ReLoC technical report

Dan Frumin   Robbert Krebbers   Lars Birkedal

April 28, 2018

Abstract

The aim of this document is to formally describe the relational logic ReLoC used for proving contextual refinement of higher-order stateful concurrent programs. The logic is based on higher-order separation logic Iris, and has been fully formalized in Coq. The repository, containing all the formalized results and examples present in this text, can be found at `https://gitlab.mpi-sws.org/dfrumin/logrel-conc`.

Contents

1 Introduction

Reasoning about equivalence of programs is an old problem in semantics of programming languages, with applications many applications including program verification and compilation. Possibly, the most widely used notion of program equivalence is *contextual equivalence*, which states that programs are equivalent if they exhibit the same observable (termination) behaviour under any contexts. However, proving contextual equivalence of two given programs is tricky, as it involves considering *all the possible* program contexts. One of the techniques proposed to resolve this are *logical relations* (originally used to show safety of the typing systems). In order to handle equivalence proofs in the presence of advance PL features such as higher order store, recursive types, and concurrency, the proof method of *Kripke logical relations* have been proposed, in which the truth value of "relatedness" may vary in different worlds.

The aim of this work is to describe a system for formal reasoning about program equivalence through logical relations for System F with state, existential types, and concurrency primitives. The system is built on top of the powerful higher-order separation logic called *Iris*. This allows the user to leverage advanced features of Iris, such as ghost state and invariants.

The current ReLoC library is developed by Dan Frumin and Robbert Krebbers, and is based the earlier formalisation of Amin Timany, Robbert Krebbers, and Lars Birkedal. The authors express their gratitude to Lars Birkedal, Amin Timany and many other people involved in the Iris project.

2 The object language

The programming language for which we construct the relational model is System F with iso-recursive types, references, and concurrency primitives CAS and fork. We abbreviate it as System $\mathsf{F}_{\mu,\mathsf{ref},\mathsf{conc},\exists}$.

2.1 Syntax and operational semantics

The expressions, values, and evaluation contexts for the language are defined below. We write closed(X, e) to denote that all free variables in the expression e are elements of the set X. By closed(e) we denote closed$(\emptyset, e)$.

Syntax:

$$
\begin{array}{llcll}
\text{Values} & v_1, v_2 \in \mathit{Val} & ::= & () \mid l \in \mathit{Loc} \mid n \in \mathbb{N} \mid b \in \mathbb{B} \mid (v_1, v_2) & \\
& & & \mid \texttt{inl}\ v_1 \mid \texttt{inr}\ v_1 \mid \texttt{fold}\ v_1 \mid \Lambda.e_1 & \text{where closed}(\emptyset, e_1) \\
& & & \mid \texttt{pack}\ v_1 \mid \mathsf{rec}\ f\ x = e_1 & \text{where closed}(\{x, f\}, e_1) \\
\text{Expressions} & e_1, e_2 \in \mathit{Expr} & ::= & v_1 \mid x \in \mathit{Var} \mid e_1\, e_2 \mid e_1 \oplus e_2 \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 & \\
& & & \mid \pi_1\ e_1 \mid \pi_2\ e_1 \mid \texttt{case}(e_1, e_2, e_3) \mid \texttt{fold}\ e & \\
& & & \mid \texttt{unfold}\ e \mid \Lambda.e \mid e\ [] \mid \texttt{pack}\ e & \\
& & & \mid \texttt{unpack}\ e_1\ \texttt{in}\ e_2 \mid \texttt{fork}\ \{e\} \mid \texttt{ref}(e) & \\
& & & \mid\ !\,e \mid e_1 \leftarrow e_2 \mid \texttt{CAS}(e_1, e_2, e_3) \mid \ \ldots & \\
\text{Evaluation contexts} & K \in \mathit{ECtx} & ::= & [\bullet] \mid K\, e \mid v\, K \mid K\ [] \mid (K, e) \mid (v, K) \mid K \oplus e & \\
& & & \mid v \oplus K \mid \pi_1 K \mid \pi_2 K \mid \texttt{inl}\ K \mid \texttt{inr}\ K \mid \texttt{case}(K, e_2, e_3) & \\
& & & \mid \texttt{if}\, K\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid \texttt{fold}\, K \mid \texttt{unfold}\, K \mid \texttt{pack}\, K & \\
& & & \mid \texttt{unpack}\ K\ \texttt{in}\ e \mid \texttt{ref}(K) \mid\ !\ K \mid K \leftarrow e \mid \mathit{Val} \leftarrow K & \\
& & & \mid \texttt{CAS}(K, e_2, e_3) \mid \texttt{CAS}(v, K, e_3) \mid \texttt{CAS}(v_1, v_2, K) & \\
\text{Configurations} & \sigma \in \mathit{State} & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} & \\
& T \in \mathit{ThreadPool} & = & \mathit{List}\ \mathit{Expr} & \\
& \rho \in \mathit{Cfg} & = & \mathit{ThreadPool} \times \mathit{State} &
\end{array}
$$

We make use the following derived forms.

Derived forms:

$$
\begin{array}{lcl}
\lambda x.\, e & := & \mathsf{rec}\ ()\ x = e \\
\texttt{let}\, x = t\, \texttt{in}\, e & := & (\lambda x.\, e)\ t \\
e_1; e_2 & := & (\lambda ().\, e_2)\ e_1
\end{array}
$$

Dynamics. The operational semantics are presented in three levels. Head reductions are standard call-by-value reduction rules for λ-calculus with store and concurrency. The head reductions are lifted to reductions under evaluation contexts (primitive steps), and those in turn are lifted to reductions of configurations (thread pool steps), which provide a concurrent interleaving semantics for the language.

Head Reductions: $(e, \sigma) \to_{\mathsf{h}} (e', \sigma', T)$

BETA
$$\frac{\text{closed}(\{x, f\}, e)}{((\mathsf{rec}\ f\ x = e)\ v, \sigma) \to_{\mathsf{h}} (e[v/x][\mathsf{rec}\ f\ x = e/f], \sigma, [])}$$

PROJ
$$(\pi_i\ (v_1, v_2), \sigma) \to_{\mathsf{h}} (v_i, \sigma, [])$$

CASE-INL
$$(\mathtt{case}(\mathtt{inl}\ v, e_1, e_2), \sigma) \to_{\mathsf{h}} (e_1\ v, \sigma, [])$$

CASE-INR
$$(\mathtt{case}(\mathtt{inr}\ v, e_1, e_2), \sigma) \to_{\mathsf{h}} (e_2\ v, \sigma, [])$$

BINOP
$$\frac{[\![\oplus]\!](v_1, v_2) = v_3}{(v_1 \oplus v_2, \sigma) \to_{\mathsf{h}} (v_3, \sigma, [])}$$

IF-TRUE
$$(\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, \sigma) \to_{\mathsf{h}} (e_1, \sigma, [])$$

IF-FALSE
$$(\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, \sigma) \to_{\mathsf{h}} (e_2, \sigma, [])$$

UNFOLD
$$(\mathtt{unfold}\ (\mathtt{fold}\ v), \sigma) \to_{\mathsf{h}} (v, \sigma, [])$$

TBETA
$$\frac{\text{closed}(\emptyset, e)}{((\Lambda.e)\ [], \sigma) \to_{\mathsf{h}} (e, \sigma, [])}$$

UNPACK
$$(\mathtt{unpack}\ (\mathtt{pack}\ v)\ \mathtt{in}\ e, \sigma) \to_{\mathsf{h}} (e\ v, \sigma, [])$$

FORK
$$(\mathtt{fork}\ \{e\}, \sigma) \to_{\mathsf{h}} ((), \sigma, [e])$$

ALLOC
$$\frac{\sigma(l) = \bot}{(\mathtt{ref}(v), \sigma) \to_{\mathsf{h}} (l, \sigma[l := v], [])}$$

LOAD
$$\frac{\sigma(l) = v}{(!\,l, \sigma) \to_{\mathsf{h}} (v, \sigma, [])}$$

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

CAS-FAIL
$$\frac{\sigma(l) \neq v_1}{(\mathtt{CAS}(l, v_1, v_2), \sigma) \to_{\mathsf{h}} (\mathtt{false}, \sigma, [])}$$

CAS-SUC
$$\frac{\sigma(l) = v_1}{(\mathtt{CAS}(l, v_1, v_2), \sigma) \to_{\mathsf{h}} (\mathtt{true}, \sigma[l := v_2], [])}$$

Primitive Reductions: $(e, \sigma) \to (e', \sigma', \vec{e_f})$

PRIM-STEP
$$\frac{(e, \sigma) \to_{\mathsf{h}} (e', \sigma', \vec{e_f})}{(K[e], \sigma) \to (K[e'], \sigma', \vec{e_f})}$$

Thread-pool Reductions: $\rho \rightarrow_{\mathsf{tp}} \rho'$

$$\frac{T(i) = e \qquad (e,\sigma) \rightarrow (e',\sigma',\vec{e_f})}{(T,\sigma) \rightarrow_{\mathsf{tp}} (T\,[i \leftarrow e'] +\!\!+ \vec{e_f}, \sigma')}\ \textsc{tp-step}$$

Remark 2.1. *Note that we don't have typing annotations in the syntax. In particular, we write $\Lambda.e$ instead of $\Lambda\alpha.e$, and similarly for type application. Correspondingly, for the purposes of the implementation, in the typing discipline we use de Bruijn indices for type variables. However, for the term variables we still employ explicit substitution in our formalisation. The reason why we can get away with this is that our semantics is call-by-value, as such we only substitute values for variables, thus rendering the capture-avoidance problem irrelevant.*

Definition 2.2. *An expression e is said to be (strongly)* atomic *if it reduces to a value in one step:*

$$\mathrm{atomic}(e) \triangleq \forall \sigma\, \sigma'\, e'\, \vec{e_f},\ (e,\sigma) \rightarrow (e',\sigma',\vec{e_f}) \implies e' \in \mathit{Val}.$$

Pure reductions. Our calculus is able to uniformly handle symbolic executions that do not change the physical state. Such reductions are called *pure*.

Definition 2.3. *A reduction $e \rightarrow e'$ is* pure *if e is reducible in every state, and all reductions from e do not change the state and end up in e'. Formally,*

$$\forall \sigma. \mathrm{reducible}(e,\sigma) \wedge \forall \sigma \sigma_2 \forall e_2.(e,\sigma) \rightarrow (e_2,\sigma') \implies \sigma_2 = \sigma \wedge e_2 = e'$$

We write $e \rightarrow_{\mathsf{pure}} e'$ for reduction which is pure.

We have the following rules for $\rightarrow_{\mathsf{pure}}$ (in Coq the rules below are implemented via type classes).

Pure executions: $e \rightarrow_{\mathsf{pure}} e'$

$$\frac{[\![\oplus]\!](v_1,v_2) = v_3}{v_1 \oplus v_2 \rightarrow_{\mathsf{pure}} v_3}\ \textsc{pure-binop} \qquad \frac{\mathrm{closed}(\{f,x\},e)}{(\mathsf{rec}\ f\ x = e)\ v \rightarrow_{\mathsf{pure}} e[v/x][(\mathsf{rec}\ f\ x = e)/f])}\ \textsc{pure-rec}$$

$$\textsc{pure-proj-i}\quad \pi_i(v_1,v_2) \rightarrow_{\mathsf{pure}} v_i \qquad \textsc{pure-unfold}\quad \mathtt{unfold}\ (\mathtt{fold}\ v) \rightarrow_{\mathsf{pure}} v$$

$$\textsc{pure-unpack}\quad \mathtt{unpack}\ (\mathtt{pack}\ v)\ \mathtt{in}\ e \rightarrow_{\mathsf{pure}} e\ v \qquad \textsc{pure-if-true}\quad \mathtt{if\,true\,then}\,e_1\,\mathtt{else}\,e_2 \rightarrow_{\mathsf{pure}} e_1$$

$$\textsc{pure-if-false}\quad \mathtt{if\,false\,then}\,e_1\,\mathtt{else}\,e_2 \rightarrow_{\mathsf{pure}} e_2 \qquad \textsc{pure-case-inl}\quad \mathtt{case}(\mathtt{inl}\ v, e_1, e_2) \rightarrow_{\mathsf{pure}} e_1\ v$$

$$\textsc{pure-case-inr}\quad \mathtt{case}(\mathtt{inr}\ v, e_1, e_2) \rightarrow_{\mathsf{pure}} e_2\ v \qquad \textsc{pure-tlam}\quad (\Lambda.e)\ [] \rightarrow_{\mathsf{pure}} e \qquad \frac{e_1 \rightarrow_{\mathsf{pure}} e_2}{K[e_1] \rightarrow_{\mathsf{pure}} K[e_2]}\ \textsc{pure-exec-fill}$$

2.2 The type system

Remark 2.4. *The syntax and the typing differ from the ones in the paper. Here we use explicit names in the language of terms and De Bruijn indices in the language of types. This is also the case in the Coq formalisation.*

Types:

$$\begin{aligned}\tau \in \mathit{Type} \quad ::= \quad & \mathbf{1} \mid \mathbf{2} \mid \mathbf{N} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \textsf{\textbf{ref}}\ \tau \\ & \mid \mu\tau \mid \forall\tau \mid \exists\tau \mid i \in \mathit{TVar}\end{aligned}$$

The typing judgements are of the form $\Gamma \vdash e : \tau$. Additionally, there is a judgement $\mathrm{EqType}(\tau)$ stating that the type τ supports (structural) equality testing.

Types with structural equality: $\mathrm{EqType}(\tau)$

$$\textsc{EqTUnit}\ \ \frac{}{\mathrm{EqType}(\mathbf{1})} \qquad \textsc{EqTNat}\ \ \frac{}{\mathrm{EqType}(\mathbf{N})} \qquad \textsc{EqTBool}\ \ \frac{}{\mathrm{EqType}(\mathbf{2})} \qquad \textsc{EqTProd}\ \ \frac{\mathrm{EqType}(\tau) \qquad \mathrm{EqType}(\tau')}{\mathrm{EqType}(\tau \times \tau')}$$

$$\textsc{EqTSum}\ \ \frac{\mathrm{EqType}(\tau) \qquad \mathrm{EqType}(\tau')}{\mathrm{EqType}(\tau + \tau')}$$

Typing judgements: $\Gamma \vdash e : \tau$

$$\text{VAR-TYPED}\ \ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{UNIT-TYPED}\ \ \frac{}{\Gamma \vdash () : \mathbf{1}} \qquad \text{NAT-TYPED}\ \ \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathbf{N}} \qquad \text{BOOL-TYPED}\ \ \frac{b \in \mathbb{B}}{\Gamma \vdash b : \mathbf{2}}$$

$$\text{BINOP-TYPED-NAT}\ \ \frac{\Gamma \vdash e_1 : \mathbf{N} \qquad \Gamma \vdash e_2 : \mathbf{N} \qquad \oplus \text{ operates on natural numbers}}{\Gamma \vdash e_1 \oplus e_2 : \mathrm{typeof}(\oplus)}$$

$$\text{BINOP-TYPED-BOOL}\ \ \frac{\Gamma \vdash e_1 : \mathbf{2} \qquad \Gamma \vdash e_2 : \mathbf{2} \qquad \oplus \text{ operates on booleans}}{\Gamma \vdash e_1 \oplus e_2 : \mathrm{typeof}(\oplus)}$$

$$\text{REFEQ-TYPED}\ \ \frac{\Gamma \vdash e_1 : \textsf{\textbf{ref}}\ \tau \qquad \Gamma \vdash e_2 : \textsf{\textbf{ref}}\ \tau}{\Gamma \vdash e_1 == e_2 : \mathbf{2}} \qquad \text{PAIR-TYPED}\ \ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\text{PROJ-TYPED}\ \ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i} \qquad \text{INJL-TYPED}\ \ \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{inl}\ e : \tau_1 + \tau_2} \qquad \text{INJR-TYPED}\ \ \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{inr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \qquad \Gamma \vdash e_1 : \tau_1 \to \tau_3 \qquad \Gamma \vdash e_2 : \tau_2 \to \tau_3}{\Gamma \vdash \mathtt{case}(e_0, e_1, e_2) : \tau_3}\ \text{CASE-TYPED}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{2} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\, e_0\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 : \tau}\ \text{IF-TYPED} \qquad \frac{x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{rec}\ f\ x = e : \tau_1 \to \tau_2}\ \text{REC-TYPED}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{APP-TYPED} \qquad \frac{(+1)\ \langle\$\rangle\ \Gamma \vdash e : \tau}{\Gamma \vdash \Lambda.e : \forall\tau}\ \text{TLAM-TYPED} \qquad \frac{\Gamma \vdash e : \forall\tau}{\Gamma \vdash e\ [] : \tau[\sigma/]}\ \text{TAPP-TYPED}$$

$$\frac{\Gamma \vdash e : \tau[\mu\tau/]}{\Gamma \vdash \mathtt{fold}\ e : \mu\tau}\ \text{FOLD-TYPED} \qquad \frac{\Gamma \vdash e : \mu\tau}{\Gamma \vdash \mathtt{unfold}\ e : \tau[\mu\tau/]}\ \text{UNFOLD-TYPED} \qquad \frac{\Gamma \vdash e : \tau[\sigma/]}{\Gamma \vdash \mathtt{pack}\ e : \exists\tau}\ \text{TPACK-TYPED}$$

$$\frac{\Gamma \vdash e_1 : \exists\tau_1 \qquad (+1)\ \langle\$\rangle\ \Gamma \vdash e_2 : \tau_1 \to (+1)\ \langle\$\rangle\ \tau_2}{\Gamma \vdash \mathtt{unpack}\ e_1\ \mathtt{in}\ e_2 : \tau_2}\ \text{TUNPACK-TYPED} \qquad \frac{\Gamma \vdash e : \mathbf{1}}{\Gamma \vdash \mathtt{fork}\ \{e\} : \mathbf{1}}\ \text{FORK-TYPED}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{ref}(e) : \mathbf{ref}\ \tau}\ \text{ALLOC-TYPED} \qquad \frac{\Gamma \vdash e : \mathbf{ref}\ \tau}{\Gamma \vdash\ !e : \tau}\ \text{LOAD-TYPED} \qquad \frac{\Gamma \vdash e_1 : \mathbf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \leftarrow e_2 : \mathbf{1}}\ \text{STORE-TYPED}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau \qquad \mathrm{EqType}(\tau)}{\Gamma \vdash \mathtt{CAS}(e_1, e_2, e_3) : \mathbf{2}}\ \text{CAS-TYPED}$$

2.3 Contextual equivalence and contextual refinement

Contextual equivalence is a formalisation of an important notion of *program equivalence*. Intuitively, two programs e_1 and e_2 are contextually equivalent if for any client p, $p(e_1)$ terminates to the same observable value as $p(e_2)$. A directed variant of contextual equivalence is *contextual refinement*. To define it formally, we employ the notion of a program context.

