Later Credits: Supplementary Material

SIMON SPIES, MPI-SWS, Germany
LENNARD GÄHER, MPI-SWS, Germany
JOSEPH TASSAROTTI, New York University, USA
RALF JUNG, MIT CSAIL, USA
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
LARS BIRKEDAL, Aarhus University, Denmark
DEREK DREYER, MPI-SWS, Germany

In this document, we expand on several technical aspects that are not fully spelled out in the paper (*e.g.*, the invariants, the ghost state, etc. for the examples). In §1, we give an overview of the Iris rules used throughout the paper, and augment them with masks and add additional rules where appropriate. In §2, we provide additional material for the counter with a backup example. In §3, we provide additional material for the reordering refinements example. In §4, we explain how we define prepaid invariants. In §5, we expand on the *reverse refinements* that we have proven (see Section 6 in the paper). In §6, we provide additional detail of the integration of later credits into the model of Iris.

Contents

Authors' addresses: Simon Spies, MPI-SWS, Germany, spies@mpi-sws.org; Lennard Gäher, MPI-SWS, Germany, gaeher@mpi-sws.org; Joseph Tassarotti, New York University, USA, jt4767@nyu.edu; Ralf Jung, MIT CSAIL, USA; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk; Derek Dreyer, MPI-SWS, Germany, dreyer@mpi-sws.org.

The separating conjunction and magic wand

SepWeaken $P * Q \vdash P$

SepTrue $P \vdash P * \mathsf{True}$

SepComm $P * Q \vdash Q * P$

SepAssoc $P * (Q * R) \dashv\vdash (P * Q) * R$

SepSplit
$$\frac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

PointstoSep $\ell \mapsto v * \ell \mapsto w \vdash \mathsf{False}$

WandIntro
$$\frac{P * Q \vdash R}{P \vdash Q \twoheadrightarrow R}$$

WandElim
$$\frac{P \vdash Q \twoheadrightarrow R}{P * Q \vdash R}$$

The later modality

LaterIntro $P \vdash \triangleright P$

LaterMono
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

Löb $(\triangleright P \Rightarrow P) \vdash P$

LaterSep $\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$

LaterExists
$$\frac{X \text{ non-empty}}{\triangleright(\exists x : X.\, P(x)) \dashv\vdash \exists x : X.\, \triangleright P(x)}$$

LaterAll $\triangleright(\forall x : X.\, P(x)) \dashv\vdash \forall x : X.\, \triangleright P(x)$

LaterPers $\triangleright \square P \dashv\vdash \square \triangleright P$

The persistence modality

PersDup $\square P \vdash (\square P) * (\square P)$

PersElim $\square P \vdash P$

PersMono
$$\frac{P \vdash Q}{\square P \vdash \square Q}$$

PersAndSep $(\square P) \wedge Q \vdash (\square P) * Q$

PersIdemp $\square P \vdash \square \square P$

PersAll $\forall x : X.\, \square P(x) \vdash \square \forall x : X.\, P(x)$

PersExists $\square \exists x : X.\, P(x) \vdash \exists x : X.\, \square P(x)$

Fig. 1. Basic Iris entailment rules

1 IRIS OVERVIEW

In the paper, we only show a selection of the proof rules that we use in Iris (with and without later credits), and we simplify some of them by omitting masks. In this section, we restate the rules with all masks, and complement them with additional rules omitted in the paper.

Basic entailment rules. In Figure 1, we list some standard rules for Iris's entailment relation $P \vdash Q$. We write $P \dashv\vdash Q$ for an entailment, which holds in both directions. These rules are used to manipulate the separating conjunction $P * Q$ and the associated notion of implication, the magic wand $P \twoheadrightarrow Q$. Moreover, these rules are used to work with the later modality $\triangleright P$ and the persistence modality $\square P$.

The persistence modality $\square P$ turns the notion of *duplicability* into a first-class principle in Iris. That is, $\square P$ can be duplicated, regardless of the proposition P. As a consequence, to prove $\square P$, one may only use other persistent resources (*e.g.,* other propositions $\square Q$, invariants, etc.), but no non-persistent resources (*e.g.,* $\ell \mapsto v$). The persistence modality "$\square$" is used, for example, in the definition of Hoare triples $\{P\}\ e\ \{v.\, Q\}_{\mathcal{E}}$ (see §6.2) to ensure that Hoare triples are always *duplicable.* For example, if we have proven $\forall n.\ \{P\}\ f(n)\ \{v.\, Q\}$, then persistence ensures we can use the Hoare triple for multiple calls of f (*e.g.,* in verifying $f(n) + f(m)$).

Structural rules

Value

$$\{\mathsf{True}\}\ v\ \{w.\ w = v\}$$

Frame

$$\frac{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}{\{P * R\}\ e\ \{v.\ Q * R\}_{\mathcal{E}}}$$

Consequence

$$\frac{P \vdash P' \qquad \{P'\}\ e\ \{v.\ Q'\}_{\mathcal{E}} \qquad \forall v.\ (Q' \vdash Q)}{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

Bind

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

Execution rules

PureStep

$$\frac{\{P\}\ e_2\ \{v.\ Q\}_{\mathcal{E}} \qquad e_1 \rightarrow_{\text{pure}} e_2}{\{\triangleright P\}\ e_1\ \{v.\ Q\}_{\mathcal{E}}}$$

Fork

$$\frac{\{P\}\ e\ \{\_.\ \mathsf{True}\}_{\top}}{\{\triangleright P * \triangleright Q\}\ \mathtt{fork}\ \{e\}\ \{v.\ v = () * Q\}_{\mathcal{E}}}$$

Ref

$$\{P\}\ ??v)\ \{w.\ \exists \ell.\ w = \ell * \ell \mapsto v * \triangleright P\}_{\mathcal{E}}$$

Store

$$\{\ell \mapsto v * \triangleright P\}\ \ell \leftarrow w\ \{u.\ u = () * \ell \mapsto w * P\}_{\mathcal{E}}$$

Load

$$\{\ell \mapsto v * \triangleright P\}\ !\ell\ \{w.\ w = v * \ell \mapsto v * P\}_{\mathcal{E}}$$

FAA

$$\{\ell \mapsto n * \triangleright P\}\ \mathtt{FAA}(\ell, m)\ \{v.\ v = n * \ell \mapsto (n+m) * P\}_{\mathcal{E}}$$

CAS-Succ

$$\frac{v \text{ comparable}}{\{\ell \mapsto v * \triangleright P\}\ \mathtt{CAS}(\ell, v, w)\ \{u.\ u = \mathtt{true} * \ell \mapsto w * P\}_{\mathcal{E}}}$$

CAS-Fail

$$\frac{v \neq v' \qquad (v \text{ comparable}) \lor (v' \text{ comparable})}{\{\ell \mapsto v' * \triangleright P\}\ \mathtt{CAS}(\ell, v, w)\ \{u.\ u = \mathtt{false} * \ell \mapsto v' * P\}_{\mathcal{E}}}$$

Fig. 2. Hoare triple rules

Hoare triple rules. In Figure 2, we list the rules for reasoning about Iris's Hoare triples without later credits (except for rules concerned with invariants, updates, and timelessness, which are discussed below). We write $\frac{P\ \ Q}{R}$ for Iris entailments $P * Q \vdash R$.

Compared to rules presented in the paper, we have annotated all Hoare triples with a mask and added the rules for values (*i.e.,* Value), the rule of consequence (*i.e.,* Consequence), and the bind rule (*i.e.,* Bind). (The Bind rule allows us the focus on an expression e in some larger evaluation context K.) Moreover, we have added the primitive stepping rules for state manipulating operations (*i.e.,* Ref, Store, Load, FAA, CAS-Succ, and CAS-Fail) and for **fork** $\{e\}$ (*i.e.,* Fork).

Basic rules

$$\textsc{UpdReturn}\quad P \vdash {\Rrightarrow}^{\mathcal{E}} P$$

$$\textsc{UpdBind}\quad ({}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P) * (P \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3} Q) \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3} Q$$

$$\textsc{UpdExec}\quad \frac{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}{\{{\Rrightarrow}^{\mathcal{E}} P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

$$\textsc{InvPers}\quad \boxed{R}^{\mathcal{N}} \vdash \Box \boxed{R}^{\mathcal{N}}$$

$$\textsc{UpdInvAlloc}\quad (\triangleright P) \vdash {\Rrightarrow}^{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

$$\textsc{InvAcc}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{R}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} \Big(\triangleright R * (\triangleright R \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}} \mathsf{True})\Big)}$$

$$\textsc{UpdMaskWeaken}\quad \frac{\mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad P \vdash {\Rrightarrow}^{\mathcal{E}_1} P}{P \vdash {\Rrightarrow}^{\mathcal{E}_2} P}$$

$$\textsc{MaskUpd}\quad \frac{P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}'} P' \qquad \{P'\}\ e\ \{v.\ Q'\}_{\mathcal{E}'} \qquad Q' \vdash {}^{\mathcal{E}'}{\Rrightarrow}^{\mathcal{E}} Q \qquad e \text{ physically atomic}}{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

Derived rules

$$\textsc{InvAlloc}\quad \frac{\boxed{R}^{\mathcal{N}} \vdash \{P\}\ e\ \{w.\ Q\}_{\mathcal{E}}}{\{P * \triangleright R\}\ e\ \{w.\ Q\}_{\mathcal{E}}}$$

$$\textsc{InvOpenUpd}\quad \frac{P * \triangleright R \vdash {\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} Q * \triangleright R \qquad \mathcal{N} \subseteq \mathcal{E}}{P * \boxed{R}^{\mathcal{N}} \vdash {\Rrightarrow}^{\mathcal{E}} Q}$$

$$\textsc{InvOpen}\quad \frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\boxed{R}^{\mathcal{N}} \vdash \{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

Fig. 3. Update and invariant rules

Update and invariant rules. In Figure 3, we list the rules for reasoning about Iris's update modality and invariants. In general, the update modality carries two masks $\mathcal{E}_1$ and $\mathcal{E}_2$, which indicate that when we prove $P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} Q$, then we get to open the invariants in $\mathcal{E}_1$, we can update the ghost state, and afterwards we have to close the invariants in $\mathcal{E}_2$. We write ${\Rrightarrow}^{\mathcal{E}} P \triangleq {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$ if both masks are the same. (In other Iris presentations, the mask for a non-mask-changing update is at the bottom (*i.e.*, "${\Rrightarrow}_{\mathcal{E}}$"). Since this position conflicts with the "le" of our later elimination updates, we move it up.)

From the basic rules for manipulating the update modality, we have discussed UpdReturn, UpdBind, and UpdExec in the paper. (In the paper, we have omitted the mask in some of them if they are of no particular interest.) There are also some basic rules that we have not discussed in the paper: InvPers, UpdInvAlloc, InvAcc, and MaskUpd. The rule InvPers makes sure that invariants are persistent, so we can duplicate them freely in our proofs. The other basic rules can be used to derive the rules presented in the paper (*i.e.*, InvAlloc, InvOpenUpd, and InvOpen). We include the more basic rules here to showcase the use of two distinct masks, before we return to updates with two masks in the definition of the weakest precondition (in §6.2).

$$\textsc{HoareTimeless}\quad \frac{\{P * Q\}\; e\; \{v.\, R\}_{\mathcal{E}} \qquad \mathsf{timeless}(Q)}{\{P * \triangleright Q\}\; e\; \{v.\, R\}_{\mathcal{E}}} \qquad \textsc{UpdTimeless}\quad \frac{\mathsf{timeless}(P)}{\triangleright P \vdash {\Rrightarrow}^{\mathcal{E}} P} \qquad \textsc{PointsToTimeless}\quad \mathsf{timeless}(\ell \mapsto v)$$

Fig. 4. Timelessness

Timelessness. In Figure 4, we list the rules for reasoning about timelessness. Recall from the paper that in Iris, there is a class of propositions, the so-called timeless propositions, which are largely independent of step-indexing *e.g.,* $\ell \mapsto v$ (see PointsToTimeless). What this means is that we can eliminate a later from them when we prove a Hoare triple (see HoareTimeless) and even when we prove an update (see UpdTimeless).

Later credits rules. In Figure 5, we list the rules for manipulating later credits and the later elimination update. We have discussed the *ghost theory* in the paper. CreditSplit allows us to combine and split credits. CreditTimeless allows us to put credits into invariants without having to deal with later troubles (see Figure 4). The rules SupplyBound, SupplyDecr, and SupplyExcl are used to prove soundness (and only there), since we define "$\Rrightarrow_{\mathsf{le}}$" in terms of $£_{\bullet}\, n$ (see §6.2).

The *update rules* for the later elimination update ${}^{\mathcal{E}_1}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}_2} P$ are essentially[1] the same as those for the standard update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$, with the addition of the later elimination rule LEUpdLater. Fittingly, we replace the standard update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ with ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ in the model of Iris and, as a consequence, we can eliminate later modalities (if we have credits) wherever we could previously open invariants and manipulate ghost state (see §6).

The *later credit stepping rules* generate an additional credit in the postcondition, or in the case of PureStep in the precondition for the next step. Note that together with LEUpdLater, these rules can be used to derive the Hoare triple rules in Figure 2.

Time receipt extension. In Figure 6, we list the rules for the time-receipt extension. The rules ReceiptTimeless, ReceiptCredits, and PureStep are discussed in the paper. The remaining rules generalize the rules from Figure 5 as one would expect.

[1]There is one kind of rules that does not hold for the later elimination update: the interaction rules with Iris's plain modality $\blacksquare\, P$. For more information, see §6.1.1.

