ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency

Dan Frumin
Radboud University
dfrumin@cs.ru.nl

Robbert Krebbers
Delft University of Technology
mail@robbertkrebbers.nl

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

Abstract

We present ReLoC: a logic for proving refinements of programs in a language with higher-order state, fine-grained concurrency, polymorphism and recursive types. The core of our logic is a judgement $e \precsim e' : \tau$, which expresses that a program e refines a program e' at type τ. In contrast to earlier work on refinements for languages with higher-order state and concurrency, ReLoC provides type- and structure-directed rules for manipulating this judgement, whereas previously, such proofs were carried out by unfolding the judgement into its definition in the model. These more abstract proof rules make it simpler to carry out refinement proofs.

Moreover, we introduce *logically atomic relational specifications*: a novel approach for relational specifications for compound expressions that take effect at a single instant in time. We demonstrate how to formalise and prove such relational specifications in ReLoC, allowing for more modular proofs.

ReLoC is built on top of the expressive concurrent separation logic Iris, allowing us to leverage features of Iris such as invariants and ghost state. We provide a mechanisation of our logic in Coq, which does not just contain a proof of soundness, but also tactics for interactively carrying out refinements proofs. We have used these tactics to mechanise several examples, which demonstrates the practicality and modularity of our logic.

CCS Concepts • **Theory of computation → Logic and verification**; **Separation logic**; *Concurrency*; *Program verification*;

Keywords Separation logic, logical relations, fine-grained concurrency, Iris, atomicity

ACM Reference Format:
Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS '18: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209174

1 Introduction

Recall that an expression e contextually refines e' if, for all contexts C, if $C[\,e\,]$ has some observable behaviour, then so does $C[\,e'\,]$, and that e and e' are contextually equivalent if e contextually refines e' and *vice versa*. Contextual equivalence and contextual refinement are often referred to as the gold standards of equivalence and refinement of program expressions: contextual equivalence of e and e' means that it is safe for a compiler to replace any occurrence of e by e', and contextual refinement is often used to specify the behaviour of programs, *e.g.*, one can show the correctness of a fine-grained concurrent implementation of an abstract data type by proving that it contextually refines a coarse-grained implementation, which is understood as the specification.

$$
\begin{aligned}
\mathsf{read} &\triangleq \lambda x\,().\ !\,x\\
\mathsf{inc}_s &\triangleq \lambda x\,l.\ \mathsf{acquire}\ l; \mathtt{let}\ n = !\,x\ \mathtt{in}\ x \leftarrow 1+n; \mathsf{release}\ l;\ n\\
\mathsf{counter}_s &\triangleq \mathtt{let}\ l = \mathsf{newlock}\,()\ \mathtt{in}\ \mathtt{let}\ x = \mathtt{ref}(0)\ \mathtt{in}\\
&\qquad (\mathsf{read}\ x, \lambda().\ \mathsf{inc}_s\ x\ l)\\
\mathsf{inc}_i &\triangleq \mathtt{rec}\ \mathit{inc}\ x = \mathtt{let}\ c = !\,x\ \mathtt{in}\\
&\qquad\qquad \mathtt{if}\ \mathsf{CAS}(x, c, 1+c)\ \mathtt{then}\ c\ \mathtt{else}\ \mathit{inc}\ x\\
\mathsf{counter}_i &\triangleq \mathtt{let}\ x = \mathtt{ref}(0)\ \mathtt{in}\ (\mathsf{read}\ x, \lambda().\ \mathsf{inc}_i\ x)
\end{aligned}
$$

Figure 1. Two concurrent counter implementations.

A simple example is the specification of a fine-grained concurrent counter by a coarse-grained version, $\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathbf{1} \to \mathbf{N}) \times (\mathbf{1} \to \mathbf{N})$, see Figure 1 for the code. The increment operation of the coarse-grained version, $\mathsf{counter}_s$ is performed inside a critical section guarded by a lock, whereas the fine-grained version, $\mathsf{counter}_i$, takes an "optimistic" lock-free approach to incrementing the value using a compare-and-set inside a loop. We will use the counter as a simple running example throughout the paper. Proving program refinements and equivalence directly is difficult because of the quantification over all contexts. As such, it is often the case that reasoning is done using the technique of logical relations. For programming languages with features such as impredicative polymorphism, recursive types, higher-order state, and concurrency logical relations models can be quite intricate. Such models usually involve recursively defined worlds (constructed using step-indexing) and various forms of resource accounting [2, 4, 5, 10]. To simplify both the definition and the application of logical relations, logical approaches to logical relations have been invented, for increasingly richer programming languages [13, 15, 26, 28].

A very recent publication [23], which is the basis for our work, shows how logical relations for $F_{\mu,\mathsf{ref},\mathsf{conc}}$, a language with impredicative polymorphism, recursive types, general references, and concurrency, can be defined in a state-of-the-art higher-order concurrent separation logic Iris [19–22].

Iris supports impredicative concurrent abstract predicates [12, 27] and includes general forms of ghost state which can be used both for the definition of binary logical relations and for reasoning about challenging program equivalences. The meta-theory of Iris is

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209174

formalised in Coq and Iris also comes equipped with a *proof mode*, an extensive set of tactics, which made it possible to formalise the definitions of logical relations in Iris in Coq [23].

However, the reasoning about logical relatedness of programs in [23] proceeds by unfolding and working with the explicit definition of the logical relations in the logic. In this paper, we abstract further and introduce a higher-order relational logic ReLoC, which extends Iris with *refinement judgements* to support first-class relational reasoning. The calculus of ReLoC provides type- and structure-directed rules for manipulating judgements of the form $\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim e' : \tau$, expressing that program e refines program e' at type τ. As a result, ReLoC allows a higher level of abstraction for proofs of contextual refinements by providing a relational logic for reasoning about logical refinements of programs. In comparison with the approach in [23], ReLoC enables more modular proofs due to the encapsulation and the first-class status of refinement judgements. This means, in particular, that a representation independence proof, a refinement of two modules, can be constructed modularly from the refinement proofs of module methods (under suitable assumptions). Moreover, we introduce *logically atomic relational specifications*: a novel approach to relational specifications for compound expressions that take effect at a single instant in time. Logically atomic relation specifications can be thought of as a relational variant of logically atomic triples in recent concurrent separation logics [11, 18, 21], and similarly to *loc. cit.* they support more modular proofs.

ReLoC is built on top of Iris, allowing the user to leverage the features of Iris such as invariants and higher-order ghost state. We have formalised ReLoC on top of the Iris formalisation in Coq [23] and also implemented new tactics which support mechanised interactive reasoning in ReLoC in a practical and modular way. To our knowledge, this is the first fully mechanised relational logic enabling reasoning about contextual refinements of programs in a concurrent higher-order imperative programming language.

Contributions and structure

- We present a novel relational logic ReLoC for reasoning about contextual refinements of concurrent higher-order imperative programs. We present our target programming language (§2), an overview of ReLoC (§3), and a more detailed definition of ReLoC (§4).
- We introduce logically atomic relational specifications to support logical atomicity for relational reasoning (§5).
- We show how to use ReLoC to prove several challenging refinements. In particular, we show its application to fine-grained concurrent algorithms (§5 and §6).
- We describe our formalisation of ReLoC in Coq [16] and explain how it supports mechanised interactive reasoning in ReLoC in a practical and modular way (§7).

We discuss further related work in §8 and conclude in §9.

