_0) \to^*_{\mathsf{tp}} (\vec{e}, \sigma)$. Here, $\mathsf{spec\_inv}(\vec{e}, \sigma)$ is a connective defined using Iris's ghost theory that keeps track of the configuration of the ghost thread pool and ensures it is consistent with the $\Mapsto$ and $\mapsto_{\mathsf{s}}$ connectives. The latter is essential, as it allows us to conclude from $\mathsf{spec\_inv}(\vec{e}, \sigma)$ and $0 \Mapsto v_2$ (as given by the post condition of the weakest precondition in the definition of the refinement judgment) that $\vec{e}$ is equal to $v_2 :: \vec{e_{f2}}$ for some $\vec{e_{f2}}$. By definition of the invariant $\mathsf{specCtx}$, this gives us a reduction $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_{f2}}, \sigma_2')$ for the right-hand side, which is needed to conclude the third step of the proof. □

8. The Coq mechanization of ReLoC

The Coq mechanization of ReLoC provides a soundness proof of ReLoC and infrastructure to carry out interactive tactic-based refinement proofs. It is built on top of the mechanization of Iris in Coq [Iri20] and the Iris Proof Mode/MoSeL framework for tactic-based proofs in separation logic [KTB17, KJJ+18]. In this section we examine the way ReLoC's language and type system are defined (Section 8.1), and how the ReLoC logic is defined on top of that (Section 8.2). We then describe ReLoC's tactic support for interactive refinement proofs,

which allows us to seamlessly carry out proofs in Coq similar to those we have seen in this paper (Section 8.3). Finally, we give an overview of the source code (Section 8.4).

8.1. **The programming language.** Iris is a programming language independent framework, which means that it can be instantiated with a programming language of choice. In this paper, we do not make use of this generality, and use HeapLang—the default language shipped with Iris's Coq development, which is essentially an untyped version of the language we considered in Section 2. HeapLang is represented via a deep embedding and comes with a set of notations so that programs can be written in Coq-style syntax. For example, the Boolean implementation bitbool of the bit module from Section 3.3 is written as follows:

```
Definition bit_bool : expr :=
  (#true, (λ: "b", ~"b"), (λ: "b", "b")).
```

Binders in HeapLang are represented as strings, which makes it possible to write programs in a human-readable way. This works well in practice because expression-level substitution only acts on closed terms, and thus does not need to be capture avoiding.

We equip HeapLang with a type system in the usual way—types `type` are defined as an inductive data type, and the typing judgment `typed` is defined as an inductive relation:

```
Inductive type :=
  | TVar : var → type
  | TProd : type → type → type
  | TArrow : type → type → type
  | TExists : {bind 1 of type} → type
  | (* ... *).
Inductive typed : stringmap type → expr → type → Prop :=
  | Var_typed Γ x τ :
     Γ !! x = Some τ → (Γ ⊢t Var x : τ)
  | Pair_typed Γ e1 e2 τ1 τ2 :
     (Γ ⊢t e1 : τ1) → (Γ ⊢t e2 : τ2) → (Γ ⊢t (e1, e2) : τ1 * τ2)
  | Fst_typed Γ e τ1 τ2 :
     (Γ ⊢t e : τ1 * τ2) → (Γ ⊢t Fst e : τ1)
  | (* ... *).
```

We use the notation `Γ ⊢t e : τ` for `typed Γ e τ`, and overload the standard Coq notations for types, *e.g.,* we use the notation $\tau_1 * \tau_2$ for `TProd` $\tau_1\ \tau_2$ and $\tau_1 \to \tau_2$ for `TArrow` $\tau_1\ \tau_2$. Since type-level substitution acts on (potentially) open terms, and therefore needs to be capture avoiding, we use De Bruijn indices to represent type-level binders through the Autosubst Coq library [STS15]. For example, the type $\mathsf{TBit} \triangleq \exists\alpha.\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool})$ from Section 3.3 is represented in Coq as follows (`#` is notation for `TVar`):

```
Definition bitτ : type := ∃: #0 * (#0 → #0) * (#0 → TBool).
```

8.2. **The ReLoC logic.** Recall from Section 7 that ReLoC is defined as a shallow definition in Iris—the ReLoC connectives are definitions in Iris, and the ReLoC proof rules are lemmas in Iris. In Coq we follow the same approach. At the core of ReLoC we have the definition `lrel` of *semantic types*, *i.e.,* persistent Iris relations over HeapLang values:

```
Record lrel Σ := LRel {
  lrel_car :> val → val → iProp Σ;
  lrel_persistent v₁ v₂ : Persistent (lrel_car v₁ v₂)
}.
```