Ghost theory

CreditSplit
$$£(n+m) \Leftrightarrow £n * £m$$

CreditTimeless
$$\mathsf{timeless}(£n)$$

SupplyBound
$$£_{\bullet}\, m * £n \vdash m \geq n$$

SupplyDecr
$$£_{\bullet}\,(n+m) * £n \vdash \Rrightarrow £_{\bullet}\, m$$

SupplyExcl
$$£_{\bullet}\, m_1 * £_{\bullet}\, m_2 \vdash \mathsf{False}$$

Update rules

LEUpdReturn
$$P \vdash \Rrightarrow_{\mathsf{le}}^{\mathcal{E}} P$$

LEUpdBind
$$({}^{\mathcal{E}_1}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}_2} P) * (P \twoheadrightarrow {}^{\mathcal{E}_2}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}_3} Q) \vdash {}^{\mathcal{E}_1}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}_3} Q$$

LEUpdExec
$$\frac{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}{\left\{\Rrightarrow_{\mathsf{le}}^{\mathcal{E}} P\right\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

LEUpdLater
$$£1 * \triangleright P \vdash \Rrightarrow_{\mathsf{le}}^{\mathcal{E}} P$$

LEUpdInvAlloc
$$P \vdash \Rrightarrow_{\mathsf{le}}^{\mathcal{E}} \boxed{P}^{\mathcal{N}}$$

LEUpdMaskWeaken
$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad P \vdash \Rrightarrow_{\mathsf{le}}^{\mathcal{E}_1} P}{P \vdash \Rrightarrow_{\mathsf{le}}^{\mathcal{E}_2} P}$$

LEInvAcc
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{R}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}\setminus\mathcal{N}} \Big(\triangleright R * \big(\triangleright R \twoheadrightarrow {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}} \mathsf{True}\big)\Big)}$$

LEMaskUpd
$$\frac{P \vdash {}^{\mathcal{E}}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}'} P' \qquad \{P'\}\ e\ \{v.\ Q'\}_{\mathcal{E}'} \qquad Q' \vdash {}^{\mathcal{E}'}{\Rrightarrow}_{\mathsf{le}}^{\mathcal{E}} Q \qquad e \text{ physically atomic}}{\{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

Later credit step rules

PureStep
$$\frac{\{P * £1\}\ e_2\ \{v.\ Q\}_{\mathcal{E}} \qquad e_1 \to_{\text{pure}} e_2}{\{P\}\ e_1\ \{v.\ Q\}_{\mathcal{E}}}$$

Fork
$$\frac{\{\triangleright P\}\ e\ \{\_.\ \mathsf{True}\}_{\top}}{\{P\}\ \mathtt{fork}\ \{e\}\ \{v.\ v = () * £1\}_{\mathcal{E}}}$$

Ref
$$\{\mathsf{True}\}\ ??v)\ \{w.\ \exists \ell.\ w = \ell * \ell \mapsto v * £1\}_{\mathcal{E}}$$

Store
$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{u.\ u = () * \ell \mapsto w * £1\}_{\mathcal{E}}$$

Load
$$\{\ell \mapsto v\}\ !\ell\ \{w.\ w = v * \ell \mapsto v * £1\}_{\mathcal{E}}$$

FAA
$$\{\ell \mapsto n\}\ \mathtt{FAA}(\ell, m)\ \{v.\ v = n * \ell \mapsto (n+m) * £1\}_{\mathcal{E}}$$

CAS-Succ
$$\frac{v \text{ comparable}}{\{\ell \mapsto v\}\ \mathtt{CAS}(\ell, v, w)\ \{u.\ u = \mathtt{true} * \ell \mapsto w * £1\}_{\mathcal{E}}}$$

CAS-Fail
$$\frac{v \neq v' \qquad (v \text{ comparable}) \vee (v' \text{ comparable})}{\{\ell \mapsto v'\}\ \mathtt{CAS}(\ell, v, w)\ \{u.\ u = \mathtt{false} * \ell \mapsto v' * £1\}_{\mathcal{E}}}$$

Fig. 5. Later credits rules

General rules

ReceiptTimeless

$$\text{timeless}(⧗n)$$

ReceiptCredits

$$\frac{\{P\}\ e\ \{v.\ Q\} \qquad e \notin \mathit{Val}}{\{P * ⧗n\}\ e\ \{v.\ Q * £n * ⧗n\}}$$

Execution rules

PureStep

$$\frac{\{P * £1 * ⧗1\}\ e_2\ \{v.\ Q\}_{\mathcal{E}} \qquad e_1 \rightarrow_{\text{pure}} e_2}{\{P\}\ e_1\ \{v.\ Q\}_{\mathcal{E}}}$$

Fork

$$\frac{\{\triangleright P\}\ e\ \{\_.\ \text{True}\}_{\top}}{\{\triangleright P\}\ \mathtt{fork}\ \{e\}\ \{v.\ v = () * £1 * ⧗1\}_{\mathcal{E}}}$$

Ref

$$\{\text{True}\}\ ??v)\ \{w.\ \exists \ell.\ w = \ell * \ell \mapsto v * £1 * ⧗1\}_{\mathcal{E}}$$

Store

$$\{\ell \mapsto v\}\ \ell \leftarrow w\ \{u.\ u = () * \ell \mapsto w * £1 * ⧗1\}_{\mathcal{E}}$$

Load

$$\{\ell \mapsto v\}\ !\ell\ \{w.\ w = v * \ell \mapsto v * £1 * ⧗1\}_{\mathcal{E}}$$

FAA

$$\{\ell \mapsto n\}\ \mathtt{FAA}(\ell, m)\ \{v.\ v = n * \ell \mapsto (n+m) * £1 * ⧗1\}_{\mathcal{E}}$$

CAS-Succ

$$\frac{v \text{ comparable}}{\{\ell \mapsto v\}\ \mathtt{CAS}(\ell, v, w)\ \{u.\ u = \mathtt{true} * \ell \mapsto w * £1 * ⧗1\}_{\mathcal{E}}}$$

CAS-Fail

$$\frac{v \neq v' \qquad (v \text{ comparable}) \vee (v' \text{ comparable})}{\{\ell \mapsto v'\}\ \mathtt{CAS}(\ell, v, w)\ \{u.\ u = \mathtt{false} * \ell \mapsto v' * £1 * ⧗1\}_{\mathcal{E}}}$$

Fig. 6. Time receipt extension

Implementation

$$\begin{aligned}
\texttt{new}() &\triangleq \textbf{let}\ (b,p) := (\textbf{ref}(0),\textbf{ref}(0));\ \textbf{fork}\ \{\texttt{bg\_thread}(b,p)\}\ ;(b,p)\\
\texttt{incr}(b,p) &\triangleq \textbf{let}\ n = \textbf{FAA}(p,1);\ \texttt{await\_backup}(b,n+1);\ n\\
\texttt{get}(b,p) &\triangleq \textbf{let}\ n = !\,p;\ \texttt{await\_backup}(b,n);\ n\\
\texttt{get\_backup}(b,p) &\triangleq\ !\,b
\end{aligned}$$

Helper Functions

$$\begin{aligned}
\texttt{bg\_thread}(b,p) &\triangleq \textbf{let}\ n = !\,p;\ b \leftarrow n;\ \texttt{bg\_thread}(b,p)\ \text{// copy primary to backup, in a loop}\\
\texttt{await\_backup}(b,n) &\triangleq \textbf{if}\ !\,b < n\ \textbf{then}\ \texttt{await\_backup}(b,n)\ \textbf{else}\ ()\ \text{// loop until } !\,b \text{ reaches } n
\end{aligned}$$

Specification

$$\vdash \{\mathsf{True}\}\ \texttt{new}()\ \{c.\ \exists \gamma.\ \mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) * \mathsf{value}_{\gamma}(0)\}$$

$$\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) \vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{incr}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n+1)\rangle_{\mathcal{N}}$$

$$\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) \vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{get}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n)\rangle_{\mathcal{N}}$$

$$\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) \vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{get\_backup}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n)\rangle_{\mathcal{N}}$$

Fig. 7. Counter with a backup

2 COUNTER WITH A BACKUP

In this section, we expand on the details of the counter with a backup (shown in Figure 7). We first define the counter predicates $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$ and $\mathsf{value}_{\gamma}(n)$. We will chose these predicates such that $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$ is persistent (*i.e.,* $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) \vdash \Box\, \mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$), such that it can be shared between threads, and $\mathsf{value}_{\gamma}(n)$ is exclusive (*i.e.,* $\mathsf{value}_{\gamma}(n) * \mathsf{value}_{\gamma}(m) \vdash \mathsf{False}$), such that owning it gives full control over the counter. After defining the predicates, we discuss their definition and how they factor into the proof of the specifications (in Figure 7).

The ghost state. To define the counter predicates, we use *ghost state.* To be precise, we will use the following kinds of ghost state:

(1) *Monotonically growing natural numbers* from the resource algebra $\mathit{Auth}(\mathbb{N}, \max)$ with a fractional authoritative element. We write $\mathsf{mono}^{\gamma}_{q}(n) \triangleq \boxed{\bullet_q\, n}^{\gamma} * \boxed{\circ\, n}^{\gamma}$ for the authoritative element and $\mathsf{lb}^{\gamma}(n) \triangleq \boxed{\circ\, n}^{\gamma}$ for the persistent lower-bound fragments. Note that $\mathsf{mono}^{\gamma}_{q}(n) * \mathsf{mono}^{\gamma}_{q'}(n') \vdash q + q' \leq 1 \land n = n'$.
(2) *Exclusive tokens* from the resource algebra $\mathit{Ex}(1)$. We write $\mathsf{tok}_{\gamma} \triangleq \boxed{\mathsf{ex}()}^{\gamma}$ for the exclusive token.
(3) *Iris's ghost maps.* We write $\mathsf{GhostMapAuth}^{\gamma}(M)$ for the authoritative element (storing the map $M : X \xrightarrow{\mathsf{fin}} Y$) and $x \overset{\gamma}{\hookrightarrow} y$ for the fragments. We write $x \overset{\gamma}{\hookrightarrow}_{\Box} y$ if the fragment is persistent and thus the value of x is unchangeable.
(4) *Iris's ghost variables.* We write $\gamma \overset{q}{\mapsto} x$ for a ghost variable γ storing x with the fractional ownership q.

The counter predicates. In the paper and in Figure 7, we use an abstract name "γ" to indicate that $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$ and $\mathsf{value}_{\gamma}(n)$ depend on ghost state. Formally, the "γ" will consist of two ghost names: γ_{back} and γ_{ex}. Besides those, we will additionally allocate several auxiliary ghost names used

$$
\begin{aligned}
&I_{\mathrm{cnt}}(b, p, \gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, \gamma_{\mathrm{prim}}, \gamma_{\mathrm{get}}, \gamma_{\mathrm{put}}) \triangleq \\
&\qquad \exists (G : \mathbb{N} \xrightarrow{\mathrm{fin}} \mathit{Gname}), (P : \mathbb{N} \xrightarrow{\mathrm{fin}} \mathit{Gname} \times \mathit{Gname}), (n_b n_p : \mathbb{N}).\ b \mapsto n_b * p \mapsto n_p * n_p \geq n_b \\
&\qquad * \mathsf{mono}^{\gamma_{\mathrm{back}}}_{1/4}(n_b) * \mathsf{mono}^{\gamma_{\mathrm{prim}}}_{1}(n_p) * \mathsf{GhostMapAuth}^{\gamma_{\mathrm{get}}}(G) * \mathsf{GhostMapAuth}^{\gamma_{\mathrm{put}}}(P) \\
&\qquad * \mathsf{gets}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, G, n_b) * \mathsf{incrs}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, P, n_b, n_p) * \mathop{\Huge *}_{n \mapsto \gamma \in G} n \overset{\gamma_{\mathrm{get}}}{\hookrightarrow}_{\square} \gamma
\end{aligned}
$$

where

$$
\begin{aligned}
\mathsf{gets}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, G, n_b) \triangleq {} & \mathop{\Huge *}_{k \mapsto \gamma \in G} \exists O : \mathit{Gname} \times \mathit{Gname} \xrightarrow{\mathrm{fin}} \mathsf{unit}.\ \mathsf{GhostMapAuth}^{\gamma}(O) \\
& * \mathop{\Huge *}_{(\gamma_1, \gamma_2) \mapsto () \in O} \exists R.\ \boxed{I_{\mathrm{get}}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, \gamma_1, \gamma_2, k, R)}^{\mathcal{N}.\mathrm{get}} * \gamma_1 \overset{1/2}{\mapsto} (n_b < k) \\
\mathsf{incrs}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, P, n_b, n_p) \triangleq {} & \mathsf{dom}(P) = \{0, \ldots, n_p - 1\} \\
& * \mathop{\Huge *}_{k \mapsto (\gamma_1, \gamma_2) \in P} \exists R.\ \boxed{I_{\mathrm{incr}}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, \gamma_1, \gamma_2, k, R)}^{\mathcal{N}.\mathrm{incr}} * \gamma_1 \overset{1/2}{\mapsto} (n_b \leq k) \\
I_{\mathrm{get}}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, \gamma_1, \gamma_2, k, R) \triangleq {} & (\mathbf{AU}(n.\ \mathsf{value}_{\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}}(n), m.\ m = n * \mathsf{value}_{\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}}(n))^{\mathcal{N}}_{R} * \gamma_1 \overset{1/2}{\mapsto} \mathtt{true} * £1) \\
& \vee (R[k/n] * \gamma_1 \overset{1/2}{\mapsto} \mathtt{false}) \\
& \vee (\mathsf{tok}_{\gamma_2} * \gamma_1 \overset{1/2}{\mapsto} \mathtt{false}) \\
I_{\mathrm{incr}}(\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, \gamma_1, \gamma_2, k, R) \triangleq {} & (\mathbf{AU}(n.\ \mathsf{value}_{\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}}(n), m.\ m = n * \mathsf{value}_{\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}}(n+1))^{\mathcal{N}}_{R} * \gamma_1 \overset{1/2}{\mapsto} \mathtt{true} * £1) \\
& \vee (R[k/n] * \gamma_1 \overset{1/2}{\mapsto} \mathtt{false}) \\
& \vee (\mathsf{tok}_{\gamma_2} * \gamma_1 \overset{1/2}{\mapsto} \mathtt{false})
\end{aligned}
$$

Fig. 8. The counter invariant

in the definition of the two predicates. We define

$$\mathsf{value}_{\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}}(n) \triangleq \mathsf{mono}^{\gamma_{\mathrm{back}}}_{1/4}(n) * \mathsf{tok}_{\gamma_{\mathrm{ex}}}$$

and

$$\mathsf{is\_counter}^{\mathcal{N}}_{\gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}}(c) \triangleq \exists b, p : \mathit{Loc}.\ \exists \gamma_{\mathrm{prim}}, \gamma_{\mathrm{get}}, \gamma_{\mathrm{put}}.\ c = (b, p) * \boxed{I_{\mathrm{cnt}}(b, p, \gamma_{\mathrm{back}}, \gamma_{\mathrm{ex}}, \gamma_{\mathrm{prim}}, \gamma_{\mathrm{get}}, \gamma_{\mathrm{put}})}^{\mathcal{N}}$$

where the invariant I_{cnt} is given in Figure 8.