2 The programming language $\mathsf{F}_{\mu,\mathrm{ref},\mathrm{conc},\exists}$

The programming language considered in this paper is $\mathsf{F}_{\mu,\mathrm{ref},\mathrm{conc},\exists}$: a typed polymorphic call-by-value λ-calculus with existential types, isorecursive types, higher-order references and `fork` $\{-\}$-based concurrency. The types are:

$$\begin{aligned}\tau \in \mathit{Type} ::= {} & \mathbf{1} \mid \mathbf{2} \mid \mathbf{N} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \mathbf{ref}\ \tau \\ & \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \alpha,\end{aligned}$$

Thread-local CBV head-reduction (omitted): $(e, \sigma) \to_{\mathsf{h}} (e', \sigma')$

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

$$\frac{(e, \sigma) \to_{\mathsf{h}} (e', \sigma')}{(\vec{e}_1\ K[\,e\,]\ \vec{e}_2, \sigma) \to_{\mathsf{tp}} (\vec{e}_1\ K[\,e'\,]\ \vec{e}_2, \sigma')}$$

$$(\vec{e}_1\ K[\,\texttt{fork}\ \{e\}\,]\ \vec{e}_2, \sigma) \to_{\mathsf{tp}} (\vec{e}_1\ K[\,()\,]\ \vec{e}_2\ e, \sigma)$$

Figure 2. Operational semantics of $\mathsf{F}_{\mu,\mathrm{ref},\mathrm{conc},\exists}$.

where α ranges over a countable infinite set *TVar* of type variables. The values and expressions are:

$$\begin{aligned}v \in \mathit{Val} ::= {} & \texttt{rec}\ f\ x = e \mid \Lambda.e \mid \texttt{fold}\ v \mid \texttt{pack}\ v \mid \ldots \\ e \in \mathit{Expr} ::= {} & x \mid \texttt{rec}\ f\ x = e \mid e_1(e_2) \mid \Lambda.e \mid e\,[] \mid \texttt{fold}\ e \mid \texttt{unfold}\ e \\ & \mid \texttt{pack}\ e \mid \texttt{unpack}\ e_1\ \texttt{in}\ e_2 \mid \texttt{fork}\ \{e\} \\ & \mid \texttt{ref}(e) \mid\ !\,e \mid e_1 \leftarrow e_2 \mid \texttt{CAS}(e_1, e_2, e_3) \mid \ldots\end{aligned}$$

(We omit the usual operations on pairs, sums, and integers.)

We use the following syntactic sugar: $\lambda x.\, e \triangleq \texttt{rec}\_\ x = e$, $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \triangleq (\lambda x.\, e_2)\, e_1$, and $e_1; e_2 \triangleq \texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$.

Terms are untyped, so type-level abstraction is written as $\Lambda.e$ and type application as $e\,[]$, as in [3]. Typing judgements 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 judgements are standard and hence omitted.

The operational semantics is split into two parts: thread-local head reduction $\to_{\mathsf{h}}$ and thread-pool reduction $\to_{\mathsf{tp}}$, see Figure 2. Both are defined using standard *call-by-value evaluation contexts*:

$$K \in \mathit{ECtx} ::= [\bullet] \mid K(e_2) \mid v_1(K) \mid K\,[] \mid \ldots$$

Thread-pool reduction 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 `fork` $\{e\}$, which spawns a thread e, and reduces itself to the unit value $()$.

An expression e_1 *contextually refines* an expression e_2 at type τ if no well-typed C context can distinguish the two:

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

The typing relation $C : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\Xi' \mid \Gamma' \vdash \tau')$ on full contexts C is standard, and can be found in the appendix [16].

3 A tour of ReLoC

We now give a brief tour of ReLoC, and demonstrate the purpose of its most important logical connectives:

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

ReLoC is an extension of Iris and thus includes all connectives of Iris, in particular, the *later* modality $\triangleright$, *persistence* modality $\Box$, *update* modality ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$, and the *invariant assertion* $\boxed{P}^{\mathcal{N}}$. We will introduce these connectives on a by need basis throughout this

section. Some of the connectives are annotated by *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ and *name spaces* $\mathcal{N} \subseteq \mathit{InvName}$, which are needed for bookkeeping related to invariants. Until we introduce invariants in §3.3, we will just omit these annotations.

An essential difference to ordinary Iris is that ReLoC has first-class *refinement judgements* $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$, which should be pronounced as "the expression e_1 refines the expression e_2 at type τ". The judgement contains two environments: Γ is a typing environment assigning types to program variables, Δ is an environment for assigning interpretations to type variables. These interpretations are given by an Iris relation of type $\mathit{Val} \times \mathit{Val} \to \mathsf{iProp}$. One such relation, the *value interpretation* relation $[\![\tau]\!]_\Delta : \mathit{Val} \times \mathit{Val} \to \mathsf{iProp}$ for each syntactic type τ will be discussed in §4. We elide the contexts in the refinement judgement whenever they are empty.

The intuitive meaning of $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is that e_1 is safe, and all of its behaviours can be simulated by e_2. One should think of e_1 being *demonic* and e_2 being *angelic*: for any behaviour (*i.e.,* order of scheduling) of e_1 we should find at least one matching behaviour of e_2. Since we often use refinement judgements to specify programs, we refer to e_1 as the *implementation* and to e_2 as the *specification*. The intuitive meaning is formally reflected by the soundness theorem w.r.t. contextual refinement of $\mathsf{F}_{\mu,\mathsf{ref},\mathsf{conc},\exists}$.

Theorem 3.1 (Soundness). *Suppose that* $\Xi = \alpha_1, \ldots, \alpha_n$ *and* $\Delta = [\alpha_1 := R_1], \ldots, [\alpha_n := R_n]$. *If the judgement* $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ *is derivable in ReLoC, then* $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$.

The proof of this theorem hinges on the following facts: **(1)** the relation $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is a precongruence, which follows from the compatibility lemmas (see §4); **(2)** soundness of weakest preconditions in Iris [23], as the relational judgement is encoded in terms of weakest preconditions. See the appendix for details [16].

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.,* [23], refinement judgements $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ in ReLoC are first class propositions. As such, we can combine them in arbitrary ways with the other logical connectives, *e.g.,*

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

which states that the refinement holds, under the assumption of another refinement, and some properties of the heap.

The fact that refinement judgements are first class plays an important role in the presentation of inferences rules: Each inference rule we present for the refinement judgements is really a shorthand for a ReLoC proposition, namely a magic wand between the separating conjunction of the antecedents and the consequent. For instance, the proposition above is presented as the following inference rule:

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

3.1 Proof of the counter refinement

Recall the counter example from Figure 1 in §1. We now show the refinement of the two concurrent modules in ReLoC:

$$\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathbf{1} \to \mathbf{N}) \times (\mathbf{1} \to \mathbf{N}).$$

The proof of the refinement will be done by:

- performing *symbolic execution*;
- establishing an *invariant* that links the values of the counters;
- verifying that the returned closures refine each other while preserving the invariant.

In this section, we will describe each of these steps. In the proof, we will use some (derived) ReLoC rules presented in Figure 3, and the *relational specifications* for locks as shown in Figure 4. The lock specification, which can be implemented by *e.g.,* a spinlock, is formulated in terms of an abstract predicate $\mathsf{isLock}(v, \texttt{false})$ (resp., $\mathsf{isLock}(v, \texttt{true})$) stating that v is a lock which is free (resp., in use).