Here, `iProp Σ` is the type of Iris propositions.[7] The record bundles together a relation together with a proof that it is persistent. The notation `:>` declares the field `lrel_car` as a coercion. In the Coq mechanization of ReLoC we generalize the refinement judgment $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ to range over semantic types (`lrel`) instead of syntactic types (`type`):

```
Definition refines (E : coPset) (e₁ e₂ : expr) (A : lrel Σ) : iProp Σ :=
  ∀ j K, spec_ctx -∗ j ⤇ fill K e₂ ={E,⊤}=∗
         WP e₁ {{ v₁, ∃ v₂, j ⤇ fill K (of_val v₂) ∗ A v₁ v₂ }}.
```

The parameter `E` corresponds to the mask $\mathcal{E}$, and the semantic type `A` corresponds to the type interpretation $[\![\tau]\!]_\Delta$. We use the notation `REL` e_1 `<<` e_2 `@ E : A` for `refines E` e_1 e_2 `A`. This definition makes use of the ghost thread pool connectives `spec_ctx` and `j ⤇ e`, as discussed in Section 7.1, and which were originally defined in Coq in [KTB17].

To formalize the refinement judgment on syntactic types, we first define the semantic interpretation $[\![\tau]\!]_\Delta$, denoted as `interp` τ Δ in Coq, which maps syntactic types τ to semantic types. To define the semantic interpretation, we define semantic type formers, which are combinators on semantic types corresponding to each syntactic type former. For example, the semantic product type is defined as follows:

```
Definition lrel_prod (A B : lrel Σ) : lrel Σ := LRel (λ v₁ v₂,
  ∃ w₁ w₂ w₁' w₂', ⌜v₁ = (w₁,w₁')%V⌝ ∧ ⌜v₂ = (w₂,w₂')%V⌝ ∧ A w₁ w₂ ∗ B w₁' w₂').
```

Here, we use Iris's notion $\ulcorner\varphi\urcorner$ to embed Coq propositions φ `: Prop` into Iris, although on paper we take the equality predicate to be primitive. With the above definitions at hand, we can now define ReLoC's refinement judgment $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ as `REL` e_1 `<<` e_2 `@ E : interp` τ Δ.

The proof rules. For example, the rule REL-LOAD-R is formalized as the following lemma:

```
Lemma refines_load_r E K l q v e₁ A :
  ↑ relocN ⊆ E →
  l ↦ₛ{q} v -∗
  (l ↦ₛ{q} v -∗ REL e₁ << fill K (of_val v) @ E : A) -∗
  REL e₁ << fill K !#l @ E : A.
```

The lemma states that, under the assumption that $\mathsf{relocN}^{\uparrow} \subseteq \mathcal{E}$ (*i.e.,* ReLoC's internal invariants are available in the mask $\mathcal{E}$), the following separation logic formula holds:

$$\ell \overset{q}{\mapsto}_{\mathsf{s}} v \mathrel{-\!\!*} (\ell \overset{q}{\mapsto}_{\mathsf{s}} v \mathrel{-\!\!*} \models_{\mathcal{E}} t \precsim K[\,v\,] : A) \mathrel{-\!\!*} \models_{\mathcal{E}} t \precsim K[\,!\,\ell\,] : A$$

[7] The parameter Σ describes the kind of ghost state available in Iris. It is an important but technical detail that can safely be ignored for the purpose of this paper. An interested reader is directed to [JKJ+18, §4.7].

This is exactly the internalization of REL-LOAD-R. The other ReLoC proof rules are mechanized in a similar way.

Soundness. The versions of ReLoC's soundness theorem for closed (Theorem 3.1) and open terms (Theorem 4.6) are stated in Coq as follows:

```
Lemma refines_sound Σ `{relocPreG Σ} e1 e2 τ :
  (∀ `{relocG Σ} Δ, ⊢ REL e1 << e2 : interp τ Δ) →
  ∅ ⊨ e1 ≾ctx e2 : τ.
Lemma refines_sound_open Σ `{relocPreG Σ} Γ e1 e2 τ :
  (∀ `{relocG Σ} Δ, ⊢ {Δ;Γ} ⊨ e1 ≾log e2 : τ) →
  Γ ⊨ e1 ≾ctx e2 : τ.
```

Here, $\Gamma \vDash e_1 \precsim_{ctx} e_2 : \tau$ is the notion of contextual refinement, $\{\Delta;\Gamma\} \vDash e_1 \precsim_{log} e_2 : \tau$ is the refinement judgment lifted to open expressions, and $\vdash P$ expresses that the Iris proposition `P` is derivable.