Program contexts:

$$\begin{aligned}
\mathcal{C} \in \mathit{Ctx} \quad ::= \quad & [\bullet] \mid \mathsf{rec}\ f\ x = \mathcal{C} \mid \mathcal{C}\ e_2 \mid e_1\ \mathcal{C} \mid (\mathcal{C}, e_2) \mid (e_1, \mathcal{C}) \mid \pi_1\ \mathcal{C} \mid \pi_2\ \mathcal{C} \mid \mathtt{inl}\ \mathcal{C} \mid \mathtt{inr}\ \mathcal{C} \\
& \mid \mathtt{case}(\mathcal{C}, e_1, e_2) \mid \mathtt{case}(e_0, \mathcal{C}, e_2) \mid \mathtt{case}(e_0, e_1, \mathcal{C}) \mid \mathcal{C} \oplus e_2 \mid e_1 \oplus \mathcal{C} \\
& \mid \mathtt{if}\, \mathcal{C}\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2 \mid \mathtt{if}\, e_0\, \mathtt{then}\, \mathcal{C}\, \mathtt{else}\, e_2 \mid \mathtt{if}\, e_0\, \mathtt{then}\, e_1\, \mathtt{else}\, \mathcal{C} \mid \mathtt{fold}\ \mathcal{C} \mid \mathtt{unfold}\ \mathcal{C} \\
& \mid \Lambda.\mathcal{C} \mid \mathcal{C}\ [] \mid \mathtt{pack}\ \mathcal{C} \mid \mathtt{unpack}\ \mathcal{C}\ \mathtt{in}\ e_2 \mid \mathtt{unpack}\ e_1\ \mathtt{in}\ \mathcal{C} \\
& \mid \mathtt{fork}\ \{\mathcal{C}\} \mid \mathtt{ref}(\mathcal{C}) \mid\ !\mathcal{C} \mid \mathcal{C} \leftarrow e_2 \mid e_1 \leftarrow \mathcal{C} \mid \mathtt{CAS}(\mathcal{C}, e_1, e_2) \mid \mathtt{CAS}(e_0, \mathcal{C}, e_2) \mid \mathtt{CAS}(e_0, e_1, \mathcal{C})
\end{aligned}$$

Notice that $\mathit{ECtx} \subsetneq \mathit{Ctx}$, i.e. every evaluation context is a program context as well. However, unlike evaluation contexts, a hole in program contexts can appear under in any position – including under a lambda. In particular, that means that the substitution of expression e for a hole – $\mathcal{C}[e]$ – can capture free variables in e.

For the purposes of contextual refinement we only consider context with a hole that "fits" a certain type. We write

$$[\,\mathcal{C}\,] : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$$

for a judgement stating that $\mathcal{C}$ is a typed context with the hole of type σ in context Δ, returning an expression of type τ in context Γ.

Context typing: $[\,\mathcal{C}\,] : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$

$$[\,[\bullet]\,] : (\Gamma \vdash \tau) \Rightarrow (\Gamma \vdash \tau) \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (x : \tau, f : (\tau \to \tau'), \Gamma \vdash \tau')}{[\,\mathsf{rec}\ f\ x = \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau \to \tau')}$$

$$\frac{\Gamma \vdash e_2 : \tau \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau \to \tau')}{[\,\mathcal{C}\ e_2\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}{[\,e_1\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')}$$

$$\frac{\Gamma \vdash e_2 : \tau' \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}{[\,(\mathcal{C}, e_2)\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau \times \tau')}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')}{[\,(e_1, \mathcal{C})\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau \times \tau')} \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau \times \tau')}{[\,\pi_1\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau \times \tau')}{[\,\pi_2\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')} \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}{[\,\mathtt{inl}\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau + \tau')}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')}{[\,\mathtt{inr}\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau + \tau')}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau' \qquad \Gamma \vdash e_2 : \tau_2 \to \tau' \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau_1 + \tau_2)}{[\,\mathtt{case}(\mathcal{C}, e_1, e_2)\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \qquad \Gamma \vdash e_2 : \tau_2 \to \tau' \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau_1 \to \tau')}{[\,\mathtt{case}(e_0, \mathcal{C}, e_2)\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')}$$

$$\frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \qquad \Gamma \vdash e_1 : \tau_1 \to \tau' \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau_2 \to \tau')}{[\,\mathtt{case}(e_0, e_1, \mathcal{C})\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau')}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{2}) \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{[\,\texttt{if}\,\mathcal{C}\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{2} \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau) \qquad \Gamma \vdash e_2 : \tau}{[\,\texttt{if}\,e_0\,\texttt{then}\,\mathcal{C}\,\texttt{else}\,e_2\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{2} \qquad \Gamma \vdash e_1 : \tau \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}{[\,\texttt{if}\,e_0\,\texttt{then}\,e_1\,\texttt{else}\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{N}) \qquad \Gamma \vdash e_2 : \mathbf{N}}{[\,\mathcal{C} \oplus e_2\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{N})}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{N} \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{N})}{[\,e_1 \oplus \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{N})} \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau[\mu\tau/])}{[\,\texttt{fold}\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mu\tau)}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mu\tau)}{[\,\texttt{unfold}\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau[\mu\tau/])} \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (+1 \langle\$\rangle\, \Gamma \vdash \tau)}{[\,\Lambda.\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \forall\tau)}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \forall\tau)}{[\,\mathcal{C}\ []\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau[\tau'/])} \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau[\tau'/])}{[\,\texttt{pack}\ \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \exists\tau)}$$

$$\frac{+1 \langle\$\rangle\, \Gamma \vdash e_2 : \tau \to +1 \langle\$\rangle\, \tau_2 \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \exists\tau)}{[\,\texttt{unpack}\,\mathcal{C}\,\texttt{in}\,e_2\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \exists\tau \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (+1 \langle\$\rangle\, \Gamma \vdash \tau \to +1 \langle\$\rangle\, \tau_2)}{[\,\texttt{unpack}\,e_1\,\texttt{in}\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau_2)}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{1})}{[\,\texttt{fork}\ \{\mathcal{C}\}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{1})} \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}{[\,\texttt{ref}(\mathcal{C})\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \textsf{\textbf{ref}}\ \tau)}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \textsf{\textbf{ref}}\ \tau)}{[\,!\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)} \qquad \frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \textsf{\textbf{ref}}\ \tau) \qquad \Gamma \vdash e_2 : \tau}{[\,\mathcal{C} \leftarrow e_2\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{1})}$$

$$\frac{\Gamma \vdash e_1 : \textsf{\textbf{ref}}\ \tau \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau)}{[\,e_1 \leftarrow \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{1})}$$

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \textsf{\textbf{ref}}\ \tau) \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \mathrm{EqType}(\tau)}{[\,\texttt{CAS}(\mathcal{C}, e_1, e_2)\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{2})}$$

$$\frac{\Gamma \vdash e_0 : \textsf{ref}\ \tau \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau) \qquad \Gamma \vdash e_2 : \tau \qquad \mathrm{EqType}(\tau)}{[\,\texttt{CAS}(e_0, \mathcal{C}, e_2)\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{2})}$$

$$\frac{\Gamma \vdash e_0 : \textsf{ref}\ \tau \qquad \Gamma \vdash e_1 : \tau \qquad [\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau) \qquad \mathrm{EqType}(\tau)}{[\,\texttt{CAS}(e_0, e_1, \mathcal{C})\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \mathbf{2})}$$

The validity of the typing rules for contexts is supported by the following lemma.

Lemma 2.5. *If* $\Gamma \vdash e : \tau$ *and* $[\,\mathcal{C}\,] : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$*, then* $\Gamma' \vdash \mathcal{C}[e] : \tau'$*.*

Proof. By induction on the derivation of the context typing. □

Informally, contextual refinement should say that if we embed the first expression into any (well-typed) program context, and the resulting program terminates to an observable value v, then plugging the second expression into the same context will also result in the value v.

Definition 2.6. *A type* τ *is* observable*, denoted as* $\mathrm{ObsType}(\tau)$*, if it is either a base type (naturals, booleans), or obtained from the observable types by means of product or sum types.*

Definition 2.7. *We say that* e_1 contextually refines e_2 *at type* τ *in context* Γ *– denoted as* $\Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$ *– if* e_1 *and* e_2 *terminate to the same observable value under any suitable program context. Formally,*

$$\begin{aligned} &\Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau \triangleq \forall \tau',\ \mathrm{ObsType}(\tau') \implies \\ &\quad \forall [\,\mathcal{C}\,] : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \tau'),\ \forall v\, T\, \sigma,\ ([\mathcal{C}[e_1]], \emptyset) \to_{\mathsf{tp}}^{*} ([v] \mathbin{+\!\!+} T, \sigma) \implies \\ &\qquad\qquad\qquad \exists T'\, \sigma',\ ([\mathcal{C}[e_2]], \emptyset) \to_{\mathsf{tp}}^{*} ([v] \mathbin{+\!\!+} T', \sigma') \end{aligned}$$

Note that we only quantify over the typed contexts with the *observable* return type. If we allow $\mathcal{C}$ to be quantified over arbitrary program contexts, then the notion of contextual refinement will be too fine for our purpose. Consider, for instance, a context $\mathcal{C} := \lambda x.\,[\,\bullet\,]$. This context has a type $[\,\mathcal{C}\,] : ([x : \mathbf{2}] \vdash \tau) \Rightarrow (\emptyset \vdash \mathbf{2} \to \tau)$. If we were to allow contexts of such type in Definition 2.7, then the notion of contextual refinement will collapse to syntactic equality: $\mathcal{C}[e]$ always terminates to a closure $\lambda x.\, e$, and $\mathcal{C}[e']$ terminates to the same value iff $e = e'$.

The notion of contextual refinement is hard to work with directly due to the fact that what we need to show an instance of contextual refinement, we need to prove something for an arbitrary program context. As we will see in later sections, a stronger notion of logical refinement is much more suitable for deductive reasoning.

Notes on formalisation

The syntax and the dynamics of $\mathsf{F}_{\mu,\mathsf{ref},\mathsf{conc},\exists}$ are defined in `F_mu_ref_conc/lang.v`. The $\to_{\mathsf{pure}}$ instances are defined in `F_mu_ref_conc/pureexec.v`. Definitions of typed contexts and contextual refinement are formalised in `F_mu_ref_conc/context_refinement.v`.

In the `F_mu_ref_conc` directory of the Coq formalisation one can also find modules containing lemmas about binders, substitution, as well as some Coq-specific things (notation for the object language, reified syntax for automatically solving questions of closedness and atomicity) and Iris-specific things (WP-calculus for $\mathsf{F}_{\mu,\mathsf{ref},\mathsf{conc},\exists}$ with the adequacy proofs).

The binders in the term language are represented using explicit names. For our purposes it is actually fine and we avoid any free variable capturing issues because the a beta reduction can be performed only if the argument is a value (and consequently is closed). This approach actually gave us some speedup, compared to using De Bruijn indices and σ-calculus as implemented by `autosubst` [5]. On the level of types, however, we still employ De Bruijn indices and `autosubst`.

The $\to_{\mathsf{pure}}$ judgement is implemented as a type class `PureExec P e e'` in Iris, where `P` is a (pure) proposition describing conditions under which `e` can be reduced to `e'` – for example `P` in PURE-BINOP ensures that the binary operation is defined on the arguments.

3 The calculus

The basic calculus of logical relations is based on the higher-order separation logic *Iris*, and is enriched with propositions of the form

$$\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim e' : \tau \qquad (1)$$

where e and e' are expressions, τ is a type, Γ is a typing environment (that is, it assigns types to variable names), Δ is an interpretation for type variables (that is, it assigns relations to type variables), and $\mathcal{E}$ is an invariant mask.

Intuitively, the proposition in Equation (1) states that for e and e' are related at type τ, in which free type variables are interpreted using Δ. The role of the mask $\mathcal{E}$ is two-fold. On one hand, it keeps track of the invariants that can still be opened, preventing the issues of reentrancy. On the other hand, if the mask is not $\top$, then it signifies that a symbolic execution step on the left hand side has been taken; it then prevents further reductions on the left hand side until all the invariants has been restored

We use the following shorthand:

- $\Delta \mid \Gamma \models e \precsim e' : \tau \triangleq \Delta \mid \Gamma \models_{\top} e \precsim e' : \tau$

The rules themselves are presented in Sections 3.1 and 3.2. Below we provide some comments.

Value interpretation. The value interpretation $[\![\tau]\!]_{\Delta}(v_1, v_2)$ is defined inductively on the structure of the type. Usually, the user of the logic would not see these kind of propositions in their proofs, apart from some places where they are crucial, for instance during the representation independence proofs.

Structural rules. FUPD-LOGREL is the rule for opening invariants around the masked refinement judgement. The rule LR-CLOSURE is crucial for reasoning about higher-order programs.

Symbolic execution. To perform actual refinement proofs in the system we need to be able to symbolically execute expressions under a given type. For reductions that do not change the state the rules are LR-PURE-L, LR-PURE-L-MASKED, and LR-PURE-R. The rules witness the fact that the refinement judgements are closed under reductions. There are general rules for pure and stateful reductions on both sides.

The rules for symbolically executing stateful reductions are more involved. Consider, for instance, the following rule for performing a store operation on the left hand side.

LR-STORE-L'

$$\frac{\triangleright l \mapsto_{\mathsf{i}} v' \qquad (l \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models K[()] \precsim e' : \tau)}{\Delta \mid \Gamma \models K[l \leftarrow v] \precsim e' : \tau}$$

This rule is suitable for symbolic executing in sequential programs. However, consider what happens when $l \mapsto_{\mathsf{i}} v$ belongs to some invariant I that links together l in the target program with l' in the source program; for instance $\boxed{\exists n. l \mapsto_{\mathsf{i}} n * l' \mapsto_{\mathsf{s}} n}^{\mathcal{N}}$. If we want to prove the refinement

$$\Delta \mid \Gamma \models l \leftarrow m \precsim l' \leftarrow m : \mathbf{1}, \tag{2}$$

then we first apply FUPD-LOGREL to get a necessary view shift to be able to open the invariant. After we open the invariant and apply LR-STORE-L' we are left with:

$$^{\top \setminus \mathcal{N}}\Rrightarrow^{\top} (l \mapsto_{\mathsf{i}} m \twoheadrightarrow \Delta \mid \Gamma \models () \precsim l' \leftarrow m : \mathbf{1})$$

Which means that we have to immediately close the invariant without being able to perform a symbolic execution step on the right hand side – this will not work because the invariant is broken at this stage. To circumvent this limitation we propose a slightly different rule LR-STORE-L and a corresponding right-hand side rule LR-STORE-R.

LR-STORE-L

$$\frac{^{\top}\Rrightarrow^{\mathcal{E}} \exists v'. \triangleright l \mapsto_{\mathsf{i}} v' * \triangleright (l \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[()] \precsim e' : \tau)}{\Delta \mid \Gamma \models K[l \leftarrow v] \precsim e' : \tau}$$

LR-STORE-R

$$\frac{l \mapsto_{\mathsf{s}} v' \qquad (l \mapsto_{\mathsf{s}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[()] : \tau) \qquad \uparrow \mathsf{logrelN} \subseteq \mathcal{E}}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[l \leftarrow v] : \tau}$$

Using those two rules we can prove the refinement $\Delta \mid \Gamma \models l \leftarrow m \precsim l' \leftarrow m : \mathbf{1}$ as follows: first we apply LR-STORE-L, and open up the invariant. This gets rid of the view shift, which allows us to frame $l \mapsto_{\mathsf{i}} n$, leaving us with $l' \mapsto_{\mathsf{s}} n$. It remains to prove $\Delta \mid \Gamma \models_{\top \setminus \mathcal{N}} () \precsim l' \leftarrow m : \mathbf{1}$ from $l \mapsto_{\mathsf{i}} m$. For this we apply LR-STORE-R resulting in a proof obligation

$$l \mapsto_{\mathsf{i}} m * l' \mapsto_{\mathsf{s}} m \vdash \Delta \mid \Gamma \models_{\top \setminus \mathcal{N}} () \precsim () : \mathbf{1}$$

It can be proven by applying FUPD-LOGREL, closing the invariant, and applying the compatibility lemma for the unit type. The full proof derivation can be found in Section 3.