3.2 Symbolic execution

Performing symbolic execution means reducing the left or right hand side of the refinement. For example, we can use the symbolic execution rules PURE-L and ALLOC-L' on the left to obtain:

$$c_i \mapsto_{\mathsf{i}} 0 \mathrel{-\!\!*} (\mathsf{read}\ c_i, \lambda().\ \mathsf{inc}_i\ c_i) \precsim \mathsf{counter}_s : (\mathbf{1} \to \mathbf{N}) \times (\mathbf{1} \to \mathbf{N}).$$

Subsequently, using the symbolic execution rules PURE-R, ALLOC-R and NEWLOCK-R on the right hand side the new goal becomes:

$$c_i \mapsto_{\mathsf{i}} 0 * c_s \mapsto_{\mathsf{s}} 0 * \mathsf{isLock}(l, \texttt{false}) \mathrel{-\!\!*}$$
$$(\mathsf{read}\ c_i, \lambda().\ \mathsf{inc}_i\ c_i) \precsim (\mathsf{read}\ c_s, \lambda().\ \mathsf{inc}_s\ c_s\ l) : (\mathbf{1} \to \mathbf{N}) \times (\mathbf{1} \to \mathbf{N}).$$

The symbolic execution rules are inspired by the "backwards" style Hoare rules of [17] and weakest-precondition rules in Iris [20, 22].

3.3 Invariants and persistent propositions

At this point we wish to prove a refinement of two closures. By the rule PAIR it would suffice to prove that both closures refine each other. However, we cannot directly use this rule because the proofs of both closures need 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 following invariant (using INV-ALLOC'):

$$\boxed{I_{\mathsf{cnt}}} = \boxed{\exists n \in \mathbb{N}.\ c_i \mapsto_{\mathsf{i}} n * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}(l, \texttt{false})}.$$

This invariant 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.

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*, if $P \mathrel{-\!\!*} \Box P$; otherwise, we say that P is *ephemeral*. To prove $\Box P$, one can only use persistent resources.

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

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} \mathsf{read}\ c_i \precsim \mathsf{read}\ c_s : \mathbf{1} \to \mathbf{N} \tag{1}$$

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} \lambda().\ \mathsf{inc}_i\ c_i \precsim \lambda().\ \mathsf{inc}_s\ c_s\ l : \mathbf{1} \to \mathbf{N} \tag{2}$$

We first describe how to prove the refinement of read. As $\lambda x.\, e$ is syntactic sugar for $\texttt{rec}\ \_\ x = e$, we can apply CLOSURE-UNIT at the function type $\mathbf{1} \to \mathbf{N}$ and obtain the new goal:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} \Box\left((\lambda().\ !\,c_i)\ () \precsim (\lambda().\ !\,c_s)\ () : \mathbf{N}\right).$$

$$\textsc{pure-l}\ \frac{e_1 \to_{\mathrm{pure}} e_1' \qquad \triangleright\, \Delta \mid \Gamma \models K[\,e_1'\,] \precsim e_2 : \tau}{\Delta \mid \Gamma \models K[\,e_1\,] \precsim e_2 : \tau} \qquad \textsc{pure-r}\ \frac{e_2 \to_{\mathrm{pure}} e_2' \qquad \Delta \mid \Gamma \models_{\mathcal{E}} e_1 \precsim K[\,e_2'\,] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

$$\textsc{alloc-l'}\ \frac{\forall \ell.\, \ell \mapsto_{\mathrm{i}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models K[\,\ell\,] \precsim e : \tau}{\Delta \mid \Gamma \models K[\,\mathtt{ref}(v)\,] \precsim e : \tau} \qquad \textsc{alloc-r}\ \frac{\forall \ell.\, \ell \mapsto_{\mathrm{s}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,\ell\,] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,\mathtt{ref}(v)\,] : \tau}$$

$$\textsc{load-r}\ \frac{\ell \mapsto_{\mathrm{s}} v \qquad \ell \mapsto_{\mathrm{s}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,v\,] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,!\ell\,] : \tau}$$

$$\textsc{store-r}\ \frac{\ell \mapsto_{\mathrm{s}} - \qquad \ell \mapsto_{\mathrm{s}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,()\,] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,\ell \leftarrow v\,] : \tau} \qquad \textsc{nat}\ \frac{n \in \mathbb{N}}{\Delta \mid \Gamma \models n \precsim n : \mathbf{N}}$$

$$\textsc{pair}\ \frac{\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau \qquad \Delta \mid \Gamma \models e_1' \precsim e_2' : \sigma}{\Delta \mid \Gamma \models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \sigma}$$

$$\textsc{closure-unit}\ \frac{\square\,(\Delta \mid \Gamma \models (\mathtt{rec}\ f\ x = e)\ () \precsim (\mathtt{rec}\ f'\ x' = e')\ () : \tau)}{\Delta \mid \Gamma \models \mathtt{rec}\ f\ x = e \precsim \mathtt{rec}\ f'\ x' = e' : \mathbf{1} \to \tau} \qquad \textsc{inv-alloc'}\ \frac{\triangleright P \qquad \boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \Delta \mid \Gamma \models e_1 \precsim e_2 : \tau}{\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau}$$

$$\textsc{load-l-inv}\ \frac{\boxed{P}^{\mathcal{N}} \qquad (\triangleright P * \mathsf{closeInv}_{\mathcal{N}}(P)) \mathrel{-\!\!*} \exists v.\, \ell \mapsto_{\mathrm{i}} v * \triangleright\left(\ell \mapsto_{\mathrm{i}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} K[\,v\,] \precsim e_2 : \tau\right)}{\Delta \mid \Gamma \models K[\,!\ell\,] \precsim e_2 : \tau}$$

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

$$\textsc{inv-restore}\ \frac{\mathsf{closeInv}_{\mathcal{N}}(P) \qquad \triangleright P \qquad \Delta \mid \Gamma \models_{\mathcal{E}} e_1 \precsim e_2 : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}\setminus\mathcal{N}} e_1 \precsim e_2 : \tau}$$

Figure 3. Selected (derived) ReLoC rules for the running example.

$$\textsc{newlock-r}\ \frac{\forall v.\, \mathsf{isLock}(v, \mathtt{false}) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,v\,] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,\mathsf{newlock}\ ()\,] : \tau} \qquad \textsc{acquire-r}\ \frac{\mathsf{isLock}(v, \mathtt{false}) \qquad \mathsf{isLock}(v, \mathtt{true}) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,()\,] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,\mathsf{acquire}\ v\,] : \tau}$$

$$\textsc{release-r}\ \frac{\mathsf{isLock}(v, b) \qquad \mathsf{isLock}(v, \mathtt{false}) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,()\,] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\,\mathsf{release}\ v\,] : \tau}$$

Figure 4. Relational specification for a lock implementation.

Note the *persistence* modality $\square$ in the premise of CLOSURE-UNIT: it ensures that we do not use ephemeral resources in the verification of the body of a closure. After all, closures can be invoked arbitrarily many times at different points in time (possibly concurrently). For example, without the $\square$ modality in the premise of CLOSURE-UNIT one would be able to prove the following unsound refinement:

$$\mathtt{let}\ \ell = \mathtt{ref}(0)\ \mathtt{in}\ \lambda().\, \ell \leftarrow 1 + \ell; !\ell \ \precsim\ \lambda().\, 1 \ :\ \mathbf{1} \to \mathbf{N},$$

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 is turned into an invariant $\boxed{P}$, one is only allowed to access P during a single *atomic* execution step on the left hand side. However, due to the simulation-like nature of the proofs, ReLoC allows one to perform 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 $\square$ and performing pure symbolic execution steps):

$$\boxed{I_{\mathrm{cnt}}} \mathrel{-\!\!*} !\, c_i \precsim !\, c_s : \mathbf{N}.$$

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

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

In our example, by applying LOAD-L-INV, we obtain $c_i \mapsto_{\mathrm{i}} n$ and $c_s \mapsto_{\mathrm{s}} n$ and $\mathsf{isLock}(l, \mathtt{false})$, for some $n \in \mathbb{N}$, reducing our goal

to $\models_{\top\setminus\mathcal{N}} n \precsim !\, c_s : \mathbf{N}$. We then use LOAD-R to reduce our goal to $\models_{\top\setminus\mathcal{N}} n \precsim n : \mathbf{N}$. Because these steps did not change the heap, INV-RESTORE's premises for closing the invariant are trivially met.