Example proof: refinement of the bit module. In order to prove the contextual refinement $\emptyset \vDash$ `bit_bool` $\precsim_{ctx}$ `bit_nat` : `bit`τ from Section 3.3, it suffices to prove the following:

```
Lemma bit_refinement Δ : ⊢ REL bit_bool << bit_nat : interp bitτ Δ.
```

To prove this lemma, we use the relation `R`, which is the same as the one in Section 3.3, but wrapped into a semantic type (`lrel`) to ensure it is persistent:

```
Definition R : lrel Σ := LRel (λ v1 v2,
  (⌜v1 = #true⌝ ∧ ⌜v2 = #1⌝) ∨ (⌜v1 = #false⌝ ∧ ⌜v2 = #0⌝)).
```

Using the relation `R`, a Coq proof of the desired refinement is as follows:

```
Lemma bit_refinement Δ : ⊢ REL bit_bool << bit_nat : interp bitτ Δ.
Proof.
  unfold bitτ; simpl. iApply (refines_exists R). (* apply REL-PACK *)
  progress repeat iApply refines_pair. (* repeatedly apply REL-PAIR *)
  - rel_values. (* apply REL-RETURN and solve the goal *)
  - (* ... *)
Qed.
```

Finally, we combine `bit_refinement` with the soundness theorem to get a closed proof of contextual refinement:

```
Theorem bit_ctx_refinement : ∅ ⊨ bit_bool ≾ctx bit_nat : bitτ
Proof. auto using (refines_sound relocΣ), bit_refinement. Qed.
```

It is important to emphasize that the contextual refinements, which we obtain in theorems like `bit_ctx_refinement` above, are closed propositions in Coq. The statement (the type) of `bit_ctx_refinement` does not refer to ReLoC or Iris. This illustrates that the only parts of the trusted code base of our development are the notions that are involved in the definition of contextual refinement, *i.e.,* the operational semantics and the typing of contexts.

```
--------------------*
P -∗ (P -∗ Q) -∗ Q
```

(A) Before executing any tactics.

```
"H1" : P
"H2" : P -∗ Q
--------------------*
Q
```

(B) After `iIntros "H1 H2".`

FIGURE 16. Interactive proof of lemma `example` in IPM.

8.3. **Tactic support for interactive proofs.** To prove refinement judgments, like the bit refinement `REL bit_bool << bit_nat : interp bit`τ Δ from the previous section, we can repeatedly apply the Iris lemmas corresponding to the ReLoC proof rules. However, doing so directly quickly becomes unwieldy, as the user has to manually provide the resources (like the precondition `l` $\mapsto_s$`{q} v` of `refines_load_r`), and manually select the evaluation context `K`. For better usability we provide tactic support for symbolic execution.

Interactive separation logic proofs. To explain the tactics for ReLoC that we have defined, let us first look at the general tactic support in Iris. The Iris Proof Mode (IPM) [KTB17] and its successor MoSeL [KJJ+18] allow us to carry out separation logic proofs interactively, in the style of regular tactic-based proofs in Coq. IPM provides a convenient representation of sequents for separation logic and tactics for manipulating them, allowing for interactive proof development in the style of regular proofs in Coq. To illustrate this, consider the following separation logic tautology:

```
Lemma example (P Q : iProp Σ) : P -∗ (P -∗ Q) -∗ Q.
Proof. iIntros "H1 H2". iApply ("H2" with "H1"). Qed.
```

The intermediate results can be seen in Figure 16. Applying `iIntros "H1 H2".` introduces the hypothesis `P` and `P -∗ Q` into the IPM context, giving them names `H1` and `H2`, respectively. Then, `iApply ("H2" with "H1").` applies the separating implication `P -∗ Q` to the goal, using the hypothesis `H1 : P` as the assumption.

Symbolic execution tactics. In addition to tactics like `iIntros` and `iApply`, IPM provides tactic for symbolic execution in weakest preconditions. We built similar tactics on top of IPM for symbolic execution in refinement judgments. For example, consider:

```
Lemma example_load l : l ↦s #0 -∗ REL #2 << (!#l + #2) : lrel_int.
Proof. iIntros "Hl". rel_load_r. rel_pures_r. rel_values. Qed.
```

The results of `rel_load_r` and `rel_pures_r` can be seen in Figure 17. The tactic `rel_load_r` symbolically executes the dereferencing operation, and the tactic `rel_pures_r` symbolically executes as many pure reduction steps as possible. The tactic `rel_values` finishes the goal since both sides are values. Similarly, we built tactics for all other language connectives (both on the left- and right-hand side). The tactics were developed in a similar way to the weakest-precondition tactics from IPM, and we refer the reader to [KTB17] for details.

```
"Hl" : l ↦ₛ #0
------------------------------*
REL #2 << (!#l + #2) : lrel_int
```

(A) Before applying `rel_load_r`.

```
"Hl" : l ↦ₛ #0
------------------------------*
REL #2 << (#0 + #2) : lrel_int
```

(B) After applying `rel_load_r`.

```
"Hl" : l ↦ₛ #0
------------------------------*
REL #2 << #(0 + 2) : lrel_int
```

(C) After applying `rel_pures_r`.