The general rules for stateful reductions are LR-WP-ATOMIC-L for atomic reductions and LR-WP-L for general reductions.

The symbolic execution rules for the reductions on the right hand side are simpler than those for the left hand side, and they can be performed under arbitrary masks. The use of the proposition $i \Rrightarrow e$ in LR-FORK-R will become clear after the introduction of the thread pool resource algebra in Section 6.1; a general rule for stateful reductions on the right hand side, from which the specific rules can be derived, is described in Section 6.3.

$$
\begin{array}{c}
\textsf{True} \vdash \Delta \mid \Gamma \models () \precsim () : \mathbf{1} \\ \hline
{}^{\top\setminus\mathcal{N}}\!\models\!\!\!\Rrightarrow^{\top} \textsf{True} \vdash {}^{\top\setminus\mathcal{N}}\!\models\!\!\!\Rrightarrow^{\top} \Delta \mid \Gamma \models () \precsim () : \mathbf{1} \\ \hline
{}^{\top\setminus\mathcal{N}}\!\models\!\!\!\Rrightarrow^{\top} \textsf{True} \vdash \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} () \precsim () : \mathbf{1} \\ \hline
\triangleright(\exists m, l \mapsto_{\mathsf{i}} m * l' \mapsto_{\mathsf{s}} m), \triangleright I \;{}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top}\, \textsf{True} \vdash \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} () \precsim () : \mathbf{1} \\ \hline
l \mapsto_{\mathsf{i}} m, l' \mapsto_{\mathsf{s}} m, \triangleright I \;{}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top}\, \textsf{True} \vdash \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} () \precsim () : \mathbf{1} \\ \hline
l \mapsto_{\mathsf{i}} m, l' \mapsto_{\mathsf{s}} n, \triangleright I \;{}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top}\, \textsf{True} \vdash \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} () \precsim l' \leftarrow m : \mathbf{1} \\ \hline
\triangleright l' \mapsto_{\mathsf{s}} n, \triangleright I \;{}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top}\, \textsf{True} \vdash \triangleright(l \mapsto_{\mathsf{i}} m \twoheadrightarrow \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} () \precsim l' \leftarrow m : \mathbf{1}) \\ \hline
\triangleright l \mapsto_{\mathsf{i}} n, \triangleright l' \mapsto_{\mathsf{s}} n, \triangleright I \;{}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top}\, \textsf{True} \vdash \models\!\!\!\Rrightarrow_{\top\setminus\mathcal{N}} \exists v', \triangleright(l \mapsto_{\mathsf{i}} v') * \triangleright(l \mapsto_{\mathsf{i}} m \twoheadrightarrow \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} () \precsim l' \leftarrow m : \mathbf{1}) \\ \hline
\boxed{I}^{\mathcal{N}} \vdash {}^{\top}\!\models\!\!\!\Rrightarrow^{\top\setminus\mathcal{N}} \exists v', \triangleright(l \mapsto_{\mathsf{i}} v') * \triangleright(l \mapsto_{\mathsf{i}} m \twoheadrightarrow \Delta \mid \Gamma \models_{\top\setminus\mathcal{N}} () \precsim l' \leftarrow m : \mathbf{1}) \\ \hline
\boxed{I}^{\mathcal{N}} \vdash \Delta \mid \Gamma \models l \leftarrow m \precsim l' \leftarrow m : \mathbf{1}
\end{array}
$$

Figure 1: Full derivation of Equation (2)

3.1 Primitive rules

Value interpretation: $[\![\tau]\!]_\Delta(v_1, v_2)$

$$
\begin{array}{c} v_1 = () \land v_2 = () \\ \hline\hline [\![\mathbf{1}]\!]_\Delta(v_1, v_2) \end{array}
\qquad
\begin{array}{c} \exists n \in \mathbb{N},\, v_1 = v_2 = n \\ \hline\hline [\![\mathbf{N}]\!]_\Delta(v_1, v_2) \end{array}
$$

$$
\begin{array}{c} v_1 = \texttt{true} \land v_2 = \texttt{true} \lor v_1 = \texttt{false} \land v_2 = \texttt{false} \\ \hline\hline [\![\mathbf{2}]\!]_\Delta(v_1, v_2) \end{array}
$$

$$
\begin{array}{c} \exists w_1\, w_2\, w_1'\, w_2',\, v_1 = (w_1, w_2) * v_2 = (w_1', w_2') * (w_1, w_1') \in [\![\tau]\!]_\Delta * (w_2, w_2') \in [\![\sigma]\!]_\Delta \\ \hline\hline [\![\tau \times \sigma]\!]_\Delta(v_1, v_2) \end{array}
$$

$$
\begin{array}{c} \exists v\, v', (v_1 = \texttt{inl}\ v * v_2 = \texttt{inl}\ v' * (v, v') \in [\![\tau]\!]_\Delta) \lor (v_1 = \texttt{inr}\ v * v_2 = \texttt{inr}\ v' * (v, v') \in [\![\sigma]\!]_\Delta) \\ \hline\hline [\![\tau + \sigma]\!]_\Delta(v_1, v_2) \end{array}
$$

$$
\begin{array}{c} (\forall (w_1, w_2) \in [\![\tau]\!]_\Delta,\, \Delta \mid \emptyset \models v_1\ w_1 \precsim v_2\ w_2 : \sigma) \qquad \text{the mask } \mathcal{E} \text{ is arbitrary} \\ \hline\hline [\![\tau \to \sigma]\!]_\Delta(v_1, v_2) \end{array}
$$

$$
\begin{array}{c} \forall R : \mathit{Val} \times \mathit{Val} \to \mathit{iProp},\, [\![\tau]\!]_{(R::\Delta)}(v_1\ [], v_2\ []) \\ \hline\hline [\![\forall(\tau)]\!]_\Delta(v_1, v_2) \end{array}
$$

$$\begin{array}{c} \exists v\, v',\, \exists R : \mathit{Val} \times \mathit{Val} \to \mathit{iProp},\, v_1 = \texttt{pack}\ v * v_2 = \texttt{pack}\ v' * [\![\tau]\!]_{(R::\Delta)}(v, v') \\ \hline\hline [\![\exists(\tau)]\!]_\Delta(v_1, v_2) \end{array}$$

$$\begin{array}{c} \exists v\, v',\, v_1 = \texttt{fold}\ v * v_2 = \texttt{fold}\ v' * \triangleright [\![\tau]\!]_{(\mu(\tau)::\Delta)}(v, v') \\ \hline\hline [\![\mu(\tau)]\!]_\Delta(v_1, v_2) \end{array}$$

$$\begin{array}{c} \boxed{I_{\mathsf{rev}}(l, l', [\![\tau]\!]_\Delta)}^{\mathsf{logN}.(l,l')} \\ \hline\hline [\![\textbf{ref}\ \tau]\!]_\Delta(l, l') \end{array} \qquad \begin{array}{c} \Box\, \Delta(i)(v_1, v_2) \\ \hline\hline [\![x_i]\!]_\Delta(v_1, v_2) \end{array} \qquad \begin{array}{c} \scriptstyle\text{INTERP-PERSISTENT} \\ [\![\tau]\!]_\Delta(v_1, v_2) \\ \hline \Box [\![\tau]\!]_\Delta(v_1, v_2) \end{array}$$

where

$$\begin{array}{lcl} I_{\mathsf{rev}} & : & \mathit{Loc} \times \mathit{Loc} \to (\mathit{Val} \times \mathit{Val} \to \mathit{iProp}) \xrightarrow{\text{ne}} \mathit{iProp} \\ I_{\mathsf{rev}}(l, l', \tau i) & \triangleq & \exists v\ v',\, l \mapsto_{\mathsf{i}} v * l' \mapsto_{\mathsf{s}} v' * \tau_i(v, v') \end{array}$$

Remark 3.1. *The value interpretation rule for the arrow type requires an invariant* $\mathsf{spec\_ctx}(\rho)$ *in the context. We can, however, always obtain such an invariant from a logical refinement judgement. See "logrel/rules.v" in the formalisation for details (*`interp_val_arrow` *and* `bin_log_related_spec_ctx`*).*

Refinement judgements: $\Delta \mid \Gamma \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$

$$\begin{array}{c} \scriptstyle\text{LR-CLOSURE} \\ \Box(\forall v\, v',\ [\![\tau]\!]_\Delta(v, v') -\!\!* \Delta \mid \Gamma \models (\mathsf{rec}\ f\ x = e)\ v \precsim (\mathsf{rec}\ f'\ x' = e')\ v' : \tau') \qquad \text{closed}(\mathsf{rec}\ f\ x = e) \qquad \text{closed}(\mathsf{rec}\ f'\ x' = e') \\ \hline \Delta \mid \Gamma \models \mathsf{rec}\ f\ x = e \precsim \mathsf{rec}\ f'\ x' = e' : \tau \to \tau' \end{array}$$

$$\begin{array}{c} \scriptstyle\text{FUPD-LOGREL} \\ {}^{\mathcal{E}_1}{\vert\!\!\Rrightarrow}^{\mathcal{E}_2} (\Delta \mid \Gamma \models_{\mathcal{E}_2} e \precsim e' : \tau) \\ \hline \Delta \mid \Gamma \models_{\mathcal{E}_1} e \precsim e' : \tau \end{array} \qquad \begin{array}{c} \scriptstyle\text{LR-WEAKEN-2} \\ \Delta \mid \Gamma \models e \precsim e' : \tau \\ \hline (R, \Delta) \mid (+1)\ \langle\$\rangle\ \Gamma \models e \precsim e' : (+1)\ \langle\$\rangle\ \tau \end{array} \qquad \begin{array}{c} \scriptstyle\text{LR-RETURN} \\ [\![\tau]\!]_\Delta(v_1, v_2) \\ \hline \Delta \mid \Gamma \models v_1 \precsim v_2 : \tau \end{array}$$

$$\begin{array}{c} \scriptstyle\text{LR-BIND-UP} \\ (R, \Delta) \mid (+1)\ \langle\$\rangle\ \Gamma \models e_1 \precsim e_2 : \tau \qquad (\forall v\, v',\ [\![\tau]\!]_{(R,\Delta)}(v, v') -\!\!* \Delta \mid \Gamma \models K[v] \precsim K'[v'] : \tau') \\ \hline \Delta \mid \Gamma \models K[e_1] \precsim K'[e_2] : \tau' \end{array}$$

$$\begin{array}{c} \scriptstyle\text{LR-PURE-L} \\ e \to_{\mathsf{pure}} e' \qquad \triangleright \Delta \mid \Gamma \models K[e'] \precsim t : \tau \\ \hline \Delta \mid \Gamma \models K[e] \precsim t : \tau \end{array} \qquad \begin{array}{c} \scriptstyle\text{LR-PURE-L-MASKED} \\ e \to_{\mathsf{pure}} e' \qquad \Delta \mid \Gamma \models_{\mathcal{E}} K[e'] \precsim t : \tau \\ \hline \Delta \mid \Gamma \models_{\mathcal{E}} K[e] \precsim t : \tau \end{array}$$

$$\begin{array}{c} \scriptstyle\text{LR-WP-ATOMIC-L} \\ {}^{\top}{\vert\!\!\Rrightarrow}^{\mathcal{E}}\, \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau\} \qquad \text{atomic}(e) \qquad \text{closed}(e) \\ \hline \Delta \mid \Gamma \models K[e] \precsim t : \tau \end{array}$$

$$\begin{array}{c} \scriptstyle\text{LR-WP-L} \\ \mathsf{wp}\ e\, \{v.\, \Delta \mid \Gamma \models K[v] \precsim t : \tau\} \qquad \text{closed}(e) \\ \hline \Delta \mid \Gamma \models K[e] \precsim t : \tau \end{array} \qquad \begin{array}{c} \scriptstyle\text{LR-PURE-R} \\ e \to_{\mathsf{pure}} e' \qquad \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e'] : \tau \qquad \uparrow\mathsf{logrelN} \subseteq \mathcal{E} \\ \hline \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e] : \tau \end{array}$$

For the reductions on the RHS it is assumed that $\uparrow\mathsf{logrelN} \subseteq \mathcal{E}$.

LR-ALLOC-R
$$\frac{\forall l,\ l \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[l] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathtt{ref}(v)] : \tau}$$

LR-LOAD-R
$$\frac{l \mapsto_{\mathsf{s}} v \qquad l \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[v] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[!\,l] : \tau}$$

LR-STORE-R
$$\frac{l \mapsto_{\mathsf{s}} - \qquad l \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[()] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[l \leftarrow v] : \tau}$$

LR-CAS-FAIL-R
$$\frac{l \mapsto_{\mathsf{s}} v' \qquad v' \neq v_1 \qquad l \mapsto_{\mathsf{s}} v_1 \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathtt{false}] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathtt{CAS}(l, v_1, v_2)] : \tau}$$

LR-CAS-SUC-R
$$\frac{l \mapsto_{\mathsf{s}} v_1 \qquad l \mapsto_{\mathsf{s}} v_2 \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathtt{true}] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathtt{CAS}(l, v_1, v_2)] : \tau}$$

LR-FORK-R
$$\frac{\forall i, (i \Rrightarrow e \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[()] : \tau) \qquad \mathrm{closed}(e)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathtt{fork}\ \{e\}] : \tau}$$

LR-VAR
$$\frac{\Gamma(x) = \tau}{\Delta \mid \Gamma \models x \precsim x : \tau}$$

LR-REC
$$\frac{\square(\Delta \mid f : (\tau \to \sigma), x : \tau, \Gamma \models e \precsim e' : \sigma) \qquad \mathrm{closed}(\{x, f\} \cup (\mathrm{dom}\,\Gamma), e) \qquad \mathrm{closed}(\{x, f\} \cup (\mathrm{dom}\,\Gamma), e')}{\Delta \mid \Gamma \models \mathtt{rec}\, f\ x = e \precsim \mathtt{rec}\, f\ x = e' : \tau \to \sigma}$$

LR-TLAM
$$\frac{\forall \tau_i : \mathit{Val} \times \mathit{Val} \to \mathit{iProp},\ \square((\tau_i, \Delta) \mid (+1)\ \langle\$\rangle\ \Gamma \models e \precsim e' : \tau)}{\Delta \mid \Gamma \models \Lambda.e \precsim \Lambda.e' : \forall \tau}$$

LR-TAPP
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \forall \tau \qquad \tau_i : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}}{(\tau_i, \Delta) \mid (+1)\ \langle\$\rangle\ \Gamma \models e\ [] \precsim e'\ [] : \tau}$$

LR-PACK
$$\frac{(\tau_i, \Delta) \mid (+1)\ \langle\$\rangle\ \Gamma \models e \precsim e' : \tau}{\Delta \mid \Gamma \models \mathtt{pack}\ e \precsim \mathtt{pack}\ e' : \exists \tau}$$

LR-UNPACK
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_1' : \exists \tau_1 \qquad (\forall \tau_i : \mathit{Val} \times \mathit{Val} \to \mathit{iProp},\ (\tau_i, \Delta) \mid (+1)\ \langle\$\rangle\ \Gamma \models e_2 \precsim e_2' : \tau_1 \to (+1)\ \langle\$\rangle\ \tau_2)}{\Delta \mid \Gamma \models \mathtt{unpack}\ e_1\ \mathtt{in}\ e_2 \precsim \mathtt{unpack}\ e_1'\ \mathtt{in}\ e_2' : \tau_2}$$

LR-FORK
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \mathbf{1}}{\Delta \mid \Gamma \models \mathtt{fork}\ \{e\} \precsim \mathtt{fork}\ \{e'\} : \mathbf{1}}$$

3.2 Derived rules

For the symbolic execution rules for the RHS it is assumed that $\uparrow\mathsf{logrelN} \subseteq \mathcal{E}$.

The following rules are derived using the $\to_{\mathsf{pure}}$ rules, LR-PURE-L, and LR-PURE-R. The rule LR-ARROW is derived from LR-CLOSURE and LR-RETURN.

The rule LR-BIND is derived from LR-BIND-UP and LR-WEAKEN-2. The difference between the two is that LR-BIND-UP contains a baked in *semantic type* R. The idea here is that we don't actually require the expressions that we bind to have the same *syntactic type*, like in LR-BIND.