The counter invariant and the distribution of ghost state. The invariant I_{cnt} is used to coordinate the counter state. It contains ownership of the physical state, *i.e.,* the locations b and p, and various ghost state constructs. We share the invariant between the forked-off background thread (*i.e.,* `bg_thread`) and the clients of the counter, which will use the counter through the counter operations `incr`, `get`, and `get_backup`.

For the physical state, the invariant contains the ownership of the primary b and the backup p. They store the value of the backup n_b and the value of the primary n_p, where the primary value is always greater than the backup value. The values n_b and n_p are existentially quantified in the invariant, so they can change over time. To ensure that they only increase in value (and never decrease), the invariant includes $\mathsf{mono}^{\gamma_{\text{back}}}_{1/4}(n_b)$ and $\mathsf{mono}^{\gamma_{\text{prim}}}_{1}(n_p)$. The latter expresses logical ownership of the primary, whereas the former expresses logical ownership of the backup—a quarter of the ownership. That is, quarter of the ownership of $\mathsf{mono}^{\gamma_{\text{back}}}_{\cdot}(n)$ resides in the shared invariant to make sure that the counter value $\mathsf{value}_{\gamma_{\text{back}},\gamma_{\text{ex}}}(n)$ matches the value of the backup stored in the invariant. The predicate $\mathsf{value}_{\gamma_{\text{back}},\gamma_{\text{ex}}}(n)$ contains the ownership of another quarter of $\mathsf{mono}^{\gamma_{\text{back}}}_{\cdot}(n_b)$. The remaining half of $\mathsf{mono}^{\gamma_{\text{back}}}_{\cdot}(n_b)$ is given to the background thread. To update the value of the backup b, one needs full ownership of $\mathsf{mono}^{\gamma_{\text{back}}}_{\cdot}(n)$, since only then $\mathsf{mono}^{\gamma_{\text{back}}}_{\cdot}(n)$ can be increased. Through this ownership split, we ensure that one can only increase the counter if *all parts* are present—the half of the background thread, the quarter in the invariant, and the quarter in $\mathsf{value}_{\gamma_{\text{back}},\gamma_{\text{ex}}}(n)$.

The remaining parts of the invariant enable the helping exchange. Let us start with the machinery in place for helping `incr`. The invariant I_{cnt} contains a map P that is used to track all the completed and pending increment operations (using a ghost map $\mathsf{GhostMapAuth}^{\gamma_{\text{put}}}(P)$). To understand how exactly this tracking works, we take a closer look at $\mathsf{incrs}(\gamma_{\text{back}}, \gamma_{\text{ex}}, P, n_b, n_p)$. The map P maps the numbers $0, \ldots, n_p - 1$ to pairs of ghost names γ_1 and γ_2. Each mapping $k \mapsto (\gamma_1, \gamma_2) \in P$ corresponds to one `incr` operation that either has been helped already, or is currently awaiting helping. The k-th increment is pending if $n_b \leq k$, and has been helped otherwise. For each increment, we store two things: First, we store the invariant I_{inc}, which is the central part of the helping exchange for the k-th increment. Second, we store the information about the status of the increment operation in a ghost variable, γ_1, with half ownership (the other half is in the invariant I_{inc}). We will describe the helping exchange more detail shortly.

Let us now turn to the machinery for helping `get`. In general, the setup is very similar to the machinery for `incr`. The most important difference is that there can be *multiple* get-operations pending at a time for the value k, while there can only ever be one increment. In the invariant I_{cnt}, the map G is used to track the completed and pending get-operations (using a ghost map $\mathsf{GhostMapAuth}^{\gamma_{\text{get}}}(G)$). In contrast to P, G only stores a single ghost name γ, but that ghost name is unchangeable (since $\ast_{n \mapsto \gamma \in G}\, n \overset{\gamma_{\text{get}}}{\hookrightarrow}_{\square} \gamma$). To understand why, we take a closer look at $\mathsf{gets}(\gamma_{\text{back}}, \gamma_{\text{ex}}, G, n_b)$. For each number $k \in \operatorname{dom} G$, we store *a map* O in $\mathsf{gets}(\gamma_{\text{back}}, \gamma_{\text{ex}}, G, n_b)$ with $\mathsf{GhostMapAuth}^{\gamma}(O)$. This map will keep track of *multiple* get-operations for the value k. (We use a finite map O with a singleton codomain, which effectively makes O a set.) As for the increment operations, we store for each helpee an invariant and a ghost variable that tracks the status of the helpee (*i.e.,* either still pending or already completed). Another important difference to `incr` is that helpees are only pending if their value k is *strictly* greater than n_b. Thus, whenever the backup n_b is increased, we *have to* linearize all pending get-operations.

The helping exchange. Let us now turn to the helping exchange. We cover the helping exchange from *three* perspectives: an increment operation, a get operation, and the background thread.

We start with *an increment operation.* When the increment operation increases the primary p from n_p to $n_p + 1$, it puts its atomic update:

$$\mathbf{AU}_{\text{inc}} \triangleq \mathbf{AU}(n.\, \mathsf{value}_{\gamma}(n), m.\, m = n * \mathsf{value}_{\gamma}(n+1))^{\mathcal{N}}_{R}$$

into a new invariant, I_{inc} together with half the ownership of a freshly allocated ghost variable $\gamma_1 \overset{1/2}{\mapsto} \mathtt{false}$. It also allocates a new token tok_{γ_2}, which it keeps for itself. The invariant I_{inc} and the other half of $\gamma \overset{1/2}{\mapsto} \mathtt{false}$ go into the shared counter invariant I_{cnt}. To be precise, the increment operation extends the map P with $n_p \mapsto (\gamma_1, \gamma_2)$. Afterwards, the increment operation waits for the backup to catch up. Once the backup has caught up, the increment operation knows $n_b \geq n_p + 1$ where n_b is the new value of the backup and n_p the old value of the primary that the increment read in the previous step. Thus, the ghost variable γ_1 must be true, and the invariant I_{inc} must be in one of the last two states. In fact, the invariant must be in the second state, because the increment operation kept tok_{γ_2} for itself. Thus, the increment operation can use tok_{γ_2} to take out R, swapping it for tok_{γ_2}.

The helping exchange for *a get operation* is very similar to an increment. Instead of extending the map P, the get-operation extends the map O for the value of the primary n_p that it has read. If n_p is not yet in P, a fresh ghost name γ is allocated for n_p. The rest of the exchange: putting the atomic update into an invariant, adding the invariant to I_{inc}, and retrieving R are analogous to an increment.

Let us now turn to the *background thread.* The background thread first reads the value of the primary n_p. Then, in a second step, it updates the value of the backup b to n_p.[2] When the background thread is updating b, it is helping all the increments for the values $n_b + 1, \ldots, n_p$ (which would read $n_b, \ldots, n_p - 1$). Moreover, the background thread is also helping all the gets for values in this range. It proceeds as follows: First, it linearizes the increment for $n_b + 1$ by (1) taking out the atomic update from I_{inc}, (2) executing the update and incrementing $\mathsf{mono}^{\gamma_{\text{back}}}_{\bullet}(n_b)$ to $\mathsf{mono}^{\gamma_{\text{back}}}_{\bullet}(n_b + 1)$, and (3) returning the result R. Afterwards, it linearizes all the gets for $n_b + 1$ by (1) taking out the atomic update from I_{get}, (2) executing the update, and (3) returning the result R. After both the increment for $n_b + 1$ and all the gets for $n_b + 1$ are complete, the background thread proceeds with $n_b + 2, \ldots, n_p$.

Later credits. The use of *later credits* in this example is pretty straightforward: In each iteration, the background thread saves one credit $£1$ from some pure step (*e.g.,* the reduction of `bg_thread()`). It uses this credit to open the *outer* invariant I_{cnt}. Then for each pending helpee operation, it opens the corresponding invariant (*i.e.,* I_{get} or I_{inc}), uses the inner credit to eliminate the guarding later from the **AU**, and executes the update. Afterwards, it has the result R and can transition to the *second* state of the invariant, which no longer requires a later credit $£1$.

[2]Technically, the primary could have advanced beyond the value n_p that the background thread has read, but that does not matter for our purposes. Those waiting increments will be helped in the next iteration.

Implementation

$$
\begin{aligned}
\texttt{promise} &: 1 \to \texttt{pr}(\tau)\\
\texttt{promise}() &\triangleq (\texttt{mklock}(), \texttt{??none}), \texttt{??}[\,]))\\
\texttt{resolve} &: \texttt{pr}(\tau) \times \tau \to 1\\
\texttt{resolve}((l, r, c), a) &\triangleq \texttt{lock}(l); \textbf{case } !r \textbf{ of } \texttt{some}(b) \Rightarrow \texttt{unlock}(l); \textbf{abort}()\\
&\qquad\qquad\qquad\qquad \mid \texttt{none} \Rightarrow r \leftarrow \texttt{some}(a); \textbf{let } fs = !c;\ c \leftarrow [\,];\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{unlock}(l);\ \texttt{app}\ (\lambda f.\, f(a))\ fs\\
\texttt{then} &: \texttt{pr}(\tau) \times (\tau \to 1) \to 1\\
\texttt{then}((l, r, c), f) &\triangleq \texttt{lock}(l); \textbf{case } !r \textbf{ of } \texttt{some}(a) \Rightarrow \texttt{unlock}(l); f(a)\\
&\qquad\qquad\qquad\qquad \mid \texttt{none} \Rightarrow c \leftarrow f :: !c; \texttt{unlock}(l)
\end{aligned}
$$

Transition System

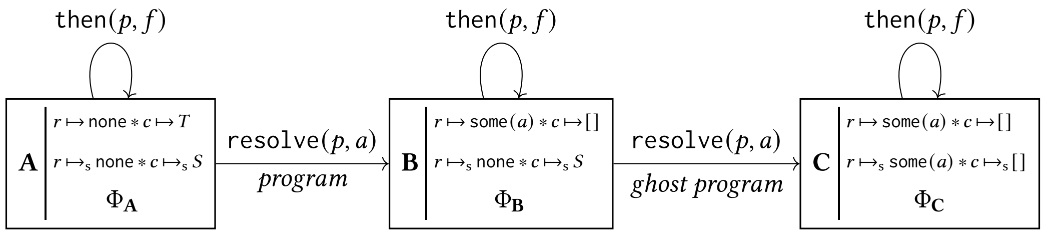

Fig. 9. Promise implementation and transition system

3 REORDERING REFINEMENTS

In this section, we expand on technical aspects of the promise reordering example that are sketched in the paper. Concretely, we elaborate on how we define the semantic interpretation of $\mathsf{pr}(\tau)$ and the transition system backing the interpretation. Before we start, recall the implementation of promises from the paper and the transition system that we set up for each promise (depicted in Figure 9).

Conceptually, each promise is represented by an instance of this transition system. In terms of Iris propositions, this means we will use ghost state in the interpretation of $\mathsf{pr}(\tau)$. Technically, we do not extend the syntax of ReLoC's types with $\mathsf{pr}(\tau)$. Instead, we define the *semantic type* $\mathsf{Pr}(A)$ for a semantic type A and prove our lemmas with respect to this semantic type.

We define $\mathsf{Pr}(A)$ using a lock and an invariant:

$$\mathsf{Pr}(A) \triangleq \{((l_t, r_t, c_t), (l_s, r_s, c_s)) | \exists \boldsymbol{\gamma}.\ \mathsf{Lock}(l_t, L_{r_t, c_t, \gamma_t}) * \boxed{I_{(l_t, r_t, c_t),(l_s, r_s, c_s), A, \boldsymbol{\gamma}}}^{\mathcal{N}}\}$$

If $\mathsf{pr}(\tau)$ were a syntactic type, we could then define its interpretation as

$$[\![\mathsf{pr}(\tau)]\!] \triangleq \mathsf{Pr}([\![\tau]\!]).$$

In our setting, we can pack up the promise implementation in an existential type

$$\exists \alpha.\ \{\mathsf{mkpromise} : 1 \to \alpha; \mathsf{resolve} : \alpha \times \tau \to_{\mathsf{re}} 1; \mathsf{then} : \alpha \times (\tau \to_{\mathsf{re}} 1) \to_{\mathsf{re}} 1\}$$

and assume that we are working in an environment where this existential type has been unpacked.[3]

[3]For the proof of this, we instantiate α precisely with $[\![\mathsf{pr}(\tau)]\!] = \mathsf{Pr}([\![\tau]\!])$.

In the rest of this section, we will explain how the lock proposition as well as the invariant are defined, and why they encode the promise transition system. For that purpose, let us fix A for the rest of the section.

To avoid any confusion with our auxiliary ghost state and clearly distinguish between the *program* and the *ghost program*, we refer to the former as the "target" and the latter as the "source".