FIGURE 17. Interactive refinement proof of lemma `example_load` in ReLoC.

8.4. **Overview of the source code.** The Coq mechanization contains around 10300 lines of code, of which approximately (1) 1315 lines for mechanization of the model of ReLoC (Section 7), including the adequacy theorem (Theorem 7.1), and the primitive and derived rules (Section 4); (2) 1200 lines for the tactics (Section 8.3); (3) 1450 lines for the mechanization of the type system (Section 2), and the soundness theorem for open term (Theorem 4.6); (4) 6050 lines for the examples and case studies (including the case studies we describe in the upcoming Sections 10.1 and 10.2); (5) and 140 lines for tests (mainly regression tests for the tactics).

9. Related work

We described some of the most closely related work in the introduction (Section 1), we now discuss other related work on logical relations models, relational logics, atomic specifications, speculative reasoning, and linearizability.

Logical relations models. Logical relations models over denotational and operational semantics have an extensive history. To cover advanced programming language features such as recursive types and higher-order references, logical relations with step-indexing have been introduced [AAV02, Ahm04, ADR09, BRS+11]. Step-indexing has shown to be very effective by a large body of work on step-indexed logical relations models, *e.g.,* [NDR11, HD11, BST12, ÇPG16, RG18]. However, in these papers step-indices appear explicitly in the definition of the logical relations model and the proofs about it. In contrast in this paper we have used the "logical approach" to step-indexed logical relations. This approach, pioneered by Dreyer *et al.* in the LSLR logic [DAB09], hides step-indices by abstracting and internalizing them in a logic using the later modality ($\triangleright$) [AMRV07]. Dreyer *et al.* used this approach to construct a binary logical relations model for System F with recursive types [DAB09], and later extended the approach as part of the LADR logic to cover existential types and references [DNRB10].

The logical approach to logical relations was further refined by Turon *et al.* [TTA+13, TDB13], culminating in the CaReSL logic, who showed how Hoare triples and ghost thread pools can be used to define a binary logical relation for fine-grained concurrency. Subsequently, a version of this binary logical relation was defined and mechanized in Iris by Krebbers *et al.* [KTB17] and Timany [Tim18]. However, in these papers, logical refinement judgments are meta-logical statements, and because of that, there are no high-level proof rules for

establishing and combining refinements. Instead, to prove a refinement judgment, the user of the logic had to unfold the definition of the refinement judgment, and reason directly in CaReSL or Iris. In this work we provide a generalization that makes refinement judgments first-class logical statements, which is crucial to reason abstractly about invariants and formulate atomic specifications. The technical differences are discussed in Section 7.3. Thus we really make use of the fact that Iris is a *higher-order* logic—CaReSL is only a second-order logic and it would not be possible to make refinement judgments first-class in CaReSL (indeed Iris is not only based on CaReSL, but just as much on the *higher-order* iCAP logic of Svendsen and Birkedal [SB14]). We also provide a mechanization in Coq with tactical support that supports the same backwards reasoning style that is employed for proving weakest preconditions in Iris [KTB17].

Apart from the directions that we explored in this paper, there has been an abundance of work on logical relations models in Iris. Binary logical relations models in Iris have been used for proving contextual equivalence in the context of Haskell's `ST` monad [TSKB18], first-class per-thread continuations [TB19], and types-and-effect systems [KJSB17]. Unary logical relations models in Iris have been used for proving type safety and data-race freedom of the Rust type system [JJKD18, DJKD20, JJKD21], type safety of session types [HLKB21], type safety of Scala's core calculus DOT [GST$^+$20], and robust safety [SGD17, SGDL20]. Logical relations in Iris have also been used for showing other relational properties such as termination-preserving refinement [TJH17], non-interference of concurrent programs [FKB21b], and recovery refinements (refinements in the presence of potential crashes) [CTKZ19]. Nearly all of the aforementioned developments have accompanying mechanizations in Coq, and in some of those mechanizations the authors define their own tactics. They define tactics for either their version of weakest preconditions or for derived operations, but, to the best of our knowledge, they do not define tactics for reasoning about the logical relation directly.