LR-ARROW

$$\frac{\Box(\forall v\, v',\ \Box(\Delta \mid \Gamma \models v \precsim v' : \tau) \twoheadrightarrow \Delta \mid \Gamma \models (\mathsf{rec}\ f\ x = e)\ v \precsim (\mathsf{rec}\ f'\ x' = e')\ v' : \tau') \qquad \mathrm{closed}(\{f, x\}, e) \qquad \mathrm{closed}(\{f', x'\}, e')}{\Delta \mid \Gamma \models \mathsf{rec}\ f\ x = e \precsim \mathsf{rec}\ f'\ x' = e' : \tau \to \tau'}$$

LR-BIND

$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau \qquad (\forall v\, v',\ [\![\tau]\!]_\Delta(v, v') \twoheadrightarrow \Delta \mid \Gamma \models K[v] \precsim K'[v'] : \tau')}{\Delta \mid \Gamma \models K[e_1] \precsim K'[e_2] : \tau'}$$

LR-REC-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[e[v/x][\mathsf{rec}\ f\ x = e/f]] \precsim t : \tau) \qquad \mathrm{closed}(\mathsf{rec}\ f\ x = e)}{\Delta \mid \Gamma \models K[(\mathsf{rec}\ f\ x = e)\ v] \precsim t : \tau}$$

LR-FST-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[v_1] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\pi_1(v_1, v_2)] \precsim t : \tau}$$

LR-SND-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[v_2] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\pi_2(v_1, v_2)] \precsim t : \tau}$$

LR-TLAM-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[e] \precsim t : \tau) \qquad \mathrm{closed}(e)}{\Delta \mid \Gamma \models K[(\Lambda.e)\ [\,]] \precsim t : \tau}$$

LR-FOLD-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[v] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\mathtt{unfold}\ (\mathtt{fold}\ v)] \precsim t : \tau}$$

LR-PACK-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[e\ v] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\mathtt{unpack}\ (\mathtt{pack}\ v)\ \mathtt{in}\ e] \precsim t : \tau}$$

LR-CASE-INL-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[e_1\ v] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\mathtt{case}(\mathtt{inl}\ v, e_1, e_2)] \precsim t : \tau}$$

LR-CASE-INR-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[e_2\ v] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\mathtt{case}(\mathtt{inl}\ v, e_1, e_2)] \precsim t : \tau}$$

LR-IF-TRUE-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[e_1] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2] \precsim t : \tau}$$

LR-IF-FALSE-L

$$\frac{\triangleright(\Delta \mid \Gamma \models K[e_2] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2] \precsim t : \tau}$$

LR-BINOP-L
$$\frac{\triangleright(\Delta \mid \Gamma \models K[k] \precsim t : \tau) \qquad k = n[\![\oplus]\!]m}{\Delta \mid \Gamma \models K[n \oplus m] \precsim t : \tau}$$

LR-REC-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e[v/x][\mathsf{rec}\ f\ x = e/f]] : \tau \qquad \mathrm{closed}(\mathsf{rec}\ f\ x = e)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[(\mathsf{rec}\ f\ x = e)\ v] : \tau}$$

LR-FST-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[v_1] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\pi_1(v_1, v_2)] : \tau}$$

LR-SND-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[v_2] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim K[\pi_2(v_1, v_2)] : \tau}$$

LR-TLAM-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e] : \tau \qquad \mathrm{closed}(e)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[(\Lambda.e)\ []] : \tau}$$

LR-FOLD-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[v] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\texttt{unfold}\ (\texttt{fold}\ v)] : \tau}$$

LR-PACK-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e\ v] : \tau \qquad \mathrm{closed}(e)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\texttt{unpack}\ (\texttt{pack}\ v)\ \texttt{in}\ e] : \tau}$$

LR-CASE-INL-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e_1\ v] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\texttt{case}(\texttt{inl}\ v, e_1, e_2)] : \tau}$$

LR-CASE-INR-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e_2\ v] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\texttt{case}(\texttt{inl}\ v, e_1, e_2)] : \tau}$$

LR-IF-TRUE-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e_1] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\texttt{if}\,\texttt{true}\,\texttt{then}\,e_1\,\texttt{else}\,e_2] : \tau}$$

LR-IF-FALSE-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e_2] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\texttt{if}\,\texttt{false}\,\texttt{then}\,e_1\,\texttt{else}\,e_2] : \tau}$$

LR-BINOP-R
$$\frac{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[k] : \tau \qquad k = [\![\oplus]\!](n, m)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[n \oplus m] : \tau}$$

Utilising LR-WP-ATOMIC-L we can prove the admissibility of the stateful reduction rules.

LR-FORK-L
$$\frac{\Delta \mid \Gamma \models K[()] \precsim t : \tau \qquad \mathsf{wp}\, e\, \{\mathsf{True}\} \qquad \mathrm{closed}(e)}{\Delta \mid \Gamma \models K[\texttt{fork}\ \{e\}] \precsim t : \tau}$$

LR-ALLOC-L
$$\frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} \triangleright(\forall l,\, l \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[l] \precsim t : \tau)}{\Delta \mid \Gamma \models K[\mathtt{ref}(v)] \precsim t : \tau}$$

LR-LOAD-L
$$\frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} (\exists v,\, \triangleright l \mapsto_{\mathsf{i}} v * \triangleright(l \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau))}{\Delta \mid \Gamma \models K[!\, l] \precsim t : \tau}$$

LR-STORE-L
$$\frac{{}^{\top}\!\Rrightarrow^{\mathcal{E}} (\triangleright l \mapsto_{\mathsf{i}} - * \triangleright(l \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[()] \precsim t : \tau))}{\Delta \mid \Gamma \models K[l \leftarrow v] \precsim t : \tau}$$

LR-CAS-L
$$\frac{\begin{array}{c}{}^{\top}\!\Rrightarrow^{\mathcal{E}} (\exists v', \triangleright l \mapsto_{\mathsf{i}} v' * (v' \neq v_1 \twoheadrightarrow \triangleright(l \mapsto_{\mathsf{i}} v' \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[\mathtt{false}] \precsim t : \tau)) \\ \wedge(v' = v_1 \twoheadrightarrow \triangleright(l \mapsto_{\mathsf{i}} v_2 \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} K[\mathtt{true}] \precsim t : \tau)))\end{array}}{\Delta \mid \Gamma \models K[\mathtt{CAS}(l, v_1, v_2)] \precsim t : \tau}$$

Note that we have only one rule for CAS, which is not the case in the WP-calculus. The reason for that is the following: if $l \mapsto_{\mathsf{i}} v'$ is stored in an invariant, we do not know, a priori, whether v' is going to be equal to v_1. The only way do decide whether the CAS succeeds is to open the invariant first. Hence, the decision must be put under the fancy update modality.

The derived compatibility rules are proven using the "monadic" rules (LR-RETURN, LR-BIND) and symbolic execution rules.

LR-VAL
$$\frac{\Rrightarrow_{\mathcal{E}} [\![\tau]\!]_{\Delta}(v, v')}{\Delta \mid \Gamma \models v \precsim v' : \tau}$$

LR-LITERAL
$$\frac{c \text{ is a literal of type } \tau}{\Delta \mid \Gamma \models c \precsim c : \tau}$$

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

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

LR-SND
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau \times \sigma}{\Delta \mid \Gamma \models \pi_2(e_1) \precsim \pi_2(e_2) : \sigma}$$

LR-APP
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau \to \sigma \qquad \Delta \mid \Gamma \models e_1' \precsim e_2' : \tau}{\Delta \mid \Gamma \models e_1\, e_1' \precsim e_2\, e_2' : \sigma}$$

LR-INJL
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau}{\Delta \mid \Gamma \models \mathtt{inl}(e_1) \precsim \mathtt{inl}(e_2) : \tau + \sigma}$$

LR-INJR
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_2 : \sigma}{\Delta \mid \Gamma \models \mathtt{inr}(e_1) \precsim \mathtt{inr}(e_2) : \tau + \sigma}$$

LR-CASE
$$\frac{\Delta \mid \Gamma \models e_0 \precsim e_0' : \tau_1 + \tau_2 \qquad \Delta \mid \Gamma \models e_1 \precsim e_1' : \tau_1 \to \tau_3 \qquad \Delta \mid \Gamma \models e_2 \precsim e_2' : \tau_2 \to \tau_3}{\Delta \mid \Gamma \models \mathtt{case}(e_0, e_1, e_2) \precsim \mathtt{case}(e_0', e_1', e_2') : \tau_3}$$

LR-IF
$$\frac{\Delta \mid \Gamma \models e_0 \precsim e_0' : \mathbf{2} \qquad \Delta \mid \Gamma \models e_1 \precsim e_1' : \tau \qquad \Delta \mid \Gamma \models e_2 \precsim e_2' : \tau}{\Delta \mid \Gamma \models \texttt{if}\, e_0\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \precsim \texttt{if}\, e_0'\, \texttt{then}\, e_1'\, \texttt{else}\, e_2' : \tau}$$

LR-BINOP
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_1' : \mathbf{N} \qquad \Delta \mid \Gamma \models e_2 \precsim e_2' : \mathbf{N}}{\Delta \mid \Gamma \models e_1 \oplus e_2 \precsim e_1' \oplus e_2' : \mathbf{N}}$$

LR-TAPP'
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \forall \tau}{\Delta \mid \Gamma \models e\; [] \precsim e'\; [] : \tau[\sigma/]}$$

LR-FOLD
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \tau[\mu(\tau)/]}{\Delta \mid \Gamma \models \texttt{fold}(e) \precsim \texttt{fold}(e') : \mu(\tau)}$$

LR-UNFOLD
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \mu(\tau)}{\Delta \mid \Gamma \models \texttt{unfold}(e) \precsim \texttt{unfold}(e') : \tau[\mu(\tau)/]}$$

LR-PACK'
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \tau[\sigma/]}{\Delta \mid \Gamma \models \texttt{pack}\; e \precsim \texttt{pack}\; e' : \exists \tau}$$

LR-ALLOC
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \tau}{\Delta \mid \Gamma \models \texttt{ref}(e) \precsim \texttt{ref}(e') : \mathbf{ref}(\tau)}$$

LR-LOAD
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \mathbf{ref}(\tau)}{\Delta \mid \Gamma \models\, !e \precsim\, !e' : \tau}$$

LR-STORE
$$\frac{\Delta \mid \Gamma \models e \precsim e' : \mathbf{ref}(\tau) \qquad \Delta \mid \Gamma \models t \precsim t' : \tau}{\Delta \mid \Gamma \models e \leftarrow t \precsim e' \leftarrow t' : \mathbf{1}}$$

LR-CAS
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_1' : \mathbf{ref}(\tau) \qquad \mathsf{EqType}(\tau) \qquad \Delta \mid \Gamma \models e_2 \precsim e_2' : \tau \qquad \Delta \mid \Gamma \models e_3 \precsim e_3' : \tau}{\Delta \mid \Gamma \models \texttt{CAS}(e_1, e_2, e_3) \precsim \texttt{CAS}(e_1', e_2', e_3') : \mathbf{2}}$$

LR-SEQ
$$\frac{(R, \Delta) \mid (+1)\ \langle\$\rangle\ \Gamma \models e_1 \precsim e_1' : \tau_1 \qquad \Delta \mid \Gamma \models e_2 \precsim e_2' : \tau_2}{\Delta \mid \Gamma \models e_1; e_2 \precsim e_1'; e_2' : \tau_2}$$

LR-SEQ'
$$\frac{\Delta \mid \Gamma \models e_1 \precsim e_1' : \tau_1 \qquad \Delta \mid \Gamma \models e_2 \precsim e_2' : \tau_2}{\Delta \mid \Gamma \models e_1; e_2 \precsim e_1'; e_2' : \tau_2}$$

3.3 Compatibility lemmas and the fundamental property

The standard proof of soundness of logical refinement judgement is done via so called "compatibility lemmas". In our deductive system they are presented as rules.

Lemma 3.2. *If* $\Gamma \vdash e : \tau$ *then* $\Delta \mid \Gamma \models e \precsim e : \tau$.

Proof. By induction on the typing derivation, using the compatibility rules from Sections 3.1 and 3.2. ☐

Notes on formalisation

The formalisation of the calculus is split across various modules in the `logrel` directory. The rule FUPD-LOGREL and the monadic rules are formalised in `logrel_binary.v`. In addition FUPD-LOGREL gives rise to several `ElimModal` instances (also defined in the same file). The primitive and derived compatibility rules are formalised in `fundamental_binary.v` alongside the fundamental property Lemma 3.2. The rest of the rules are formalised in `rules.v` module.

4 Introductory example: fine-grained concurrent counter

In this section we go over an illustrative example: a lock-free concurrent counter implementation that uses atomic CAS refines an implementation that uses locking. The source code for the two counters is in Figure 2, $\mathsf{counter}_i$ is a fine-grained counter whereas $\mathsf{counter}_s$ is a coarse-grained counter implemented using locks.

4.1 General form of relational specifications: a library for locks

The course grained counter, which we will use as a specification, is implemented using locks. The locks themselves in turn are implemented using atomic compare-and-swap. In this subsection we sketch the lemmas provided by the lock library.

Implementation:

$$\begin{array}{lcl}
\mathsf{TLock} & := & \mathbf{ref\ 2} \\
\mathsf{newlock} & : & \mathsf{TLock} \\
\mathsf{newlock} & := & \mathtt{ref}(\mathtt{false}) \\
\mathsf{acquire} & : & \mathsf{TLock} \to \mathbf{1} \\
\mathsf{acquire} & := & \mathsf{rec\ acquire}\ x = \mathtt{if}\ \mathtt{CAS}(x, \mathtt{false}, \mathtt{true})\ \mathtt{then}\ ()\ \mathtt{else}\ \mathsf{acquire}\ x \\
\mathsf{release} & : & \mathsf{TLock} \to \mathbf{1} \\
\mathsf{release} & := & \lambda x.\, x \leftarrow \mathtt{false}
\end{array}$$

Relational specifications: where $\uparrow\mathsf{logrelN} \subseteq \mathcal{E}$

LR-NEWLOCK-R

$$\frac{\forall l,\ l \mapsto_{\mathsf{s}} \mathtt{false} \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[l] : \tau}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathsf{newlock}] : \tau}$$

LR-ACQUIRE-R

$$\frac{l \mapsto_{\mathsf{s}} \mathtt{false} \qquad (l \mapsto_{\mathsf{s}} \mathtt{true} \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[()] : \tau)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathsf{acquire}\ l] : \tau}$$

LR-RELEASE-R

$$\frac{l \mapsto_{\mathsf{s}} b \qquad (l \mapsto_{\mathsf{s}} \mathtt{false} \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[()] : \tau)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathsf{release}\ l] : \tau}$$

As one can observe, the relational specifications for symbolic execution on the right hand side follow a certain pattern. For an expression e that under precondition P reduces to v with postcondition $Q(v)$, the rule has the following form:

$$P * (\forall v,\ Q(v) \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[v] : \tau) \twoheadrightarrow \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e] : \tau$$

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

$$P * (\forall v,\, Q(v) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[v] \precsim t : \tau) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[e] \precsim t : \tau$$

Notice that for the left hand side rule, the masks in the judgement have to be the same. This means, in particular, that such rules cannot be applied in combination with opening an invariant. We will see how to mitigate this issue in Section 8.1.

Hoare triples and relational specification. In fact, we can take the general form of the relational specification from the previous paragraph as a basis for defining a "relational Hoare triple for the left hand side":

$$\Delta \mid \Gamma \models \{P\}\; e\; \{Q\} \triangleq \forall K\, t\, \tau,\, \square(P * (\forall v,\, Q(v) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[v] \precsim t : \tau) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[e] \precsim t : \tau)$$

Lemma 4.1. *For any Δ, Γ it is the case that $\{P\}\, e\, \{Q\}_{\mathcal{E}} \mathrel{-\!\!*} \Delta \mid \Gamma \models \{P\}\; e\; \{Q\}$*

Proof. Unfolding the definitions and using LR-WP-L. □

4.2 Coarse-grained and fine-grained counters

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

Figure 2: Fine-grained and coarse-grained counters

The invariant that is going to link two implementations is

$$I_{\mathsf{cnt}}(l, c_i, c_s) \triangleq \exists n,\, l \mapsto_{\mathsf{s}} \mathtt{false} * c_s \mapsto_{\mathsf{s}} n * c_i \mapsto_{\mathsf{i}} n$$

Our goal is to show $\vdash \Delta \mid \emptyset \models \mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathbf{1} \to \mathbf{N}) \times (\mathbf{1} \to \mathbf{N})$. To do this we can perform symbolic execution until we reach pairs on both sides; our goal then becomes

$$l \mapsto_{\mathsf{s}} \mathtt{false}, c_i \mapsto_{\mathsf{i}} 0, c_s \mapsto_{\mathsf{s}} 0 \vdash \Delta \mid \emptyset \models (\mathsf{read}\ c_i, \lambda().\, \mathsf{inc}_i\ c_i) \precsim (\mathsf{read}\ c_s, \lambda().\, \mathsf{inc}_s\ c_s\ l) : (\mathbf{1} \to \mathbf{N}) \times (\mathbf{1} \to \mathbf{N})$$

At this point we can establish the invariant $\boxed{I_{\mathsf{cnt}}(l, c_i, c_s)}^{\mathcal{N}}$ and apply LR-PAIR.