Let us take a look at the rules LOAD-L-INV and INV-RESTORE in more detail. A crucial aspect of these rules is that they ensure that access to the invariant $\boxed{P}$ is *temporary*, *i.e.,* that P is only used during a single symbolic execution step on the left hand side (but possibly many on the right). This is achieved by tagging each invariant $\boxed{P}^{\mathcal{N}}$ with a name space $\mathcal{N} \subseteq \mathit{InvName}$, 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 judgements $\Gamma \mid \Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ are annotated with a *mask* $\mathcal{E} \subseteq \mathit{InvName}$ of accessible invariants. By default all invariants are accessible, so we write $\Gamma \mid \Delta \models e_1 \precsim e_2 : \tau$ for $\Gamma \mid \Delta \models_{\top} e_1 \precsim e_2 : \tau$, where $\top$ is the set of all invariant names.

When accessing an invariant, *e.g.,* using LOAD-L-INV or CAS-L-INV, its namespace is removed from the mask annotation of the judgement. 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 INV-RESTORE).

One may wonder why refinement judgements are annotated with a mask instead of a Boolean that indicates if an invariant has been opened. As we will show in §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 are *impredicative* [20, 27]. This means that P is allowed to contain other invariant assertions $\boxed{Q}^{\mathcal{N}'}$ or even refinement judgements. 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.,* they consist solely of the connectives of first-order logic and *e.g.,* heap assertions), these modalities can be soundly omitted.

3.4 Later modality and Löb induction

As is custom in logics based on step-indexing [6], like Iris, the later modality $\triangleright$ and Löb induction are used to reason about recursive functions. In our example, this means that by Löb induction, we may prove $\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ l : \mathbf{N}$, under the assumption of the induction hypothesis $\triangleright(\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ l : \mathbf{N})$. 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. Let us see how it works in the example. We use PURE-L to arrive at:

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

By monotonicity of $\triangleright$, we can now remove the $\triangleright$ from the induction hypothesis. Subsequently, we symbolically execute the load operation using the invariant, just like in the previous section, reaching:

$$\mathtt{if}\ \mathtt{CAS}(c_i, n, 1+n)\ \mathtt{then}\ n\ \mathtt{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ l : \mathbf{N}$$

for some $n \in \mathbb{N}$. In order to symbolically execute the $\mathtt{CAS}(-,-,-)$, we use the rule 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 $\mathtt{CAS}(-,-,-)$ or not.

1. Suppose that the value of the counter c_i has changed. In that case the $\mathtt{CAS}(-,-,-)$ operation fails and we are left with:

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

   for some $m \neq n$. Because the symbolic heap has not been changed, we can easily restore the invariant and execute the $\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ \ldots\ \mathtt{else}\ \ldots$ to obtain $\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ l : \mathbf{N}$, which is exactly our induction hypothesis.
2. If the value has not changed, then the $\mathtt{CAS}(-,-,-)$ succeeds and we are left with the new goal:

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

   At this point we use the symbolic execution rules STORE-R, LOAD-R and the lock specifications from Figure 4 to symbolically execute the right hand side of the refinement and update the resources to match:

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

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

4 A closer look at ReLoC

In this section we explain some of the more technical details of ReLoC and show how to obtain the principles that we have used in §3.1 from more primitive and generic rules. Some of these primitive rules are shown in Figure 5 and Figure 6.

First, we describe how to work with invariants using Iris's update modality $\Rightarrow$. Then we go through a selection of primitive rules of ReLoC and explain how the symbolic execution and compatibility rules can be derived from them.

Invariants and the update modality The rules for invariants as presented in §3.3 are fairly restricted, *e.g.,* they allow at most one invariant to be opened at the same time. We now show ReLoC's more generic rules, which integrate Iris's flexible mechanism for invariants and ghost state.

Invariants and ghost state in Iris are primarily controlled via the *(fancy) 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. Furthermore, this modality allows one to perform changes to Iris's ghost state via *frame preserving updates*, but we defer the description of that to [20].

Before discussing the rules for the update modality, let us recap some syntactic sugar used in 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 $P\ {}^{\mathcal{E}_1}\!\Rightarrow\!\!\!*\!\!\!\!\!\!*^{\mathcal{E}_2}\ Q$ for $P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q$, and follow the same conventions for omitting masks in $\Rightarrow\!\!\!*\!\!\!\!\!\!*$ as used for $\Rightarrow$.

ReLoC's main rule for interacting with the update modality is UPD-LOGREL. It allows to eliminate an update modality around a

$$\textsc{upd-logrel}\ \frac{\mathcal{E}_1 \Rrightarrow \mathcal{E}_2\, P \qquad P \twoheadrightarrow \Gamma \mid \Delta \models_{\mathcal{E}_2} e \precsim e' : \tau}{\Gamma \mid \Delta \models_{\mathcal{E}_1} e \precsim e' : \tau} \qquad \textsc{upd-upd}\ \frac{\mathcal{E}_1 \Rrightarrow \mathcal{E}_2\, P \qquad P \twoheadrightarrow {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} Q}{\mathcal{E}_1 \Rrightarrow \mathcal{E}_3\, Q}$$

$$\textsc{inv-alloc}\ \triangleright P \Rrightarrow\!\!\!\ast_{\mathcal{E}} \boxed{P}^{\mathcal{N}} \qquad \textsc{inv-access}\ \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}}\ \mathcal{E} \Rrightarrow\!\!\!\ast^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \ {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!\!\!\ast^{\mathcal{E}}\ \mathsf{True})}$$

Figure 5. A selection of primitive ReLoC rules for invariants (UPD-UPD, INV-ALLOC and INV-ACCESS are borrowed from Iris).

$$\textsc{load-l}\ \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\begin{array}{l} \exists v.\, \ell \mapsto_{\mathsf{i}} v * \\ \quad \triangleright \left(\ell \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[\, v\,] \precsim e : \tau\right) \end{array} \right)}{\Delta \mid \Gamma \models K[\, !\,\ell\,] \precsim e : \tau} \qquad \textsc{store-l}\ \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\begin{array}{l} \ell \mapsto_{\mathsf{i}} - * \\ \triangleright \left(\ell \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[\,()\,] \precsim e : \tau\right) \end{array} \right)}{\Delta \mid \Gamma \models K[\,\ell \leftarrow v\,] \precsim e : \tau}$$

$$\textsc{cas-l}\ \frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\begin{array}{l} \exists v'.\, \ell \mapsto_{\mathsf{i}} v' * \\ \quad \triangleright \left(v' \neq v_1 \twoheadrightarrow \triangleright(\ell \mapsto_{\mathsf{i}} v' \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[\,\mathtt{false}\,] \precsim e : \tau)\right) \wedge \\ \quad \triangleright \left(v' = v_1 \twoheadrightarrow \triangleright(\ell \mapsto_{\mathsf{i}} v_2 \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[\,\mathtt{true}\,] \precsim e : \tau)\right) \end{array} \right)}{\Delta \mid \Gamma \models K[\,\mathtt{CAS}(\ell, v_1, v_2)\,] \precsim e : \tau}$$