Relational logics. Logics for proving relational properties of programs have a long history, going back to the earlier work of Plotkin and Abadi [PA93]. Since then many relational logics have been developed addressing various applications, *e.g.,* probabilistic properties in security [BGZB09, BKOZB12, BDG$^+$13] and cost analysis [ÇBG$^+$17, RBG$^+$18]. Here we discuss some more recent work on relational logics that are capable of proving program refinements, with a focus on logics with support for higher-order languages, languages with mutable state, and languages with concurrency.

Earlier work on relational logics targeted programming languages with mutable state, but no concurrency. Relational Hoare logic [Ben04] and Relational Separation logic [Yan07] can be used for reasoning about relational properties for first-order imperative programs, and they have inspired several extensions, for example to probabilistic languages [BGZB09].

Relational Higher Order Logic (RHOL) [ABG$^+$19] is a recent relational higher-order logic for reasoning about relational properties of programs using relational refinement types. The main judgment of RHOL allows one to prove that a relational formula φ holds for two expressions, which do not necessarily have the same type. While it is not directly possible to reason about expressions with different types in ReLoC, we can relate them by using a type variable α and a suitable interpretation of α in the environment Δ. The authors prove soundness of RHOL and show how to embed a number of type systems into it. They provide proofs of various relational properties such as non-interference and relative cost, as provided by the systems they embed into RHOL. In our work we consider only one (family of) relation(s), namely the logical relation for contextual refinement. The programming

language considered in RHOL is a pure terminating variant of simply-typed PCF, while we consider a much richer programming language with general references and concurrency.

Liang and Feng developed a relational rely-guarantee style logic [LF13], which can be used to prove refinement for fine-grained concurrent algorithms (including those with helping) but, in contrast to ReLoC, it can only be used to reason about first-order programs.

A relational logic for a sequential class-based language with dynamically allocated objects has been introduced by Banerjee *et al.* [BNN16]. Their relational logic is based on region logic [BNR13], a first-order logic, which is amenable to SMT-based automation. Their relational logic is aimed at proving refinement and non-interference. The approach was further extended in [NBN19] to cover representation independence proofs using per-modules invariants and coupling relations. In contrast, we focus on reasoning about refinements, but also treat concurrent programs and higher-order store, and we provide tool support for tactic-based interactive verification in Coq.

While not a logic in the strict sense, Relational Hoare Type Theory (RHTT) [NBG13] is a dependent type theory for specification and verification of relational properties of higher-order programs with mutable first-order state, capable of expressing information flow and access control properties. The object programming language of RHTT and the type system itself are shallowly embedded in Coq.

Atomic specifications. To our knowledge, we are the first to study logically atomic specifications in the relational setting. Logically atomic specifications originate in Hoare-style program logics. Jacobs and Piessens [JP11] have originally developed a methodology for specifying logically atomic operations. In their approach, specifications are parameterized by auxiliary code that is performed at the linearization point. This approach was refined to what we refer to as HOCAP-style specifications, originally introduced in the context of the eponymous logic [SBP13], where the role of auxiliary code is filled by *view shifts* [DBG$^+$13], which in this paper are given by Iris's update modality $\Rightarrow\!\!\!\!\ast$ (Section 5.4). Compared to the original Jacobs-Piessens approach, in HOCAP-style specifications, the physical state that a logically atomic function operates on is hidden behind an abstract predicate. Furthermore, HOCAP-style specifications can also be formulated for non-logically atomic operations, as we have seen in Section 5.4.4. The HOCAP-style specifications were later adopted in the iCAP logic [SB14] and Iris logic [BB20, Chapter 11].

Because Jacobs-Piessens and HOCAP-style specifications require parameterizing the (ghost) functions that are executed at the linearization points, such specifications are often referred to as higher-order. As an alternative to this higher-order approach, da Rocha Pinto *et al.* have introduced the notion of logically atomic triples in their program logic TaDA [dRPDG14, dRP17]. Logically atomic triples are a first-order construct, built in as a primitive construct into the logic, which can be used to specify the atomic updates that a program performs. The atomic triples can be systematically composed in the style of Hoare logic. A more detailed comparison between the first-order and higher-order approach is given in [DYdRPG18]. TaDA-style logically atomic triples were adapted for Iris by Jung *et al.* [JSS$^+$15, JLP$^+$20]. Specifically, they are encoded as derived constructs, using the Jacobs-Piessens approach, that satisfy the TaDA-style rules.

Speculative reasoning. To facilitate speculative reasoning, we employ the mechanism for prophecy variables recently introduced in Iris [JLP$^+$20]. Prophecy variables were first introduced by Abadi and Lamport [AL91] for the purpose of proving refinements of state

machines. The idea to use prophecy variables in program logic originates in the rely-guarantee style logic of Vafeiadis [Vaf08], although his treatment of prophecy variables is informal, and he appeals to Abadi and Lamport [AL91] for soundness.