Thus we have to prove

$$\boxed{I_{\mathsf{cnt}}(l, c_i, c_s)}^{\mathcal{N}} \vdash \Delta \mid \emptyset \models \lambda().\,\mathsf{inc}_i\ c_i \precsim \lambda().\,\mathsf{inc}_s\ c_s\ l : \mathbf{1} \to \mathbf{N} \tag{3}$$

$$\boxed{I_{\mathsf{cnt}}(l, c_i, c_s)}^{\mathcal{N}} \vdash \Delta \mid \emptyset \models \mathsf{read}\ c_i \precsim \mathsf{read}\ c_s : \mathbf{1} \to \mathbf{N} \tag{4}$$

(Proof of Equation (3)*).* By LR-CLOSURE and LR-PURE-L, LR-PURE-R it suffices to prove

$$\Delta \mid \emptyset \models \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ l : \mathbf{N}$$

under the assumption that we have the invariant $\boxed{I_{\mathsf{cnt}}(l, c_i, c_s)}^{\mathcal{N}}$. Performing some symbolic execution on both sides our goal becomes

$$\Delta \mid \emptyset \models \mathtt{let}\, c = !\, c_i\ \mathtt{in}\, \mathtt{if}\ \mathtt{CAS}(c_i, c, 1 + c)\ \mathtt{then}\, c\, \mathtt{else}\, \mathsf{inc}_i\ c_i \precsim \mathsf{acquire}\ l; \ldots : \mathbf{N}$$

We proceed by Löb induction; that is, we get an assumption

$$\triangleright(\Delta \mid \emptyset \models \mathtt{let}\, c = !\, c_i\ \mathtt{in}\, \mathtt{if}\ \mathtt{CAS}(c_i, c, 1{+}c)\ \mathtt{then}\, c\, \mathtt{else}\, \mathsf{inc}_i\ c_i \precsim \mathsf{acquire}\ l; \ldots : \mathbf{N}).$$

We will get rid of $\triangleright$ for this hypothesis as soon as we perform a symbolic execution step on the left (using the monotonicity of $\triangleright$).

At this point we apply LR-REC-L and LR-LOAD-L to get the goal

$$^{\top}\!\!\Rrightarrow^{\top\setminus\mathcal{N}} (\exists v, \triangleright c_i \mapsto_{\mathsf{i}} v * (c_i \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \mid \emptyset \models_{\top\setminus\mathcal{N}} K[v] \precsim \mathsf{acquire}\ l; \ldots : \mathbf{N}).$$

We can then use the invariant opening rule to obtain

1. The lock resource: $l \mapsto_{\mathsf{s}} \mathtt{false}$;
2. The counter resources: $c_s \mapsto_{\mathsf{s}} n$ and $c_i \mapsto_{\mathsf{i}} n$ for some $n \in \mathbb{N}$;
3. The invariant closing rule: $\exists n, l \mapsto_{\mathsf{s}} \mathtt{false} * c_s \mapsto_{\mathsf{s}} n * c_i \mapsto_{\mathsf{i}} n \ {}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top}$ True;
4. And the goal without the ${}^{\top}\!\!\Rrightarrow^{\top\setminus\mathcal{N}}$ modality.

We can then frame $\triangleright c_i \mapsto_{\mathsf{i}} n$, and introduce $c_i \mapsto_{\mathsf{i}} n$ to obtain a new goal

$$\Delta \mid \emptyset \models_{\top\setminus\mathcal{N}} \mathtt{let}\, c = n\, \mathtt{in}\ \ldots \precsim \mathsf{acquire}\ l; \ldots : \mathbf{N}.$$

At this point we cannot really continue the symbolic execution, so we close the invariant (as we can easily do, because we haven't actually changed any of the resources that we were holding) using FUPD-LOGREL. Our new goal is

$$\Delta \mid \emptyset \models \mathtt{let}\, c = n\, \mathtt{in}\ \ldots \precsim \mathsf{acquire}\ l; \ldots : \mathbf{N}$$

and we do not hold any resources. After performing a number of pure reductions on the left had side we reach the goal

$$\Delta \mid \emptyset \models \mathtt{if}\ \mathtt{CAS}(c_i, n, n + 1)\ \mathtt{then}\, n\, \mathtt{else}\, \mathsf{inc}_i\ c_i \precsim \mathsf{acquire}\ l; \ldots : \mathbf{N}.$$

At this point we apply LR-CAS-L, open the invariant and consider two cases:

1. The new value of the counter has changed and is no longer n. In that case CAS fails. However, the state has not been changed and we can easily close the invariant leaving us with the goal:

   $$\Delta \mid \emptyset \models \texttt{if}\,\texttt{false}\,\texttt{then}\,n\,\texttt{else}\,\mathsf{inc}_i\ c_i \precsim \mathsf{acquire}\ l; \ldots : \mathbf{N}.$$

   which we solve by applying LR-IF-FALSE-L and using the induction hypothesis.

2. The counter value has not changed. In this case the goal is

   $$c_i \mapsto_{\mathsf{i}} (n+1) \twoheadrightarrow$$
   $$\Delta \mid \emptyset \models_{\top \setminus \mathcal{N}} \texttt{if}\,\texttt{true}\,\texttt{then}\,n\,\texttt{else}\,\mathsf{inc}_i\ c_i \precsim \mathsf{acquire}\ l; \texttt{let}\ n = !c_s\ \texttt{in}\ c_s \leftarrow n+1; \mathsf{release}\ l; n : \mathbf{N}.$$

   Then, the operation have succeeded. It remains, however, to perform the counter update on the right hand side. Because the invariant is still open, we have access to $l \mapsto_{\mathsf{s}} \texttt{false}$ and $c \mapsto_{\mathsf{s}} n$. Using LR-ACQUIRE-R, LR-STORE-R, and LR-RELEASE-R we can reduce this to

   $$c_i \mapsto_{\mathsf{i}} (n+1) * c_s \mapsto_{\mathsf{s}} (n+1) * l \mapsto_{\mathsf{s}} \texttt{false} \twoheadrightarrow$$
   $$\Delta \mid \emptyset \models_{\top \setminus \mathcal{N}} \texttt{if}\,\texttt{true}\,\texttt{then}\,n\,\texttt{else}\,\mathsf{inc}_i\ c_i \precsim n : \mathbf{N}.$$

   After this we can close the invariant using FUPD-LOGREL and use LR-PURE-L to finish up with an instance of LR-VAL:

   $$\Delta \mid \emptyset \models n \precsim n : \mathbf{N}.$$

□

Notes on formalisation

The counter refinement is implemented in `examples/counter.v` using logically atomic rules described in Section 8.1. The rules for the lock are derived in `examples/lock.v`.

5 Ticket lock from the counter specification

In this section we present the details of the ticket-lock vs spin lock refinement described in the LICS paper. The purpose of this section is to give an detailed description of the way the rules of ReLoC are used for an actual proof.

$$\textsc{ticket-nondup}\quad \frac{\mathsf{ticket}_\gamma(n) \qquad \mathsf{ticket}_\gamma(n)}{\mathsf{False}} \qquad\qquad \textsc{newIssuedTickets}\quad \Rrightarrow \exists\gamma,\ \mathsf{issuedTickets}_\gamma(0)$$

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

Figure 3: Properties of abstract predicates.

Abstract predicates. We use the following abstract predicates:

- $\mathsf{ticket}_\gamma(m)$ representing a ticket with the id m from the ticket dispensing machine with the name γ;
- $\mathsf{issuedTickets}_\gamma(m)$ stating that a total of m tickets have been issued for the dispensing machine γ;

The predicates themselves are implemented in Iris using ghost state over the resource algebra $\textsc{Auth}(\mathcal{P}_{\text{DISJ}}(\mathbb{N}))$. For the purpose of the proof, we are not concerned with the implementations of the predicates and only require that they satisfy the rules presented in Figure 3.

The relation linking together two modules (serving as the interpretation for α) is:

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

Lemma 5.1. *The following refinement holds:*

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathit{newlock}_i \precsim \mathit{newlock}_s : \mathbf{1} \to \alpha.$$

Proof. By LR-CLOSURE it suffices to show:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathsf{newlock}_i\ () \precsim \mathsf{newlock}_s() : \alpha.$$

Performing symbolic execution on the left and the right hand sides we get $lo \mapsto_{\mathsf{i}} 0 * ln \mapsto_{\mathsf{i}} 0 * \mathsf{isLock}(l', \texttt{false})$ and the goal:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models (lo, ln) \precsim l' : \alpha.$$

By FUPD-LOGREL and LR-RETURN it suffices to prove:

$$\Rrightarrow \mathsf{lockInt}((lo, ln), l').$$

In other words:

$$\Rrightarrow \exists \gamma.\, \mathsf{lockInv}_\gamma(lo, ln, l')$$

To prove this goal we first create a new ticket dispensing machine with a fresh name γ using NEWISSUEDTICKETS. Together with the resources that we already had, $\mathsf{issuedTickets}_\gamma(0)$ comprises $\mathsf{lockInv}_\gamma(lo, ln, l')$. □

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

Lemma 5.2. *Assume the ticket* $\mathsf{ticket}_\gamma(m)$*, and the invariant:*

$$\boxed{\mathsf{lockInv}_\gamma(lo, ln, l')}^{\mathcal{N}},$$

linking the two locks together. Then:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathit{wait\_loop}\ m\ lo \precsim \mathit{acquire}_s\ l' : \mathbf{1}$$

Proof. By Löb induction it suffice to show goal from an assumption:

$$\triangleright(\mathsf{ticket}_\gamma(m) \twoheadrightarrow$$
$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathsf{wait\_loop}\ m\ lo \precsim \mathsf{acquire}_s\ l' : \mathbf{1}).$$

(We will get rid of the later modality after performing a symbolic execution step—so we will ignore the later modality from now on.)

After performing pure symbolic reductions on the left had side our goal becomes:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models$$
$$\mathtt{if}\ (m = !\, lo)\ \mathtt{then}\ ()\ \mathtt{else}\ \mathsf{wait\_loop}\ m\ lo \precsim \mathsf{acquire}_s\ l' : \mathbf{1}.$$

At this point we apply the rule LR-LOAD-L, which allows us to open the invariant $\mathcal{N}$ to get:

- the resources $lo \mapsto_{\mathsf{i}} o * ln \mapsto_{\mathsf{i}} n * \mathsf{isLock}(l', b)$ for some o, n, b;
- $\mathsf{issuedTickets}_\gamma(n)$ and if b then $\mathsf{ticket}_\gamma(o)$, for some γ.

After framing and introducing resources our goal is:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models_{\top \setminus \mathcal{N}}$$
$$\mathtt{if}\ (m = o)\ \mathtt{then}\ ()\ \mathtt{else}\ \mathsf{wait\_loop}\ m\ lo \precsim \mathsf{acquire}_s\ \ell' : \mathbf{1}.$$

Here we distinguish two cases:

1. **Case $m = o$.** In this situation we know that our turn to enter the critical section has arrived, *i.e.,* it must be the case that $b = \mathtt{false}$. This is the case because if $b = \mathtt{true}$, then have $\mathsf{ticket}_\gamma(o)$ from the invariant $\mathcal{N}$ and $\mathsf{ticket}_\gamma(m)$ by assumption. This yields a contradiction by TICKET-NONDUP.

Since $b = \texttt{false}$ we can apply LR-ACQUIRE-R to update the lock to $\mathsf{isLock}(l', \texttt{true})$ and reduce the goal to:

$$\begin{aligned}&[\alpha := \mathsf{lockInt}] \mid \emptyset \models_{\top\setminus\mathcal{N}} \\ &\quad \texttt{if}\ (o = o)\ \texttt{then}\ ()\ \texttt{else}\ \mathsf{wait\_loop}\ m\ lo \precsim () : \mathbf{1}.\end{aligned}$$

We can close the invariant by giving up the original ticket $\mathsf{ticket}_\gamma(o)$. The goal then holds by LR-PURE-L and the compatibility property for the unit type.

2. **Case $m \neq o$.** We can immediately close the invariant to restore the masks on the relational judgement, and reduce the goal to the original statement of this lem. Finally, we discharge the goal by the induction hypothesis. □

Lemma 5.3. *The following refinement holds:*

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathit{acquire}_i \precsim \mathit{acquire}_s : \alpha \to \mathbf{1}.$$

Proof. By LR-CLOSURE it suffices to assume the invariant:

$$\boxed{\mathsf{lockInv}_\gamma(lo, ln, l')}^{\mathcal{N}}$$

for some γ, and show:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathsf{acquire}_i\ (lo, ln) \precsim \mathsf{acquire}_s\ l' : \mathbf{1}.$$

After applying LR-PURE-L, the goal becomes:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models K[\mathsf{inc}_i\ ln] \precsim \mathsf{acquire}_s\ \ell' : \mathbf{1}$$

where $K \triangleq \texttt{let}\ n = [\bullet]\ \texttt{in}\ \mathsf{wait\_loop}\ n\ lo$.

At this point we can use the atomic rule for the fine-grained counter FG-INCREMENT-ATOMIC-L with the parameters $\mathcal{E} \triangleq \top\setminus\mathcal{N}$ and $R(n) \triangleq \mathsf{issuedTickets}_\gamma(n)$. We have to show:

$$\begin{aligned}&{}^{\top}\!\!\Rrightarrow^{\top\setminus\mathcal{N}} \exists n.\ ln \mapsto_{\mathsf{i}} n * \mathsf{issuedTickets}_\gamma(n) * \\ &\qquad\qquad \left(\begin{aligned}&(ln \mapsto_{\mathsf{i}} n * \mathsf{issuedTickets}_\gamma(n)\ {}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top}\ \mathsf{True}) \wedge \\ &(ln \mapsto_{\mathsf{i}} (n+1) * \mathsf{issuedTickets}_\gamma(n) \mathrel{-\!\!*} \\ &\ \models_{\top\setminus\mathcal{N}} K[n] \precsim \mathsf{acquire}_s\ l' : \mathbf{1})\end{aligned}\right)\end{aligned} \tag{5}$$

At this point we can introduce the update modality by opening the invariant $\mathcal{N}$ and obtaining:

- the resources $lo \mapsto_{\mathsf{i}} o * ln \mapsto_{\mathsf{i}} n * \mathsf{isLock}(l', b)$ for some o, n, b;
- $\mathsf{issuedTickets}_\gamma(n)$ and if b then $\mathsf{ticket}_\gamma(o)$, for some γ.

We can frame $ln \mapsto_{\mathsf{i}} n$ and $\mathsf{issuedTickets}_\gamma(n)$ in Equation (5), it then remains to show the conjunction:

$$\left(\begin{array}{l}(ln \mapsto_{\mathsf{i}} n * \mathsf{issuedTickets}_\gamma(n) \ {}^{\top\setminus\mathcal{N}}\Rrightarrow\!\!*^{\top}\ \mathsf{True}) \wedge \\ (ln \mapsto_{\mathsf{i}} (n+1) * \mathsf{issuedTickets}_\gamma(n) \mathrel{-\!\!*} \\ \quad \models_{\top\setminus\mathcal{N}} K[n] \precsim \mathsf{acquire}_s\ l' : \mathbf{1})\end{array}\right).$$

For the first conjunct we just apply the invariant closing proposition and show that the invariant $\mathsf{lockInv}_\gamma(lo, ln, l')$ still holds. Since we have not changed any ghost state it is trivial.

For the second conjunct, assume that we have $\mathtt{ln} \mapsto_{\mathsf{i}} (\mathtt{n}+1)$ and $\mathsf{issuedTickets}_\gamma(n)$. We can apply the update ISSUENEWTICKET to get:

$$\mathsf{issuedTickets}_\gamma(n+1) * \mathsf{ticket}_\gamma(n)$$

We restore the invariant using these resources and $\mathtt{ln} \mapsto_{\mathsf{i}} (\mathtt{n}+1)$, which leaves us with the goal:

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathsf{wait\_loop}\ n\ lo \precsim \mathsf{acquire}_s\ l' : \mathbf{1}$$

which reduces to the statement of Lemma 5.2. □

Similarly we can show the refinement for $\mathsf{release}$.

Lemma 5.4. *The following refinement holds:*

$$[\alpha := \mathsf{lockInt}] \mid \emptyset \models \mathit{release}_i \precsim \mathit{release}_s : \alpha \to \mathbf{1}.$$

Theorem 5.5. *The following refinement holds:*

$$\begin{array}{l}\mathtt{pack}(\mathit{newlock}_s, \mathit{acquire}_s, \mathit{release}_s) \\ \qquad\qquad \precsim \mathtt{pack}(\mathit{newlock}_i, \mathit{acquire}_i, \mathit{release}_i) \\ \qquad\qquad\qquad\qquad : \exists \alpha. (\mathbf{1} \to \alpha) \times (\alpha \to \mathbf{1}) \times (\alpha \to \mathbf{1}).\end{array}$$

Proof. The theorem follows from LR-PACK (with $\mathsf{lockInt}$ as the witness for the existential type), LR-PAIR and Lemmas 5.1, 5.3 and 5.4. □

6 Interpretation in Iris

The calculus defined in Section 3 is interpreted in Iris. The interpretation of the judgements are not very different from the encoding of [3].

6.1 Ghost thread pool

The thread pool (unital) resource algebra is defined as follows.

$$\mathrm{TP} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \textsc{Ex}(\mathit{Expr})$$

Given a thread pool $T \in \mathit{ThreadPool}$ we can obtain $\bar{T} \in \mathrm{TP}$ by folding over the list, i.e. $\overline{[e_1, \ldots, e_n]} = \{1 \mapsto e_1, \ldots, n \mapsto e_n\}$. The resource algebra of configurations is obtained as a product

$$\textsc{Cfg} \triangleq \textsc{Auth}(\mathrm{TP} \times \mathrm{H})$$

where H is the heap resource algebra $\mathit{Loc} \xrightarrow{\text{fin}} \mathbb{Q} \times \textsc{Ag}(\mathit{Val})$. The basic assertions are then defined as follows.

$$l \mapsto_{\mathsf{s}} v \triangleq \circ\,(\emptyset, \{l \mapsto (1, \mathsf{ag}(v))\})$$

$$j \Mapsto e \triangleq \circ\,(\{j \mapsto \mathsf{ex}(e)\}, \emptyset)$$

Notice that, as usual, $l \mapsto_{\mathsf{s}} v$ is a timeless proposition.