$$\textsc{closure}\ \frac{\square \left(\begin{array}{l} \forall v\, v'.\, [\![\tau]\!]_\Delta(v, v') \twoheadrightarrow \\ \quad \Delta \mid \Gamma \models (\mathtt{rec}\, f\, x = e)\, v \precsim (\mathtt{rec}\, f'\, x' = e')\, v' : \sigma \end{array} \right)}{\Delta \mid \Gamma \models \mathtt{rec}\, f\, x = e \precsim \mathtt{rec}\, f'\, x' = e' : \tau \to \sigma}$$

$$\textsc{tlam}\ \frac{\forall R : \mathit{Val} \times \mathit{Val} \to \mathsf{iProp}.\ \square\left([\alpha := R], \Delta \mid \Gamma \models e \precsim e' : \tau\right)}{\Delta \mid \Gamma \models \Lambda.e \precsim \Lambda.e' : \forall \alpha.\tau} \qquad \textsc{pack}\ \frac{[\alpha := R], \Delta \mid \Gamma \models e \precsim e' : \tau}{\Delta \mid \Gamma \models \mathtt{pack}\ e \precsim \mathtt{pack}\ e' : \exists \alpha.\, \tau}$$

Figure 6. A selection of primitive ReLoC rules.

refinement judgement. To get an idea of how this rule is used, let us take a look at the primitive rule for allocating an invariant INV-ALLOC. As one can see, the derived rule INV-ALLOC' from Figure 3 is just a composition of UPD-LOGREL and INV-ALLOC.

By combining UPD-LOGREL with Iris's rule 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 $\triangleright P \ {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!\!\!\ast^{\mathcal{E}}\ \mathsf{True}$. It is important to notice that by using the combination of these rules, the mask on the refinement judgement 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 UPD-LOGREL with the premise $\triangleright P \ {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!\!\!\ast^{\mathcal{E}}\ \mathsf{True}$. This requires one to give up P, and in turn transforms the mask of the judgement back into $\mathcal{E}$. Note that one can use INV-ACCESS multiple times to open multiple invariants.

Invariants and symbolic execution. The way of opening invariants by using UPD-LOGREL and INV-ACCESS in the way described above is fairly limited. Once we open an invariant, the mask at the refinement judgement changes from $\top$ into $\top \setminus \mathcal{N}$, which prevents any symbolic execution on the left hand side. This is because the rules for symbolic execution on that side require a $\top$ mask.

As we discussed in §3.3 already, the restriction to the $\top$ mask on symbolic execution rules for the left hand side is crucial. It is unsound to perform multiple symbolic execution steps on the left while an invariant is opened. Instead, 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 LOAD-L, STORE-L and CAS-L.

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

$${}^{\top}\!\Rrightarrow^{\top\setminus\mathcal{N}} \left(\begin{array}{l} \exists v.\, \ell \mapsto_{\mathsf{i}} v * \\ \quad \triangleright \left(\ell \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} K[\, v\,] \precsim e : \tau\right) \end{array} \right)$$

We then use INV-ACCESS and UPD-UPD to get the premise of LOAD-L-INV. In the same way CAS-L-INV can be derived from CAS-L.

Finally, the closing rule INV-RESTORE is a consequence of the definition of $\mathsf{closeInv}_{\mathcal{N}}(P)$ and UPD-LOGREL.

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

$$\textsc{store-l'}\ \frac{\ell \mapsto_{\mathsf{i}} v \qquad \triangleright \left(\ell \mapsto_{\mathsf{i}} w \twoheadrightarrow \Delta \mid \Gamma \models K[\,()\,] \precsim e : \tau\right)}{\Delta \mid \Gamma \models K[\,\ell \leftarrow w\,] \precsim e : \tau}$$

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 like LOAD-R, STORE-R, which are of a similar shape, can be performed even with invariants open because they allow the mask to be arbitrary.

Value interpretation and monadic rules. We now present some rules for the value interpretation $[\![\tau]\!]_\Delta : \mathit{Val} \times \mathit{Val} \to \mathsf{iProp}$. These rules are mostly used in a few select places when doing representation independence proofs.

$$\textsc{value-nat}\ \frac{\exists n \in \mathbb{N}.\, v_1 = v_2 = n}{[\![\mathbf{N}]\!]_\Delta(v_1, v_2)} \qquad \textsc{value-var}\ \frac{\square\, \Delta(\alpha)(v_1, v_2)}{[\![\alpha]\!]_\Delta(v_1, v_2)}$$

$$\textsc{value-arr}\ \frac{\square(\forall w_1\, w_2.\, [\![\tau]\!]_\Delta(w_1, w_2) \twoheadrightarrow \Delta \mid \emptyset \models v_1\ w_1 \precsim v_2\ w_2 : \sigma)}{[\![\tau \to \sigma]\!]_\Delta(v_1, v_2)}$$

The value interpretation is used in the following "monadic" rules for the relational judgements:

$$\textsc{return}\ \frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \mid \Gamma \models v_1 \precsim v_2 : \tau}$$

$$\textsc{bind}\ \frac{\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau \qquad \left(\begin{array}{l}\forall v_1\, v_2.\ [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} \\ \qquad \Delta \mid \Gamma \models K_1[\,v_1\,] \precsim K_2[\,v_2\,] : \sigma\end{array}\right)}{\Delta \mid \Gamma \models K_1[\,e_1\,] \precsim K_2[\,e_2\,] : \sigma}$$

The monadic rules are used to derive compatibility rules in the system, as we will see later in this section.

Note that the value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$ should be persistent because our type system enjoys contraction (*i.e.,* typing is not substructural). As such, morally any interpretation relation in the context Δ should be persistent. While persistence is not enforced by rules like PACK, the rule VALUE-VAR includes a persistence modality, which instead guarantees persistence of $[\![\tau]\!]_\Delta(v_1, v_2)$. As such, although one can use non-persistent interpretations in Δ, it would make representation independence proofs like the ones in §6 generally impossible: when one has to establish a refinement at type α, the $\Box$ modality in VALUE-VAR ensures that only persistent resources can be used to prove $\Delta(\alpha)(v_1, v_2)$.

Compatibility rules. Compatibility rules are type-directed rules that relate two terms of a similar shape. The rules correspond to "compatibility lemmas" in the logical relation literature and are crucial for proving soundness of ReLoC (Theorem 3.1).

The sole *primitive* compatibility rules of ReLoC are the those for `rec` $f\ x = e$, $\Lambda.e$, $e\ []$, `pack`(e), `unpack` e_1 `in` e_2, and `fork` $\{e\}$. The others can be derived using the monadic rules RETURN and BIND and the symbolic execution rules. As an example, consider the compatibility lemma for the first projection π_1.

Lemma 4.1. *The following rule is derivable:*

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

Proof. By BIND it suffices to show:

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

According to the value interpretation we have values v_i, w_i for $i \in \{0, 1\}$ such that $v = (v_1, v_2)$ and $w = (w_1, w_2)$ and $[\![\tau]\!](v_1, w_1) * [\![\sigma]\!](v_2, w_2)$. Using PURE-L and PURE-R we reduce the goal to $\Delta \mid \Gamma \models v_1 \precsim w_1 : \tau$. At this point we just apply RETURN. $\Box$

Using the compatibility rules we can show a standard result:

Theorem 4.2 (Fundamental theorem). *Suppose that* $\Xi = \alpha_1, \ldots, \alpha_n$ *and* $\Delta = [\alpha_1 := R_1], \ldots, [\alpha_n := R_n]$. *If* $\Xi \mid \Gamma \vdash e : \tau$, *then* $\Delta \mid \Gamma \models e \precsim e : \tau$ *is derivable in ReLoC.*

5 Relational specifications and atomicity

In our tour of ReLoC (§3) we saw an example of how relational specifications support modularity: to prove properties of a client of a module (in the example, a lock module) we do not need to know anything about the source code of the module.

Relational specifications for symbolic execution on the right hand side, such as the one used in §3, see Figure 4, follow a certain pattern. For an expression e_2 that under precondition P reduces to v with postcondition $Q(v)$, the rule has the following form:

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

The symbolic execution rules for the left hand side can be presented in a similar way:

$$\frac{P \qquad \forall v.\ Q(v) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[\,v\,] \precsim e_2 : \tau}{\Delta \mid \Gamma \models K[\,e_1\,] \precsim e_2 : \tau} \tag{3}$$

However, this specification for a left hand side program e_1 is *sequential* in the sense that the mask on the relational judgements is $\top$, which means that we cannot use such a specification if the resources mentioned in propositions P and Q are located in an invariant. In this section we will see how to formulate and use *logically atomic relational specifications* for resolving this issue.

5.1 Formulating atomic relational specifications

For any primitive operation of $\mathsf{F}_{\mu,\mathrm{ref},\mathrm{conc},\exists}$ we have a symbolic execution rule that allows the operation to access shared resources stored in an invariant, *e.g.,* the rules LOAD-L and STORE-L. These rules are sound because said operations are *physically atomic*—*i.e.,* they reduce in one step. Methods of a concurrent module are typically compound programs and hence not physically atomic. However, such operations often behave as if they were atomic, from a point of view of an arbitrary client, and we wish to express that.

Consider, for example, the counter increment program $\mathsf{inc}_i\ x$ for some counter x. It is a compound program which does not reduce to a value in one step. Nevertheless, during the execution of this program there is a single instant at which the whole operation actually appears to take the effect—namely the successful reduction of the `CAS`(-, -, -). Such instant is called a *linearisation point*. What it means is that, for an outside observer, the program $\mathsf{inc}_i\ x$ behaves "as if" it was atomic. This phenomenon is called *logical atomicity* in the literature [11, 21].

The idea behind *logically atomic relational specifications* is to provide (derived) proof rules for logically atomic operations that allows them to access shared resources. Taking inspiration from the encoding of atomic Hoare triples in [21] we write down the logically atomic rule for inc_i in Jacobs-Piessens style [18]:

$$\textsc{fg-increment-atomic-l}\ \frac{\Box\, {}^{\top}\!\Rightarrow^{\mathcal{E}} \left(\begin{array}{l}\exists n.\ x \mapsto_{\mathsf{i}} n * R(n) * \\ \left((x \mapsto_{\mathsf{i}} n * R(n) \ {}^{\mathcal{E}}\!\Rightarrow\!\!\ast^{\top} \mathsf{True}) \wedge \right. \\ \left.(x \mapsto_{\mathsf{i}} (n+1) * R(n) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau)\right)\end{array}\right)}{\Delta \mid \Gamma \models K[\,\mathsf{inc}_i\ x\,] \precsim e : \tau}$$

Contrary to the sequential rule (Equation (3)), we do not have $x \mapsto_{\mathsf{i}} n$ as a premise directly, but instead the premise contains a way of obtaining this resource. The typical way of obtaining $x \mapsto_{\mathsf{i}} n$ is by accessing an invariant using the update modality ${}^{\top}\!\Rightarrow^{\mathcal{E}}$ in the premise combined with INV-ACCESS. However, an invariant typically contains more resources than just $x \mapsto_{\mathsf{i}} n$, which we collect in a frame $R(n)$ instead of throwing them away.

To justify the remaining part of the premise we have to think about the behaviour of $\mathsf{inc}_i\ x$. The compare-and-set operation in $\mathsf{inc}_i\ x$ can either succeed or fail. If it succeeds, then we have managed to update our resources to $x \mapsto_{\mathsf{i}} (n+1)$, and we can proceed

with proving $\Delta \mid \Gamma \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau$ under that premise. This explains the $(x \mapsto_{\mathsf{i}} (n+1) * R(n) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau)$ clause.

If, however, the compare-and-set fails, then we need to be able to restart the whole computation. For that we must be able to return $x \mapsto_{\mathsf{i}} n$ to the invariant. Hence the $(x \mapsto_{\mathsf{i}} n * R(n) \;{}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True})$ clause. The same clause is used for performing operations that do not modify the state, such as dereferencing.

Finally, we know that the computation either succeeds or has to be restarted—but not both. Hence the last two clauses described here are connected by an intuitionistic conjunction ($\wedge$), instead of the separating conjunction ($*$).

5.2 Using atomic relational specifications

We can now use FG-INCREMENT-ATOMIC-L to prove the refinement that we have seen in §3.3 more modularly:

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

At this point we apply FG-INCREMENT-ATOMIC-L with:

$$R(n) \triangleq \mathsf{isLock}(l, \mathtt{false}) * c_s \mapsto_{\mathsf{s}} n.$$

After getting rid of the persistence modality, we get a new goal:

$$\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}} \mathrel{-\!\!*} {}^{\top}\!\Rrightarrow^{\top\setminus\mathcal{N}} \exists n.\, c_i \mapsto_{\mathsf{i}} n * R(n) * \ldots$$

At this point we can open up the invariant, and thereby introduce the update modality. The contents of the invariant provides us with a witness for the existential quantifier and allows us to frame the first two conjuncts. We are left with proving the conjunction:

$$\begin{aligned}&(c_i \mapsto_{\mathsf{i}} n * \mathsf{isLock}(l, \mathtt{false}) * c_s \mapsto_{\mathsf{s}} n \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}) \wedge\\ &(c_i \mapsto_{\mathsf{i}} (n+1) * \mathsf{isLock}(l, \mathtt{false}) * c_s \mapsto_{\mathsf{s}} n \mathrel{-\!\!*}\\ &\qquad \models_{\top\setminus\mathcal{N}} n \precsim \mathsf{inc}_s\ c_s\ l : \mathbf{N})\end{aligned}$$

under the premise of the invariant closing resource:

$$\triangleright\, I_{\mathsf{cnt}} \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}.$$

The first conjunct follows directly from the invariant closing resource. It thus remains to prove $n \precsim \mathsf{inc}_s\ c_s\ l : \mathbf{N}$ from:

$$(\triangleright\, I_{\mathsf{cnt}} \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}) * c_i \mapsto_{\mathsf{i}} (n+1) * \mathsf{isLock}(l, \mathtt{false}) * c_s \mapsto_{\mathsf{s}} n.$$

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. See the appendix and Coq formalisation [16] for the full proof.

5.3 General form of logically atomic specifications

The general form of logically atomic rules for logical refinements is thus the following:

$$\frac{R_2 \qquad \square\, {}^{\top}\!\Rrightarrow^{\mathcal{E}} \left(\begin{array}{l}\exists x.\, P(x) * R_1(x) * \\ \quad \big((P(x) * R_1(x) \;{}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True}) \wedge \\ \quad (\forall v.\, Q(x,v) * R_1(x) * R_2 \mathrel{-\!\!*} \\ \qquad \Delta \mid \Gamma \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau)\big)\end{array}\right)}{\Delta \mid \Gamma \models K[\,e_1\,] \precsim e_2 : \tau}$$

where $P : X \to \mathsf{iProp}$ is a predicate describing consumed resources and $Q : X \times \mathit{Val} \to \mathsf{iProp}$ is a predicate describing produced resources, both dependent on a type X supplied by the implementer of the rule. When using the rule, the client chooses an *invariant frame* $R_1 : X \to \mathsf{iProp}$ (that comprises the persistent resource $P(x) * R_1(x)$ together with the precondition) and an *ephemeral frame* $R_2 : \mathsf{iProp}$ containing all the ephemeral resources we had prior to applying the rule. We get access to those resources once again when we are ready to prove the new goal $\Delta \mid \Gamma \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau$.