Prophecy variables are not the only tool for carrying out speculative reasoning. Both CaReSL [TDB13] and extended LRG [LF13] are program logics capable of proving refinements of programs with future-dependent linearization points. Both employ, albeit in different forms, a mechanism for recording multiple potential logical states of the program. These multiple states can then be coalesced into a single one, once the linearization point is determined, and that resulting state is used for establishing the refinement.

Other approaches [KDGP17, DSNB17] for proving linearizability of algorithms with future-dependent linearization points use Hoare logics with auxiliary state to track the abstract history of a program as a partial order. The crucial property is that all total extensions of the partial order result in valid linear histories of the program.

Other work on linearizability. One of the main application of ReLoC is to prove linearizability of concurrent algorithms, by reducing it to contextual refinements. Proving linearizability has a long history, and the program logic based approach is not the only one. Other methods include automated model checking based solutions [LCLS09, VYY09, ČRZ+10, BDMT10] and static analysis, in particular shape analysis, [ARR+07, BLAM+08, Vaf09]. The model checking approaches in question do not prove linearizability, but automatically *check* execution traces for linearizability, bounding the heap or the number of threads. Indeed, model checking approaches are designed to find bugs in a "push-button" fashion and can generate counterexample traces. Approaches based on static analysis are usually sound even for unbounded heaps and threads, but limited to first-order programs.

10. Discussion and conclusion

In this paper we have presented ReLoC—the first mechanized relational logic for proving refinements of fine-grained concurrent higher-order programs. We have demonstrated that ReLoC is expressive enough to formally prove contextual refinements of concurrent programs in a modular way, by employing relational specifications of programs. Moreover, the mechanization of ReLoC in Coq allows us to carry out tactic-based interactive proofs in an intuitive way, by using ReLoC's type-directed structural rules and symbolic execution rules, coupled with the powerful mechanisms from Iris, such as invariants, ghost state, and prophecy variables.

In the remainder of this paper we discuss other case studies that we have mechanized in ReLoC (Section 10.1), discuss the "escape hatch" of ReLoC (Section 10.2) for verifying programs that cannot be handled by ReLoC, and outline some directions for future work (Section 10.3).

10.1. **Other examples and case studies.**

In addition to the examples that we have presented in the paper, we have mechanized a number of examples from the literature on logical relations in ReLoC in Coq. Below we give a short summary of those examples.

- Linearizability of the Treiber stack [Tre86];
- Refinement of higher-order cell objects from [KW06, ADR09];
- Refinement of a symbol lookup table and a name generation module from [ADR09];

- Many equivalences from [DNB12], adapted for the concurrent setting, including variations of the "awkward example" from [PS98], and the "higher-order profiling" example modified to use the atomic increment function inc_i;
- Equivalence between different ways of defining the fixed point combinators;
- Equivalence between late-choice and early-choice examples from [TTA+13];
- Algebraic laws for the parallel composition operation and its interaction with non-deterministic choice and sequential composition, inspired by the work on Concurrent Kleene Algebra [HMSW11];
- Linearizability of the Michael-Scott queue [MS96], mechanized by Friis Vindum and Birkedal [VB21].

10.2. **The "escape hatch".** The rules of ReLoC are sound, but not complete. In particular, there are some examples that cannot be verified in ReLoC completely. One class of such examples that we know of, are refinements of fine-grained concurrent data structures with *external* linearization points (as opposed to fixed linearization points or future-dependent linearization points; see [DD15] for a survey outlining the differences). Such external linearization points are present, for example, in algorithms that use *helping* or *work-stealing*. Fortunately, ReLoC's model on top of Iris provides an "escape hatch" that still allows us to verify some data structures with helping.

In the appendix [FKB21a] we consider an example of such a data-structure: a fine-grained concurrent stack with helping, a simplified version of the elimination-backoff stack from [HSY04]. We prove that this stack with helping refines a coarse-grained stack (thus showing that the stack with helping is linearizable). The stack with helping is interesting because two threads that perform a push and pop operation concurrently can *eliminate* each other, by exchanging data through a side channel, thus reducing the contention for the top node of the stack. To verify this example we make use of ReLoC's "escape hatch"—we unfold the definition of ReLoC's refinement judgment, and perform an explicit proof in terms of ReLoC's model in Iris so we can explicitly manipulate the ghost thread pool. As we demonstrate, the "escape hatch" does not render ReLoC useless for this example: we still use ReLoC's proof rules to carry out the majority of the proof. Only for a small part of the proof we need to work in the model. This is achieved by encapsulating the *elimination* mechanism of the stack, for which we can provide a logically atomic relational specification that is proved in the model of ReLoC. This specification can then be used through ReLoC's high-level rules to verify the complete data structure without further breaking the abstraction.