For the rest of this section we assume that Iris is instantiated with the configuration RA under the name γ_{cfg}. The global invariant that we want to maintain for the configuration RA is spec_ctx (we implicitly coerce thread pools and states to the corresponding RAs).

$$\mathsf{spec\_ctx}(\rho) \triangleq \boxed{\exists T\ \sigma, \ulcorner \rho \to^* (T, \sigma) \urcorner * \boxed{\bullet\,(T, \sigma)}^{\gamma_{cfg}}}^{\mathsf{specN}}$$

The invariant states that the current configuration that we own is reachable from some original configuration ρ.

Rules for Cfg. We have the following "symbolic execution" updates for the configuration RA.

$$\textsc{step-pure}\quad \frac{e \to_{\mathsf{pure}} e' \qquad \uparrow\mathsf{specN} \subseteq \mathcal{E} \qquad \mathsf{spec\_ctx}(\rho)}{j \Mapsto K[e] \;{}^{\mathcal{E}}\!\Rrightarrow\!\!*^{\mathcal{E}}\; j \Mapsto K[e']}$$

$$\textsc{step-alloc}\quad \frac{\uparrow\mathsf{specN} \subseteq \mathcal{E} \qquad \mathsf{spec\_ctx}(\rho)}{j \Mapsto K[\texttt{fork}\ \{e\}] \;{}^{\mathcal{E}}\!\Rrightarrow\!\!*^{\mathcal{E}}\; \exists i, j \Mapsto K[()] * i \Mapsto e}$$

$$\textsc{step-alloc}\quad \frac{\uparrow\mathsf{specN} \subseteq \mathcal{E} \qquad \mathsf{spec\_ctx}(\rho)}{j \Mapsto K[\texttt{ref}(v)] \;{}^{\mathcal{E}}\!\Rrightarrow\!\!*^{\mathcal{E}}\; \exists l, j \Mapsto K[l] * l \mapsto_{\mathsf{s}} v}$$

STEP-LOAD

$$\frac{l \mapsto_{\mathsf{s}} v \qquad \uparrow\mathsf{specN} \subseteq \mathcal{E} \qquad \mathsf{spec\_ctx}(\rho)}{j \Rrightarrow K[!\ l] \ {}^{\mathcal{E}}{\Rrightarrow}\!\!\!\ast^{\mathcal{E}}\ j \Rrightarrow K[v] * l \mapsto_{\mathsf{s}} v}$$

STEP-STORE

$$\frac{l \mapsto_{\mathsf{s}} v \qquad \uparrow\mathsf{specN} \subseteq \mathcal{E} \qquad \mathsf{spec\_ctx}(\rho)}{j \Rrightarrow K[l \leftarrow v'] \ {}^{\mathcal{E}}{\Rrightarrow}\!\!\!\ast^{\mathcal{E}}\ j \Rrightarrow K[()] * l \mapsto_{\mathsf{s}} v'}$$

STEP-CAS-FAIL

$$\frac{l \mapsto_{\mathsf{s}} v \qquad v \neq v_1 \qquad \uparrow\mathsf{specN} \subseteq \mathcal{E} \qquad \mathsf{spec\_ctx}(\rho)}{j \Rrightarrow K[\mathtt{CAS}(l, v_1, v_2)] \ {}^{\mathcal{E}}{\Rrightarrow}\!\!\!\ast^{\mathcal{E}}\ j \Rrightarrow K[\mathtt{false}] * l \mapsto_{\mathsf{s}} v}$$

STEP-CAS-SUC

$$\frac{l \mapsto_{\mathsf{s}} v_1 \qquad \uparrow\mathsf{specN} \subseteq \mathcal{E} \qquad \mathsf{spec\_ctx}(\rho)}{j \Rrightarrow K[\mathtt{CAS}(l, v_1, v_2)] \ {}^{\mathcal{E}}{\Rrightarrow}\!\!\!\ast^{\mathcal{E}}\ j \Rrightarrow K[\mathtt{true}] * l \mapsto_{\mathsf{s}} v_2}$$

6.2 Encoding logical relations

The semantic domain, in which we are going to interpret the types is a set of persistent predicates over $\mathit{Val} \times \mathit{Val}$:

$$\mathcal{D} \triangleq \mathit{Val} \times \mathit{Val} \xrightarrow{\text{ne}} \mathit{iProp}$$

An interpretation function $[\![-]\!]$ takes a type and a list $\mathit{List}\ \mathcal{D}$ of semantic types, which is used to interpret type variables. The interpretation of types and the interpretation of expressions are defined simultaneously.

$$[\![-]\!]_e(\mathcal{E}) : (\mathit{List}\ \mathcal{D} \xrightarrow{\text{ne}} \mathcal{D}) \xrightarrow{\text{ne}} \mathit{List}\ \mathcal{D} \xrightarrow{\text{ne}} \mathit{Expr} \times \mathit{Expr} \xrightarrow{\text{ne}} \mathit{iProp}$$

$$[\![\tau]\!]_e(\mathcal{E})(\Delta)(e_1, e_2) \triangleq \forall j\ K,\ j \Rrightarrow K[e_2] \ {}^{\top}{\Rrightarrow}\!\!\!\ast^{\mathcal{E}}\ \mathsf{wp}\ e_1\ \{v.\ \exists v',\ j \Rrightarrow K[v'] * \tau(\Delta)(v, v')\}$$

We describe the interpretation of types using set-theoretic notation; it is straightforward to transform every such description into a predicate.

$$\begin{array}{lll}
[\![\mathbf{1}]\!]_\Delta & \triangleq & \{((), ())\} \\
[\![\mathbf{2}]\!]_\Delta & \triangleq & \{(\mathtt{true}, \mathtt{true}), (\mathtt{false}, \mathtt{false})\} \\
[\![\mathbf{N}]\!]_\Delta & \triangleq & \{(n, n) \mid n \in \mathbb{N}\} \\
[\![\tau \times \sigma]\!]_\Delta & \triangleq & \{((v_1, v_2), (v_1', v_2')) \mid (v_1, v_1') \in [\![\tau]\!]_\Delta * (v_2, v_2') \in [\![\sigma]\!]_\Delta\} \\
[\![\tau + \sigma]\!]_\Delta & \triangleq & \{(\mathtt{inl}\ v, \mathtt{inl}\ v') \mid (v, v') \in [\![\tau]\!]_\Delta\} \cup \{(\mathtt{inr}\ v, \mathtt{inr}\ v') \mid (v, v') \in [\![\sigma]\!]_\Delta\} \\
[\![\tau \to \sigma]\!]_\Delta & \triangleq & \{(v, v') \mid \Box(\forall (w, w') \in [\![\tau]\!]_\Delta,\ [\![\sigma]\!]_e(\top)(\Delta)(v\ w, v'\ w')\} \\
[\![\forall(\tau)]\!]_\Delta & \triangleq & \{(v, v') \mid \Box(\forall \tau i \in \mathcal{D},\ [\![\tau]\!]_e(\top)(\tau i :: \Delta)(v\ [], v'\ [])\} \\
[\![\exists(\tau)]\!]_\Delta & \triangleq & \{(\mathtt{pack}\ v, \mathtt{pack}\ v') \mid \Box(\exists \tau i \in \mathcal{D},\ [\![\tau]\!]_{(\tau i :: \Delta)}(v, v')\} \\
[\![\mu(\tau)]\!]_\Delta & \triangleq & \{(\mathtt{fold}\ v, \mathtt{fold}\ v') \mid \triangleright [\![\tau]\!]_{(\mu(\tau) :: \Delta)}(v, v')\} \\
I_{\mathsf{rev}} & : & \mathit{Loc} \times \mathit{Loc} \to \mathcal{D} \xrightarrow{\text{ne}} \mathit{iProp} \\
I_{\mathsf{rev}}(l, l', \tau i) & \triangleq & \exists v\ v',\ l \mapsto_{\mathsf{i}} v * l' \mapsto_{\mathsf{s}} v' * \tau_i(v, v') \\
[\![\mathbf{ref}\ \tau]\!]_\Delta & \triangleq & \{(l, l') \mid \boxed{I_{\mathsf{rev}}(l, l', [\![\tau]\!]_\Delta)}^{\mathsf{logN}.(l,l')}\} \\
[\![x_i]\!]_\Delta & \triangleq & \Box\, \Delta(i)
\end{array}$$

Note that in the interpretation of the recursive types, the truth-value $[\![\tau]\!]_{(\mu(\tau)::\Delta)}(v, v')$ is under the later modality, which allows to define the interpretation of $[\![\mu(\tau)]\!]$ as a fixed point.

Remark 6.1. *In the interpretation of type variables we use the persistence modality. This ensures that the value interpretation $[\![\tau]\!]_\Delta$ is persistent, even if some relations in Δ are not. However, morally any relation in Δ should be persistence. Consider, for instance, a refinement at type α. By the definition of the value interpretation we would have to prove $\Delta(\alpha)$ using only persistent resources, and that might be hard or impossible if $\Delta(\alpha)$ is not persistent itself. For instance, if $\Delta(\alpha)(v_1, v_2)$ is a heap assertion, then $\Box\, \Delta(\alpha)(v_1, v_2)$ is logically equivalent to False.*

Remark 6.2. *In the definitions related to the ghost thread pool we use the invariant name specN, whereas for the interpretation of the reference types we use the invariant name logN. To that extent we assume that both specN and logN share common namespace $\mathsf{logrelN}$, which is used in the rules in Section 3.*

Proposition 6.3. *The value interpretation is persistent: $\forall \tau\, \Delta\, w,\ \mathsf{persistent}([\![\tau]\!]_\Delta(w))$.*

Proof. By induction on τ. ☐

The interpretation of environments is defined as follows.

$$\begin{array}{lcl} [\![\Gamma]\!]_* & : & (\mathit{List}\ \mathcal{D}) \to (\mathit{Map}\ \mathit{Var}\ (\mathit{Val} \times \mathit{Val})) \to \mathit{iProp} \\ [\![\Gamma]\!]_{*\Delta}(\vec{v}) & \triangleq & \ulcorner \mathrm{dom}(\vec{v}) = \mathrm{dom}(\Gamma) \urcorner * \forall (x,\tau) \in \Gamma,\ [\![\tau]\!]_\Delta(\vec{v}(x)) \end{array}$$

Proposition 6.4. *For any Δ, Γ, τ, $\vec{v} \in \mathit{Map}\ \mathit{Var}\ (\mathit{Val} \times \mathit{Val})$, and for any $(v, v') \in \mathit{Val} \times \mathit{Val}$, $x \in \mathit{Var}$,*

$$[\![\tau]\!]_\Delta(v, v') * [\![\Gamma]\!]_{*\Delta}(\vec{v}) \vdash [\![x : \tau, \Gamma]\!]_{*\Delta}(\vec{v}\,[x \leftarrow (v, v')])$$

Note that the other direction does not hold if $\Gamma(x)$ is already defined.

Proposition 6.5. *For any Δ, Γ, τ, $\vec{v} \in \mathit{Map}\ \mathit{Var}\ (\mathit{Val} \times \mathit{Val})$, and for any $(v, v') \in \mathit{Val} \times \mathit{Val}$, $x \in \mathit{Var}$, such that $x \notin \mathrm{dom}(\Gamma)$:*

$$[\![\tau]\!]_\Delta(v, v') * [\![\Gamma]\!]_{*\Delta}(\vec{v}) \dashv\vdash [\![x : \tau, \Gamma]\!]_{*\Delta}(\vec{v}\,[x \leftarrow (v, v')])$$

Next we define an environment substitution. Given a map $\vec{v} \in \mathit{Map}\ \mathit{Var}\ (\mathit{Val} \times \mathit{Val})$,

$$\begin{array}{lcll} \vec{v}_1(e) & \triangleq & e[/\vec{v}.*1] & \text{where } e[/m] \text{ is parallel substitution} \\ \vec{v}_2(e) & \triangleq & e[/\vec{v}.*2] & \text{and } \vec{v}.*i \text{ is } \pi_i\ \langle\$\rangle\ \vec{v} \end{array}$$

Refinement judgement. The interpretation of the logical refinement judgements is given by

$$\Delta \mid \Gamma \models_{\mathcal{E}} e \precsim e' : \tau \triangleq \forall \vec{v}\,\rho,\ \mathsf{spec\_ctx}(\rho) \twoheadrightarrow \Box\, [\![\Gamma]\!]_{*\Delta}(\vec{v}) \twoheadrightarrow [\![\tau]\!]_e(\mathcal{E})(\Delta)(\vec{v}_1(e), \vec{v}_2(e'))$$

6.3 Deriving the symbolic execution rules

In this section we examine how to derive the most general rules such as LR-PURE-L, LR-PURE-R, LR-WP-ATOMIC-L, and a new rule LR-STEP-R.

Rules on the left hand side.

Lemma 6.6. *The proof rule* LR-PURE-L *is sound.*

Proof. We wish to prove $\Delta \mid \Gamma \models K[e] \precsim t : \tau$ from $\triangleright \Delta \mid \Gamma \models K[e'] \precsim t : \tau$, $e \to_{\mathsf{pure}} e'$ with e and e' closed. Unfolding the definitions, we are to show

$$\mathsf{spec\_ctx}(\rho) * \Box\, [\![\Gamma]\!]_{*\Delta}(\vec{v}) * j \Rrightarrow \vec{v}_2(K'[t]) \vdash {\Rrightarrow}_{\top}\, \mathsf{wp}\, \vec{v}_1(K[e])\, \{v.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_{\Delta}(v, v')\}$$

We can get rid of the fancy update modality and rewrite $\vec{v}_1(K[e])$ as $\vec{v}_1(K)[\vec{v}_1(e)]$ where we extend the definition of substitution to evaluation contexts. Furthermore, since e is closed, $\vec{v}_1(e) = e$. Thus we are left with proving.

$$\mathsf{wp}\, \vec{v}_1(K)[e]\, \{v.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_{\Delta}(v, v')\}$$

And according to WP-BIND, it suffices to show

$$\mathsf{wp}\, e\, \{v.\, \mathsf{wp}\, \vec{v}_1(K)[v]\, \{w.\, \exists w',\, j \Rrightarrow K'[w'] * [\![\tau]\!]_{\Delta}(w, w')\}\}$$

We can then apply WP-LIFT-LR-PURE-STEP to obtain the goal

$$\begin{aligned}&\mathsf{spec\_ctx}(\rho), \Box\, [\![\Gamma]\!]_{*\Delta}(\vec{v}), j \Rrightarrow \vec{v}_2(K'[t]),\\ &\qquad \triangleright \Delta \mid \Gamma \models K[e'] \precsim t : \tau \vdash \triangleright\, \mathsf{wp}\, e'\, \{v.\, \mathsf{wp}\, \vec{v}_1(K)[v]\, \{w.\, \exists w',\, j \Rrightarrow K'[w'] * [\![\tau]\!]_{\Delta}(w, w')\}\}\end{aligned}$$

We can then get rid of the later modalities on both sides of the turnstile and apply WP-BIND-INV; it then remains to show

$$\begin{aligned}&\mathsf{spec\_ctx}(\rho), \Box\, [\![\Gamma]\!]_{*\Delta}(\vec{v}), j \Rrightarrow \vec{v}_2(K'[t]),\\ &\qquad \Delta \mid \Gamma \models K[e'] \precsim t : \tau \vdash \mathsf{wp}\, \vec{v}_1(K)[e']\, \{v.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_{\Delta}(v, v')\}\end{aligned}$$

This follows from instantiating $\Delta \mid \Gamma \models K[e'] \precsim t : \tau$ with $\mathsf{spec\_ctx}(\rho)$, $\Box\, [\![\Gamma]\!]_{*\Delta}(\vec{v})$, and $j \Rrightarrow \vec{v}_2(K'[t])$. □

Lemma 6.7. *The proof rule* LR-WP-ATOMIC-L *is sound.*

Proof. Assume that ${}^{\top}{\Rrightarrow}^{\mathcal{E}}\, \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau\}$. Let $\vec{v}$ be such that $[\![\Gamma]\!]_{*\Delta}(\vec{v})$ and let j and K' be such that $j \Rrightarrow \vec{v}_2(K'[t])$. We are to show

$$\mathsf{wp}\, \vec{v}_1(K[e])\, \{v.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_{\Delta}(v, v')\}.$$

Because e is closed, $\vec{v}_1(K[e]) = \vec{v}_1(K)[e]$. By WP-BIND it suffices to show

$$\mathsf{wp}\, e\, \{v.\, \mathsf{wp}\, \vec{v}_1(K)[v]\, \{v_0.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_{\Delta}(v_0, v')\}\}.$$

Applying WP-MASK-MONO, WP-ATOMIC and FUPD-MONO, we can get rid of the fancy update modality, resulting in the sequent

$$j \Rrightarrow \vec{v}_2(K'[t]) * \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \left\{v.\, {}^{\top}{\Rrightarrow}^{\mathcal{E}}\, \mathsf{wp}\, \vec{v}_1(K)[v]\, \{v_0.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_{\Delta}(v_0, v')\}\right\}$$

Finally, we can apply WP-MONO, and the result follows from the definition of $\Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau$. □

Rules on the right hand side.