The ghost state. We start with the ghost state that we will use both components. The invariant is parameterized over the ghost names $\boldsymbol{\gamma} = (\gamma_{\mathsf{st}}, \gamma_{\mathsf{s}}, \gamma_{\mathsf{t}}, \gamma_{\mathsf{hist}}, \gamma_{\mathsf{cb}})$ for the ghost state that we use:

- an exclusive token $\mathsf{tok}_t \triangleq \boxed{1}^{\gamma_{\mathsf{t}}}$ of the fractional algebra *Frac* that connects the target lock and the invariant,
- a fractional token $\mathsf{tok}_s(q) \triangleq \boxed{q}^{\gamma_{\mathsf{t}}}$ of the fractional algebra *Frac* that enables the source `resolve` to transition to state **C** of the transition system,
- the oneshot algebra $Ex(1) + Ag(List(Val \times Val) \times Val \times Val)$ that gets fired when the target resolves the promise (*i.e.,* when we transition from state **A** to state **B**). We define the elements $\mathsf{pend} \triangleq \boxed{\mathsf{inl}(\mathsf{ex}(()))}^{\gamma_{\mathsf{st}}}$ that gives the exclusive permission to fire the oneshot, as well as the persistent element $\mathsf{shot}(H, v_t, v_s) \triangleq \boxed{\mathsf{inr}(\mathsf{ag}(H, v_t, v_s))}^{\gamma_{\mathsf{st}}}$,
- an indexed list (modelled with a ghost map) tracking the callbacks in the order in which they arrived in the target, with the authoritative element $\bullet_{\mathsf{Hist}}(H)$, stating that the full history is $H : List(Val \times Val)$, and fragments $i \mapsto_{\mathsf{Hist}} (c_t, c_s)$, stating that the i-th callback is c_t in the target and c_s in the source,
- an indexed list (with indices synchronized to the history list) supplying exclusive tokens for the state of callbacks (so the elements are in 1), with authoritative element $\bullet_{\mathsf{Cb}}(B)$ (we do not care about the actual values in B) and exclusive tokens cb_i. Logically, the tokens cb_i will be provided to the `then` proof in the source to trade for a callback execution that was generated by the `then` proof in the target.

The invariant and the lock. With the ghost state in hand, we can turn to the definition of the invariant and the lock:

$$\begin{aligned} I_{(l_t,r_t,c_t),(l_s,r_s,c_s),A,\boldsymbol{\gamma}} \triangleq \exists H, B.\ & \boldsymbol{\gamma} = (\gamma_{\mathsf{st}}, \gamma_{\mathsf{s}}, \gamma_{\mathsf{t}}, \gamma_{\mathsf{hist}}, \gamma_{\mathsf{cb}}) \\ & * |B| = |H| * \bullet_{\mathsf{Hist}}(H) * \bullet_{\mathsf{Cb}}(B) \\ & * (I^{\mathbf{A}}_{H,\gamma_{\mathsf{st}},\gamma_{\mathsf{s}},\gamma_{\mathsf{t}},\gamma_{\mathsf{hist}},\gamma_{\mathsf{cb}}} \vee I^{\mathbf{B}}_{H,\gamma_{\mathsf{st}},\gamma_{\mathsf{s}},\gamma_{\mathsf{t}},\gamma_{\mathsf{hist}},\gamma_{\mathsf{cb}}} \vee I^{\mathbf{C}}_{H,\gamma_{\mathsf{st}},\gamma_{\mathsf{s}},\gamma_{\mathsf{t}},\gamma_{\mathsf{hist}},\gamma_{\mathsf{cb}}}) \\ L_{r_t,c_t,\gamma_{\mathsf{t}}} \triangleq\ & \mathsf{tok}_t * c_t \overset{1/2}{\mapsto} \_ * (r_t \overset{1/2}{\mapsto} \mathsf{none} \vee r_t \overset{1/2}{\mapsto} \mathsf{some}\_) \end{aligned}$$

Matching the transition system, the invariant has three states: $I^{\mathbf{A}}$, $I^{\mathbf{B}}$, and $I^{\mathbf{C}}$. We will expand on these states shortly. Besides the states, the invariant always keeps track of the history (with $\bullet_{\mathsf{Hist}}(H)$) and the callback list (with $\bullet_{\mathsf{Cb}}(B)$).

Besides the invariant I, we also use a lock in the definition of $\mathsf{Pr}(A)$. That is, each promise is protected by a *lock* in our implementation. For the target lock, we use Iris's standard locks to guard ownership of $L_{r_t,c_t,\gamma_{\mathsf{t}}}$. Here, c_t is the target location for the callback list and r_t is the target location for the resolved value. The lock owns one half of the target locations, as well as the exclusive target token. For the source lock, we proceed differently. The source lock is managed by the invariant I—the invariant ensures that the source is always locked, when the invariant is closed.

Let us now turn to the propositions $I^{\mathbf{A}}$, $I^{\mathbf{B}}$, and $I^{\mathbf{C}}$, which encode the states state **A**, state **B**, and state **C**. The proposition Φ will not appear directly in each of these propositions—it is the remainder one obtains after removing the ownership of the callback list locations and the promise reference.

Invariant state A. In this state, neither the target nor source resolve has happpened yet.

$$
\begin{aligned}
I^{\mathbf{A}}_{H,\gamma_{\mathrm{st}},\gamma_{\mathrm{s}},\gamma_{\mathrm{t}},\gamma_{\mathrm{hist}},\gamma_{\mathrm{cb}}} \triangleq\ & \exists c_t', c_s'.\ c_t \overset{1/2}{\mapsto} c_t' * c_s \mapsto_{\mathrm{s}} c_s' \\
& * \mathsf{LinkedList}(c_t', \mathsf{rev}(H.1)) \\
& * (r_t \overset{1/2}{\mapsto} \mathsf{none} \lor \mathsf{tok}_t) * r_s \mapsto_{\mathrm{s}} \mathsf{none} \\
& * l_s \mapsto_{\mathrm{s}} \mathsf{false} \text{ // the source lock is unlocked} \\
& * \mathsf{pend} * \mathsf{tok}_s(1) \\
& * \mathop{\Huge *}_{(v_t,v_s)\in H} (v_t, v_s) \in A \to_{\mathsf{re}} 1 \text{ // semantic typing for all callbacks} \\
& * \exists (m : \mathit{List}(\mathbb{B}))(C_s : \mathit{List}(\mathit{Val})).\ \text{ // subset } C_s \text{ of callbacks is also in the source} \\
& \qquad \mathsf{LinkedList}(c_s', \mathsf{rev}(C_s)) * \mathsf{indexed\_subset}(m, H.2, C_s) \\
& \qquad \text{// the token is here if the source then has been executed} \\
& \qquad * \mathop{\Huge *}_{i \mapsto \_ \in H} (m[i] = \mathsf{true} * \mathsf{tok}_i) \lor (m[i] = \mathsf{false})
\end{aligned}
$$

Here, $\mathsf{indexed\_subset}(m, \mathsf{full}, l)$ states that l contains exactly the elements (by index) of full for which the mask m (a list of Booleans) is true, in an arbitrary permutation. Concretely, it is defined as follows:

$$\mathsf{indexed\_subset}(m, \mathsf{full}, l) \triangleq |m| = |\mathsf{full}| * \mathsf{Permutation}(l, \mathsf{filter\_mask}(m, \mathsf{full}))$$

where

$$
\begin{aligned}
\mathsf{filter\_mask}([], \_) &\triangleq [] \\
\mathsf{filter\_mask}(\mathsf{true} :: m', h :: l') &\triangleq h :: \mathsf{filter\_mask}(m', l') \\
\mathsf{filter\_mask}(\mathsf{false} :: m', h :: l') &\triangleq \mathsf{filter\_mask}(m', l') \\
\mathsf{filter\_mask}(\_, []) &\triangleq []
\end{aligned}
$$

Invariant state B. In this state, the target resolve has been initiated (but the target may still be executing the list of callbacks), while the source resolve has not happened yet. We remember the original history H at the moment where the target resolve "happened" as H_1 (*i.e.,* the moment when the reference r_t was updated), while H_2 contains the callbacks that arrived concurrently or reentrantly (and have been directly executed in the target).

$$
\begin{aligned}
I^{\mathbf{B}}_{H,\gamma_{\mathrm{st}},\gamma_{\mathrm{s}},\gamma_{\mathrm{t}},\gamma_{\mathrm{hist}},\gamma_{\mathrm{cb}}} \triangleq\ & \exists H_1, H_2, v_t, v_s, c'_t, c'_s.\ H = H_1 \mathbin{+\!\!+} H_2 * c_t \overset{1/2}{\mapsto} c'_t * c_s \mapsto_{\mathrm{s}} c'_s \\
& * \mathsf{LinkedList}(c'_t, [\,]) \ \text{// the target list has been emptied} \\
& * r_t \overset{1/2}{\mapsto} \mathsf{some}(v_t) * r_s \mapsto_{\mathrm{s}} \mathsf{none} \\
& * l_s \mapsto_{\mathrm{s}} \mathsf{false} \ \text{// the source lock is unlocked} \\
& * \mathsf{shot}(H_1, v_t, v_s) * \mathsf{tok}_s(1/2) \ \text{// the source resolve proof has one half of the token} \\
& * (v_t, v_s) \in A \ \text{// semantic typing for resolved values} \\
& * \exists (m : \mathit{List}(\mathbb{B}))(C_s : \mathit{List}(\mathit{Val})).\ \text{// subset } C_s \text{ of callbacks is also in the source} \\
& \quad \mathsf{LinkedList}(c'_s, \mathsf{rev}(C_s)) * \mathsf{indexed\_subset}(m, H.2, C_s) \\
& \quad * \mathop{\Huge *}_{i \mapsto (c_t, c_s) \in H} \quad \begin{aligned}[t] & \text{// callback not in the source yet or the token is here} \\ & (m[i] = \mathsf{false} \lor \mathsf{tok}_i) \\ & \text{// we already have the source execution} \\ & \text{// \dots or it will be produced by the target resolve} \\ & * (c_s(v_s) \rightsquigarrow_{\mathrm{ghost}} () \lor i < |H_1|) \end{aligned}
\end{aligned}
$$

Invariant state state C. In this state, the resolve has also happened in the source.

$$
\begin{aligned}
I^{\mathbf{C}}_{H,\gamma_{\mathrm{st}},\gamma_{\mathrm{s}},\gamma_{\mathrm{t}},\gamma_{\mathrm{hist}},\gamma_{\mathrm{cb}}} \triangleq\ & \exists H_1, H_2, v_t, v_s, c'_t, c'_s.\ H = H_1 \mathbin{+\!\!+} H_2 * c_t \overset{1/2}{\mapsto} c'_t * c_s \mapsto_{\mathrm{s}} c'_s \\
& * \mathsf{LinkedList}(c'_t, [\,]) * \mathsf{LinkedList}(c'_s, [\,]) \ \text{// both lists have been emptied} \\
& * r_t \overset{1/2}{\mapsto} \mathsf{some}(v_t) * r_s \mapsto_{\mathrm{s}} \mathsf{some}(v_s) \\
& * l_s \mapsto_{\mathrm{s}} \mathsf{false} \ \text{// the source lock is unlocked} \\
& * \mathsf{shot}(H_1, v_t, v_s) * \mathsf{tok}_s(1) \\
& * (v_t, v_s) \in A \ \text{// semantic typing for resolved values} \\
& * \mathop{\Huge *}_{i \mapsto (c_t, c_s) \in H} \quad \begin{aligned}[t] & \mathsf{tok}_i \ \text{// the callback has already been executed} \\ & \lor c_s(v_s) \rightsquigarrow_{\mathrm{ghost}} () \ \text{// or we have deposited the source execution} \end{aligned}
\end{aligned}
$$

4 PREPAID INVARIANTS WITH LATER CREDITS

As mentioned in Section 6 of the paper, with later credits we can define a new kind of invariant, a *prepaid invariant* $\boxed{R}^{\mathcal{N}}_{\text{pre}}$, which can be opened around atomic expressions *without* a guarding later:

$$\textsc{InvPreOpen}\quad \frac{\{R * P\}\ e\ \{v.\ R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\boxed{R}^{\mathcal{N}}_{\text{pre}} \vdash \{P\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

The crux of prepaid invariants is that the later elimination has been prepaid—accounted for in amortized fashion by an earlier step of computation. To explain how that works and where later credits fit in, we first introduce a basic version $\boxed{R}^{\mathcal{N}}_{\text{basic}}$, which requires paying a later credit $£1$ to close it, and then show how to get the full version $\boxed{R}^{\mathcal{N}}_{\text{pre}}$ satisfying InvPreOpen.

Basic prepaying. In the examples in the paper, we have seen *prepaid reasoning*. That is, we have seen how to obtain a credit $£1$, keep it for several steps, and then spend it to eliminate the guarding later (*e.g.,* from the contents of an invariant). For our basic prepaid invariants, we use the same idea, but *bundle a credit with an invariant.* Specifically, we define $\boxed{R}^{\mathcal{N}}_{\text{basic}} \triangleq \boxed{R * £1}^{\mathcal{N}}$. As we explain below, we can now always use the credit in the invariant to remove the guarding later from R when we open the invariant. We obtain two proof rules for basic prepaid invariants:

$$\textsc{InvBasicAlloc}\quad \frac{\left\{P * \boxed{R}^{\mathcal{N}}_{\text{basic}}\right\}\ e\ \{v.\ Q\}}{\{P * £1 * \triangleright R\}\ e\ \{v.\ Q\}} \qquad \textsc{InvBasicOpen}\quad \frac{\{R * P\}\ e\ \{v.\ £1 * R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\left\{\boxed{R}^{\mathcal{N}}_{\text{basic}} * P\right\}\ e\ \{v.\ Q\}_{\mathcal{E}}}$$

To explain the use of later credits, we sketch the proof of InvBasicOpen. Since e is atomic, we can open the invariant behind $\boxed{R}^{\mathcal{N}}_{\text{basic}}$ with InvAcc. We have to prove $\{\triangleright(R * £1) * P\}\ e\ \{v.\ \triangleright(R * £1) * Q\}_{\mathcal{E}\setminus\mathcal{N}}$. The later credits $£n$ are timeless (see CreditTimeless), and hence we can use LaterSep and Hoare-Timeless to eliminate the guarding later from $£1$. Thus, we are left to prove

$$\{\triangleright R * £1 * P\}\ e\ \{v.\ \triangleright(R * £1) * Q\}_{\mathcal{E}\setminus\mathcal{N}}.$$

Next, we can use the later credit $£1$ to eliminate the guarding later in front of R with LEUpdLater and LEUpdExec. Thus, we are left with $\{R * P\}\ e\ \{v.\ \triangleright(R * £1) * Q\}_{\mathcal{E}\setminus\mathcal{N}}$. We apply LaterIntro in the postcondition, and are left with the goal $\{R * P\}\ e\ \{v.\ R * £1 * Q\}_{\mathcal{E}\setminus\mathcal{N}}$, which is our assumption.

The basic prepaid invariants $\boxed{R}^{\mathcal{N}}_{\text{basic}}$ are already quite useful: when we open these invariants, we do not have to eliminate any later modalities, since R is not guarded by a later modality. In exchange for this liberty, we have to put back one credit $£1$ after executing e—a credit which we typically obtain from the step of e (*e.g.,* with $\{\ell \mapsto v\}\ !\ell\ \{w.\ w = v * \ell \mapsto v * £1\}$). Unfortunately, there is a limitation. If we want to open *two* invariants $\boxed{P}^{\mathcal{N}_1}_{\text{basic}}$ and $\boxed{Q}^{\mathcal{N}_2}_{\text{basic}}$ while reasoning about the same single physical step of executing e, then we need to give back *two* credits, but the execution of e only generates *one* credit. So, next, we show how we extend the basic $\boxed{R}^{\mathcal{N}}_{\text{basic}}$ to the full $\boxed{R}^{\mathcal{N}}_{\text{pre}}$, which enables opening multiple and nested invariants.

Time receipt extension. The problem that prohibits opening multiple basic prepaid invariants is that we do not generate enough credits. Thus, to get prepaid invariants that satisfy InvPreOpen, we need to increase the number of credits that are generated on each program step. Toward this end, we use the time receipts extension discussed in the paper (rules shown in Figure 6). Recall that the idea of the time receipts extension is that with each program step, we obtain a time receipt ⧗1—a witness for the program step. (They were used originally to prove lower bounds on time complexity.)

Combined with later credits, this means that each execution step now produces both a later credit *and* a time credit (*e.g.,* see PureStep). In addition, we can use the time receipts to get later credits with ReceiptCredits: if we own a receipt for n steps, then we can leverage it to generate n fresh later credits after the execution of e.

With time receipts in hand, we define $\boxed{R}^{\mathcal{N}}_{\mathsf{pre}} \triangleq \boxed{R * £1 * ⧗1}^{\mathcal{N}}$ and obtain the rules:

InvPreAlloc

$$\frac{\left\{P * \boxed{R}^{\mathcal{N}}_{\mathsf{pre}}\right\}\ e\ \left\{v.\ Q\right\}}{\{P * £1 * ⧗1 * \triangleright R\}\ e\ \{v.\ Q\}}$$

InvPreOpen

$$\frac{\{R * P\}\ e\ \{v.\ R * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\left\{\boxed{R}^{\mathcal{N}}_{\mathsf{pre}} * P\right\}\ e\ \left\{v.\ Q\right\}_{\mathcal{E}}}$$

To understand how time receipts help us, we proceed with the proof of InvPreOpen. As in the proof of InvBasicOpen, we open the invariants behind $\boxed{R}^{\mathcal{N}}_{\mathsf{pre}}$ to obtain $\triangleright(R * £1 * ⧗1)$. We then eliminate the later from $£1$ and $⧗1$, leveraging the fact that $⧗1$ is, ironically, timeless (see ReceiptTimeless). Once again, we use the later credit $£1$ to eliminate the later from $\triangleright R$, obtaining R. Then, after applying LaterIntro in the postcondition, we are left with the goal $\{R * P * ⧗1\}\ e\ \{v.\ R * £1 * ⧗1 * Q\}_{\mathcal{E}\setminus\mathcal{N}}$. We use our receipt $⧗1$ to generate a new credit with ReceiptCredits, and the resulting goal matches the premise of InvPreOpen.

With the time receipt extension of later credits, we have unlocked the full power of prepaid invariants. When we create one with InvPreAlloc, we have to give up one credit $£1$ and one time receipt $⧗1$. Afterwards, we can always use InvPreOpen to open the invariants around Hoare triples without a later guarding the contents. Unlike InvBasicOpen, we can apply InvPreOpen multiple times if we want to open two (or more) invariants $\boxed{P}^{\mathcal{N}_1}_{\mathsf{pre}}$ and $\boxed{Q}^{\mathcal{N}_2}_{\mathsf{pre}}$. In particular, we can use InvPreOpen to open nested invariants (*e.g.,* $\boxed{\exists \ell.\ \boxed{\exists n : \mathbb{N}.\ \ell \mapsto n}^{\mathcal{N}_1}_{\mathsf{pre}} * \gamma \mapsto_{\mathsf{ghost}} \ell}^{\mathcal{N}_2}_{\mathsf{pre}}$ from the introduction of the paper) without any difficulty.

Limitations. Above, we have seen that prepaid invariants can be opened around Hoare triples without a guarding later (see InvPreOpen). Naturally, this begs the question: what about opening prepaid invariants around updates (*e.g.,* see InvOpenUpd). Sadly, even with prepaid invariants, the guarding laters remain in the rule. This is not by accident: Krebbers et al. [2017a] have shown that an invariant opening rule without laters for updates is not sound.

Since logically atomic updates are defined solely in terms of updates, we also do not inherit a later-free rule for opening invariants around atomic updates from prepaid invariants. Nevertheless, as explained in the paper, later credits still bring significant benefits to logical atomicity proofs.

```
checkCache c x ≜ case !c of none ⇒ none | some(y, v) ⇒ if x = y then some(v) else none
        memo f ≜ let c = ??none);
                 λx. case checkCache c x of some(v) ⇒ v
                                          | none ⇒ let v = f x; c ← some(x, v); v
```

Fig. 10. Implementation of a concurrent memoization function.

5 REVERSE REFINEMENTS

We show how later credits can be used to address a limitation with step-indexed logical relations described by Svendsen et al. [2016]. The issue arises when there is a function $f : \tau \to \tau$ and we want to show for all $e : \tau$ that $f(e)$ is contextually equivalent to e at type τ, written $f(e) \equiv_{\mathsf{ctx}} e : \tau$. One strategy to show such an equivalence is to split it into proving two contextual *refinements*: we show $f(e) \leq_{\mathsf{ctx}} e : \tau$ and $e \leq_{\mathsf{ctx}} f(e) : \tau$ where $\leq_{\mathsf{ctx}}$ is contextual refinement. To prove these contextual refinements, we show that the expressions *logically* refine each other, according to a step-indexed logical relation. That is, we show $f(e) \leq_{\mathsf{log}} e : \tau$ and $e \leq_{\mathsf{log}} f(e) : \tau$, where $\leq_{\mathsf{log}}$ is a step-indexed judgment which (by the logical relation's soundness theorem) implies $\leq_{\mathsf{ctx}}$.

Generally, when proving a logical refinement of the form $e_1 \leq_{\mathsf{log}} e_2 : \tau$, steps of e_1 allow elimination of laters. Thus, showing $f(e) \leq_{\mathsf{log}} e : \tau$ is usually relatively straight-forward, as far as step-indexing goes, since evaluating $f(e)$ takes steps, which provides opportunities to eliminate laters. On the other hand, showing $e \leq_{\mathsf{log}} f(e) : \tau$, which we call the *reverse refinement*, can be problematic. Since we want to show $e \leq_{\mathsf{log}} f(e) : \tau$ for all $e : \tau$, we cannot use steps taken by e to eliminate laters because e could be a value that does not take any steps at all.

To address this issue with later elimination in the reverse refinement, Svendsen et al. [2016] use a *transfinite* step-indexed logical relation. Transfinite step-indexed models come with some drawbacks (discussed in the main paper), so here we show that later credits are an alternative solution for proving reverse refinements that avoids the need for transfinite step indexing. As for reordering refinements, we demonstrate this by modifying ReLoC [Frumin et al. 2018, 2021], the framework for proving contextual equivalences using step-indexed logical relations encoded in Iris. Using this new version of ReLoC we have verified the reverse refinement example considered by Svendsen et al. [2016], as well as a more complicated concurrent memoization example. We focus on the memoization example here.

The key idea is simple: rather than trying to prove $\vdash e \leq_{\mathsf{log}} f(e) : \tau$, we instead prove $£n \ast ⧗n \vdash e \leq_{\mathsf{log}} f(e) : \tau$, where n is some number of additional later credits and time receipts that are needed to establish the refinement. The logical relation's soundness theorem is extended to say that $£n \ast ⧗n \vdash e_1 \leq_{\mathsf{log}} e_2 : \tau$ implies $e_1 \leq_{\mathsf{ctx}} e_2 : \tau$ for any n. Taking advantage of these additional credits and receipts requires changing the interpretation of types in the logical relation, as we will see. But before we get there, we describe our memoization example in more detail.

Example: memoization of repeatable functions. Memoization is a strategy to improve performance of an algorithm by caching computed values. Figure 10 gives a simple implementation of a concurrent memoization routine that caches the most recently computed value of a function. Given a function f as input, memo f first allocates a reference cell c to store cached results. Then, it returns a new function that, when applied to an argument x, first checks the contents of c to see if there is a cached value for the argument x, and returns it if so. If not, it evaluates $f\,x$ to get some value v, and then stores some (x, v) in c.

Suppose $f : \tau_1 \to \tau_2$. When is memo $f \equiv_{\mathrm{ctx}} f : \tau_1 \to \tau_2$? First, we need to require τ_1 to be an *equality type*, meaning that values of type τ_1 can be tested for equality (e.g. `int`, or $\tau_1 \times \tau_2$, where τ_1 and τ_2 are respectively equality types). Second, if f has observable side-effects, then this equivalence may not hold, since the memoized version will run f fewer times if there is a cache hit. However, if re-running $(f\ x)$ always returns the same value and has no observable side-effects after the first run, then we should expect memo $f \equiv_{\mathrm{ctx}} f : \tau_1 \to \tau_2$. We call such functions *repeatable*.

The existing type system for the language found in ReLoC, which is a version of System F extended with recursive types, mutable references, and concurrency features, is not rich enough to state that a function is repeatable in this sense. To be able to state a contextual equivalence formally, we thus extend the type system in ReLoC with a new type $\tau_1 \to^{\mathsf{rep}} \tau_2$ of repeatable functions. The typing rule for this type uses a new typing relation, $\Gamma \vdash^{\mathsf{rep}} e : \tau$, which implies that e is a repeatable expression of type τ. This judgment is a restriction of the standard typing judgment $\vdash$ that removes the rules for operations that have side effects.[4] We then extend the standard typing judgment $\vdash$ with two new rules for introducing and eliminating terms of type $\to^{\mathsf{rep}}$:

$$\frac{\Gamma, x : \tau_1, f : \tau_1 \to^{\mathsf{rep}} \tau_2 \vdash^{\mathsf{rep}} e : \tau_2}{\Gamma \vdash (\mathbf{rec}\, f\ x = e) : \tau_1 \to^{\mathsf{rep}} \tau_2} \qquad \frac{\Gamma \vdash f : \tau_1 \to^{\mathsf{rep}} \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash f\ e : \tau_2}$$

With these definitions in place, we can formally state the desired result: for all f, if $f : \tau_1 \to^{\mathsf{rep}} \tau_2$ and τ_1 is an equality type, then memo $f \equiv_{\mathrm{ctx}} f : \tau_1 \to \tau_2$. Proving this contextual equivalence runs into the issue with the reverse refinement described above: when trying to show $f \leq_{\mathrm{log}}$ memo f, we have the problem that we need to eliminate laters, but f is an arbitrary function value that may not have any spare steps to take. Adding later credits to the interpretation of function types will allow us to work around this issue.

5.1 Logical Relations in Iris

We first recall the basics of how the step-indexed logical relation in ReLoC is defined. ReLoC uses Iris's program logic to specify the behavior of programs. This means we need a way to do relational reasoning about pairs of programs in Iris, instead of the unary reasoning about a single program that we have seen so far. To do relational reasoning inside of Iris's unary logic, ReLoC uses a technique from CaReSL [Turon et al. 2013], in which a second program is represented by ghost state in Iris (this technique has also been used in other formalization of logical relations in Iris [Krogh-Jespersen et al. 2017; Krebbers et al. 2017b; Tassarotti et al. 2017; Timany et al. 2018; Spies et al. 2021]). This ghost state has assertions of the form $j \Rrightarrow e$ which mean that thread j in this ghost program is executing expression e. Similarly, there are ghost assertions of the form $\ell \mapsto_{\mathrm{s}} v$ which mean that location ℓ points to v in the ghost program's state. The ghost program is "executed" by modifying the ghost state with the update modality. For example, to perform a store of w to reference cell ℓ in the ghost program, we have the rule $j \Rrightarrow (\ell \leftarrow w) * \ell \mapsto_{\mathrm{s}} v \vdash \,\Rrightarrow^{\mathcal{N}_{\mathrm{reloc}}} j \Rrightarrow () * \ell \mapsto_{\mathrm{s}} w$, which reflects that the store returns the unit value $()$, and the reference cell now contains the value w.

To prove a relational property about programs e_1 and e_2, it suffices to prove a Hoare triple about e_1 in which the precondition has a ghost thread running e_2 in an arbitrary evaluation context K. We define an assertion in Iris that expresses this relational pattern. Given $P : (Val \times Val) \to iProp$ where $iProp$ is the type of Iris assertions, we define

$$e_1 \leq e_2 : P \triangleq \forall j, K.\ \{j \Rrightarrow K[e_2]\}\ e_1\ \{v_1.\ \exists v_2.\ j \Rrightarrow K[v_2] * P(v_1, v_2)\}$$

[4]The fragment $\vdash^{\mathsf{rep}}$ is somewhat limited since expressions may not contain instructions with side-effects. It is possible to bend this limitation. In the following, we develop a logical relation for $\vdash^{\mathsf{rep}}$ which admits additional terms that cannot be typed syntactically in the side-effect free fragment $\vdash^{\mathsf{rep}}$, but are *semantically repeatable*. For example, $\lambda().\ \mathbf{let}\ r = ??41); !r{+}1$ is semantically repeatable even though it has side effects from the perspective of $\vdash^{\mathsf{rep}}$.

$$\begin{aligned}
[\![\texttt{int}]\!] &\triangleq \lambda(v_1, v_2).\, \exists z \in \mathbb{Z}.\, v_1 = v_2 = z \\
[\![\tau \times \tau']\!] &\triangleq \lambda(v_1, v_2).\, \exists v_a, v_a', v_b, v_b'.\, v_1 = (v_a, v_a') * v_2 = (v_b, v_b') * [\![\tau]\!](v_a, v_b) * [\![\tau']\!](v_a', v_b') \\
[\![\tau + \tau']\!] &\triangleq \lambda(v_1, v_2).\, \exists u_1, u_2.\, (v_1 = \textbf{inl}(u_1) * v_2 = \textbf{inl}(u_2) * [\![\tau]\!](u_1, u_2)) \lor \\
&\qquad\qquad\qquad\qquad\qquad (v_1 = \textbf{inr}(u_1) * v_2 = \textbf{inr}(u_2) * [\![\tau']\!](u_1, u_2)) \\
[\![\tau \rightarrow \tau']\!] &\triangleq \lambda(v_1, v_2).\, \forall u_1, u_2.\, \Box([\![\tau]\!](u_1, u_2) \mathrel{-\!\!*} (v_1\ u_1) \le (v_2\ u_2) : [\![\tau']\!]) \\
[\![\text{ref}\ \tau]\!] &\triangleq \lambda(v_1, v_2).\, \exists \ell_1, \ell_2.\, v_1 = \ell_1 * v_2 = \ell_2 * \boxed{\exists u_1, u_2.\, \ell_1 \mapsto u_1 * \ell_2 \mapsto_s u_2 * [\![\tau]\!](u_1, u_2)}^{\mathcal{N}.\ell_1.\ell_2}
\end{aligned}$$

Fig. 11. Type interpretation $[\![-]\!]$ in ReLoC. (Polymorphic types and recursive types omitted.)

The adequacy theorem of Iris then ensures that, if $\vdash e_1 \le e_2 : P$, and e_1 terminates with value v_1, there exists an execution of e_2 in which it terminates with a value v_2 such that $P(v_1, v_2)$ holds.

With this method of encoding relational properties, we can now define a logical relation. To simplify the explanation here, we leave out the details of how this approach scales to polymorphic and recursive types, since the addition of later credits that we describe later does not affect that part of the logical relation. The logical relation is defined in three steps:

(1) First, we define a type interpretation function $[\![-]\!] : \mathit{Type} \rightarrow (\mathit{Val} \times \mathit{Val}) \rightarrow \mathit{iProp}$, That is, for each type τ, $[\![\tau]\!]$ is an Iris relation on values.
(2) Next, this type interpretation is used to define a logical refinement relation on closed expressions $(e_1 \le_{\log} e_2 : \tau) \triangleq (e_1 \le e_2 : [\![\tau]\!])$. This definition of logical refinement uses the above encoding of relational reasoning in Iris, with a postcondition that requires the values the expressions reduce to to be related according to $[\![\tau]\!]$.
(3) Finally, the logical refinement relation is lifted from closed expressions to open expressions. Given a type context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and open terms e, e', we define:

$$\Gamma \models e \le_{\log} e' : \tau \triangleq \forall v_1, v_1', \ldots, v_n, v_n'.$$
$$\Big(\mathop{\Large *}_{i=1,\ldots,n} [\![\tau_i]\!](v_i, v_i') \Big) \vdash e[v_1/x_1] \cdots [v_n/x_n] \le_{\log} e'[v_1'/x_1] \cdots [v_n'/x_n] : \tau$$

Figure 11 gives an excerpt of the definition of $[\![-]\!]$ in ReLoC. The two most interesting cases are for $\tau \rightarrow \tau'$ and ref τ. The former says that two values are related at type $\tau \rightarrow \tau'$, if whenever they are applied to values that are assumed to be related at type τ, the resulting application expressions are related at the interpretation of $[\![\tau']\!]$. In the definition of $[\![\tau \rightarrow \tau']\!]$, we use Iris's *persistence* modality $\Box P$, which ensures that the assertion P does not own any non-duplicable resources. We will expand on the persistence modality and its use shortly. For ref τ, the relation says that the two values must be locations, and we use an Iris invariant assertion that requires the two locations to always point to values that are related at type τ. (To keep the changes to ReLoC specific to reverse refinements, we do not use a prepaid invariant here.) The invariant here is implicitly making use of Iris's step-indexing, which is what allows us to avoid the usual circularity issues that arise when trying to define logical relations for systems with higher-order mutable state [Ahmed 2004; Birkedal et al. 2011].

The logical relation has the following two key properties:

Theorem 5.1 (Soundness). *If* $\Gamma \models e_1 \le_{\log} e_2 : \tau$ *then* $\Gamma \vdash e_1 \le_{\text{ctx}} e_2 : \tau$

Theorem 5.2 (Fundamental Property). *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \models e \le_{\log} e : \tau$

The soundness theorem is what ensures that the logical relation is useful for proving contextual equivalences, and it follows from the adequacy of Iris. Meanwhile, the fundamental property lets us automatically deduce that a syntactically well typed term is logically related to itself. This theorem is proved by showing that the logical relation is a congruence relation wrt. all typing rules.

Because the type system here is not sub-structural, a key component of this proof is that the $[\![-]\!]$ predicate is duplicable for all types. This means that when trying to prove $\Gamma \models e_1 \leq_{\log} e_2 : \tau$, we can duplicate the assumptions about the values substituted in for the variables in Γ. This duplicability requirement is what forces us to include the $\square$ modality in the definition of $[\![\tau \to \tau']\!]$ above. The modality $\square\, P$ requires us to prove P without any assertions that we own exclusively (*e.g.,* $\ell \mapsto v$). We may only use duplicable assertions (*e.g.,* $\boxed{P}^{\mathcal{N}}$) and, as a result, $\square\, P$ is also duplicable.

5.2 Extending ReLoC with Repeatability and Later Credits

To extend this logical relation to support the repeatability type judgment ($\vdash^{\mathsf{rep}}$) and repeatable function type ($\to^{\mathsf{rep}}$), we need an Iris assertion that captures that an expression is repeatable. The impredicative features of Iris make this relatively straightforward. First, for repeatability of ghost state programs, we define:

$$\mathsf{repGhost}(e, v) \triangleq \square(\forall j, K.\ j \Rrightarrow K[e] \twoheadrightarrow \mathrel{|\!\!\Rrightarrow}^{\top} j \Rrightarrow K[v])$$

That is, $\mathsf{repGhost}(e, v)$ says that for any ghost thread running e in an evaluation context K, we can perform a ghost update to "execute" e to the value v. The persistence modality $\square$ ensures that this can be done as often as we like. For non-ghost code, we have

$$\mathsf{repImpl}(e, v) \triangleq \{\mathsf{True}\}\ e\ \{v'.\ v = v'\}$$

Just as with repGhost, if $\mathsf{repImpl}(e, v)$ holds, then in the course of a proof we can "run" e and it can only terminate in value v. We do not need $\square$ here, because it is baked into Hoare triples.

Using these definitions, we define a repeatable form of $e_1 \leq e_2 : P$ as follows:

$$(e_1 \leq^{\mathsf{rep}} e_2 : P) \triangleq (e_1 \leq e_2 : \lambda(v_1, v_2).P(v_1, v_2) * \mathsf{repImpl}(e_1, v_1) * \mathsf{repGhost}(e_2, v_2))$$

This requires that the results of evaluating e_1 and e_2 not only need to be related according to P, but also we must have proofs that the expressions can repeatably run to those values. Repeatable versions of the key definitions in the logical relation are then obtained by using $\leq^{\mathsf{rep}}$ in place of $\leq$:

$$[\![\tau \to^{\mathsf{rep}} \tau']\!] \triangleq \lambda v_1, v_2.\ \forall u_1, u_2.\ \square([\![\tau]\!](u_1, u_2) \twoheadrightarrow (v_1\ u_1) \leq^{\mathsf{rep}} (v_2\ u_2) : [\![\tau']\!])$$

$$(e_1 \leq^{\mathsf{rep}}_{\log} e_2 : \tau) \triangleq (e_1 \leq^{\mathsf{rep}} e_2 : [\![\tau]\!])$$

$$\Gamma \models e \leq^{\mathsf{rep}}_{\log} e' : \tau \triangleq \forall v_1, v_1', \ldots, v_n, v_n'.$$
$$\left(\mathop{\Huge *}_{i=1,\ldots,n} [\![\tau_i]\!](v_i, v_i')\right) \vdash e[v_1/x_1]\cdots[v_n/x_n] \leq^{\mathsf{rep}}_{\log} e'[v_1'/x_1]\cdots[v_n'/x_n] : \tau$$

In addition to soundness, the resulting logical relation has an extended form of the fundamental property, as follows:

Theorem 5.3 (Fundamental Property).
(1) *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \models e \leq_{\log} e : \tau$.
(2) *If* $\Gamma \vdash^{\mathsf{rep}} e : \tau$ *then* $\Gamma \models e \leq^{\mathsf{rep}}_{\log} e : \tau$.

The need for later credits. Despite the addition of repeatability to the logical relation, the definition of the logical relation is still problematic if we try to prove $f \leq_{\mathsf{ctx}} \mathsf{memo}\ f : \tau_1 \to \tau_2$, the reverse refinement in our memoization example. Let us see where the issue is. We assume that we have a value $f : \tau_1 \to^{\mathsf{rep}} \tau_2$, where τ_1 is an equality type and we want to show that $f \leq_{\log} \mathsf{memo}\ f : \tau_1 \to \tau_2$. By the fundamental lemma, we know that $f \leq_{\log} f : \tau_1 \to^{\mathsf{rep}} \tau_2$. Proceeding with the proof by unfolding the definitions and introducing universally quantified variables, we need to prove $\{j \Rrightarrow K[\mathsf{memo}\ f]\}\ f\ \{v.\ \exists v'.\ j \Rrightarrow K[v'] * [\![\tau_1 \to \tau_2]\!](v, v')\}$.

By performing updates to evaluate memo f, we first allocate some ghost reference cell c for storing cached values and obtain $c \mapsto_s \mathsf{none}$. We are left with a value f', where

$$\begin{aligned} f' = \lambda x.\, \mathbf{case}\ \mathsf{checkCache}\ c\ x\ \mathbf{of}\ & \mathsf{some}(v) \Rightarrow v \\ & \mid \mathsf{none} \Rightarrow \mathbf{let}\, v = f\ x; c \leftarrow \mathsf{some}(x, v); v \end{aligned}$$

and we must prove $[\![\tau_1 \to \tau_2]\!](f, f')$. Our proof of this must be duplicable (due to $\Box$), yet we will need to use the points-to fact for c when reasoning about the execution of f', and points-to facts are not duplicable. To resolve this issue, we create the following invariant for the points-to fact of c:

$$\boxed{\begin{array}{l}(\exists y, v, v'.\, c \mapsto_s \mathsf{some}(y, v') * \mathsf{repImpl}(f\ y, v) * \mathsf{repGhost}(f\ y, v') * [\![\tau_1]\!](y, y) * [\![\tau_2]\!](v, v')) \\ \quad \lor (c \mapsto_s \mathsf{none})\end{array}}^{\mathcal{N}}$$

This invariant requires that if c contains a cached value, meaning $c \mapsto_s \mathsf{some}(y, v')$ for some y and v', then there is some v such that $f\ y$ repeatably runs to v on the implementation level and v' on the ghost level, where v and v' are related according to the interpretation of τ_2.

Assuming this invariant, we must now show that for any arguments x, x' such that $[\![\tau_1]\!](x, x')$, we have $f\ x \le f'\ x' : [\![\tau_2]\!]$. Here, we use that since τ_1 is an equality type, $[\![\tau_1]\!](x, x')$ implies $x = x'$. Thus, unfolding the definitions, that means we must show

$$\{j \Rrightarrow K[f'\ x]\}\ f\ x\ \{v.\, \exists v'.\, j \Rrightarrow K[v'] * [\![\tau_2]\!](v, v')\}$$

To execute ghost steps of $f'\ x$ we need access to the points-to assertion for c in the invariant. To do so, we use the feature of Iris's invariants that they can be opened as part of an update:

InvOpenUpd

$$\frac{P * \triangleright R \vdash \Rrightarrow^{\mathcal{E} \setminus \mathcal{N}} Q * \triangleright R \qquad \mathcal{N} \subseteq \mathcal{E}}{P * \boxed{R}^{\mathcal{N}} \vdash \Rrightarrow^{\mathcal{E}} Q}$$

With InvOpenUpd, we can open them (guarded by a later), perform some ghost updates, and then close them again—*without* taking a step. (From a concurrency perspective, using the invariant for *zero* steps is fine, because no other thread can observe it in between.)

In our case, once we open the invariant (and use suitable commuting rules), we have to consider two cases. If we are in the "$c \mapsto_s \mathsf{none}$" branch, there is no issue: both the real code and the ghost code will end up executing $f\ x$. We can update the source to advance to the execution of $f\ x$ and then execute $f\ x$ in both ghost code and implementation. From the definition of $[\![\tau_1 \to^{\mathsf{rep}} \tau_2]\!]$, after this code executes, we get $\mathsf{repImpl}(f\ x, v)$ and $\mathsf{repGhost}(f\ x, v')$ for the returned values, which we store in the invariant (again opening it for zero steps).

For the "$c \mapsto_s \mathsf{some}(y, v')$" branch, we obtain:

$$\exists y, v, v'.\, c \mapsto_s \mathsf{some}(y, v') * \triangleright \mathsf{repImpl}(f\ y, v) * \triangleright \mathsf{repGhost}(f\ y, v') * \triangleright [\![\tau_1]\!](y, y) * \triangleright [\![\tau_2]\!](v, v')$$

after applying commuting rules and using timelessness of $c \mapsto_s \mathsf{some}(y, v')$. If the cached argument y is not equal to x, the argument is similar to the "$c \mapsto_s \mathsf{none}$" case. The difficult case is when there is a cache hit (*i.e.,* $x = y$). Then, we use the $c \mapsto_s \mathsf{some}(x, v')$ to execute the steps of $f'\ x$ in the ghost program, which will return the cached value to the point where we own $j \Rrightarrow K[v']$. Turning to reasoning about $f\ x$ in the implementation (*i.e.,* the Hoare triple), we have $\triangleright \mathsf{repImpl}(f\ x, v)$ from the invariant. Now, we would like to use this to argue that $f\ x$ must return v. But $\mathsf{repImpl}(f\ x, v)$ is not timeless (it is a Hoare triple), so we cannot eliminate the $\triangleright$ and our proof attempt is stuck.[5]

[5]We might try to take a step (by doing a beta reduction of $f\ x$) to remove the later, but then we would have to prove a triple about the expression that arises from taking the step instead of $f\ x$, so we would no longer be able to use $\mathsf{repImpl}(f\ x, v)$.

Adding later credits to the logical relation. We would like to have a later credit to eliminate the later guarding $\mathsf{repImpl}(f\ x, v)$ in the above proof. To do so, we will change the definition of $[\![\tau_1 \rightarrow \tau_2]\!]$ and $[\![\tau_1 \rightarrow^{\mathsf{rep}} \tau_2]\!]$ to allow functions to *demand* later credits. This change creates a tension: since functions can now demand credits, to prove the fundamental lemma for the new definition, we need to be able to *provide* them whenever the function is applied to an argument. To resolve this tension we define *receipt pools* (using the time receipt extension depicted in Figure 6). A receipt pool, written $\mathsf{pool}(n)$, where n is a natural number, consists of n different invariants, each containing a time receipt $⧗1$. This assertion supports the following rules:

$$\textsc{PoolExtend} \quad ⧗1 * \mathsf{pool}(n) \vdash \Rrightarrow^{\top} \mathsf{pool}(n+1) \qquad\qquad \textsc{PoolCredits}\quad \frac{\{P * £\,(n+1)\}\ e_2\ \{v.\ Q\} \qquad e_1 \rightarrow_{\mathrm{pure}} e_2}{\{P * \mathsf{pool}(n)\}\ e_1\ \{v.\ Q\}}$$

That is, given a time receipt, we can extend an existing pool by 1. And, if we have a $\mathsf{pool}(n)$ we can generate $n+1$ later credits after taking a step.[6] A receipt pool is duplicable because it consists of multiple invariants, each of which is duplicable.

We then redefine the interpretation of $\rightarrow$ to use receipt pools and later credits:

$$\begin{aligned}[\![\tau \rightarrow \tau']\!] \triangleq \lambda v_1, v_2.\ \exists n, x_1, f_1, e_1, x_2, f_2, e_2. v_1 = (\mathbf{rec}\ f_1\ x_1 = e_1) * v_2 = (\mathbf{rec}\ f_2\ x_2 = e_2) * \mathsf{pool}(n)\\ * \forall u_1, u_2.\ \Box\,([\![\tau]\!](u_1, u_2) \twoheadrightarrow £\,(n+1) \twoheadrightarrow (e_1[u_1/x_1][v_1/f_1]) \leq (e_2[u_2/x_2][v_2/f_2]) : [\![\tau']\!])\end{aligned}$$

Let us break this definition down into pieces. First, we assert v_1 and v_2 are in fact functions. Then, for some existentially quantified n, there must be a $\mathsf{pool}(n)$. Finally, given values u_1 and u_2 related according to $[\![\tau]\!]$, as well as $£\,(n+1)$, we substitute u_1 and u_2, and the recursive definitions of the functions into the function bodies. The resulting expressions must be related according to $[\![\tau']\!]$. That is, whereas the earlier interpretation of $\rightarrow$ was stated in terms of the *applications* of $v_1\ u_1$ and $v_2\ u_2$, here we consider the terms after the application has been reduced by one step, performing the substitution. The interpretation of $\tau \rightarrow^{\mathsf{rep}} \tau'$ is similar, using $\leq^{\mathsf{rep}}$ in place of $\leq$.

The fundamental property holds with this new definition. The biggest change in proving the fundamental property is the case for the function application typing rule. In that case, given $[\![\tau \rightarrow \tau']\!](v_1, v_2)$ and $[\![\tau]\!](u_1, u_2)$, we must prove that $v_1\ u_1 \leq v_2\ u_2 : [\![\tau']\!]$. (This case was trivial with the original definition of $[\![\tau \rightarrow \tau']\!]$, since the desired conclusion was precisely the definition.) Under the new definition, we have $\mathsf{pool}(n)$ for some n and need $£\,(n+1)$ after reducing the application by 1 step. We obtain this $£\,(n+1)$ by using the rule PoolCredits for the β-reduction step, and the rest of the case is straightforward.

In addition, we obtain a stronger version of the soundness theorem, where we may now assume $£n * ⧗n$ when proving two expressions are logically related:

Theorem 5.4 (Soundness). *If* $(£n * ⧗n \vdash \Gamma \models e_1 \leq_{\log} e_2 : \tau)$ *then* $\Gamma \vdash e_1 \leq_{\mathrm{ctx}} e_2 : \tau$

5.3 Memoization with Later Credits

Returning to our memoization example, we use this extended soundness theorem to be able to start with an initial later credit and time receipt. That is, assuming $f : \tau_1 \rightarrow^{\mathsf{rep}} \tau_2$, where τ_1 is an equality type, we prove $£1 * ⧗1 \vdash f \leq_{\log} \mathsf{memo}\ f : \tau_1 \rightarrow \tau_2$. As before, we apply the fundamental lemma to our assumption about f to get that it is logically related to itself. From the new definition of $[\![\tau_1 \rightarrow^{\mathsf{rep}} \tau_2]\!]$, we know that $f = (\mathbf{rec}\ f_0\ z = e)$ for some z, f_0, e, that there is $\mathsf{pool}(n)$ for some existentially quantified n, and that f demands $£\,(n+1)$ to be executed. Using PoolExtend, we exchange our $⧗1$ for $\mathsf{pool}(n+1)$.

[6]This rule relies on a modification of ReceiptCredits.

As before, we execute the ghost code in memo f to allocate the reference cell c and obtain f'. Our invariant storing the points-to for c is slightly different, storing repeatability facts about the result of substituting values into the body e of f:

$$\boxed{\begin{array}{l}(\exists y, v, v'.\, c \mapsto_{\mathsf{s}} \mathsf{some}(y, v') * \mathsf{repImpl}(e[y/z][f/f_0], v) * \mathsf{repGhost}(e[y/z][f/f_0], v') \\ \qquad * [\![\tau_1]\!](y, y) * [\![\tau_2]\!](v, v')) \vee (c \mapsto_{\mathsf{s}} \mathsf{none})\end{array}}^{\mathcal{N}}$$

Now, when proving $[\![\tau_1 \to \tau_2]\!](f, f')$ we choose the existentially quantified natural number in the interpretation of the arrow type to be $n + 1$, since we have $\mathsf{pool}(n + 1)$. This choice means we get $£(n + 2)$ now when reasoning about f and f' applied to some arguments that have been substituted in. The non-cache hit cases proceed as before, except that we have to give up $£(n + 1)$ of our $£(n + 2)$ to use our assumptions about f. In the case of a cache hit, we use $£1$ to remove the later from repImpl. This repImpl lets us show that the non-memoized code will return the value that is cached in the ghost code version, and thus complete the proof.

6 THE MODEL OF IRIS

In this section, we explain how the later credit mechanism is integrated into the model of Iris. Without later credits, Iris factors into two layers: *a base logic* and *a program logic*. The *base logic*, in short, is a step-indexed logic of bunched implications [O'Hearn and Pym 1999]. What that means is that (1) the base logic is step-indexed (*e.g.*, it has the later modality $\triangleright P$), (2) it has the distinguishing connectives of separation logic (*e.g.*, $P * Q$ and $P \mathrel{-\!\!*} Q$), and (3) it has support for ghost state through updates and resources. Importantly, the base logic does not yet have any notion of programs, state, or Hoare triples. These notions are defined in the *program logic*, which is built on top of the base logic (for a discussion of the definition see [Jung et al. 2018]). The program logic defines a number of connectives that we have already encountered in terms of the base logic: updates with masks, invariants, Hoare triples, and logically atomic triples.

We insert later credits as an additional layer between the base logic and the program logic. That is, we define the later credits mechanism in terms of the base logic without changing Iris's model of propositions, since the base logic is already expressive enough to define later credits $£n$ and the later elimination update $\Rrightarrow_{\mathsf{le}} P$ (in §6.1). Equipped with the later credits mechanism, we then *redefine* the program logic of Iris. For the most part, this change involves replacing standard updates with later elimination updates. The non-trivial change to the program logic is the way we alter Hoare triples, which we will explain step-by step (in §6.2).

6.1 Update Modalities

Technically, Iris has two kinds of update modalities: the *ghost state update* $\dot{\Rrightarrow} P$ (notice the dot!) and the *fancy update* ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$. The former enables only manipulation of ghost state $\boxed{r}^{\gamma}$, whereas the latter enables manipulation of ghost state and additionally invariants $\boxed{P}^{\mathcal{N}}$. The difference between both updates is often swept under the rug, since ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ generalizes $\dot{\Rrightarrow} P$, meaning $\dot{\Rrightarrow} P \vdash {\Rrightarrow}^{\mathcal{E}} P$. However, to explain how later credits fit in, we distinguish between both notions here. The update $\dot{\Rrightarrow} P$ is a primitive of the *base logic*, whereas ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ is a derived notion of the *program logic*.

To integrate later credits, we define two new modalities: $\dot{\Rrightarrow}_{\mathsf{le}} P$ and ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}_{\mathsf{le}} P$. The former is defined in terms of $\dot{\Rrightarrow} P$, and the latter is defined in terms of $\dot{\Rrightarrow}_{\mathsf{le}} P$ (analogous to how ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ is defined in terms of $\dot{\Rrightarrow} P$). We start with the ghost state update (in §6.1.1) and then proceed with the fancy update (in §6.1.2).

6.1.1 Ghost State Updates. The ghost state update $\dot{\Rrightarrow} P$ is one of the central pieces for working with ghost state in Iris. It obeys the following rules:

$$\textsc{UpdReturn}\quad P \vdash \dot{\Rrightarrow} P \qquad \textsc{UpdBind}\quad (\dot{\Rrightarrow} P) * (P \mathrel{-\!\!*} \dot{\Rrightarrow} Q) \vdash \dot{\Rrightarrow} Q \qquad \textsc{UpdGhost}\quad \frac{r \rightsquigarrow r'}{\boxed{r}^{\gamma} \vdash \dot{\Rrightarrow} \boxed{r'}^{\gamma}}$$

where $\boxed{r}^{\gamma}$ is Iris's ghost state ownership connective, and $r \rightsquigarrow r'$ is the underlying notion of "frame preserving update" for Iris's resources.

We use this update modality and Iris's ghost state as the building blocks for later credits. That is, we define the ghost state connectives $£n \triangleq \boxed{\circ n}^{\gamma_{\mathsf{lc}}}$ and $£_{\bullet}\, n \triangleq \boxed{\bullet n}^{\gamma_{\mathsf{lc}}}$ (for some fixed ghost name γ_{lc}), where the resources are drawn from Iris's $\mathit{Auth}(\mathbb{N}, +)$ resource algebra. Moreover, we define the later elimination update (for ghost state) as:

$$\dot{\Rrightarrow}_{\mathsf{le}} P \triangleq \forall n.\, £_{\bullet}\, n \mathrel{-\!\!*} \dot{\Rrightarrow}((£_{\bullet}\, n * P) \vee (\exists m < n.\, £_{\bullet}\, m * \triangleright \dot{\Rrightarrow}_{\mathsf{le}} P))$$

We have discussed the definition of this modality already in the paper in Section 5. In short, the update "$\dot{\Rrightarrow}$" ensures that $\dot{\Rrightarrow}_{\mathsf{le}} P$ allows ghost state updates (*i.e.*, $\dot{\Rrightarrow} P \vdash \dot{\Rrightarrow}_{\mathsf{le}} P$), the later "$\triangleright$" ensures that $\dot{\Rrightarrow}_{\mathsf{le}} P$ allows later eliminations (*i.e.,* $£\,1 * \triangleright P \vdash \dot{\Rrightarrow}_{\mathsf{le}} P$), the ever-decreasing credit supply "$£_\bullet\, n$" ensures that the number of later eliminations is not unbounded, and the recursive occurrence "$\dot{\Rrightarrow}_{\mathsf{le}} P$" ensures transitivity (*i.e.,* $\dot{\Rrightarrow}_{\mathsf{le}} \dot{\Rrightarrow}_{\mathsf{le}} P \vdash \dot{\Rrightarrow}_{\mathsf{le}} P$). The recursive definition is solved using Iris's guarded fixed points [Jung et al. 2018], a version of Banach's fixed points. Since they are unique and the occurrence is positive, it does not matter which kind of fixed point one uses (*i.e.,* least, greatest, or guarded).

The later elimination ghost state update "$\dot{\Rrightarrow}_{\mathsf{le}}$" inherits almost all properties of the update "$\dot{\Rrightarrow}$". The only rule that is lost is the elimination of the modality in front of Iris's so-called *plain* propositions:

$$\dot{\Rrightarrow} \blacksquare P \vdash P$$

This elimination rule is rarely used in Iris developments. Its main use case is in the runST work [Timany et al. 2018] (see also Section 5 in the paper).

6.1.2 Fancy Updates. Let us turn to *fancy updates* ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$. In Iris, these updates are built on top of ghost state updates $\dot{\Rrightarrow} P$. Thus, we build them here on top of the later elimination ghost state updates $\dot{\Rrightarrow}_{\mathsf{le}} P$. As a consequence, they inherit most of the properties[7] of $\dot{\Rrightarrow} P$ and, additionally, the ability to eliminate later modalities.

Formally, Iris's fancy update is defined as follows (the details do not matter too much for understanding how later credits factor in):

$${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P \triangleq \mathsf{wsat} * \boxed{\mathcal{E}_1}^{\gamma_{\mathrm{en}}} \mathrel{-\!\!*} \dot{\Rrightarrow} \diamond (\mathsf{wsat} * \boxed{\mathcal{E}_2}^{\gamma_{\mathrm{en}}} * P)$$

The world satisfaction wsat ensures that all currently enabled invariants hold, while the ghost state $\boxed{\mathcal{E}}^{\gamma_{\mathrm{en}}}$ is linked up to that and asserts that the invariants $\mathcal{E}$ are currently enabled.

The change in the fancy update is as simple as one would hope for:

$${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}_{\mathsf{le}} P \triangleq \mathsf{wsat} * \boxed{\mathcal{E}_1}^{\gamma_{\mathrm{en}}} \mathrel{-\!\!*} \dot{\Rrightarrow}_{\mathsf{le}} \diamond (\mathsf{wsat} * \boxed{\mathcal{E}_2}^{\gamma_{\mathrm{en}}} * P)$$

Note that we write ${\Rrightarrow}^{\mathcal{E}} P \triangleq {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} P$ (and ${\Rrightarrow}^{\mathcal{E}}_{\mathsf{le}}$ respectively) if both masks are the same. (In other Iris presentations, the mask for a non-mask-changing update is at the bottom (*i.e.,* "${\Rrightarrow}_{\mathcal{E}}$"). Since this position conflicts with the "le" of our later elimination updates, we move it up.)

6.2 Weakest Precondition

Recall the definition of Hoare triples from the paper:

$$\{P\}\ e\ \{v.\, Q\}_{\mathcal{E}} \triangleq \square(P \mathrel{-\!\!*} \mathsf{wp}^{\mathcal{E}}\ e\ \{v.\, Q\})$$

In the following, we discuss the definition of the weakest precondition $\mathsf{wp}^{\mathcal{E}}\ e\ \{v.\, Q\}$. We discuss multiple versions. We start with the traditional version without later credits (in §6.2.1). Then, we discuss how later credits can be integrated into this definition (in §6.2.2). Subsequently, we discuss the extension of Jourdan [2021] to the traditional weakest precondition to allow eliminating an increasing number of laters per step (in §6.2.3). Finally, we discuss how later credits can be integrated with this extension (in §6.2.4).

Note that in all of the definitions, we will ignore Iris's support for *prophecy variables*. The integration of later credits is orthogonal to this feature, so we omit them here.

[7] Analogously to the ghost state update, the later elimination update also does not enjoy some of the plain rules (see Iris Coq code).

6.2.1 Weakest Precondition without Later Credits. Compared to the simplified version of the weakest precondition shown in Section 5 of the paper, the full weakest precondition supports concurrency and uses *fancy updates* ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$ to support invariants:

$$\begin{aligned}
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq {\Rrightarrow}^{\mathcal{E}} Q[e/v] && \text{if } e \in \mathit{Val}\\
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq \forall \sigma, n_t.\ S(\sigma, n_t) \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\emptyset}\,\mathrm{red}(e, \sigma) * && \text{if } e \notin \mathit{Val}\\
&\qquad (\forall e', \sigma', \vec{e}.(e, \sigma) \rightarrow (e', \sigma', \vec{e}) \mathrel{-\!\!*} {}^{\emptyset}{\Rrightarrow}^{\emptyset} \triangleright {}^{\emptyset}{\Rrightarrow}^{\mathcal{E}}\\
&\qquad\quad S(\sigma', |\vec{e}| + n_t) * \mathrm{wp}^{\mathcal{E}}\ e'\ \{v.\ Q\} *\\
&\qquad\quad \mathop{\Huge *}_{e_f \in \vec{e}} \mathrm{wp}^{\top}\ e_f\ \{v.\ \mathsf{True}\})
\end{aligned}$$

To support invariants, the weakest precondition is parameterized by the mask $\mathcal{E}$, describing the set of invariants that are currently active. In the step case, the weakest precondition allows us to open all invariants (and update ghost state) to justify the step, by showing a mask-changing update "${}^{\mathcal{E}}{\Rrightarrow}^{\emptyset}$" that deactivates all invariants. After the step, the definition requires us to establish them again with an update "${}^{\emptyset}{\Rrightarrow}^{\mathcal{E}}$". The later $\triangleright$ in between enables us to eliminate a later from our assumptions when taking a step.

To integrate concurrency, the operational semantics provides a list $\vec{e}$ of forked-off expressions in the step case. For each of the forked-off expressions $e_f \in \vec{e}$, the definition requires us to prove a weakest precondition with a trivial postcondition. The state interpretation is additionally parameterized by the number of threads n_t.

6.2.2 Weakest Precondition with Later Credits. For the extension with later credits, two changes are necessary: First, we use the fancy updates that support later elimination "${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}_{\mathsf{le}}$". Second, the definition of the weakest precondition is changed to grant access to one later credit in the step case.

$$\begin{aligned}
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq {\Rrightarrow}^{\mathcal{E}}_{\mathsf{le}} Q[e/v] && \text{if } e \in \mathit{Val}\\
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq \forall \sigma, n_t.\ S(\sigma, n_t) \mathrel{-\!\!*} {}^{\mathcal{E}}{\Rrightarrow}^{\emptyset}_{\mathsf{le}}\,\mathrm{red}(e, \sigma) * && \text{if } e \notin \mathit{Val}\\
&\qquad (\forall e', \sigma', \vec{e}.(e, \sigma) \rightarrow (e', \sigma', \vec{e}) \mathrel{-\!\!*} \pounds 1 \mathrel{-\!\!*} {}^{\emptyset}{\Rrightarrow}^{\emptyset}_{\mathsf{le}} \triangleright {}^{\emptyset}{\Rrightarrow}^{\mathcal{E}}_{\mathsf{le}}\\
&\qquad\quad S(\sigma', |\vec{e}| + n_t) * \mathrm{wp}^{\mathcal{E}}\ e'\ \{v.\ Q\} *\\
&\qquad\quad \mathop{*}_{e_f \in \vec{e}} \mathrm{wp}^{\top}\ e_f\ \{v.\ \mathsf{True}\})
\end{aligned}$$

Note that we keep the later that was there previously. We do so to simplify backwards compatibility. That is, without any trouble, this definition validates the rules in Figure 2 *and* in Figure 5. Thus, adapting for example Iris's proof mode [Krebbers et al. 2017b] is straightforward. In the adequacy proof, dealing with this additional later is simple: we know $\triangleright P \vdash \pounds 1 \mathrel{-\!\!*} \dot{\Rrightarrow}_{\mathsf{le}} P$. Thus, in the adequacy proof, we can replace the later with just another "$\pounds 1 \mathrel{-\!\!*} \dot{\Rrightarrow}_{\mathsf{le}}$" and simply allocate $\pounds 2n$ initially.

6.2.3 Weakest Precondition with Increasing Laters without Later Credits. Recently, Iris's weakest precondition has been extended to support an increasing number of later eliminations per step, depending on how many program steps have already happened [Jourdan 2021]. For this change, the state interpretation is extended with a parameter n_s determining the number of steps that have already been taken, which is increased with each step. This number n_s is then mapped to the the number of laters that can be eliminated in a step through a function $\Xi : \mathbb{N} \rightarrow \mathbb{N}$.[8] To allow updating

[8]The definition of the weakest precondition adds 1 to this, to ensure that at least one later can be eliminated in each step (and to make the definition contractive).

ghost state interleaved with later elimination, the weakest precondition interleaves fancy updates and laters (changes from §6.2.1 highlighted in blue):

$$\begin{aligned}
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq {\mid\!\Rrightarrow}^{\mathcal{E}} Q[e/v] && \text{if } e \in \mathit{Val}\\
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq \forall \sigma, n_s, n_t.\ S(\sigma, n_s, n_t) \mathrel{-\!\!*} {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\emptyset}\, \mathrm{red}(e, \sigma) * && \text{if } e \notin \mathit{Val}\\
&\qquad (\forall e', \sigma', \vec{e}. (e, \sigma) \rightarrow (e', \sigma', \vec{e}) \mathrel{-\!\!*} ({\mid\!\Rrightarrow}_{\emptyset} \triangleright {\mid\!\Rrightarrow}_{\emptyset})^{1+\Xi(n_s)}\, {}^{\emptyset}{\mid\!\Rrightarrow}^{\mathcal{E}} \\
&\qquad\quad S(\sigma', 1 + n_s, |\vec{e}| + n_t) * \mathrm{wp}^{\mathcal{E}}\ e'\ \{v.\ Q\} * \\
&\qquad\quad \mathop{\Large *}_{e_f \in \vec{e}} \mathrm{wp}^{\top}\ e_f\ \{v.\ \mathsf{True}\})
\end{aligned}$$

6.2.4 Weakest Precondition with Increasing Later Credits. We describe how the definition needs to be changed to obtain a flexible number of later credits per step. Instead of putting an iterated alternation of fancy updates and laters into the definition, we get $1 + \Xi(n_s)$ credits per step. The definition we obtain is (changes from §6.2.2 highlighted in blue):

$$\begin{aligned}
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq {\mid\!\Rrightarrow}^{\mathcal{E}}_{\mathsf{le}} Q[e/v] && \text{if } e \in \mathit{Val}\\
&\mathrm{wp}^{\mathcal{E}}\ e\ \{v.\ Q\} \triangleq \forall \sigma, n_s, n_t.\ S(\sigma, n_s, n_t) \mathrel{-\!\!*} {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\emptyset}_{\mathsf{le}}\, \mathrm{red}(e, \sigma) * && \text{if } e \notin \mathit{Val}\\
&\qquad (\forall e', \sigma', \vec{e}. (e, \sigma) \rightarrow (e', \sigma', \vec{e}) \mathrel{-\!\!*} £(1 + \Xi(n_s)) \mathrel{-\!\!*} {}^{\emptyset}{\mid\!\Rrightarrow}^{\emptyset}_{\mathsf{le}} \triangleright {}^{\emptyset}{\mid\!\Rrightarrow}^{\mathcal{E}}_{\mathsf{le}} \\
&\qquad\quad S(\sigma', 1 + n_s, |\vec{e}| + n_t) * \mathrm{wp}^{\mathcal{E}}\ e'\ \{v.\ Q\} * \\
&\qquad\quad \mathop{\Large *}_{e_f \in \vec{e}} \mathrm{wp}^{\top}\ e_f\ \{v.\ \mathsf{True}\})
\end{aligned}$$

The number of steps n_s is connected to the time receipts $⧗ n$ in the state interpretation S. That is, the time receipts are elements of the resource algebra $\mathit{Auth}(\mathbb{N}, +)$. The authoritative element $⧗_{\bullet}\, n_s \triangleq \boxed{\bullet n_s}^{\gamma_{\mathrm{tr}}}$ resides in the state interpretation, and the receipts $⧗ n \triangleq \boxed{\circ n}^{\gamma_{\mathrm{tr}}}$ are the fragments.

6.3 Unbounded Credits are Unsound with Finite Step-indexing

In Section 6 of the paper, we claim that the rule CreditsPost that gives us an arbitrary number of credits in the postcondition is unsound under finite step-indexing:

$$\textsc{CreditsPost}\quad \frac{\{P\}\ e\ \{v.\ Q\} \qquad e \notin \mathit{Val}}{\{P\}\ e\ \{v.\ Q * £ n\}}$$

To prove this, we show that $\{\mathsf{True}\}\ \mathsf{Skip}\ \{\_.\ \mathsf{False}\}$, which by Iris's adequacy theorem entails False at the meta-level.

Proof. We use the well-known fact that $\vdash \exists n.\ \triangleright^n \mathsf{False}$ holds in Iris (the proof goes by Löb induction). Thus, we show $\triangleright^n \mathsf{False} \vdash \{\mathsf{True}\}\ \mathsf{Skip}\ \{\_.\ \mathsf{False}\}$ for some n.

Moreover, we claim that $(\triangleright^n \mathsf{False}) * £ n \vdash {\mid\!\Rrightarrow}_{\mathsf{le}} \mathsf{False}$: the proof is by induction on n. In the base case, we are done by assumption. In the inductive step, we eliminate one later, using one of the credits we have, by the rule LEUpdLater, before using the inductive hypothesis:

$$\textsc{LEUpdLater}\quad £1 * \triangleright P \vdash \dot{\mid\!\Rrightarrow}_{\mathsf{le}} P$$

Thus, it suffices to show that $\triangleright^n \mathsf{False} \vdash \{\mathsf{True}\}\ \mathsf{Skip}\ \{\_.\ (\triangleright^n \mathsf{False}) * £ n\}$ by the rule of consequence. We frame $\triangleright^n \mathsf{False}$ and are left to prove $\vdash \{\mathsf{True}\}\ \mathsf{Skip}\ \{\_.\ £ n\}$. We apply CreditsPost and it remains to show $\vdash \{\mathsf{True}\}\ \mathsf{Skip}\ \{\_.\ \mathsf{True}\}$. This holds trivially by executing the Skip. □

REFERENCES

Amal Ahmed. 2004. *Semantics of types for mutable state.* Ph. D. Dissertation. Princeton University.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *POPL*. 119–132. https://doi.org/10.1145/1926385.1926401

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*. 442–451. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *LMCS* 17, 3 (2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Jacques-Henri Jourdan. 2021. Flexible number of logical steps per physical step. https://gitlab.mpi-sws.org/iris/iris/-/merge_requests/595 Iris merge request 595.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3093333.3009855

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*. 218–231. https://doi.org/10.1145/3093333.3009877

Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *Bulletin of Symbolic Logic* 5, 2 (June 1999), 215–244. https://doi.org/10.2307/421090

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*. 80–95. https://doi.org/10.1145/3453483.3454031

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite step-indexing: Decoupling concrete and logical steps. In *ESOP (LNCS, Vol. 9632)*. 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A higher-order logic for concurrent termination-preserving refinement. In *ESOP (LNCS, Vol. 10201)*. 909–936. https://doi.org/10.1007/978-3-662-54434-1_34

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. 377–390. https://doi.org/10.1145/2500365.2500600