We include the ephemeral frame R_2 in the rule for the following reason. The second premise of the rule resides behind the persistence modality. In order to prove such a premise we have to give up all the ephemeral resources. However, we do not want to throw away all the ephemeral resources that we have (as they might be needed to close the invariants afterwards or to proceed with the proof), so we give them up only temporarily.

6 Case studies

To evaluate the feasibility of our approach we have formalised several non-trivial example refinements including:

1. Generative ADTs from [5], such as a symbol generation and lookup table.
2. Higher-order functions with state from [14].
3. A port of the Treiber stack refinement from [23] to ReLoC.
4. A ticket lock specification in terms of a spinlock.

The examples from the first two points were adapted to work in a concurrent setting. They also demonstrate how Iris ghost state may be tightly integrated into the proofs, to enforce protocols such as monotonicity of the symbol table. Some of the proofs also involve the relational specifications for locks and for the atomic counter increment presented earlier. In the rest of this section we elaborate on the ticket lock refinement.

6.1 Ticket lock refinement

In this example, our goal is to prove the refinement of a spinlock by a ticket-based lock. This refinement demonstrates several important features of ReLoC. In particular, it demonstrates modularity and compositionality of proofs in ReLoC by employing logically atomic relational specifications from §5 and by splitting the module refinement proof into separate reusable refinement proofs of module methods. Furthermore, the proof highlights the integration of Iris ghost state to facilitate CAP-like [12] reasoning with abstract predicates, and it demonstrates our general approach to refinements of ADTs (detailed below).

Spinlock. As a specification program we consider the following simple implementation of a spinlock.

$$\begin{aligned}\mathsf{newlock}_s &\triangleq \lambda().\, \mathtt{ref}(\mathtt{false})\\ \mathsf{acquire}_s &\triangleq \mathtt{rec}\ \mathit{acquire}\ x =\\ &\qquad \mathtt{if}\ \mathtt{CAS}(x, \mathtt{false}, \mathtt{true})\ \mathtt{then}\ ()\ \mathtt{else}\ \mathit{acquire}\ x\\ \mathsf{release}_s &\triangleq \lambda x.\, x \leftarrow \mathtt{false}\end{aligned}$$

The relational specification for the spinlock is presented in Figure 4. We omit the proofs of the relational specifications for the spinlock and instead refer the reader to the accompanying Coq source code [16]. After establishing the soundness of the relational specification, we no longer need to appeal to the actual source code for the spinlock. This allows us to reason on a more abstract level and makes our proofs more resilient to change.

Ticket lock. As a more efficient version of a spinlock we consider the following ticket-based lock implementation:

$$\mathsf{newlock}_i \triangleq \lambda().\, (\mathtt{ref}(0), \mathtt{ref}(0))$$

$$\begin{aligned}\mathsf{wait\_loop} &\triangleq \mathtt{rec}\ \mathit{wait\_loop}\ n\ \mathit{lo} = \\ &\quad \mathtt{if}\ (n = !\,\mathit{lo})\ \mathtt{then}\ ()\ \mathtt{else}\ \mathit{wait\_loop}\ n\ \mathit{lo}\\ \mathsf{acquire}_i &\triangleq \lambda(\mathit{lo}, \mathit{ln}).\ \mathtt{let}\ n = \mathsf{inc}_i\ \mathit{ln}\ \mathtt{in}\ \mathsf{wait\_loop}\ n\ \mathit{lo}\\ \mathsf{release}_i &\triangleq \lambda(\mathit{lo}, \mathit{ln}).\ \mathit{lo} \leftarrow\ !\,\mathit{lo} + 1\end{aligned}$$

The two locations associated with the lock, *lo* and *ln*, point to the ID of the current owner of the lock and to the total number of issued tickets. When a thread wants to enter a critical section, it first requests a new ticket (by atomically increasing the value of *ln*), and then spins until the value of the current owner of the lock matches the ticket number. The ticket lock is fair—threads racing to enter a critical section will gain access to it in the order of arrival.

Proof setup. We show that:

$$\begin{aligned}&\mathsf{pack}(\mathsf{newlock}_i, \mathsf{acquire}_i, \mathsf{release}_i)\\ &\qquad \precsim \mathsf{pack}(\mathsf{newlock}_s, \mathsf{acquire}_s, \mathsf{release}_s)\\ &\qquad\qquad : \exists\alpha.\, (\mathbf{1} \to \alpha) \times (\alpha \to \mathbf{1}) \times (\alpha \to \mathbf{1})\end{aligned}$$

The proof follows our general strategy for proving refinements of stateful fine-grained concurrent ADTs in ReLoC:

1. We define a predicate $\mathsf{lockInv}_{-}(-,-,-)$ linking together the underlying representations of each individual pair of locks.
2. As the witness for the existential type, we pick a relation on the underlying representation of the two locks stating that there is an invariant linking the locks together via $\mathsf{lockInv}_{-}(-,-,-)$.
3. We prove the refinements for each method in the signature. Finally, we combine those proofs together into a module refinement proof. This is what we refer to as a *component-wise* refinement proof.

Due to space limitations, we are not able to present the refinement proof in details in the main part of the paper. We chose to sketch the main ideas pertaining to the points above in the remainder of this section, while spelling out the proofs themselves in the appendix [16].

Abstract predicates and the representations of locks. The *lock invariant* describes the relation between the values representing locks, $(\mathit{lo}, \mathit{ln})$ for the ticket lock and l' for the spinlock:

$$\begin{aligned}\mathsf{lockInv}_\gamma(\mathit{lo}, \mathit{ln}, l') \triangleq\ &\exists(o\, n : \mathbb{N})\,(b : \mathbb{B}).\\ &\mathit{lo} \mapsto_{\mathsf{i}} o * \mathit{ln} \mapsto_{\mathsf{i}} n * \mathsf{isLock}(l', b) *\\ &\mathsf{issuedTickets}_\gamma(n) * (\mathsf{if}\ b\ \mathsf{then}\ \mathsf{ticket}_\gamma(o)\ \mathsf{else}\ \mathsf{True})\end{aligned}$$

It refers to abstract predicates $\mathsf{ticket}_\gamma(n)$ and $\mathsf{issuedTickets}_\gamma(m)$. The former represents a ticket with id n and the latter states that a total of m tickets have been issued. Each ticket lock is associated with its own ticket dispensing machine—a ghost state gadget. The index γ in $\mathsf{ticket}_\gamma(-)$ and $\mathsf{issuedTickets}_\gamma(-)$ is an Iris ghost name of the associated dispensing machine. The abstract predicates are defined in terms of Iris ghost state, but for the purposes of the refinement proof, we only require them to satisfy certain rules (presented in the appendix) and we do not refer to the underlying definition in terms of ghost state and resource algebras.

Then, the relation linking together the two modules is:

$$\mathsf{lockInt}((\mathit{lo}, \mathit{ln}), l') \triangleq \exists\gamma.\, \boxed{\mathsf{lockInv}_\gamma(\mathit{lo}, \mathit{ln}, l')}^{\mathcal{N}}.$$

Refinement proof. The refinement proof is subdivided into three refinements for its components:

1. $[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathsf{newlock}_i \precsim \mathsf{newlock}_s : \mathbf{1} \to \alpha$;
2. $[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathsf{acquire}_i \precsim \mathsf{acquire}_s : \alpha \to \mathbf{1}$;
3. $[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathsf{release}_i \precsim \mathsf{release}_s : \alpha \to \mathbf{1}$.

The proofs of these refinements are done without exposing the underlying definitions of the abstract predicates and without references to the source code of the spinlock. We also stress that the proof of the $\mathsf{acquire}_i$ refinement does *not* rely on the source code of inc_i, but only refers to its logically atomic specification. Without its logically atomic relational specification, this would not have been possible (since atomicity is required for reasoning about updates to the invariant when proving the $\mathsf{acquire}_i$ refinement).

In order to prove the main statement we apply the compatibility lemmas PACK and PAIR, followed by the refinements above.

7 Coq formalisation

We have implemented the calculus presented here in Coq, building on the formalisation of Iris [20] and Iris Proof Mode (IPM) [23]. The formalisation contains a machine-checked proof of soundness directly against the operational semantics of $\mathsf{F}_{\mu,\mathsf{ref},\mathsf{conc},\exists}$, and all the examples presented and mentioned in this paper. The Treiber stack refinement has already been formalised in [23] as a monolithic proof. Our approach allowed for splitting the proof into distinct pieces combinable together. The compilation time for the example refinements has improved compared to *loc. cit.* , but we are not sure if that can be attributed to the increased modularity of the proofs or other optimisations, like the usage of explicit names and a better performing substitution function in our formalisation *vis-à-vis* a general purpose library for de Bruijn indices used in [23].

The backward-style reasoning is suitable for interactive proving, as it is already the style of reasoning employed in Coq. The formalisation contains machinery around the proof calculus, including an array of tactics for executing the proofs. Primitive rules of ReLoC are formalised as lemmas in Coq; the tactic mechanism is then used to automatically figure out the parameters for the proof rule (*e.g.,* the evaluation context K) and discharge proof obligations, if possible. Altogether this allows for seamless reasoning in the logic, as if the proofs were done on a whiteboard.

8 Related work

We described the most closely related work in the introduction (§1), we now discuss other related work on relational logics.

The rules for symbolic execution in ReLoC are similar to corresponding rules in the relational LSLR logic [13] for System F with recursive types and in the relational LADR logic [15] for System F with recursive types and references. However, ReLoC also includes symbolic execution rules for a programming language with concurrency. Furthermore, ReLoC uses general Iris invariants, whereas LSLR did not include support for invariants (since the programming language did not include mutable state) and LADR had support for more specialised invariants. None of these earlier relational logics came with mechanised tool support.

Liang and Feng present a relational rely-guarantee style logic [24], which can be used to prove refinement for fine-grained concurrent algorithms but, in contrast to ReLoC, it can only be used to reason about *first-order* programs. Unlike ReLoC, the logic of *loc. cit.* has not been mechanised.

RHOL [1] is a relational higher-order logic for reasoning about relational properties of programs in (a terminating variant of) PCF. The main judgement of the logic allows one to prove that a relational formula φ holds for two expressions (not necessarily of the same type). The authors demonstrate the soundness of the logic and show how to embed a number of type systems into their framework. In our case, we take a more type-directed approach and relate terms of the same type only. However, we can relate terms of different type by relating them at some type variable α and picking a suitable interpretation for it to substitute for α in the environment Δ. The authors of RHOL demonstrate proofs of various relational properties (like those provided by the systems that they embed); in our work we consider only one (family of) relation(s), namely the logical relation for contextual refinement. On the other hand, 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.

Earlier work has also included relational logics for (higher-order) programming languages with mutable state, but no concurrency. Relational Hoare logic [9] and Relational Separation logic [29] can be used for reasoning about relational properties for first-order imperative programs, and they have inspired several extensions. Relational Hoare Type Theory [25] is a dependent type theory for specification and verification of information flow and access control properties of higher-order programs with dynamically allocated mutable first-order state, defined in Coq, A relational logic for a sequential class-based language with dynamically allocated objects has been introduced by Banerjee *et al.* [7]. The relational logic is based on region logic [8], a first-order logic, which is amenable to SMT-based automation. The relational logic is aimed at proving refinement and noninterference. In contrast, we focus on reasoning about refinement, but also treat concurrent programs and higher-order store, and we provide tool support for interactive verification.

9 Conclusion and further work

We have presented ReLoC, a relational logic for abstract reasoning about contextual refinement of fine-grained concurrent higher-order imperative programs. ReLoC enables modular proofs due to the first-class status of refinement judgements and the support of a novel form of logically atomic relational specifications. We have provided a mechanisation of our logic in Coq, which does not just contain a proof of soundness, but also tactics for interactively carrying out refinements proofs. We have used these tactics to mechanise several examples, which demonstrates the practicality and modularity of our logic.

One possible direction of further work is increased support for showing refinements of programs that involve helping through side channels. We have formalised (in Coq) a refinement of a coarse-grained concurrent stack by a stack with helping; however, that proof requires us to appeal to the interpretation of the logical relation judgements in the Iris logic and is thus perhaps not as abstract as one could hope for, although parts of the proof are still carried out in ReLoC. Another interesting problem that is not addressed by the calculus is reasoning that involves speculating on possible values in the program or in the heap. We are also interested in exploring extensions of ReLoC to richer type-and-effect systems and cross-language logical relations.

Acknowledgments

The authors thank Amin Timany and the anonymous reviewers for their useful comments. This research was supported in part by the STW project 14319, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO); and the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *PACMPL* 1, ICFP (2017), 21:1–21:29.
[2] Amal Ahmed. 2004. *Semantics of types for mutable state.* Ph.D. Dissertation. Princeton University.
[3] Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP (LNCS)*, Vol. 3924. 69–83.
[4] Amal Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A stratified semantics of general references embeddable in higher-order logic. In *LICS*. 75–86.
[5] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *POPL*. 340–353.
[6] Andrew Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683.
[7] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2016. Relational logic with framing and hypotheses. In *FSTTCS (LIPIcs)*, Vol. 65. 11:1–11:16.
[8] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2013. Local reasoning for global invariants, part I: region logic. *JACM* 60, 3 (2013), 18:1–18:56.
[9] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *POPL*. 14–25.
[10] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *POPL*. 119–132.
[11] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP (LNCS)*, Vol. 8586. 207–231.
[12] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP (LNCS)*, Vol. 6183. 504–528.
[13] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical step-indexed logical relations. In *LICS*. 71–80.
[14] Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *JFP* 22, 4-5 (2012), 477–528.
[15] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *POPL*. 185–198.
[16] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. Appendix and Coq development of *ReLoC: A mechanised relational logic for fine-grained concurrency*. (2018). Available at http://cs.ru.nl/~dfrumin/reloc/.
[17] Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an assertion language for mutable data structures. In *POPL*. 14–26.
[18] Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *POPL*. 271–282.
[19] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.
[20] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Submitted for publication* (2017).
[21] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. 637–650.
[22] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *ESOP (LNCS)*, Vol. 10201. 696–723.
[23] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217.
[24] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *PLDI*. 459–470.
[25] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent type theory for verification of information flow and access control policies. *TOPLAS* 35, 2 (2013), 6:1–6:41.
[26] Gordon D. Plotkin and Martín Abadi. 1993. A logic for parametric polymorphism. In *TLCA (LNCS)*, Vol. 664. 361–375.
[27] Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS)*, Vol. 8410. 149–168.
[28] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. 377–390.
[29] Hongseok Yang. 2007. Relational separation logic. *TCS* 375, 1-3 (2007), 308–334.