Lemma 6.8. *The proof rule* LR-PURE-R *is sound.*

Proof. Unfolding the definitions, we see that we have to show

$$^{\top}{\Rrightarrow}^{\mathcal{E}}\, \mathsf{wp}\, \vec{v}_1(t)\, \{v.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_\Delta(v, v')\}$$

from $\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e'] : \tau$, $\Box [\![\Gamma]\!]_{*\Delta}(\vec{v})$, and $j \Rrightarrow K'[\vec{v}_2(K[e])]$. The latter resource can be rewritten as

$$j \Rrightarrow K'[\vec{v}_2(K)[e]]$$

since e is closed. By FUPD-TRANS it suffices to show

$$\ldots, j \Rrightarrow K'[\vec{v}_2(K)[e]] \vdash {\Rrightarrow}_{\top}\, ^{\top}{\Rrightarrow}^{\mathcal{E}}\, \mathsf{wp}\, \vec{v}_1(t)\, \{v.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_\Delta(v, v')\}$$

Then, by STEP-PURE, we can update this resource to ${\Rrightarrow}_{\top}\, j \Rrightarrow K'[\vec{v}_2(K)[e']]$, and cancel the fancy update modality on both sides of the turnstile to obtain

$$\ldots, j \Rrightarrow K'[\vec{v}_2(K)[e']] \vdash\, ^{\top}{\Rrightarrow}^{\mathcal{E}}\, \mathsf{wp}\, \vec{v}_1(t)\, \{v.\, \exists v',\, j \Rrightarrow K'[v'] * [\![\tau]\!]_\Delta(v, v')\}$$

The result then follows by instantiating $\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e'] : \tau$ with the appropriate resources. ☐

Using the notation of Section 6.1 we can formulate a general rule for performing symbolic execution on the right hand side of the refinement judgement.

LR-STEP-R

$$\frac{\forall \rho\ j\ K',\ \mathsf{spec\_ctx}(\rho) \mathrel{-\!\!*} (j \Rrightarrow K'[e] \Rrightarrow\!\!\!*_{\mathcal{E}}\ \exists v,\ (j \Rrightarrow K'[v]) * \Phi(v)) \qquad \forall v,\ \Phi(v) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[v] : \tau \qquad \mathrm{closed}(e)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[e] : \tau}$$

Using LR-STEP-R we can derive all stateful symbolic execution rules for the right hand side (Section 3.1).

6.4 Soundness

The proof that our logical relation is sound w.r.t. contextual refinement follows a fairly standard strategy, and it relies on the *adequacy* of the weakest precondition calculus in Iris [1].

Definition 6.9. *A program e in an initial state σ is* adequate *for a pure predicate $\varphi : Val \to \mathsf{Prop}$ if for any thread pool T and a state σ' such that $([e], \sigma) \to_{\mathsf{tp}}^{*} (T, \sigma')$:*

1. (Safety) *For any $e' \in T$ either e' is a value, or (e', σ') is reducible;*
2. (Result) *If $v \in T$ is a value, then $\varphi(v)$ holds.*

Note that adequacy itself is a *pure statement*, formulated outside separation logic.

Theorem 6.10 ([1, Theorem 6])**.** *If φ is a pure predicate, and $\mathsf{wp}\, e\, \{v. \ulcorner \varphi(v) \urcorner\}$ is derivable in Iris, then e is adequate for φ w.r.t. any initial state σ.*

Lemma 6.11. *If* $\mathrm{closed}(\mathrm{dom}\,\Gamma, e)$ *and* $[\,\mathcal{C}\,] : (\Gamma \vdash \tau) \Rightarrow (\Gamma' \vdash \tau')$*, then* $\mathrm{closed}(\mathrm{dom}\,\Gamma', \mathcal{C}[e])$*.*

Proof. By induction on the derivation of the context typing, using the fact that if $\Delta \vdash t : \sigma$, then $\mathrm{closed}(\mathrm{dom}\,\Delta, t)$. □

Lemma 6.12 (Precongruence)**.** *If* $\mathrm{closed}(\mathrm{dom}(\Gamma'), e)$ *and* $\mathrm{closed}(\mathrm{dom}(\Gamma'), e')$*, and* $[\,\mathcal{C}\,] : (\Gamma' \vdash \tau') \Rightarrow (\Gamma \vdash \tau)$ *then*

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

Proof. By induction on the context typing derivation. Most of the cases are trivial. For those context typing judgements that contain typing assumptions we need to use the fundamental property (Lemma 3.2). For instance, for one of the cases in which $[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau' \to \tau)$ we have to show

$$\Delta \mid \Gamma \models \mathcal{C}[e]\ e_2 \precsim \mathcal{C}[e']\ e_2 : \tau$$

from the assumptions that $\Gamma \vdash e_2 : \tau'$ and the induction hypothesis

$$\Delta \mid \Gamma \models \mathcal{C}[e] \precsim \mathcal{C}[e'] : \tau' \to \tau.$$

For this we apply the fundamental property to obtain $\Delta \mid \Gamma \models e_2 \precsim e_2 : \tau'$ and then use LR-APP.

The most tricky case is the context typing rule

$$\frac{[\,\mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (x : \tau, f : (\tau \to \tau'), \Gamma \vdash \tau')}{[\,\mathsf{rec}\ f\ x = \mathcal{C}\,] : (\Gamma' \vdash \sigma) \Rightarrow (\Gamma \vdash \tau \to \tau')}$$

The goal is

$$\Delta \mid \Gamma \models \mathsf{rec}\ f\ x = \mathcal{C}[e] \precsim \mathsf{rec}\ f\ x = \mathcal{C}[e'] : \tau \to \tau'$$

and the induction hypothesis then gives us

$$\Delta \mid (x : \tau, f : (\tau \to \tau'), \Gamma) \models \mathcal{C}[e] \precsim \mathcal{C}[e'] : \tau'$$

with the assumption

$$\Box(\forall \Delta,\ \Delta \mid \Gamma' \models e \precsim e' : \tau).$$

To reduce the goal we apply the rule LR-REC. We than have to show some closedness conditions, which are discharged using the closedness assumptions on e and e' and Lemma 6.11. The reduced goal that we get is

$$\Box(\Delta \mid (x : \tau, f : (\tau \to \tau'), \Gamma) \models \mathcal{C}[e] \precsim \mathcal{C}[e'] : \tau')$$

which can be obtained from the induction hypothesis. Note the presence of the $\Box$ modality in the reduced goal – that is the reason why the assumption for the lemma had to put under the $\Box$ modality. □

Lemma 6.13 (Adequacy of logical relations)**.** *If* $\Delta \mid \emptyset \models e \precsim e' : \tau$ *is derivable in logic, then* e *is adequate w.r.t. any initial state for the predicate*

$$\varphi(v) = \exists T'\, \sigma'\, v',\ ([e'], \emptyset) \to_{\mathsf{tp}}^{*} ([v'] \mathbin{+\!\!+} T', \sigma') \wedge (\mathrm{ObsType}(\tau') \to v = v')$$

One immediate implication of the adequacy lemma is the type safety of the target language.

Theorem 6.14 (Type safety)**.** *If* $\emptyset \vdash e : \tau$*, then* e *is* safe*, i.e. if* $([e], \sigma) \to_{\mathsf{tp}}^{*} (T, \sigma')$ *and* $e' \in T$*, then either* e' *is a value or it is reducible.*

Proof. By the fundamental property of logical relations (Lemma 3.2) we obtain that $\emptyset \mid \emptyset \models e \precsim e : \tau$.

Then, by Lemma 6.13, it is the case that e is adequate for some predicate w.r.t any initial state. Adequacy trivially implies safety. □

Theorem 6.15 (Soundness)**.** *If* closed(dom(Γ), e) *and* closed(dom(Γ), e')*, and* $(\forall \Delta,\ \Delta \mid \Gamma \models e \precsim e' : \tau)$ *is derivable in logic, then* $\Gamma \vdash e \precsim_{ctx} e' : \tau$

Proof.

1. Suppose that $(\forall \Delta,\ \Delta \mid \Gamma \models e \precsim e' : \tau)$ and $[\,\mathcal{C}\,] : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \tau')$ for some observable type τ'
2. Furthermore, suppose that $([\mathcal{C}[e_1]], \emptyset) \to_{\mathsf{tp}}^{*} ([v] \mathbin{+\!\!+} T, \sigma)$. We are to show: $\exists T'\, \sigma',\ ([\mathcal{C}[e_2]], \emptyset) \to_{\mathsf{tp}}^{*} ([v] \mathbin{+\!\!+} T', \sigma')$.
3. By precongruence (Lemma 6.12), $(\forall \Delta,\ \Delta \mid \emptyset \models \mathcal{C}[e] \precsim \mathcal{C}[e'] : \tau')$.
4. The result the follows by adequacy (Lemma 6.13) and the two previous points.

□

Notes on formalisation

The encoding presented in this section has been initially formalised by Amin Timany, Robbert Krebbers, and Lars Birkedal [3, 6]. The main difference is that we have extended the interpretation with masks, allowing the users of the logic to open invariants around the refinement judgements.

The ghost thread pool construction and its basic properties are described in `logrel/threadpool.v`. The rules for the ghost thread pool are proved in `logrel/rules_threadpool.v`. The encoding of the value and expression interpretations is located in `logrel/semtypes.v`. The refinement judgement is defined in `logrel/logrel_binary.v`. Symbolic execution rules (and some other primitive rules) are proved in `logrel/rules.v`. Lemma 6.12 and Theorem 6.15 are proved in `logrel/contextual_refinement.v` and `logrel/soundness_binary.v` resp.

7 Further examples

In this section we present some further examples demonstrating various features of the system.

7.1 Representation independence

In this example we will demonstrate how to prove refinement of two representations of the same abstract data type. We are going to consider an abstract data type which interface is provided by the following type

$$\mathsf{TBit} \triangleq \exists \alpha. \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathbf{2})$$

This is a simple type representing a *bit*, which consists of the initial state of the bit, the function that flips the bit, and the function that converts the bit to a boolean value.

Boolean bit. Perhaps, the simplest implementation of the bit interface is the one that uses booleans for the internal state.

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

Natural numbers bit. Our second implementation models a bit by a natural number from a set $\{0, 1\}$.

$$\mathsf{flipnat} \triangleq \lambda n.\, \texttt{if}\, n = 0\, \texttt{then}\, 1\, \texttt{else}\, 0 : \mathbf{N} \to \mathbf{N}$$

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

Refinement. Let Γ, Δ be arbitrary. We are to prove the following refinement.

Theorem 7.1. *The following judgement is derivable.*

$$\Delta \mid \Gamma \models \mathit{bitbool} \precsim \mathit{bitnat} : \mathit{TBit}$$

Proof. (We ignore the liftings of Γ since it is not really important). In order to prove the refinement, we appeal to LR-PACK. Thus, we have to pick a relation τ_i that would link the underlying types of the two representation. A good candidate is

$$\tau_i(b, n) \triangleq (b = \texttt{true} \wedge n = 1) \vee (b = \texttt{false} \wedge n = 0).$$

It remains to show

$$(\tau_i, \Delta) \mid \Gamma \models (\texttt{true}, \lambda b.\, b {\oplus} \texttt{true}, \lambda b.\, b) \precsim (1, \mathsf{flipnat}, \lambda n.\, n = 1) : \alpha {\times} (\alpha \to \alpha) {\times} (\alpha \to \mathbf{2}).$$

By repeatedly applying LR-PAIR we get three new goals

- $(\tau_i, \Delta) \mid \Gamma \models \texttt{true} \precsim 1 : \alpha$, which amounts to showing $\tau_i(\texttt{true}, 1)$ by LR-VAL; this holds trivially.
- $(\tau_i, \Delta) \mid \Gamma \models \lambda b.\, b \oplus \texttt{true} \precsim \mathsf{flipnat} : \alpha \to \alpha$; by LR-CLOSURE and LR-REC-L,LR-REC-R it suffices to show

  $$(b = \texttt{true} \wedge n = 1) \vee (b = \texttt{false} \wedge n = 0) \vdash (\tau_i, \Delta) \mid \Gamma \models b \oplus \texttt{true} \precsim \texttt{if}\, n = 0\, \texttt{then}\, 1\, \texttt{else}\, 0 : \alpha.$$

  That statement is proved by case analysis on b and n, appealing to LR-VAL.
- $(\tau_i, \Delta) \mid \Gamma \models \lambda b.\, b \precsim \lambda n.\, n = 1 : \alpha \to \mathbf{2}$; similar to the previous item, it suffices to show

  $$(b = \texttt{true} \wedge n = 1) \vee (b = \texttt{false} \wedge n = 0) \vdash (\tau_i, \Delta) \mid \Gamma \models b \precsim n = 1 : \mathbf{2}.$$

  This is proved by case analysis and LR-LITERAL.

□

"Heapification". Given a module m : TBit that implements a bit interface, we can construct a module heapify(m) we implements the bit interface by holding a value of the underlying type of bit from m in a reference, and performing all the operations on that reference. This is done by the following function.

```
heapify(m) = unpack m as (init, flip, view) in
  let x = init in
  let l = newlock () in
  let flip' () = acquire l; x ← flip (!x); release l in
  let view' () = view (!x) in
  pack ((), flip', view')
```

Theorem 7.2. *For any Δ and Γ,*

$$\Delta \mid \Gamma \models \mathit{heapify}(\mathit{bitbool}) \precsim \mathit{heapify}(\mathit{bitnat}) : \mathit{TBit}$$

Proof. Note that $\Gamma \vdash_t \mathsf{heapify}(-) : \mathsf{TBit} \to \mathsf{TBit}$. Hence, by the fundamental property, $\Delta \mid \Gamma \models \mathsf{heapify}(-) \precsim \mathsf{heapify}(-) : \mathsf{TBit} \to \mathsf{TBit}$. The result then follows by LR-APP and Theorem 7.1. □

7.2 Irreversible state change

Consider the following Pitts and Stark's "awkward" example [4]:

$$\begin{aligned} e_1 &:= \texttt{let}\, x = \texttt{ref}(0)\, \texttt{in}\, \lambda f.\, (x \leftarrow 1; f\ (); !\, x) \\ e_2 &:= \texttt{let}\, x = \texttt{ref}(1)\, \texttt{in}\, \lambda f.\, (f\ (); !\, x) \\ e_3 &:= \lambda f.\, (f\ (); 1) \end{aligned}$$

All the functions have the type $(\mathbf{1} \to \mathbf{1}) \to \mathbf{1}$ and are contextually equivalent. We will show it through a chain of refinements: $e_1 \precsim e_2 \precsim e_3 \precsim e_1$ and use the transitivity of contextual refinement. Intuitively, the reason why those functions are equivalent is because the variable x is *local* and f can only affect the value of x if it invokes the closure itself.

Notably, the following program is *not* equivalent to any of the above:

$$e' := \texttt{let}\, x = \texttt{ref}(0)\, \texttt{in}\, \lambda f.\, (x \leftarrow 0; f\ (); x \leftarrow 1; !\, x)$$

The reason for that is that the callback f can spawn another thread invoking the closure. Then, depending on the scheduler, this thread can enter the callback directly before the $!\,x$ operation of the original thread commences. Specifically, consider the following program context K:

```
let g = [•] in
let f = fun () => fork { g (fun () => ()) }
g f
```

Then $K[e_3]$ always terminates with 1 as the value; on the other hand there is an execution of $K[e']$ which terminates in 0:

1. $e'\ f$ starts executing, assigning value 0 to x;
2. it then spawns a thread i which is going to execute $(e'\ \mathsf{id})$;
3. the main thread continues its executing assigning 1 to x;
4. the main thread then yields control to thread i which enters the body of $(e'\ \mathsf{id})$ and assigns 0 to x;
5. thread i yields to the main thread which performs $!\,x$ and returns 0.

Lemma 7.3. *For any* Γ *and* Δ *it is the case that*

$$\Delta \mid \Gamma \models e_2 \precsim e_3 : (\mathbf{1} \to \mathbf{1}) \to \mathbf{N}$$

Proof. After applying LR-ALLOC-L we are left with the goal

$$x \mapsto_{\mathsf{i}} 1 \vdash \Delta \mid \Gamma \models \lambda f.\, (f\ (); !\, x) \precsim \lambda f.\, (f\ (); 1) : (\mathbf{1} \to \mathbf{1}) \to \mathbf{N}.$$

At this point we would like to prove the refinement of closures using LR-ARROW. However, the only resources that are going to be available for the refinement proof are the persistent ones. Intuitively, the reason for that is a closure can be stored somewhere and invoked at an arbitrary point in the future, when we might or might not have some non-persistent resources.

For that purpose we are going to put the resource $x \mapsto_{\mathsf{i}} 1$ in an invariant $\boxed{x \mapsto_{\mathsf{i}} 1}^{\mathcal{N}}$, which amounts to saying that each atomic operation has to ensure that the invariant is still maintained after the execution. Formally, we are to show

$$\boxed{x \mapsto_{\mathsf{i}} 1}^{\mathcal{N}} \vdash \Delta \mid \Gamma \models \lambda f.\, (f\ (); !\, x) \precsim \lambda f.\, (f\ (); 1) : (\mathbf{1} \to \mathbf{1}) \to \mathbf{N}.$$

After applying LR-ARROW and LR-REC-L, LR-REC-R we are to show

$$\boxed{x \mapsto_{\mathsf{i}} 1}^{\mathcal{N}} \vdash \Delta \mid \Gamma \models (f_1\ (); !\, x) \precsim (f_2\ (); 1) : (\mathbf{1} \to \mathbf{1}) \to \mathbf{N}$$

under the assumption

$$\Delta \mid \Gamma \models f_1 \precsim f_2 : \mathbf{1} \to \mathbf{1}.$$

Using LR-SEQ we decompose our goal into two:

1. $\Delta \mid \Gamma \models f_1\ () \precsim f_2\ () : \mathbf{1}$;
2. $\Delta \mid \Gamma \models !\, x \precsim 1 : \mathbf{N}$.

The former goal follows from the assumption on f_1 and f_2 and the compatibility lemmas LR-APP, LR-VAL.

The later goal is established as follows. First, we apply LR-LOAD-L resulting in

$$\boxed{x \mapsto_{\mathsf{i}} 1}^{\mathcal{N}} \vdash {}^{\top}\!\!\Rrightarrow^{\top \setminus \mathcal{N}} \exists v, \triangleright x \mapsto_{\mathsf{i}} v * \triangleright(x \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\top \setminus \mathcal{N}} v \precsim 1 : \mathbf{N})$$

This allows us to open the invariant to get to

$$\cdots * x \mapsto_{\mathsf{i}} 1 \vdash \exists v, \triangleright x \mapsto_{\mathsf{i}} v * \triangleright(x \mapsto_{\mathsf{i}} v \twoheadrightarrow \Delta \mid \Gamma \models_{\top \setminus \mathcal{N}} v \precsim 1 : \mathbf{N})$$

which we can reduce to

$$\cdots * x \mapsto_{\mathsf{i}} 1 \vdash \Delta \mid \Gamma \models_{\top \setminus \mathcal{N}} 1 \precsim 1 : \mathbf{N}$$

at this point we have no other choice but to close the invariant (using FUPD-LOGREL) and end up with

$$\boxed{x \mapsto_{\mathsf{i}} 1}^{\mathcal{N}} \vdash \Delta \mid \Gamma \models 1 \precsim 1 : \mathbf{N}$$

which is an instance of LR-VAL. □

Lemma 7.4. *For any* Γ *and* Δ *it is the case that*

$$\Delta \mid \Gamma \models e_1 \precsim e_2 : (\mathbf{1} \to \mathbf{1}) \to \mathbf{N}$$

Proof. The proof is similar to the previous one, using a different invariant:

$$\boxed{(x \mapsto_{\mathsf{i}} 0 * \mathsf{pending} \vee x \mapsto_{\mathsf{i}} 1 * \mathsf{shot}) * y \mapsto_{\mathsf{s}} 1}^{\mathcal{N}}.$$

□

Lemma 7.5. *For any* Γ *and* Δ *it is the case that*

$$\Delta \mid \Gamma \models e_3 \precsim e_1 : (\mathbf{1} \to \mathbf{1}) \to \mathbf{N}$$

Proof. The proof is similar to the previous one, using a different invariant:

$$\boxed{x \mapsto_{\mathsf{s}} 0 * \mathsf{pending} \vee x \mapsto_{\mathsf{s}} 1 * \mathsf{shot}}^{\mathcal{N}}.$$

□

Notes on formalization

The representation independence example is formalized in `examples/bit.v`.
The irreversible state change example are `refinement1`, `refinement2` and `refinement25` in `examples/various.v`.

8 Notes on logical atomicity

8.1 Logically atomic symbolic execution rules for compound commands

To facilitate composability, we would like to provide free-standing rules for symbolic execution of *compound* statements on both sides of the refinement judgement. Let's return to the counter example from Section 4.2. For instance, in order to prove Equation (4), we would like to have rules for symbolically executing read on the LHS and on the RHS, and then use those rules for proving the refinement. However, consider what happens if we write a lemma for symbolically executing read on the left hand side, in the style of the rules from Section 3.2.

COUNTER-READ-L
$$\frac{c_i \mapsto_{\mathsf{i}} n \qquad \Delta \mid \Gamma \models K[n] \precsim t : \tau}{\Delta \mid \Gamma \models K[\mathsf{read}\ c_i\ ()] \precsim t : \tau}$$

Such rule, albeit sound, is not going to be helpful with proving Equation (4): in order to apply the rule we need to obtain $c_i \mapsto_{\mathsf{i}} n$; for that we have to open up the invariant. However, once the invariant is open, we are left with a masked logical relation of the form $\Delta \mid \Gamma \models_{\mathcal{E}\setminus\uparrow\mathcal{N}} e \precsim t : \tau$. Hence, the COUNTER-READ-L is not applicable. Furthermore, we cannot write down a sound rule for read that would worked for arbitrary masked refinement judgement. The same argument applies to a seemingly standard rule for inc_i:

FG-INCREMENT-L
$$\frac{x \mapsto_{\mathsf{i}} n \qquad x \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[()] \precsim t : \tau}{\Delta \mid \Gamma \models K[\mathsf{inc}_i\ x] \precsim t : \tau}$$

The reason for this is neither read nor inc_i are atomic, as they are compound expressions. However, the expressions are *logically* atomic, i.e. it behaves "as if" it is physically atomic. In a sense both of those functions have a single determined linearisation point. To provide sensible reusable rules we take inspiration from the encoding of logically atomic Hoare triples. The proposed rules are thus:

COUNTER-READ-ATOMIC-L
$$\frac{\begin{array}{ccc} & & \square({}^{\top}\!\!\models\!\!\!\Rrightarrow^{\mathcal{E}} \exists n,\, x \mapsto_{\mathsf{i}} n * R(n) * \\ (x \mapsto_{\mathsf{i}} n * R(n)\ {}^{\mathcal{E}}\!\!\Rrightarrow\!\!\!*^{\top}\ \mathsf{True}) & \wedge & (x \mapsto_{\mathsf{i}} n * R(n) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} K[n] \precsim t : \tau)) \end{array}}{\Delta \mid \Gamma \models K[\mathsf{read}\ x\ ()] \precsim t : \tau}$$

FG-INCREMENT-ATOMIC-L
$$\frac{\begin{array}{ccc} & & \square({}^{\top}\!\!\models\!\!\!\Rrightarrow^{\mathcal{E}} \exists n,\, x \mapsto_{\mathsf{i}} n * R(n) * \\ (x \mapsto_{\mathsf{i}} n * R(n)\ {}^{\mathcal{E}}\!\!\Rrightarrow\!\!\!*^{\top}\ \mathsf{True}) & \wedge & (x \mapsto_{\mathsf{i}} (n+1) * R(n) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} K[n] \precsim t : \tau)) \end{array}}{\Delta \mid \Gamma \models K[\mathsf{inc}_i\ x] \precsim t : \tau}$$

Consider the inc_i rule. Informally, the reason why $\mathsf{inc}_i\ x$ is logically atomic is because it does only two things with the heap: it either reads the value of x (this cannot break any invariants or resources held by other threads), and it either succeeds in incrementing the counter (in an atomic fashion, using compare-and-swap) or it fails to do so, and starts over. In order to understand the logically atomic rule we must think of a way of (symbolically) performing those three steps whenever the resources that we need are shared between threads.

First of all, instead of requiring the resource $x \mapsto_{\mathsf{i}} n$, we require a way of obtaining such a resource. One such a way of obtaining $x \mapsto_{\mathsf{i}} n$ is by opening an invariant; however, an invariant will typically contain more resources then needed. In order not to throw those resources away we collected them in a frame $R(n)$.

Secondly, the atomic compare-and-swap 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 $\Delta \mid \Gamma \models_{\mathcal{E}} K[n] \precsim t : \tau$ with that information. This explains the $(x \mapsto_{\mathsf{i}} (n+1) * R(n) \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} K[()] \precsim t : \tau)$ clause.

If, however, the compare-and-swap 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.

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 ($*$).

Symbolic execution of compound statements on the right hand side. There is no need of writing a logically atomic rule for the right-hand side of a logical refinement. The reason for that is that we can always make *multiple* steps on the right hand side for each single step on the left hand side, even under an opened invariant. The following rule is thus provable using LR-ACQUIRE-R, LR-RELEASE-R:

CG-INCREMENT-R

$$\frac{x \mapsto_{\mathsf{s}} n \qquad l \mapsto_{\mathsf{s}} \texttt{false} \qquad (x \mapsto_{\mathsf{s}} (n+1) * l \mapsto_{\mathsf{s}} \texttt{false} \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[n] : \tau)}{\Delta \mid \Gamma \models_{\mathcal{E}} t \precsim K[\mathsf{inc}_s \ x \ l] : \tau}$$

Using the logically atomic rule. We can now use FG-INCREMENT-ATOMIC-L to actually prove refinement (3).

$$\boxed{I_{\mathsf{cnt}}(l, c_i, c_s)}^{\mathcal{N}} \vdash \Delta \mid \emptyset \models \lambda().\, \mathsf{inc}_i \ c_i \precsim \lambda().\, \mathsf{inc}_s \ c_s \ l : \mathbf{1} \to \mathbf{N}$$

Since the expressions on both sides are functions, we can apply LR-CLOSURE and LR-PURE-L, LR-PURE-R to reduce the goal to:

$$\boxed{I_{\mathsf{cnt}}(l, c_i, c_s)}^{\mathcal{N}} \vdash \Delta \mid \emptyset \models \mathsf{inc}_i \ c_i \precsim \mathsf{inc}_s \ c_s \ l : \mathbf{N}$$

At this point we apply FG-INCREMENT-ATOMIC-L with $R(n) = \mathsf{isLock}(\ell, \texttt{false}) * c_s \mapsto_{\mathsf{s}} n$. After getting rid of the persistence modality, we get a new goal:

$$\boxed{I_{\mathsf{cnt}}(l, c_i, c_s)}^{\mathcal{N}} \vdash {}^{\top}\!\!\Rrightarrow^{\top \setminus \uparrow \mathcal{N}} \exists n, c_i \mapsto_{\mathsf{i}} n * l \mapsto \texttt{false} * c_s \mapsto_{\mathsf{s}} n * \ldots$$

At this point we can open up the invariant, thus getting rid of the fancy update modality. The contents of the invariant provides us with a witness for the existential quantifier and allows us to frame the first three conjuncts. We are left with showing the conjunction

$$(c_i \mapsto_{\mathsf{i}} n * l \mapsto_{\mathsf{s}} \texttt{false} * c_s \mapsto_{\mathsf{s}} n \; {}^{\top\setminus\uparrow\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\top} \; \mathsf{True}) \wedge$$
$$(c_i \mapsto_{\mathsf{i}} (n+1) * l \mapsto_{\mathsf{s}} \texttt{false} * c_s \mapsto_{\mathsf{s}} n \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\top\setminus\uparrow\mathcal{N}} n \precsim \ldots : \mathbf{N})$$

from the invariant closing formula

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

The former conjunct follows direction from the invariant closing formula. It thus remains to show $\Delta \mid \Gamma \models_{\top\setminus\uparrow\mathcal{N}} n \precsim \mathsf{inc}_s \; c_s \; l : \mathbf{N}$ from the resources

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

The proof then proceeds as usual, by symbolically executing the right hand side and closing the invariant.

8.2 General form of a logically atomic relational specification

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

$$\frac{\begin{array}{ccc} & R_2 & \Box({}^{\top}\!\!\Rrightarrow^{\mathcal{E}} \exists x,\, P(x) * R_1(x) * \\ (P(x) * R_1(x) \; {}^{\mathcal{E}}\!\!\Rrightarrow\!\!\ast^{\top} \; \mathsf{True}) & \wedge & (\forall v,\, Q(x,v) * R_1(x) * R_2 \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau)) \end{array}}{\Delta \mid \Gamma \models K[e] \precsim t : \tau}$$

where $P : X \to \mathit{iProp}$ is a predicate describing consumed resources and $Q : X \times \mathit{Val} \to \mathit{iProp}$ is a predicated describing produced resources. In this version, in addition to having an *invariant frame* $R_1 : X \to \mathit{iProp}$ that comprises the persistent resource $P(x) * R_1(x)$ together with the "precondition", we add an *ephemeral frame* R_2 containing all the non-persistent 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 $\Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau$.

The reason for including this frame is mainly technical: the other premise of the rule resides behind the persistently modality. In order to prove such a premise we have to give up all the ephemeral resources. However, we don't really want to throw away all the non-persistent resources that we have, so we give them up only temporarily. Such resources might be required, for example, to close the invariant once the new goal is obtained.

8.3 Atomic triples

Recall the general form of symbolic execution rules and relational triples from Section 4.1. The idea here is that in Iris one defines Hoare triples through weakest precondition. In a similar way we define relational triples through the general for of a symbolic execution rule.

$$\Delta \mid \Gamma \models \{P\}\; e\; \{Q\} \triangleq \forall K\, t\, \tau,\; \Box(P * (\forall v,\; Q(v) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[v] \precsim t : \tau) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[e] \precsim t : \tau)$$

We would like to take the same approach to define a relational version of *logically atomic triples* (see [2, Section 7] and the documentation for `iris-atomic`[1]). Moreover we would like to have something similar to Lemma 4.1 allowing us to reuse proofs of regular atomic triples.

Atomic triples. The following is the definition of logically atomic triples from `iris-atomic`.

For $\alpha : A \to \mathit{iProp}$ and $\beta : A \times \mathit{Val} \to \mathit{iProp}$ and masks $\mathcal{E}_i, \mathcal{E}_o$ define

$$\begin{aligned}
&\langle x.\alpha(x)\rangle\; e\; \langle v, \beta(x,v)\rangle_{\mathcal{E}_i,\mathcal{E}_o} \triangleq \forall P\, Q,\\
&\qquad (P \;{}^{\mathcal{E}_o}{\Rrightarrow\!\!\ast}^{\mathcal{E}_i}\; \exists x : A, \alpha(x) *\\
&\qquad ((\alpha(x) \;{}^{\mathcal{E}_i}{\Rrightarrow\!\!\ast}^{\mathcal{E}_o}\; P) \wedge (\forall v,\; \beta(x,v) \;{}^{\mathcal{E}_i}{\Rrightarrow\!\!\ast}^{\mathcal{E}_o}\; Q(v)))) \mathrel{-\!\!*} \{P\}\, e\, \{Q\}_{\top}
\end{aligned}$$

Relational atomic triples.

$$\begin{aligned}
&\Delta; \Gamma \models_{\mathcal{E}} \langle x.\alpha(x)\rangle\; e\; \langle v, \beta(x,v)\rangle \triangleq \forall K\, t\, \tau\, R_1\, R_2,\\
&\qquad (R_2 * \Box({}^{\top}{\mid\!\!\Rrightarrow}^{\mathcal{E}}\; \exists x : A, \alpha(x) * R_1(x) *\\
&\qquad (\alpha(x) * R_1(x) \;{}^{\mathcal{E}}{\Rrightarrow\!\!\ast}^{\top}\; \mathsf{True}) \wedge\\
&(\forall v,\; \beta(x,v) * R_1(x) * R_2 \mathrel{-\!\!*} \Delta \mid \Gamma \models_{\mathcal{E}} K[v] \precsim t : \tau))) \mathrel{-\!\!*} \Delta \mid \Gamma \models K[e] \precsim t : \tau
\end{aligned}$$

Some differences with the regular/unary version:

- We explicitly have two types of frames R_1 and R_2;
- We can close the invariant (in case the computation is unsuccessful, or access the state without changing it);
- When we succeed we do not close the invariant directly – rather, we get a masked logical relation as a goal. This is needed in case we have to perform some executions on the right hand side before we can close the invariant.

It is the last point that actually prevents us from lifting atomic Hoare triples to relational Hoare triples inside the logic. However, we can prove such a lifting inside the model:

Theorem 8.1. *Given* $\langle x.\alpha(x)\rangle\, e\, \langle v, \beta(x,v)\rangle_{\mathcal{E},\top}$ *one can obtain* $\Delta; \Gamma \models_{\mathcal{E}} \langle x.\alpha(x)\rangle\; e\; \langle v, \beta(x,v)\rangle$ *for any* Δ, Γ.

Proof. After unfolding the definitions, we get the resource $j \Mapsto K'[\,t\,]$ for the RHS. Then we apply the logically atomic triple with $P \triangleq R_2 * j \Mapsto K'[\,t\,]$. The viewshifts can be provided directly. □

[1] `https://gitlab.mpi-sws.org/FP/iris-atomic`

References

[1] Ralf Jung et al. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *Submitted for publication* (2017).

[2] Ralf Jung et al. “Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning”. In: *POPL*. 2015, pp. 637–650.

[3] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive proofs in higher-order concurrent separation logic”. In: *POPL*. 2017, pp. 205–217.

[4] A. M. Pitts and I. D. B. Stark. “Higher Order Operational Techniques in Semantics”. In: ed. by Andrew D. Gordon and Andrew M. Pitts. New York, NY, USA: Cambridge University Press, 1998. Chap. Operational Reasoning for Functions with Local State, pp. 227–274. ISBN: 0-521-63168-8. URL: http://dl.acm.org/citation.cfm?id=309656.309671.

[5] Steven Schäfer, Tobias Tebbi, and Gert Smolka. “Autosubst: Reasoning with de bruijn terms and parallel substitutions”. In: *ITP*. Vol. 9236. LNCS. 2015, pp. 359–374.

[6] Amin Timany, Robbert Krebbers, and Lars Birkedal. “Logical Relations in Iris”. In: *CoqPL 2017: The Third International Workshop on Coq for Programming Languages*. CoqPL. 2017.