10.3. **Future work.** In future work we would like to examine the possibility of a more principled approach to specifying and verifying algorithms with helping, without having to reason in the model of ReLoC. In addition, it would be interesting to explore alternative approaches to speculative reasoning that do not involve prophecy variables. Furthermore, we would like to study applications of ReLoC to type-directed program transformations (for example typed closure conversion [AB08]) and message-passing programs (for example, by integration with the Iris-based Actris logic [HBK20, HLKB21]).

It would also be interesting to see how the ReLoC approach can be used for verifying other kinds of refinements, for example termination-sensitive refinements [TJH17] or refinements in the presence of crashes [CTKZ19].

Acknowledgments

We thank the anonymous reviewers of this paper and the conference version at LICS'18 for their comments and suggestions. We thank Herman Geuvers and Simon Friis Vindum for discussions, and Amin Timany for his contributions to the linearizability proof of the stack with helping.

Dan Frumin was supported by the Dutch Research Council (NWO) under STW project 14319 (Sovereign) and VIDI Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software). Robbert Krebbers was supported by the Dutch Research Council (NWO), project 016.Veni.192.259. Lars Birkedal was supported by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation and by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *LICS*, pages 75–86, 2002.

[AB08] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ICFP*, pages 157–168, 2008.

[ABG+19] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. A relational logic for higher-order programs. *Journal of Functional Programming*, 29:e16, 2019.

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL*, pages 340–353, 2009.

[Ahm04] Amal Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[Ahm06] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, volume 3924 of *LNCS*, pages 69–83, 2006.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *POPL*, pages 109–122, 2007.

[ARR+07] Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, volume 4590 of *LNCS*, pages 477–490, 2007.

[BB20] Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. `https://iris-project.org/tutorial-material.html`, 2020.

[BBS13] Lars Birkedal, Aleš Bizjak, and Jan Schwinghammer. Step-indexed relational reasoning for countable nondeterminism. *Logical Methods in Computer Science*, 9(4), 2013.

[BDG+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *FOSAD*, volume 8604 of *LNCS*, pages 146–166, 2013.

[BDMT10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, pages 330–340, 2010.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101, 2009.

[BKOZB12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL*, pages 97–110, 2012.

[BLAM+08] Josh Berdine, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Mooly Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, volume 5123 of *LNCS*, pages 399–413, 2008.

[BNN16] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. Relational logic with framing and hypotheses. In *FSTTCS*, volume 65 of *LIPIcs*, pages 11:1–11:16, 2016.

[BNR13] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Local reasoning for global invariants, part I: Region logic. *JACM*, 60(3):18:1–18:56, 2013.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *TCS*, 375(1-3):227–270, 2007.

[BRS+11] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, pages 119–132, 2011.

[BST12] Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. A concurrent logical relation. In *CSL*, volume 16 of *LIPIcs*, pages 107–121, 2012.

[ÇBG+17] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *POPL*, pages 316–329, 2017.

[ÇPG16] Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. A type theory for incremental computational complexity with control flow changes. In *ICFP*, pages 132–145, 2016.

[ČRZ+10] Pavol Černý, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model checking of linearizability of concurrent list implementations. In *CAV*, volume 6174 of *LNCS*, pages 465–479, 2010.

[CTKZ19] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *SOSP*, pages 243–258, 2019.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *LICS*, pages 71–80, 2009.

[DBG+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *POPL*, pages 287–300, 2013.

[DD15] Brijesh Dongol and John Derrick. Verifying linearizability: A comparative survey. *ACM Computing Surveys*, 48(2):19:1–19:43, 2015.

[DDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, volume 6183 of *LNCS*, pages 504–528, 2010.

[DJKD20] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. RustBelt meets relaxed memory. *PACMPL*, 4(POPL):34:1–34:29, 2020.

[DNB12] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.

[DNRB10] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, pages 185–198, 2010.

[dRP17] Pedro da Rocha Pinto. *Reasoning with time and data abstractions*. PhD thesis, Imperial College London, 2017.

[dRPDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, volume 8586 of *LNCS*, pages 207–231, 2014.

[DSNB17] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent data structures linked in time. In *ECOOP*, volume 74 of *LIPIcs*, pages 8:1–8:30, 2017.

[DYdRPG18] Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Philippa Gardner. A perspective on specifying and verifying concurrent modules. *Journal of Logical and Algebraic Methods in Programming*, 98:1–25, August 2018.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[FKB18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*, pages 442–451, 2018.

[FKB21a] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Appendix and Coq development of ReLoC, 2021. Available at `https://iris-project.org/reloc/`.

[FKB21b] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs, 2021. To appear in Security & Privacy 2021.

[FORY10] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.

[Gor99] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1-2):5–47, 1999.

[GST+20] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step: Soundness for DOT with step-indexed logical relations in Iris, 2020.

[Har16] Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016.

[HBK20] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *PACMPL*, 4(POPL):6:1–6:30, 2020.

[HD11] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, pages 133–146, 2011.

[HLKB21] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing, 2021. To appear at CPP'21.

[HMSW11] Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene algebra and its foundations. *The Journal of Logic and Algebraic Programming*, 80(6):266–296, 2011.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, pages 206–215, 2004.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[IO01] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[Iri20] Iris team. The Iris Project website and Coq development, 2020. https://iris-project.org/.

[JJKD18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.

[JJKD21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in Rust: The promise and the challenge, 2021. To appear in CACM.

[JKBD16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[JLP+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. *PACMPL*, 4(POPL):45:1–45:32, 2020.

[JP11] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[JSS+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 *POPL*, pages 637–650, 2015.

[JSV10] Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *LICS*, pages 209–218, 2010.

[KDGP17] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. Proving linearizability using partial orders. In *ESOP*, volume 10201 of *LNCS*, pages 639–667, 2017.

[KJB+17] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, volume 10201 of *LNCS*, pages 696–723, 2017.

[KJJ+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. *PACMPL*, 2(ICFP):77:1–77:30, 2018.

[KJSB17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*, pages 218–231, 2017.

[Koz94] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017.

[KW06] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, pages 141–152, 2006.

[LCLS09] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In *FM*, volume 5850 of *LNCS*, pages 321–337, 2009.

[LF13] Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[Mit86] John Mitchell. Representation independence and data abstraction. In *POPL*, pages 263–276, 1986.

[MS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1):21–65, 1991.

[MS96] Maged Michael and Michael Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[NBG13] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *TOPLAS*, 35(2):6:1–6:41, 2013.

[NBN19] Mohammad Nikouei, Anindya Banerjee, and David A. Naumann. Data Abstraction and Relational Program Logic. *arXiv e-prints*, page arXiv:1910.14560, October 2019.

[NDR11] Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *Journal of Functional Programming*, 21(4-5):497–562, 2011.

[O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1-3):271–307, 2007.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.

[PA93] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In *TLCA*, volume 664 of *LNCS*, pages 361–375, 1993.

[Pit00] Andrew M. Pitts. Operational semantics and program equivalence. In *APPSEM*, volume 2395 of *LNCS*, pages 378–412, 2000.

[Pit05] Andrew M. Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. MIT Press, 2005.

[Plo76] Gordon Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[PS98] Andrew M. Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–274. Cambridge University Press, 1998.

[RBG+18] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *PACMPL*, 2(POPL):36:1–36:32, 2018.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris*, volume 19 of *LNCS*, pages 408–423, 1974.

[RG18] Vineet Rajani and Deepak Garg. Types for information flow control: Labeling granularity and semantic models. In *CSF*, pages 233–246. IEEE, 2018.

[SB14] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168, 2014.

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, volume 7792 of *LNCS*, pages 169–188, 2013.

[SGD17] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *PACMPL*, 1(OOPSLA):89:1–89:26, 2017.

[SGDL20] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *PACMPL*, 4(POPL):32:1–32:32, 2020.

[Smy76] Michael Smyth. Powerdomains. In *International Symposium on Mathematical Foundations of Computer Science*, pages 537–543. Springer, 1976.

[SP07] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *JACM*, 54(5):26, 2007.

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *ITP*, volume 9236 of *LNCS*, pages 359–374, 2015.

[SV20] Alex Simpson and Niels Voorneveld. Behavioural equivalence via modalities for algebraic effects. *TOPLAS*, 42(1):4:1–4:45, 2020.

[TB19] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *PACMPL*, 3(ICFP):105:1–105:28, 2019.

[TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[Tim18] Amin Timany. *Contributions in programming languages theory: Logical relations and type theory*. PhD thesis, KU Leuven, 2018.

[TJH17] Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In *ESOP*, volume 10201 of *LNCS*, pages 909–936, 2017.

[Tre86] R. Kent Treiber. Systems programming: Coping with parallelism. Technical report, Thomas J. Watson Research Center, 1986.

[TSKB18] 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. *PACMPL*, 2(POPL):64:1–64:28, 2018.

[TTA+13] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356, 2013.

[Vaf08] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[Vaf09] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCI*, volume 5403 of *LNCS*, pages 335–348, 2009.

[VB21] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl), 2021. To appear at CPP'21.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *SPIN*, volume 5578 of *LNCS*, pages 261–278, 2009.

[Yan07] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany