RustBelt Meets Relaxed Memory: Technical Appendix

HOANG-HAI DANG, MPI-SWS, Germany
JACQUES-HENRI JOURDAN, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, France
JAN-OLIVER KAISER, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

Disclaimer. This document is only intended to aid the approach to the technical details of this work. As such, it may be outdated or contain serious typos. When one is in doubt, please confer the authoritative Coq formalization.

This work is accompanied by a Coq formalization, which includes all definitions, theorems, lemmas and proofs in this appendix, with the exception of the correspondence proof (§2).

Authors' addresses: Hoang-Hai Dang, MPI-SWS, Saarland Informatics Campus, Germany, haidang@mpi-sws.org; Jacques-Henri Jourdan, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France, jacques-henri.jourdan@lri.fr; Jan-Oliver Kaiser, MPI-SWS, Saarland Informatics Campus, Germany, janno@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

2475-1421/2020/1-ART99
https://doi.org/

Contents

1 LANGUAGE

1.1 Grammar

Our language is an extension of the original RustBelt's λ_{Rust} with the relaxed memory semantics of ORC11 (§1.2). λ_{Rust} is a lambda calculus with integers, locations with explicit allocation and deallocation, and a notion of poison value ☣. Instead of **sc** for atomic accesses, we use release **rel**, acquire **acq**, and relaxed **rlx** accesses together with fences.

The grammar is given in Fig. 1. Several syntactic sugars are taken as-is from the original RustBelt, given in Fig. 2. We refer the reader to the original RustBelt appendix ([Jung et al. 2017]) for more explanation of the grammar and syntactic sugars.

1.2 Operational Semantics

Following iGPS ([Kaiser et al. 2017]) we use an operational semantics for relaxed memory so that it can be instantiated in Iris. For this work, we extend iGPS's operational semantics for RA+NA to include relaxed accesses and fences.

$$
\begin{array}{rl}
 & z \in \mathbb{Z} \\
\mathit{Expr} \ni e ::= & \mid v \mid x \\
 & \mid e.e \mid e + e \mid e - e \mid e \leq e \mid e == e \\
 & \mid e(\overline{e}) \\
 & \mid {}^{*o}e \mid e_1 :=_o e_2 \mid \textsf{CAS}(e_0, e_1, e_2, o_f, o_r, o_w) \\
 & \mid \textbf{alloc}(e) \mid \textbf{free}(e_1, e_2) \\
 & \mid \textbf{case}\ e\ \textbf{of}\ \overline{e} \\
 & \mid \textbf{fork}\ \{\ e\ \} \\
 & \mid \textbf{fence}_o \\
\mathit{Val} \ni v ::= & \text{☣} \mid \ell \mid z \mid \textbf{rec}\ f(\overline{x}) := e \\
\mathit{Loc} \ni \ell ::= & (i, n) \qquad i \in \mathbb{N}^+, n \in \mathbb{Z} \\
\mathit{Order} \ni o ::= & \textbf{acq} \mid \textbf{rel} \mid \textbf{rlx} \mid \textbf{na} \\
\mathit{Ctx} \ni K ::= & \mid \bullet \\
 & \mid K.e \mid v.K \mid K + e \mid v + K \mid K - e \mid v - K \\
 & \mid K \leq e \mid v \leq K \mid K == e \mid v == K \\
 & \mid K(\overline{e}) \mid v(\overline{v} +\!\!+ [K] +\!\!+ \overline{e}) \\
 & \mid {}^{*o}K \mid K :=_o e \mid v :=_o K \\
 & \mid \textsf{CAS}(K, e_1, e_2, o_f, o_r, o_w) \\
 & \mid \textsf{CAS}(v_0, K, e_2, o_f, o_r, o_w) \\
 & \mid \textsf{CAS}(v_0, v_1, K, o_f, o_r, o_w) \\
 & \mid \textbf{alloc}(K) \mid \textbf{free}(K, e_2) \mid \textbf{free}(e_1, K) \\
 & \mid \textbf{case}\ K\ \textbf{of}\ \overline{e}
\end{array}
$$

Fig. 1. Language syntax.

$$
\begin{aligned}
\textbf{funrec}\, f(\overline{x})\, \textbf{ret}\, k := e &:= \textbf{rec}\, f([k] \mathbin{+\!\!+} \overline{x}) := e \\
\textbf{let}\, x = e\, \textbf{in}\, e' &:= (\textbf{rec}\, \_([x]) := e')(e) \\
e'; e &:= \textbf{let}\, \_ = e'\, \textbf{in}\, e \\
\textbf{letcont}\, k(\overline{x}) := e\, \textbf{in}\, e' &:= \textbf{let}\, k = (\textbf{rec}\, k(\overline{x}) := e)\, \textbf{in}\, e' \\
\textbf{jump}\, k(\overline{e}) &:= k(\overline{e}) \\
\textbf{call}\, f(\overline{e})\, \textbf{ret}\, k &:= f([k] \mathbin{+\!\!+} \overline{e})
\end{aligned}
$$

$$
\begin{aligned}
\textbf{false} &:= 0 \\
\textbf{true} &:= 1 \\
\textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 &:= \textbf{case}\, e_0\, \textbf{of}\, [e_1, e_2]
\end{aligned}
$$

$$
\begin{aligned}
{}^{*}e &:= {}^{*\mathsf{na}}e \\
e_1 := e_2 &:= e_1 :=_{\mathsf{na}} e_2 \\
\textbf{new} &:= \textbf{rec}\, \mathtt{new}(\mathtt{size}) := \\
&\qquad \textbf{if}\, \mathtt{size} == 0\, \textbf{then}\, (42, 1337)\, \textbf{else}\, \textbf{alloc}(\mathtt{size}) \\
\textbf{delete} &:= \textbf{rec}\, \mathtt{delete}(\mathtt{size}, \mathtt{ptr}) := \\
&\qquad \textbf{if}\, \mathtt{size} == 0\, \textbf{then}\, \text{☣}\, \textbf{else}\, \textbf{free}(\mathtt{size}, \mathtt{ptr}) \\
\textbf{memcpy} &:= \textbf{rec}\, \mathtt{memcpy}(\mathtt{dst}, \mathtt{len}, \mathtt{src}) := \\
&\qquad \textbf{if}\, \mathtt{len} \leq 0\, \textbf{then}\, \text{☣}\, \textbf{else} \\
&\qquad \mathtt{dst}.0 := \mathtt{src}.0; \\
&\qquad \mathtt{memcpy}(\mathtt{dst}.1, \mathtt{len} - 1, \mathtt{src}.1) \\
e_1 :=_n {}^{*}e_2 &:= \mathtt{memcpy}(e_1, n, e_2) \\
e \overset{\mathsf{inj}\, i}{:==} () &:= e.0 := i \\
e_1 \overset{\mathsf{inj}\, i}{:==} e_2 &:= e_1.0 := i; e_1.1 := e_2 \\
e_1 \overset{\mathsf{inj}\, i}{:==}_n {}^{*}e_2 &:= e_1.0 := i; e_1.1 :=_n {}^{*}e_2
\end{aligned}
$$

$$
\begin{aligned}
\textbf{skip} &:= \textbf{let}\, x = \text{☣}\, \textbf{in}\, \text{☣} \\
\textbf{newlft} &:= \text{☣} \\
\textbf{endlft} &:= \textbf{skip}
\end{aligned}
$$

Fig. 2. Syntactic sugars.

The semantics, called ORC11, is defined by three sub semantics: the expressions semantics (Fig. 5), the machine semantics (Fig. 7), and the race-detecting semantics (Fig. 8 and Fig. 9). The combined thread pool semantics is given in Fig. 10 and Fig. 11. In §2, we sketch a proof of correspondence that relates ORC11 to the axiomatic semantics from Lahav et al. [2017].

$$
\begin{aligned}
\pi \in \mathit{Thread} &::= \mathbb{N} \\
t \in \mathit{Time} &::= \mathbb{N}^{+} \\
\omega \in \mathit{MsgVal} &::= \dagger \mid \maltese \mid v \in \mathit{Val} \\
\mathit{ActionIds} &::= 2^{\mathbb{N}^{+}} \\
V \in \mathit{View} &::= \mathit{Loc} \xrightarrow{\text{fin}} \{\mathsf{w} : \mathit{Time}, \mathsf{aw} : \mathit{ActionIds}, \mathsf{nr} : \mathit{ActionIds}, \mathsf{ar} : \mathit{ActionIds}\} \\
\mathcal{V} \in \mathit{ThreadView} &::= \left\{\mathsf{rel} : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{View}, \mathsf{frel} : \mathit{View}, \mathsf{cur} : \mathit{View}, \mathsf{acq} : \mathit{View}\right\} \\
m \in \mathit{ExtMsg} &::= \left\{\mathsf{ts} : \mathit{Time}, \mathsf{val} : \mathit{MsgVal}, \mathsf{view} : \mathit{View}^{?}\right\} \\
\mathcal{M} \in \mathit{MsgPool} &::= \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Time} \xrightarrow{\text{fin}} \left\{\mathsf{val} : \mathit{MsgVal}, \mathsf{view} : \mathit{View}^{?}\right\} \\
\mathcal{N} \in V_{\mathbf{Race}} &::= \mathit{View} \\
\varsigma \in \mathit{GlobalState} &::= \mathit{MsgPool} \times V_{\mathbf{Race}} \\
\mathit{MemEvent} \ni \varepsilon &::= \mid \langle \mathrm{Alloc}, \ell, n \in \mathbb{N}^{+} \rangle \mid \langle \mathrm{Dealloc}, \ell, n \in \mathbb{N}^{+} \rangle \\
&\mid \langle \mathrm{Read}, \ell, v, o \rangle \mid \langle \mathrm{Write}, \ell, v, o \rangle \mid \langle \mathrm{Update}, \ell, v_r, v_w, o_r, o_w \rangle \\
&\mid \langle \mathrm{Fence}, o \rangle
\end{aligned}
$$

Fig. 3. Machine state definitions.

$$\omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}) := \exists t.\, \mathcal{M}(\ell)(t) = (\omega, \_) \wedge t \leq \mathcal{V}.\text{cur}(\ell)$$

MsgVal Injection. $\boxed{\omega \equiv v}$

$$v \equiv v \qquad\qquad \text{☥} \equiv \text{☣}$$

Unallocated. $\boxed{\ell \in \text{unalloc}(\mathcal{M})}$

$$\frac{\ell \notin \text{dom}(\mathcal{M})}{\ell \in \text{unalloc}(\mathcal{M})} \qquad\qquad \frac{\exists t.\, \mathcal{M}(\ell)(t) = (\text{⳩}, \_)}{\ell \in \text{unalloc}(\mathcal{M})}$$

Val Equality. $\boxed{\mathcal{M} \vdash v_1 = v_2}$

$$\mathcal{M} \vdash z = z \qquad\qquad \mathcal{M} \vdash \ell = \ell \qquad\qquad \frac{\ell_1 \in \text{unalloc}(\mathcal{M}) \vee \ell_2 \in \text{unalloc}(\mathcal{M})}{\mathcal{M} \vdash \ell_1 = \ell_2}$$

Val Inequality. $\boxed{\vdash v_1 \neq v_2}$

$$\frac{z_1 \neq z_2}{\vdash z_1 \neq z_2} \qquad\qquad \frac{\ell_1 \neq \ell_2}{\vdash \ell_1 \neq \ell_2} \qquad\qquad \vdash \ell \neq 0 \qquad\qquad \vdash 0 \neq \ell$$

Val Comparibility. $\boxed{\vdash v_1 =^? v_2}$

$$\vdash z_1 =^? z_2 \qquad\qquad \vdash \ell_1 =^? \ell_2 \qquad\qquad \vdash \ell =^? 0 \qquad\qquad \vdash 0 =^? \ell$$

Order's Lattice. $\boxed{o_1 \sqsubseteq o_2}$

$$\texttt{na} \sqsubseteq \texttt{rlx} \qquad \texttt{na} \sqsubseteq \texttt{acq} \qquad \texttt{na} \sqsubseteq \texttt{rel} \qquad \texttt{rlx} \sqsubseteq \texttt{acq} \qquad \texttt{rlx} \sqsubseteq \texttt{rel}$$

Fig. 4. Auxilliary relations.

Expression Step. $\boxed{\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon^?} e_1', e_2'^?}$

OE-ECTX
$$\frac{e \to e_1', e_2'^?}{\mathcal{M}, \mathcal{V} \vdash K[e] \to K[e_1'], e_2'^?}$$

OE-PROJ
$$\mathcal{M}, \mathcal{V} \vdash \ell.n \to \ell + n$$

OE-ADD
$$\frac{z_1 + z_2 = z'}{\mathcal{M}, \mathcal{V} \vdash z_1 + z_2 \to z'}$$

OE-SUB
$$\frac{z_1 - z_2 = z'}{\mathcal{M}, \mathcal{V} \vdash z_1 - z_2 \to z'}$$

OE-LE-TRUE
$$\frac{z_1 \leq z_2}{\mathcal{M}, \mathcal{V} \vdash z_1 \leq z_2 \to 1}$$

OE-LE-FALSE
$$\frac{z_1 > z_2}{\mathcal{M}, \mathcal{V} \vdash z_1 \leq z_2 \to 0}$$

OE-EQ-TRUE
$$\frac{\mathcal{M} \vdash v_1 = v_2}{\mathcal{M}, \mathcal{V} \vdash v_1 == v_2 \to 1}$$

OE-EQ-FALSE
$$\frac{\vdash v_1 \neq v_2}{\mathcal{M}, \mathcal{V} \vdash v_1 == v_2 \to 0}$$

OE-ALLOC
$$\frac{n > 0}{\mathcal{M}, \mathcal{V} \vdash \mathtt{alloc}(n) \xrightarrow{\langle \text{Alloc}, \ell, n \rangle} \ell}$$

OE-FREE
$$\frac{n > 0}{\mathcal{M}, \mathcal{V} \vdash \mathtt{free}(n, \ell) \xrightarrow{\langle \text{Dealloc}, \ell, n \rangle} \text{☣}}$$

OE-READ
$$\mathcal{M}, \mathcal{V} \vdash {*^o}\ell \xrightarrow{\langle \text{Read}, \ell, v, o \rangle} v$$

OE-WRITE
$$\mathcal{M}, \mathcal{V} \vdash \ell :=_o v \xrightarrow{\langle \text{Write}, \ell, v, o \rangle} \text{☣}$$

OE-CAS-FAIL
$$\frac{\begin{array}{l}\mathtt{rlx} \sqsubseteq o_f \\ \mathtt{rlx} \sqsubseteq o_r \\ \mathtt{rlx} \sqsubseteq o_w\end{array} \qquad (\forall \omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}).\, \exists v'.\, \omega \equiv v' \wedge \vdash v_1 =^? v') \qquad \vdash v_1 \neq v_r}{\mathcal{M}, \mathcal{V} \vdash \mathsf{CAS}(\ell, v_1, v_2, o_f, o_r, o_w) \xrightarrow{\langle \text{Read}, \ell, v_r, o_f \rangle} 0}$$

OE-CAS-SUC
$$\frac{\begin{array}{l}\mathtt{rlx} \sqsubseteq o_f \\ \mathtt{rlx} \sqsubseteq o_r \\ \mathtt{rlx} \sqsubseteq o_w\end{array} \qquad (\forall \omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}).\, \exists v'.\, \omega \equiv v' \wedge \vdash v_1 =^? v') \qquad \mathcal{M} \vdash v_1 = v_r}{\mathcal{M}, \mathcal{V} \vdash \mathsf{CAS}(\ell, v_1, v_2, o_f, o_r, o_w) \xrightarrow{\langle \text{Update}, \ell, v_r, v_2, o_r, o_w \rangle} 1}$$

OE-FENCE
$$\mathcal{M}, \mathcal{V} \vdash \mathtt{fence}_o \xrightarrow{\langle \text{Fence}, o \rangle} \text{☣}$$

OE-CASE
$$\mathcal{M}, \mathcal{V} \vdash \mathbf{case}\ i\ \mathbf{of}\ (\overline{e}) \to \overline{e}_i$$

OE-APP
$$\mathcal{M}, \mathcal{V} \vdash (\mathbf{rec}\ f(\overline{x}) := e)(\overline{v}) \to e[\mathbf{rec}\ f(\overline{x}) := e/f, \overline{v}/\overline{x}]$$

OE-FORK
$$\mathcal{M}, \mathcal{V} \vdash \mathbf{fork}\,\{\, e \,\} \to \text{☣}, e$$

Fig. 5. Expression semantics.

OM-READ-HELPER

$$\frac{\begin{array}{c} cur(\ell).\mathrm{w} \leq t \qquad R(\ell) \leq t \\ V = [\ell \leftarrow \{\mathrm{w} := t, \mathrm{aw} := \emptyset, \mathrm{nr} := \mathbf{if}\ o = \mathbf{na}\ \mathbf{then}\ \{r\}\ \mathbf{else}\ \emptyset, \mathrm{ar} := \mathbf{if}\ o \sqsubseteq \mathbf{rlx}\ \mathbf{then}\ \{r\}\ \mathbf{else}\ \emptyset\}] \\ cur' = \mathbf{if}\ \mathbf{acq} \sqsubseteq o\ \mathbf{then}\ cur \sqcup V \sqcup R\ \mathbf{else}\ cur \sqcup V \\ acq' = \mathbf{if}\ \mathbf{rlx} \sqsubseteq o\ \mathbf{then}\ acq \sqcup V \sqcup R\ \mathbf{else}\ acq \sqcup V \end{array}}{(rel, frel, cur, acq) \xrightarrow{\langle \mathrm{R}:o,\ell,t,R\rangle, r} (rel, frel, cur', acq')}$$

OM-WRITE-HELPER

$$\frac{\begin{array}{c} cur(\ell).\mathrm{w} < t \\ V = [\ell \leftarrow \{\mathrm{w} := t, \mathrm{aw} := \mathbf{if}\ \mathbf{rlx} \sqsubseteq o\ \mathbf{then}\ \{t\}\ \mathbf{else}\ \emptyset, \mathrm{nr} := \emptyset, \mathrm{ar} := \emptyset\}] \\ cur' = cur \sqcup V \qquad acq' = acq \sqcup V \\ V' = rel(\ell) \sqcup \mathbf{if}\ \mathbf{rel} \sqsubseteq o\ \mathbf{then}\ cur'\ \mathbf{else}\ V \qquad rel' = rel[\ell \leftarrow V'] \\ R_w = \mathbf{if}\ \mathbf{rlx} \sqsubseteq o\ \mathbf{then}\ V' \sqcup frel \sqcup R_r\ \mathbf{else}\ \bot \end{array}}{(rel, frel, cur, acq) \xrightarrow{\langle \mathrm{W}:o,\ell,t,R_r,R_w\rangle} (rel', frel, cur', acq')}$$

Fig. 6. View-helper relations.

Machine Step. $\boxed{\mathcal{M} \mid \mathcal{V} \xrightarrow{\varepsilon, t^?, m^*} \mathcal{M}' \mid \mathcal{V}'}$

OM-ALLOC

$$\frac{\begin{array}{c} \ell = (i, n') \qquad \{i\} \times \mathbb{N} \mathrel{\#} \mathrm{dom}(\mathcal{M}) \\ \mathcal{M}' = \mathcal{M}\,[\ell + m \leftarrow [t_m \leftarrow (\dagger, \bot)] \mid m \in [<n]] \\ \mathcal{V} \xrightarrow{\langle \mathrm{W{:}}\textbf{na}, \ell+0, t_0, \bot, \bot\rangle} \cdots \xrightarrow{\langle \mathrm{W{:}}\textbf{na}, \ell+m, t_m, \bot, \bot\rangle} \cdots \xrightarrow{\langle \mathrm{W{:}}\textbf{na}, \ell+(n-1), t_{(n-1)}, \bot, \bot\rangle} \mathcal{V}' \\ ms = [(t_m, \dagger, \bot) \mid m \in [<n]] \end{array}}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\langle \mathrm{Alloc}, \ell, n\rangle, \bot, ms} \mathcal{M}' \mid \mathcal{V}'}$$

OM-FREE

$$\frac{\begin{array}{c} \ell = (i, n') \qquad \mathrm{dom}(\mathcal{M}) \cap \{i\} \times \mathbb{N} = \{i\} \times ([\geq n', <n' + n]) \\ \forall m \in [<n], t \in \mathrm{dom}(\mathcal{M}(\ell + m)).\, t \leq \mathcal{V}.\mathrm{cur}(\ell + m).\mathrm{w} < t_m \wedge \mathcal{M}(\ell + m)(t).\mathrm{val} \neq \maltese \\ \forall m \in [<n].\, \mathrm{dom}(\mathcal{M}(\ell + m)) \neq \emptyset \\ \mathcal{M}' = \mathcal{M}\,[\ell + m \leftarrow [t_m \leftarrow (\maltese, \bot)] \mid m \in [<n]] \\ \mathcal{V} \xrightarrow{\langle \mathrm{W{:}}\textbf{na}, \ell+0, t_0, \bot, \bot\rangle} \cdots \xrightarrow{\langle \mathrm{W{:}}\textbf{na}, \ell+m, t_m, \bot, \bot\rangle} \cdots \xrightarrow{\langle \mathrm{W{:}}\textbf{na}, \ell+(n-1), t_{(n-1)}, \bot, \bot\rangle} \mathcal{V}' \\ ms = [(t_m, \dagger, \bot) \mid m \in [<n]] \end{array}}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\langle \mathrm{Dealloc}, \ell, n\rangle, \bot, ms} \mathcal{M}' \mid \mathcal{V}'}$$

OM-READ

$$\frac{\begin{array}{c} \ell \notin \mathrm{unalloc}(\mathcal{M}) \qquad \mathcal{M}(\ell)(t) = (\omega, R) \qquad \omega \equiv v \\ \mathcal{V} \xrightarrow{\langle \mathrm{R}{:}o, \ell, t, R\rangle, r} \mathcal{V}' \end{array}}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\langle \mathrm{Read}, \ell, v, o\rangle, r, []} \mathcal{M} \mid \mathcal{V}'}$$

OM-WRITE

$$\frac{\begin{array}{c} \ell \notin \mathrm{unalloc}(\mathcal{M}) \qquad t \notin \mathcal{M}(\ell) \\ \mathcal{M}' = \mathcal{M}\,[\ell \leftarrow \mathcal{M}(\ell)\,[t \leftarrow (v, R)]] \\ \mathcal{V} \xrightarrow{\langle \mathrm{W}{:}o, \ell, t, \bot, R\rangle} \mathcal{V}' \end{array}}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\langle \mathrm{Write}, \ell, v, o\rangle, \bot, [(t, v, R)]} \mathcal{M}' \mid \mathcal{V}'}$$

OM-UPDATE

$$\frac{\begin{array}{c} \ell \notin \mathrm{unalloc}(\mathcal{M}) \qquad \mathcal{M}(\ell)(t_r) = (v_r, R_r) \qquad t_w = t_r + 1 \qquad t_w \notin \mathcal{M}(\ell) \\ \mathcal{M}' = \mathcal{M}\,[\ell \leftarrow \mathcal{M}(\ell)\,[t_w \leftarrow (v_w, R_w)]] \\ \mathcal{V} \xrightarrow{\langle \mathrm{R}{:}o_r, \ell, t_r, R_r\rangle, r} \xrightarrow{\langle \mathrm{W}{:}o_w, \ell, t_w, R_r, R_w\rangle} \mathcal{V}' \end{array}}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\langle \mathrm{Update}, \ell, v_r, v_w, o_r, o_w\rangle, r, [(t_w, v_w, R_w)]} \mathcal{M}' \mid \mathcal{V}'}$$

OM-ACQ-FENCE

$$\mathcal{M} \mid \mathcal{V} \xrightarrow{\langle \mathrm{Fence}, \textbf{acq}\rangle, \bot, []} \mathcal{M} \mid (\mathcal{V}.\mathrm{rel}, \mathcal{V}.\mathrm{frel}, \mathcal{V}.\mathrm{acq}, \mathcal{V}.\mathrm{acq})$$

OM-REL-FENCE

$$\mathcal{M} \mid \mathcal{V} \xrightarrow{\langle \mathrm{Fence}, \texttt{rel}\rangle, \bot, []} \mathcal{M} \mid ([\ell \leftarrow \mathcal{V}.\mathrm{cur} \mid \ell \in \mathrm{dom}(\mathcal{V}.\mathrm{rel})], \mathcal{V}.\mathrm{cur}, \mathcal{V}.\mathrm{cur}, \mathcal{V}.\mathrm{acq})$$

Fig. 7. Machine semantics.

DRE Precondition. $\boxed{\mathcal{M}, \mathcal{N}, V \vdash \mathrm{RaceFree}(\varepsilon)}$

$$\textsc{DRF-read-na}\quad \frac{\forall t \in \mathrm{dom}(\mathcal{M}(\ell)).\, t \leq \mathit{cur}(\ell).\mathrm{w} \qquad \mathcal{N}(\ell).\mathrm{aw} \sqsubseteq \mathit{cur}(\ell).\mathrm{aw}}{\mathcal{M}, \mathcal{N}, (\mathit{rel}, \mathit{frel}, \mathit{cur}, \mathit{acq}) \vdash \mathrm{RaceFree}(\langle \mathrm{Read}, \ell, v, \mathbf{na}\rangle)}$$

$$\textsc{DRF-write-na}\quad \frac{\mathcal{N}(\ell).\mathrm{aw} \sqsubseteq \mathit{cur}(\ell).\mathrm{aw} \qquad \mathcal{N}(\ell).\mathrm{nr} \sqsubseteq \mathit{cur}(\ell).\mathrm{nr} \qquad \mathcal{N}(\ell).\mathrm{ar} \sqsubseteq \mathit{cur}(\ell).\mathrm{ar} \\ \forall t \in \mathrm{dom}(\mathcal{M}(\ell)).\, t \leq \mathit{cur}(\ell).\mathrm{w} < t_w}{\mathcal{M}, \mathcal{N}, (\mathit{rel}, \mathit{frel}, \mathit{cur}, \mathit{acq}) \vdash \mathrm{RaceFree}(\langle \mathrm{Write}, \ell, v, \mathbf{na}\rangle)}$$

$$\textsc{DRF-read-at}\quad \frac{\texttt{rlx} \sqsubseteq o \qquad \mathcal{N}(\ell).\mathrm{w} \leq \mathit{cur}(\ell).\mathrm{w}}{\mathcal{M}, \mathcal{N}, (\mathit{rel}, \mathit{frel}, \mathit{cur}, \mathit{acq}) \vdash \mathrm{RaceFree}(\langle \mathrm{Read}, \ell, v, o\rangle)}$$

$$\textsc{DRF-write-at}\quad \frac{\texttt{rlx} \sqsubseteq o \qquad \mathcal{N}(\ell).\mathrm{w} \leq \mathit{cur}(\ell).\mathrm{w} \qquad \mathcal{N}(\ell).\mathrm{nr} \sqsubseteq \mathit{cur}(\ell).\mathrm{nr}}{\mathcal{M}, \mathcal{N}, (\mathit{rel}, \mathit{frel}, \mathit{cur}, \mathit{acq}) \vdash \mathrm{RaceFree}(\langle \mathrm{Write}, \ell, v, o\rangle)}$$

$$\textsc{DRF-update}\quad \frac{\mathcal{M}, \mathcal{N}, V \vdash \mathrm{RaceFree}(\langle \mathrm{Read}, \ell, v_r, o_r\rangle) \qquad \mathcal{M}, \mathcal{N}, V \vdash \mathrm{RaceFree}(\langle \mathrm{Write}, \ell, v_w, o_w\rangle)}{\mathcal{M}, \mathcal{N}, V \vdash \mathrm{RaceFree}(\langle \mathrm{Update}, \ell, v_r, v_w, o_r, o_w\rangle)}$$

$$\textsc{DRF-alloc}\quad \mathcal{M}, \mathcal{N}, V \vdash \mathrm{RaceFree}(\langle \mathrm{Alloc}, \ell, n\rangle)$$

$$\textsc{DRF-dealloc}\quad \frac{\forall i \in [<n], t' \in \mathrm{dom}(\mathcal{M}(\ell + i)).\, t' \leq \mathit{cur}(\ell).\mathrm{w} \qquad \forall i \in [<n].\, \mathcal{N}(\ell + i).\mathrm{aw} \sqsubseteq \mathit{cur}(\ell + i).\mathrm{aw} \\ \forall i \in [<n].\, \mathcal{N}(\ell + i).\mathrm{nr} \sqsubseteq \mathit{cur}(\ell + i).\mathrm{nr} \qquad \forall i \in [<n].\, \mathcal{N}(\ell + i).\mathrm{ar} \sqsubseteq \mathit{cur}(\ell + i).\mathrm{ar}}{\mathcal{M}, \mathcal{N}, (\mathit{rel}, \mathit{frel}, \mathit{cur}, \mathit{acq}) \vdash \mathrm{RaceFree}(\langle \mathrm{Dealloc}, \ell, n\rangle)}$$

Fig. 8. Data-race-free (DRF) pre condition, detailing the exact requirements on the local and global race detector state for any particular memory event.

DRF Postcondition. $\boxed{\mathcal{N} \xRightarrow{\varepsilon, t^?, m^*} \mathcal{N}'}$

DRF-Post-read-na
$$\frac{r \notin \mathcal{N}(\ell).\mathsf{nr} \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\ \mathbf{with}\ \mathsf{nr} := \mathcal{N}(\ell).\mathsf{nr} \cup \{r\}\}]}{\mathcal{N} \xRightarrow{\langle \mathrm{Read}, \ell, v, \mathbf{na} \rangle, r, []} \mathcal{N}'}$$

DRF-Post-write-na
$$\frac{\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\ \mathbf{with}\ \mathsf{w} := m.\mathsf{ts}\}]}{\mathcal{N} \xRightarrow{\langle \mathrm{Write}, \ell, v, \mathbf{na} \rangle, \bot, [m]} \mathcal{N}'}$$

DRF-Post-read-at
$$\frac{\mathbf{rlx} \sqsubseteq o \qquad r \notin \mathcal{N}(\ell).\mathsf{ar} \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\ \mathbf{with}\ \mathsf{ar} := \mathcal{N}(\ell).\mathsf{ar} \cup \{r\}\}]}{\mathcal{N} \xRightarrow{\langle \mathrm{Read}, \ell, v, o \rangle, r, []} \mathcal{N}'}$$

DRF-Post-write-at
$$\frac{\mathbf{rlx} \sqsubseteq o \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\ \mathbf{with}\ \mathsf{aw} := \mathcal{N}(\ell).\mathsf{aw} \cup \{m.\mathsf{ts}\}\}]}{\mathcal{N} \xRightarrow{\langle \mathrm{Write}, \ell, v, o \rangle, \bot, [m]} \mathcal{N}'}$$

DRF-Post-update
$$\frac{\begin{array}{c} r \notin \mathcal{N}(\ell).\mathsf{ar} \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\ \mathbf{with}\ \mathsf{ar} := \mathcal{N}(\ell).\mathsf{ar} \cup \{r\}\}] \\ \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\ \mathbf{with}\ \mathsf{aw} := \mathcal{N}(\ell).\mathsf{aw} \cup \{m.\mathsf{ts}\}\}] \end{array}}{\mathcal{N} \xRightarrow{\langle \mathrm{Update}, \ell, v_r, v_w, o_r, o_w \rangle, r, [m]} \mathcal{N}'}$$

DRF-Post-alloc
$$\frac{\mathcal{N}' = \mathcal{N}[\ell + i \leftarrow \{\mathsf{w} := m_i.\mathsf{ts}, \mathsf{aw} := \emptyset, \mathsf{nr} := \emptyset, \mathsf{ar} := \emptyset\} \mid i \in [<n]]}{\mathcal{N} \xRightarrow{\langle \mathrm{Alloc}, \ell, n \rangle, \bot, [m_0 \ldots m_{n-1}]} \mathcal{N}'}$$

DRF-Post-dealloc
$$\frac{\mathcal{N}' = \mathcal{N}[\ell + i \leftarrow \{\mathcal{N}(\ell + i)\ \mathbf{with}\ \mathsf{w} := m_i.\mathsf{ts}\}) \mid i \in [<n]]}{\mathcal{N} \xRightarrow{\langle \mathrm{Dealloc}, \ell, n \rangle, \bot, [m_0 \ldots m_{n-1}]} \mathcal{N}'}$$

Fig. 9. Data-race-free (DRF) post condition, detailing the change to the global race detector state on a per-event basis.

Combined Step. $\boxed{\varsigma \mid (e, \mathcal{V}) \xrightarrow{\varepsilon^?, e_f^*} \varsigma' \mid (e', \mathcal{V}')}$

CombRed-pure

$$\frac{\mathcal{M}, \mathcal{V} \vdash e \to e', es}{(\mathcal{M}, \mathcal{N}) \mid (e, \mathcal{V}) \xrightarrow{\bot, es} (\mathcal{M}, \mathcal{N}) \mid (e', \mathcal{V})}$$

CombRed-event

$$\frac{\begin{array}{c}\forall \varepsilon, \mathcal{M}', \mathcal{V}', e', r', ms'.\, \mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon} e', [] \land \mathcal{M} \mid \mathcal{V} \xrightarrow{\varepsilon, r', ms'} \mathcal{M}'' \mid \mathcal{V}'' \implies \mathcal{M}, \mathcal{N}, \mathcal{V} \vdash \mathrm{RaceFree}(\varepsilon) \\ \mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon} e', [] \qquad \mathcal{M} \mid \mathcal{V} \xrightarrow{\varepsilon, r, ms} \mathcal{M}' \mid \mathcal{V}' \qquad \mathcal{N} \overset{\varepsilon, r, ms}{\rightsquigarrow} \mathcal{N}'\end{array}}{(\mathcal{M}, \mathcal{N}) \mid (e, \mathcal{V}) \xrightarrow{\varepsilon, []} (\mathcal{M}', \mathcal{N}') \mid (e', \mathcal{V}')}$$

Fig. 10. Combined machine and expression semantics

Threadpool Step. $\boxed{\varsigma \mid \mathcal{TS} \to \varsigma' \mid \mathcal{TS}'}$

$$\mathrm{ForkView}(\mathcal{V}) ::= (\emptyset, \emptyset, \mathcal{V}.\mathrm{cur}, \mathcal{V}.\mathrm{cur})$$

OT-step

$$\frac{\begin{array}{c}\mathcal{TS}(\pi) = (e, \mathcal{V}) \\ (\mathcal{M}, \mathcal{N}) \mid (e, \mathcal{V}) \xrightarrow{\varepsilon, [e_{f_0}, \ldots, e_{f_n}]} (\mathcal{M}', \mathcal{N}') \mid (e', \mathcal{V}') \qquad \{\rho_0 \ldots \rho_n\} \cap \mathrm{dom}(\mathcal{TS}) = \emptyset\end{array}}{(\mathcal{M}, \mathcal{N}) \mid \mathcal{TS} \to (\mathcal{M}', \mathcal{N}') \mid \mathcal{TS}\,[\pi \leftarrow (e', \mathcal{V}')] \left[\rho_i \leftarrow (e_{f_i}, \mathrm{ForkView}(\mathcal{V}')) \,\middle|\, i \in [<n]\right]}$$

Fig. 11. Threadpool semantics.

2 CORRESPONDENCE OF ORC11 TO RC11

The memory model of ORC11 is modeled after Lahav et al. [2017] (referred to as "RC11" from now on) without SC accesses and SC fences. It is worth noting that the memory model of ORC11 is more conservative and declares more programs racy than RC11. To prove this, we show that any program that is racy under RC11 is also considered racy by ORC11. We make this claim more precise below.

The race detector in ORC11 (and the one in the intermediate OGS machine) is stronger, *i.e.*, detects more races, than RC11. In particular, ORC11 does not permit reducing a **CAS** expression with order **acq** in the presence of an unsynchronized non-atomic read *even* when the **CAS** itself synchronizes with the non-atomic read. In contrast, the self-synchronizing nature of **CAS** leads to RC11 accepting this particular behavior as non-racy.

To simplify the proof, we allow RC11 to take expression reduction steps that are disallowed in ORC11. In particular, the declarative semantics in RC11 may compare arbitrary values with each other, whereas ORC11 will get stuck in some of these cases (see Fig. 5). A potential theorem to prove would then be that ORC11 detects any RC11 race or gets stuck for other reasons. Fortunately, the race detector in ORC11 already models races as being stuck and so the theorem statement simply becomes: *Any program that is racy under RC11 will get stuck under ORC11* (see Theorem 1).

We decompose the proof into 2 steps. First, we prove that any racy RC11 execution of the program can be replayed as a racy execution in the Operational Graph Semantics (OGS, §2.3). Second, we prove that the racy OGS execution can be replayed as a racy execution in ORC11 (§2.4). The OGS is designed to be an intermediate mixture of RC11 and ORC11.

Definition 1 (Extended Order) The set of *extended orders ExtOrder* is defined by

$$o \in \mathit{ExtOrder} := \mathit{Order} \uplus \{\mathtt{relacq}\}.$$

Note that $\mathtt{relacq} \sqsupseteq o$ for any (extended) order o. We define $o.w$ and $o.r$ s.t.

$$o.w, o.r := \begin{cases} \mathtt{rel}, \mathtt{acq} & \text{if } o = \mathtt{relacq} \\ \mathtt{rel}, \mathtt{rlx} & \text{if } o = \mathtt{rel} \\ \mathtt{rlx}, \mathtt{acq} & \text{if } o = \mathtt{acq} \\ \mathtt{rlx}, \mathtt{rlx} & \text{if } o = \mathtt{rlx} \\ \mathtt{na}, \mathtt{na} & \text{if } o = \mathtt{na} \end{cases}$$

Definition 2 (Labels) The set of *labels*, *Label*, is defined by the following (tagged) union of events:

$$\begin{aligned} \gamma \in \mathit{Label} := \quad & \{\mathsf{R}^o(\ell, v) \mid o \in \mathit{Order}, \ell \in \mathit{Loc}, v \in \mathit{Val}\} \\ & \cup \{\mathsf{W}^o(\ell, v) \mid o \in \mathit{Order}, \ell \in \mathit{Loc}, v \in \mathit{Val}\} \\ & \cup \left\{\mathsf{U}^o(\ell, v_r, v_w) \mid o \in \mathit{ExtOrder}, \ell \in \mathit{Loc}, v_r \in \operatorname{codom}(\vdash \cdot \overset{?}{=} \cdot), v_w \in \mathit{Val}\right\} \\ & \cup \{\mathsf{F}^o \mid o \in \{\mathtt{rel}, \mathtt{acq}\}\} \\ & \cup \{\mathsf{Fork}^\rho \mid \rho \in \mathit{Thread}\} \end{aligned}$$

We write $\gamma \sim \varepsilon$ when γ corresponds a memory event ε (mapping all labels except Fork to their corresponding counterparts in *MemEvent*).

2.1 Executions

An execution G is defined by:

(1) a finite set of events $\mathrm{E} \subseteq \mathbb{N}$. with events $\mathrm{E} \supseteq \mathrm{E}_0 := \left\{a_0^\ell \mid \ell \in \mathcal{L}\right\}$.

(2) a labelling function $\mathrm{lab} \in \mathrm{E} \to \mathit{Label}$, with projections typ, mod, loc, val_r, val_w where defined.
(3) a function tid assigning a thread identifier to every event in E. We write E^π to denote the events in E with $\mathrm{tid}(a) = \pi$.
(4) a strict partial order $\mathsf{sb} \subseteq \mathrm{E} \times \mathrm{E}$ which is total on E^π for every thread π, and which puts all events in E_0 before all other events.
(5) a binary relation $\mathtt{rf} \subseteq [\mathsf{WU}]; =_{\mathrm{loc}}; [\mathsf{RU}]$ such that
  (a) $\forall \langle a, b\rangle \in \mathtt{rf}.\ \mathrm{val}_w(a) = \mathrm{val}_r(b)$
  (b) $\forall b, \langle a_1, b\rangle \in \mathtt{rf}, \langle a_2, b\rangle \in \mathtt{rf}.\ a_1 = a_2$.
(6) a family of strict total orders $\{\mathtt{mo}_\ell\}_{\ell \in \mathcal{L}}$ and $\mathtt{mo} := \biguplus_{\ell \in \mathcal{L}} \mathtt{mo}_\ell$.

2.1.1 *Consistent Executions.*

Definition 3 (Completeness) An execution G is called *complete* if and only if for every $a \in \mathsf{R}$ we have $\mathrm{val}_r(a) = ☣ \vee \exists b \in \mathsf{W}_{\mathrm{loc}(a)}.\ \langle b, a\rangle \in \mathtt{rf}$. Note that this condition is weaker than in RC11 as it allows reads from uninitialized locations (signified by the value ☣).

Definition 4 (Auxiliary relations)

$$\mathtt{rb} := \mathtt{rf}^{-1}; \mathtt{mo} \qquad \textit{reads-before}$$
$$\mathtt{eco} := (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^+ \qquad \textit{extended-coherence}$$
$$\mathtt{rs} := [\mathsf{WU}]; \mathsf{sb}|^?_{=_{\mathrm{loc}}}; [(\mathsf{WU})^{\sqsupseteq \mathtt{rlx}}]; (\mathtt{rf}; [U])^* \qquad \textit{release-sequence}$$
$$\mathtt{asw} := [\mathsf{Fork}_\rho]; (\mathsf{sb}|^?_{\mathrm{tid}=\rho}); [\mathrm{E}^\rho] \qquad \textit{additionally-synchronized-with}$$
$$\mathtt{sw} := \mathtt{asw} \cup \left([\mathrm{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathsf{F}]; \mathsf{sb})^?; \mathtt{rs}; \mathtt{rf}; [(\mathsf{RU})^{\sqsupseteq \mathtt{rlx}}]; (\mathsf{sb}; [\mathsf{F}])^?; [\mathrm{E}^{\sqsupseteq \mathtt{acq}}]\right) \qquad \textit{synchronized-with}$$
$$\mathtt{hb} := (\mathsf{sb} \cup \mathtt{sw})^+ \qquad \textit{happens-before}$$

For intuition of these definitions, please confer the RC11 paper [Lahav et al. 2017].

Definition 5 (Consistency) An execution is called RC11-*consistent* (simply "consistent" from now on) if it is complete **and**

- $\mathtt{hb}; \mathtt{eco}^?$ is irreflexive (COHERENCE)
- $\mathsf{sb} \cup \mathtt{rf}$ is acyclic (NO-THIN-AIR)

This definition does not include RC11's SC axiom.

2.2 Declarative Semantics

The following definitions are taken from iGPS [Kaiser et al. 2017] and, if necessary, adapted to our setting. Below we define threadpool reduction that generates *traces*. Note that we circumvent checks (such as those for legal comparisons) in the expression reduction by providing existentially quantified memory $\mathcal{M}$ and local view $\mathcal{V}$. RC11 originally does not involve such checks.

$$\textsc{Trace-Red-Silent}\ \frac{\mathcal{M}, \mathcal{V} \vdash \mathcal{TS}(\pi) \to e, []}{\mathcal{TS} \overset{\epsilon}{\Rightarrow}^{\pi} \mathcal{TS}[\pi \mapsto e]} \qquad \textsc{Trace-Red-Mem}\ \frac{\gamma \sim \varepsilon \qquad \mathcal{M}, \mathcal{V} \vdash \mathcal{TS}(\pi) \xrightarrow{\varepsilon} e, []}{\mathcal{TS} \overset{\gamma}{\Rightarrow}^{\pi} \mathcal{TS}[\pi \mapsto e]}$$

$$\textsc{Trace-Red-Fork}\ \frac{\mathcal{V}(\pi) = (e, V) \qquad \mathcal{M}, \mathcal{V} \vdash \mathcal{TS}(\pi) \to e', e_f \qquad \rho \notin \mathrm{dom}(\mathcal{TS})}{\mathcal{TS} \overset{\mathsf{Fork}_\rho}{\Longrightarrow}^{\pi} \mathcal{TS}[\pi \mapsto e'] \uplus [\rho \mapsto e_f]}$$

We write $\mathcal{TS} \Rightarrow^{\pi} \mathcal{TS}'$ if $\mathcal{TS} \xRightarrow{x}^{\pi} \mathcal{TS}'$ for some transition label x; $\mathcal{TS} \xRightarrow{x} \mathcal{TS}'$ if $\mathcal{TS} \xRightarrow{x}^{\pi} \mathcal{TS}'$ for some thread identifier π; and $\mathcal{TS} \Rightarrow \mathcal{TS}'$ if $\mathcal{TS} \xRightarrow{x}^{\pi} \mathcal{TS}'$ for some transition label x and thread identifier π. A threadpool is called *final* if $\mathcal{TS}(\pi) \in \mathit{Val}$ for every $\pi \in \mathrm{dom}(\mathcal{TS})$.

Definition 6 (Traces) A *trace* is a sequence of pairs $\langle \gamma_1, \pi_1 \rangle, \dots, \langle \gamma_n, \pi_n \rangle$. We say that $tr = \langle \gamma_1, \pi_1 \rangle, \dots, \langle \gamma_n, \pi_n \rangle$ is a trace of an expression e if $[0 \mapsto e] \xRightarrow{\epsilon}^{*} \xRightarrow{\gamma_1}^{\pi_1} \xRightarrow{\epsilon}^{*} \dots \xRightarrow{\epsilon}^{*} \xRightarrow{\gamma_n}^{\pi_n} \xRightarrow{\epsilon}^{*} \mathcal{TS}$ for some thread π and threadpool $\mathcal{TS}$. When $\mathcal{TS}$ is final, we call tr a *full* trace.

Definition 7 A trace $tr = \langle \gamma_1, \pi_1 \rangle, \dots, \langle \gamma_n, \pi_n \rangle$ induces partial order on indices $\mathsf{sb}(tr)$, called *sequenced-before*, and a relation on indices $\mathsf{asw}(tr)$, called *additional-synchronized-with*. They are defined by:

$$\frac{i < j \qquad \pi_i = \pi_j}{\langle i, j \rangle \in \mathsf{sb}(tr)} \qquad\qquad \frac{\langle i, j \rangle \in \mathsf{sb}(tr) \qquad \langle j, k \rangle \in \mathsf{sb}(tr)}{\langle i, k \rangle \in \mathsf{sb}(tr)}$$

$$\frac{i < j \qquad \gamma_i = \mathtt{Fork}_{\pi_j}}{\langle i, j \rangle \in \mathsf{asw}(tr)}$$

Lemma 1 Let tr be a trace of an expression e. Then

- Any prefix of tr is also a trace of e.
- Any permutation tr' of tr with $\mathsf{sb}(tr') = \mathsf{sb}(tr)$ and $\mathsf{asw}(tr') = \mathsf{asw}(tr)$ is a trace of e.

Definition 8 An execution G *follows* a trace $tr = \langle \gamma_1, \pi_1 \rangle, \dots, \langle \gamma_n, \pi_n \rangle$ if:

- $\mathrm{E} = \{a_1, \dots, a_n\}$ such that $\mathrm{lab}(a_k) = \gamma_k$ and $\mathrm{tid}(a_k) = \pi_k$ for every $1 \le k < n$
- $\mathsf{sb} = \{\langle a_i, a_j \rangle \mid \langle i, j \rangle \in \mathsf{sb}(tr)\}$.

We call G an execution of expression e if G follows some trace of e.

Definition 9 (Conflict) Two events a and b are called *conflicting* in an execution G if $a, b \in \mathrm{E}$, $\{\mathrm{typ}(a), \mathrm{typ}(b)\} \cap \{\mathtt{W}, \mathtt{U}\} \neq \emptyset$, $a \neq b$, and $\mathrm{loc}(a) = \mathrm{loc}(b)$.

Definition 10 (Races) A pair $\langle a, b \rangle$ is called a *race* in G if a and b are conflicting events in G, and $\langle a, b \rangle \notin \mathsf{hb} \cup \mathsf{hb}^{-1}$. An execution G is called *racy* if there is some race $\langle a, b \rangle$ in G with $\mathbf{na} \in \{\mathrm{mod}(a), \mathrm{mod}(b)\}$.

Definition 11 (Bugginess) An execution G is buggy if it is racy. An expression e is buggy if some consistent execution of e is buggy.

2.3 Operational Graph Semantics (OGS)

We now introduce an operationalized account of RC11 (*OGS*, short for Operational Graph Semantics), in which we build up executions step by step. This serves as an important stepping stone towards a our correspondence proof with ORC11.

Definition 12 (Execution Extension: Memory Accesses) We write $G' \in \mathrm{Add}(G, \pi, \rho, \gamma)$ if there exists an event a s.t.

- $G'.\mathrm{E} = G.\mathrm{E} \uplus \{a\}$, $G'.\mathrm{tid} = G.\mathrm{tid} \cup \{a \mapsto \rho\}$, $G'.\mathrm{lab} = G.\mathrm{lab} \cup \{a \mapsto \gamma\}$
- if $\rho \neq \pi$ then $\rho \notin \mathrm{codom}(G.\mathrm{tid})$
- $G'.\mathsf{sb} = (G.\mathsf{sb} \uplus (G.\mathrm{E}^{\rho} \times \{a\}))^{+}$
- $G'.\mathsf{rf} \supseteq G.\mathsf{rf}$
- $G'.\mathsf{mo} \supseteq G.\mathsf{mo}$ and if $\gamma = \mathtt{W}^{\mathbf{na}}(\_, \_)$ then a is mo-maximal in G'

Definition 13 (Race Predicate) We define a predicate $\mathrm{Race}(G, \pi)$ which holds for all memory events from π that would cause a data race in execution G. Note that this race detector models exactly the rules implement in ORC11. Thus, it detects more races than RC11 *but* only in (potentially non-buggy) executions following buggy expressions.

$$\textsc{Race-I}\quad \frac{o \sqsupseteq \texttt{rlx} \qquad \gamma \in (\mathsf{RU})^{o}_{\ell} \qquad \exists a \in \mathsf{W}^{\mathsf{na}}_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b \rangle \notin \mathsf{hb}^{*}}{\gamma \in \mathrm{Race}(G, \pi)}$$

$$\textsc{Race-II}\quad \frac{\gamma = \mathsf{R}^{\mathsf{na}}(\ell, \_) \qquad \exists a \in (\mathsf{WU})_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b \rangle \notin \mathsf{hb}^{*}}{\gamma \in \mathrm{Race}(G, \pi)}$$

$$\textsc{Race-III}\quad \frac{\gamma = \mathsf{W}^{\mathsf{na}}(\ell, \_) \qquad \exists a \in (\mathsf{RWU})_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b \rangle \notin \mathsf{hb}^{*}}{\gamma \in \mathrm{Race}(G, \pi)}$$

$$\textsc{Race-IV}\quad \frac{o \sqsupseteq \texttt{rlx} \qquad \gamma = \mathsf{W}^{o}_{\ell} \qquad \exists a \in (\mathsf{RW})^{\mathsf{na}}_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b \rangle \notin \mathsf{hb}^{*}}{\gamma \in \mathrm{Race}(G, \pi)}$$

$$\textsc{Race-V}\quad \frac{\gamma = \mathsf{U}^{o}_{\ell} \qquad \exists a \in (\mathsf{RW})^{\mathsf{na}}_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b \rangle \notin \mathsf{hb}^{*}}{\gamma \in \mathrm{Race}(G, \pi)}$$

Definition 14 (OGS Reductions)

$$\textsc{OGS-Memory-Step}\quad \frac{\begin{array}{c}\gamma \in \{\mathsf{R}^{o}(\ell, v), \mathsf{W}^{o}(\ell, v), \mathsf{F}^{o}\} \\ \gamma \notin \mathrm{Race}(G, \pi) \\ G' \in \mathrm{Add}(G, \pi, \pi, \gamma) \\ G' \text{ is consistent}\end{array}}{G \xrightarrow{\gamma}{}^{\pi} G'}$$

$$\textsc{OGS-Fork}\quad \frac{\begin{array}{c}G' \in \mathrm{Add}(G, \pi, \rho, \mathsf{Fork}_{\rho}) \\ G' \text{ is consistent}\end{array}}{G \xrightarrow{\mathsf{Fork}_{\rho}}{}^{\pi} G'}$$

$$\textsc{OGS-Race}\quad \frac{\gamma \in \mathrm{Race}(G, \pi)}{G \xrightarrow{\gamma}{}^{\pi} \bot_{\mathrm{race}}}$$

We define combined machine and expression semantics for OGS. We once again allow expression reductions to proceed independent of the current state, thus capturing more behaviors than those allowed by ORC11.

$$\textsc{OGS-CombRed-pure}\quad \frac{\mathcal{M}, \mathcal{V} \vdash e \rightarrow e', []}{G \mid e \xrightarrow{\bot, []}{}^{\pi} G \mid e'}$$

$$\textsc{OGS-CombRed-event}\quad \frac{\begin{array}{c}\forall \varepsilon, e'.\, \mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon} e', [] \implies \neg(G \xrightarrow{\varepsilon}{}^{\pi} \bot_{\mathrm{race}}) \\ \mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon} e', [] \qquad G \xrightarrow{\varepsilon}{}^{\pi} G'\end{array}}{G \mid e \xrightarrow{\varepsilon, []}{}^{\pi} G' \mid e'}$$

$$\textsc{OGS-CombRed-fork}\quad \frac{\mathcal{M}, \mathcal{V} \vdash e \rightarrow e', [e_f] \qquad G \xrightarrow{\mathsf{Fork}_{\rho}}{}^{\pi} G'}{G \mid e \xrightarrow{\bot, [e_f]}{}^{\pi} G' \mid e'}$$

$$\textsc{OGS-CombRed-race}\quad \frac{\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon} e', [] \qquad G \xrightarrow{\varepsilon}{}^{\pi} \bot_{\mathrm{race}}}{G \mid e \xrightarrow{\varepsilon, []}{}^{\pi} \bot_{\mathrm{race}}}$$

OGS-OT-STEP

$$\frac{\mathcal{TS}(\pi) = e \qquad G \mid e \xrightarrow{\varepsilon,[e_{f_0},\dots,e_{f_n}]}{}^{\pi} G' \mid e' \qquad \{\rho_0 \dots \rho_n\} \cap \mathrm{dom}(\mathcal{TS}) = \emptyset}{G \mid \mathcal{TS} \to G' \mid \mathcal{TS}\,[\pi \leftarrow e']\,\big[\rho_i \leftarrow e_{f_i} \,\big|\, i \in [<n]\big]}$$

OGS-OT-RACE

$$\frac{\mathcal{TS}(\pi) = e \qquad G \mid e \xrightarrow{\varepsilon,[]}{}^{\pi} \bot_{\text{race}}}{G \mid \mathcal{TS} \to \bot_{\text{race}}}$$

We define G_0 to be an execution in which all locations are allocated with an initial value of ☨.

Lemma 2 (Non-buggy Reductions) Let G_1 be a non-buggy and consistent execution such that $G_1 \xrightarrow{\gamma_1}{}^{\pi_1} \dots G_n \xrightarrow{\gamma_n}{}^{\pi_n} G_{n+1}$. Then $G_1, \dots, G_n, G_{n+1}$ are all non-buggy and consistent executions.

Lemma 3 (Inclusion of Behaviors (I)) Let G be a non-buggy, consistent execution of expression e. Then there exists a trace $tr = \langle\gamma_1, \pi_1\rangle \dots \langle\gamma_n, \pi_n\rangle$ of e such that $G_0 \xrightarrow{\gamma_1}{}^{\pi_1} \dots \xrightarrow{\gamma_n}{}^{\pi_n} G \lor \exists j \le n.\, G_0 \xrightarrow{\gamma_1}{}^{\pi_1} \dots \xrightarrow{\gamma_j}{}^{\pi_j} \bot_{\text{race}}$.

Proof. As G is consistent, we have that $\mathsf{sb} \cup \mathsf{rf}$ is acyclic. Let $a_1, \dots, a_n$ be an enumeration of E that respects $(\mathsf{sb} \cup \mathsf{rf})^+$. For every $1 \le i \le n$, let $\pi_i := \mathrm{tid}(a_i)$, $\gamma_i = \mathrm{lab}(a_i)$, and $tr = \langle\gamma_1, \pi_1\rangle \dots \langle\gamma_n, \pi_n\rangle$. Adding events $a_1, \dots, a_n$ one-by-one we can thus establish either $G_0 \xrightarrow{\gamma_1}{}^{\pi_1} \dots \xrightarrow{\varepsilon_n}{}^{\pi_n} G$, or—if in any step $j \le n$ the race predicate detects a spurious race—$G_0 \xrightarrow{\gamma_1}{}^{\pi_1} \dots \xrightarrow{\gamma_j}{}^{\pi_j} \bot_{\text{race}}$. ◻

Lemma 4 (Inclusion of Behaviors (II)) Let e be a buggy expression. Then $G_0 \mid [0 \mapsto e] \to^* \bot_{\text{race}}$.

Proof. We have that e is buggy and, thus, a consistent execution G which is buggy. Let $a_1, \dots, a_n$ be an enumeration of E that respects $\mathsf{sb} \cup \mathsf{rf}$. Let k be the minimal index such that $G \cap \{a_1, \dots, a_k\}$ is buggy, *i.e.*, racy.

We thus have that $G \cap \{a_1, \dots, a_k\}$ is racy. Let $j < k$ be the minimal index such that $\mathrm{loc}(a_k) = \mathrm{loc}(a_j)$, $\langle a_k, a_j\rangle \notin \mathsf{hb} \cup \mathsf{hb}^{-1}$, and one of the following holds:

- $a_k \in (\mathsf{WU})^{\sqsupseteq \mathtt{rlx}} \land a_j \in \mathsf{R}^{\mathtt{na}} \lor a_k \in \mathsf{W}^{\mathtt{na}}$
- $a_j \in (\mathsf{WU})^{\sqsupseteq \mathtt{rlx}} \land a_k \in \mathsf{R}^{\mathtt{na}} \lor a_j \in \mathsf{W}^{\mathtt{na}}$

Note that we have $\mathrm{tid}(a_k) \ne \mathrm{tid}(a_j)$, as otherwise these events would be related by $G.\mathsf{sb}$, and, thus $G.\mathsf{hb}$.

(1) $a_k \in \mathsf{W}^{\mathtt{na}}$. We define $B := \big\{a \in \mathrm{E} \,\big|\, \langle a, a_j\rangle \in G.\mathsf{hb} \lor \langle a, a_k\rangle \in G.\mathsf{hb}^*\big\}$ and $G' := G \cap B$. Note that G' is non-empty, consistent, and does not contain a_j (which is minimal in causing the race), thus not buggy. Also note that G' is an execution of e. By Lemma 3, we have that $G_0 \xrightarrow{\gamma_1}{}^{\pi_1} \dots \xrightarrow{\gamma_n}{}^{\pi_n} G' \lor \exists j \le n.\, G_0 \xrightarrow{\gamma_1}{}^{\pi_1} \dots \xrightarrow{\gamma_j}{}^{\pi_j} \bot_{\text{race}}$ for some trace $\langle\gamma_1, \pi_1\rangle, \dots, \langle\gamma_n, \pi_n\rangle$ of e. In the latter case our proof is done. Otherwise we have $\langle a_k, a_j\rangle \notin G'.\mathsf{hb}$ and we show that $G' \xrightarrow{\mathrm{lab}(a_j)}{}^{\mathrm{tid}(a_j)} \bot_{\text{race}}$.
By Definition 13 (using whichever case corresponds to $\mathrm{lab}(a_j)$), it suffices to show that $\langle a_k, b\rangle \notin G'.\mathsf{hb}^*$ for all $b \in \mathrm{E}^{\mathrm{tid}(a_j)}$. By way of contradiction, assume $b \in \mathrm{E}^{\mathrm{tid}(a_j)}$ and $\langle a_k, b\rangle \in G'.\mathsf{hb}^*$. By definition of G', we have $\langle b, a_j\rangle \in G.\mathsf{hb} \lor \langle b, a_k\rangle \in G.\mathsf{hb}^*$.
   (a) $\langle b, a_j\rangle \in G.\mathsf{hb}$. By transitivity, we have $\langle a_k, a_j\rangle \in G.\mathsf{hb}$, which contradicts our assumption.
   (b) $\langle b, a_k\rangle \in G.\mathsf{hb}^*$. From $\mathrm{tid}(a_k) \ne \mathrm{tid}(a_j)$ we have that $b \ne a_k$. Thus, $\langle b, a_k\rangle \in G.\mathsf{hb}$. By transitivity, we have $\langle b, b\rangle \in G.\mathsf{hb}$, which contradicts hb's irreflexivity.

As $\langle\gamma_1, \pi_1\rangle, \ldots, \langle\gamma_n, \pi_n\rangle, \langle \mathrm{lab}(a_j), \mathrm{tid}(a_j)\rangle$ is a valid trace for e, we have that $G_0 \mid [0 \mapsto e] \longrightarrow^* \bot_{\mathrm{race}}$.

(2) $a_j \in \mathsf{W}^{\mathbf{na}}$ is symmetric to the case above.

(3) $a_k \in (\mathsf{WU})^{\sqsupseteq\mathbf{rlx}} \land a_j \in \mathsf{R}^{\mathbf{na}}$. We define $B := \left\{a \in \mathsf{E} \,\middle|\, \langle a, a_j\rangle \in G.\mathsf{hb} \lor \langle a, a_k\rangle \in G.\mathsf{hb}^*\right\}$ and $G' := G \cap B$. Note that G' is consistent and not buggy. By Lemma 3, we have that $G_0 \xrightarrow{\gamma_1}_{\pi_1} \ldots \xrightarrow{\gamma_n}_{\pi_n} G' \lor \exists j \leq n.\, G_0 \xrightarrow{\gamma_1}_{\pi_1} \ldots \xrightarrow{\gamma_j}_{\pi_j} \bot_{\mathrm{race}}$ for some trace $\langle\gamma_1, \pi_1\rangle, \ldots, \langle\gamma_n, \pi_n\rangle$ of e. In the latter case our proof is done. Otherwise we have $\langle a_k, a_j\rangle \notin G'.\mathsf{hb}$ and we show that $G' \xrightarrow{\mathrm{lab}(a_j)}_{\mathrm{tid}(a_j)} \bot_{\mathrm{race}}$.
By Definition 13, it suffices to show that $\langle a_k, b\rangle \notin G'.\mathsf{hb}^*$ for all $b \in \mathsf{E}^{\mathrm{tid}(a_j)}$. By way of contradiction, assume $b \in \mathsf{E}^{\mathrm{tid}(a_j)}$ and $\langle a_k, b\rangle \in G'.\mathsf{hb}^*$. By definition of G', we have $\langle b, a_j\rangle \in G.\mathsf{hb} \lor \langle b, a_k\rangle \in G.\mathsf{hb}^*$.
 (a) $\langle b, a_j\rangle \in G.\mathsf{hb}$. By transitivity, we have $\langle a_k, a_j\rangle \in G.\mathsf{hb}$, which contradicts our assumption.
 (b) $\langle b, a_k\rangle \in G.\mathsf{hb}^*$. From $\mathrm{tid}(a_k) \neq \mathrm{tid}(a_j)$ we have that $b \neq a_k$. Thus, $\langle b, a_k\rangle \in G.\mathsf{hb}$. By transitivity, we have $\langle b, b\rangle \in G'.\mathsf{hb}$, which contradicts hb's irreflexivity.

As $\langle\gamma_1, \pi_1\rangle, \ldots, \langle\gamma_n, \pi_n\rangle, \langle \mathrm{lab}(a_j), \mathrm{tid}(a_j)\rangle$ is a valid trace for e, we have that $G_0 \mid [0 \mapsto e] \longrightarrow^* \bot_{\mathrm{race}}$.

(4) $a_j \in \mathsf{W}^{\sqsupseteq\mathbf{rlx}} \land a_k \in \mathsf{R}^{\mathbf{na}}$. This case is symmetric to the one above.

□

2.4 OGS to ORC11

Definition 15 We define auxiliary relations $\{\mathtt{auxrel}\}_\ell$, $\mathtt{auxfrel}$, and $\mathtt{auxacq}$. Note that by $(\mathsf{RU})^{\mathbf{rlx}}$ we mean read and update events with the **rlx** *read* mode.

$$\begin{aligned}
\mathtt{auxrel}_\ell &:= \mathsf{hb}; [(\mathsf{WU})_\ell^{\sqsupseteq\mathbf{rel}}] \\
\mathtt{auxfrel} &:= \mathsf{hb}; [\mathsf{F}^{\mathbf{rel}}] \\
\mathtt{auxacq} &:= \mathsf{hb}; ([\mathsf{E}^{\mathbf{rel}}]; ([\mathsf{F}]; \mathsf{sb})^?; \mathsf{rs}; \mathsf{rf}; [(\mathsf{RU})^{\mathbf{rlx}}])^?
\end{aligned}$$

Definition 16 (Event Injection) Let G be an execution and $a \in \mathsf{E}$. We define an injection into natural numbers, written $\mathrm{Inj}(G, a)$, as follows.

$$\mathrm{Inj}(G, a) := \mathrm{prime}(\mathrm{tid}(a))^{|\{b \mid \langle b, a\rangle \in \mathsf{sb}\}|}$$

where $\mathrm{prime}(n)$ is the n^{th} prime number. Note that Inj is injective and that performing a machine step $G \longrightarrow G'$ implies $\mathrm{Inj}(G', a) = \mathrm{Inj}(G, a)$ for any $a \in G$.

We write $\mathrm{Inj}(G, X)$ for $\{\mathrm{Inj}(G, a) \mid a \in X\}$. We also write $a \in Y$ for $\mathrm{Inj}(G, a) \in Y$ if Y is defined as $\mathrm{Inj}(G, X)$ for some X. (Note that this implies $a \in X$.)

Definition 17 (Timestamp Assignment) A *timestamp assignment* for an execution graph G is a function $ts : \mathsf{WU} \longrightarrow \mathit{Time}$, that satisfies $ts(a) < ts(b)$ whenever $\langle a, b\rangle \in G.\mathsf{mo}$.

Definition 18 (Message Reconstruction) Given a timestamp assignment ts for G and an event $a \in \mathsf{WU}$, we define the (X, R)-restricted event map, denoted $map^{G,ts}(a, X, R)$, the X-restricted write map, denoted $map_w^{G,ts}(a, X)$, the X-restricted read map, denoted $map_r^{G,ts}(a, X)$, the proto write view, denoted $view_w(a, G, ts)$, the proto read view, denoted $view_r(a, G, ts)$, and the message induced by a in G according to ts, denoted $msg(a, G, ts)$, as follows.

$$map^{G,ts}(a, X, R) = \{b \mid b \in X, \langle b, a\rangle \in R\}$$

$$map_w^{G,ts}(a, X) = \begin{cases} map^{G,ts}(a, X, \mathtt{hb}^*) & \text{if } \mathrm{mod}(a) = \mathbf{rel} \\ map^{G,ts}(a, X, (\mathtt{auxfrel} \cup \mathtt{auxrel}_\ell)^?; \mathtt{hb}^*) & \text{if } \mathrm{mod}(a) = \mathbf{rlx} \\ \bot & \text{otherwise} \end{cases}$$

$$map_r^{G,ts}(a, X) = map(a, G, ts, X, \mathtt{auxacq})$$

$$\begin{aligned} view_w(a, G, ts) = \lambda\ell.\ \{\ & \mathsf{w} := \max\left\{ts(b) \,\middle|\, b \in map_w^{G,ts}(a, (\mathsf{WU})_\ell)\right\}, \\ & \mathsf{aw} := \left\{ts(b) \,\middle|\, b \in map_w^{G,ts}(a, (\mathsf{WU})_\ell^{\sqsupseteq\mathbf{rlx}})\right\}, \\ & \mathsf{nr} := \mathrm{Inj}(G, map_w^{G,ts}(a, \mathsf{R}_\ell^{\mathbf{na}})), \\ & \mathsf{ar} := \mathrm{Inj}(G, map_w^{G,ts}(a, (\mathsf{RU})_\ell^{\sqsupseteq\mathbf{rlx}})) \\ \}\ & \end{aligned}$$

$$\begin{aligned} view_r(a, G, ts) = \lambda\ell.\ \{\ & \mathsf{w} := \max\left\{ts(b) \,\middle|\, b \in map_r^{G,ts}(a, (\mathsf{WU})_\ell)\right\}, \\ & \mathsf{aw} := \left\{ts(b) \,\middle|\, b \in map_r^{G,ts}(a, (\mathsf{WU})^{\sqsupseteq\mathbf{rlx}})\right\}, \\ & \mathsf{nr} := \mathrm{Inj}(G, map_r^{G,ts}(a, \mathsf{R}_\ell^{\mathbf{na}})), \\ & \mathsf{ar} := \mathrm{Inj}(G, map_r^{G,ts}(a, (\mathsf{RU})_\ell^{\sqsupseteq\mathbf{rlx}})) \\ \}\ & \end{aligned}$$

$$msg(a, G, ts) = \begin{cases} (\mathrm{val}_w(a), view_w(a, G, ts)) & \text{if } a = \mathsf{W} \\ (\mathrm{val}_w(a), view_w(a, G, ts) \sqcup view_r(a, G, ts)) & \text{if } a = \mathsf{U} \end{cases}$$

In these definitions, we take $\bot$ to be the maximum of an empty set.

Definition 19 Let G be an execution and ts be a timestamp assignment for G. We define the physical state $(\mathcal{M}_G^{ts}, \mathcal{N}_G^{ts}, \mathcal{V}_G^{ts})$ as follows.

- The memory is defined by $\mathcal{M}_G^{ts} := \lambda\ell.\,\lambda t.\ \begin{cases} msg(a, G, ts) & \text{if } \exists a.\, t = ts(a) \wedge \ell = \mathrm{loc}(a) \\ \bot & \text{otherwise} \end{cases}$

- The thread views Υ^{ts}_G are defined by

$$
\begin{aligned}
\mathsf{ThEvs}(X, S, R) &:= \{a \in S \mid \exists b \in X.\, \langle a, b\rangle \in R^*\} \\
t_{\max}(X, S, R) &:= \max\{ts(a) \mid a \in \mathsf{ThEvs}(X, S, R)\} \\
V(X, R) &:= \lambda \ell.\ \{\ \mathsf{w} := t_{\max}(X, (\mathsf{WU})_\ell, R), \\
&\qquad \mathsf{aw} := \left\{ts(a) \mid a \in \mathsf{ThEvs}(X, (\mathsf{WU})^{\sqsupseteq \mathtt{rlx}}_\ell, R)\right\}, \\
&\qquad \mathsf{nr} := \mathsf{Inj}(G, \mathsf{ThEvs}(X, \mathsf{R}^{\mathbf{na}}_\ell, R)), \\
&\qquad \mathsf{ar} := \mathsf{Inj}(G, \mathsf{ThEvs}(X, (\mathsf{RU})^{\sqsupseteq \mathtt{rlx}}_\ell, R)) \\
&\quad \} \\
\Upsilon^{ts}_G(\pi) &:= \{\mathsf{rel} := \lambda \ell'.\, V(\mathrm{E}^\pi, \mathtt{auxrel}_{\ell'}), \\
&\qquad \mathsf{frel} := V(\mathrm{E}^\pi, \mathtt{auxfrel}), \\
&\qquad \mathsf{cur} := V(\mathrm{E}^\pi, \mathsf{hb}), \\
&\qquad \mathsf{acq} := V(\mathrm{E}^\pi, \mathtt{auxacq}) \\
&\quad \}
\end{aligned}
$$

- The global race detector state $\mathcal{N}^{ts}_G$ is defined by

$$
\begin{aligned}
\mathcal{N}^{ts}_G := \lambda \ell.\ \{ & \\
& \mathsf{w} := t_{\max}(\mathrm{E}, \mathsf{W}^{\mathbf{na}}_\ell, (=)), \\
& \mathsf{aw} := \left\{ts(a) \mid a \in (\mathsf{WU})^{\sqsupseteq \mathtt{rlx}}_\ell\right\}, \\
& \mathsf{nr} := \mathsf{Inj}(G, \mathsf{R}^{\mathbf{na}}_\ell), \\
& \mathsf{ar} := \mathsf{Inj}(G, (\mathsf{RU})^{\sqsupseteq \mathtt{rlx}}_\ell) \\
\} &
\end{aligned}
$$

In these definitions, we take $\perp$ to be the maximum of an empty set.

We say that G *relates* to a physical state $(\mathcal{M}, \mathcal{N}, \Upsilon)$, denoted $G \sim_{ts} (\mathcal{M}, \mathcal{N}, \Upsilon)$, if and only if $(\mathcal{M}^{ts}_G, \mathcal{N}^{ts}_G, \Upsilon^{ts}_G,) = (\mathcal{M}, \mathcal{N}, \Upsilon)$.

Definition 20 In the following, we lift ORC11's machine semantics to thread views such that

$$(\mathcal{M}, \mathcal{N}, \Upsilon) \xrightarrow{\varepsilon}{}^{\pi} (\mathcal{M}, \mathcal{N}', \Upsilon') := (\mathcal{M}, \mathcal{N}) \mid \Upsilon(\pi) \xrightarrow{\varepsilon} (\mathcal{M}', \mathcal{N}') \mid \mathcal{V}' \wedge \Upsilon' = \Upsilon[\pi \leftarrow \mathcal{V}']$$

Lemma 5 Suppose $G \xrightarrow{\gamma}{}^{\pi} G'$, $\gamma \sim \varepsilon$ and let ts' be a timestamp assignment for G'. Then $ts = ts'|_{G.\mathtt{W}}$ is a timestamp assignment for G and $(\mathcal{M}^{ts}_G, \mathcal{N}^{ts}_G, \Upsilon^{ts}_G) \xrightarrow{\varepsilon}{}^{\pi} (\mathcal{M}^{ts'}_{G'}, \mathcal{N}^{ts'}_{G'}, \Upsilon^{ts'}_{G'})$.

In the remainder of this section, when ts is uniquely identifiable, we simply write $G \sim (\mathcal{M}, \mathcal{N}, \Upsilon)$ to mean $G \sim_{ts|_{G.\mathtt{W}}} (\mathcal{M}, \mathcal{N}, \Upsilon)$.

Lemma 6 (Inclusion of Behaviors (I)) Suppose $G \xrightarrow{\gamma_1}{}^{\pi_1} \ \ldots\ \xrightarrow{\gamma_n}{}^{\pi_n} G_n$, and ts is a timestamp assignment for G_n, and $G \sim (\mathcal{M}_1, \mathcal{N}_1, \Upsilon_1)$. Then either

- there exist $\varepsilon_1 \ldots \varepsilon_n$, $G_2 \sim (\mathcal{M}_2, \mathcal{N}_2, \Upsilon_2) \ldots G_n \sim (\mathcal{M}_n, \mathcal{N}_n, \Upsilon_n)$ such that $(\mathcal{M}_1, \mathcal{N}_1, \Upsilon_1) \xrightarrow{\varepsilon_1}{}^{\pi_1} \ldots\ldots \xrightarrow{\varepsilon_n}{}^{\pi_n} (\mathcal{M}_n, \mathcal{N}_n, \Upsilon_n)$.
- or there exist $j < n$, $\varepsilon_1 \ldots \varepsilon_{j+1}$, $G_2 \sim (\mathcal{M}_2, \mathcal{N}_2, \Upsilon_2) \ldots G_j \sim (\mathcal{M}_j, \mathcal{N}_j, \mathcal{V}_j)$ such that $(\mathcal{M}_1, \mathcal{N}_1, \Upsilon_1) \xrightarrow{\varepsilon_1}{}^{\pi_1} \ldots\ldots \xrightarrow{\varepsilon_j}{}^{\pi_j} (\mathcal{M}_j, \mathcal{N}_j, \Upsilon_j) \wedge \neg\left((\mathcal{M}_j, \mathcal{N}_j, \Upsilon_j) \xrightarrow{\varepsilon_{j+1}}{}^{\pi_{j+1}} \_\right)$.

Lemma 7 (Inclusion of Behaviors (II)) Let G be a consistent execution that is not buggy, ts a timestamp assignment for G, $G \sim (\mathcal{M}, \mathcal{N}, \Upsilon)$, $\gamma \sim \varepsilon$, and $G \xrightarrow{\gamma}^{\pi} \bot_{\text{race}}$. Then $\neg\left((\mathcal{M}, \mathcal{N}, \Upsilon) \xrightarrow{\varepsilon}^{\pi} \_\right)$.

Proof. We consider the following cases.

(1) $\gamma \in \{\mathsf{R}^o(\ell, \_), \mathsf{U}^o(\ell, \_, \_)\} \wedge o \sqsupseteq \mathbf{rlx} \wedge \exists a \in \mathsf{W}^{\mathbf{na}}_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b\rangle \notin \mathsf{hb}^*$. (Race-I)
We show $\neg(\mathcal{M}, \mathcal{N}, \Upsilon(\pi) \vdash \mathrm{RaceFree}(\langle \mathrm{Read}, \ell, \_, o\rangle))$. It suffices to show that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < \mathcal{N}(\ell).\mathrm{w}$. Let $a_m \in \mathsf{W}^{\mathbf{na}}_{\ell}$ be the mo-maximal non-atomic write event on ℓ, which implies $ts(a_m) \geq ts(a)$. Then $\mathcal{N}(\ell).\mathrm{w} = ts(a_m)$. It thus suffices to show that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < ts(a_m)$. By way of contradiction, assume that $\Upsilon(\pi).\mathrm{cur}(\ell) \geq ts(a_m)$. Then, there exists $c \in (\mathsf{WU})_{\ell}$ and $b \in \mathrm{E}^{\pi}$ s.t. $\langle c, b\rangle \in \mathsf{hb}^* \wedge ts(c) \geq ts(a_m)$. From $\langle a, a_m\rangle \in \mathsf{mo}^*$, COHERENCE, and G being non-racy we have that $\langle a, a_m\rangle \in \mathsf{hb}^*$. As G is non-racy, we also have $c = a_m \vee \langle a_m, c\rangle \in \mathsf{hb} \vee \langle c, a_m\rangle \in \mathsf{hb}$.

(a) $c = a_m$. We have $\langle a_m, b\rangle \in \mathsf{hb}^*$. Then, by transitivity, we have $\langle a, b\rangle \in \mathsf{hb}^*$ which contradicts our initial assumption.

(b) $\langle a_m, c\rangle \in \mathsf{hb}$. By transitivity, we have $\langle a_m, b\rangle \in \mathsf{hb}^*$, and, thus, $\langle a, b\rangle \in \mathsf{hb}^*$. This contradicts our initial assumption.

(c) $\langle c, a_m\rangle \in \mathsf{hb} \wedge c \neq a$. By COHERENCE, we have $\langle a_m, c\rangle \notin \mathsf{mo}$ and, thus, $\langle c, a_m\rangle \in \mathsf{mo}$. This contradicts $ts(c) \geq ts(a_m)$.

(2) $\gamma = \mathsf{R}^{\mathbf{na}}(\ell, \_) \wedge \exists a \in (\mathsf{WU})_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b\rangle \notin \mathsf{hb}^*$. (Race-II)
We show $\neg(\mathcal{M}, \mathcal{N}, \Upsilon(\pi) \vdash \mathrm{RaceFree}(\langle \mathrm{Read}, \ell, \_, \mathbf{na}\rangle))$. It suffices to show that either there exists t', $(v', V') = \mathcal{M}(\ell)(t')$ s.t. $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < t'$ *or* $\mathcal{N}(\ell).\mathrm{aw} \not\sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{aw}$.
We consider two cases:

(a) $\mathrm{mod}(a) = \mathbf{na}$.
We choose $t' = ts(a)$ and $(v', V') := msg(a, G, ts)$. It suffices to show $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < ts(a)$. There exists $c \in (\mathsf{WU})_{\ell}$ and $b \in \mathrm{E}^{\pi}$ s.t. $\langle c, b\rangle \in \mathsf{hb}^*$ and $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} = ts(c)$. We show $ts(c) < ts(a)$. By way of contradiction, assume $ts(c) \geq ts(a)$. We have $c \neq a$ as otherwise $\langle a, b\rangle \in \mathsf{hb}^*$, contradicting our assumption. Thus we have $ts(c) > ts(a)$ and $\langle a, c\rangle \in \mathsf{mo}$. From G being non-racy, COHERENCE, and $\langle a, c\rangle \in \mathsf{mo}$ we have that $\langle a, c\rangle \in \mathsf{hb}$. By transitivity, $\langle a, b\rangle \in \mathsf{hb}^*$, which contradicts our assumption.

(b) $\mathrm{mod}(a) = \mathbf{rlx}$. We show $\mathcal{N}(\ell).\mathrm{aw} \not\sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{aw}$. By way of contradiction, assume that $\mathcal{N}(\ell).\mathrm{aw} \sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{aw}$. We have $a \in \mathcal{N}(\ell).\mathrm{aw}$ and, thus, $a \in \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{aw}$. Hence, there exists $b' \in \mathrm{E}^{\pi}$ s.t. $\langle a, b'\rangle \in \mathsf{hb}^*$, which contradicts our assumption.

(3) $\gamma = \mathsf{W}^{\mathbf{na}}(\ell, v) \wedge \exists a \in (\mathsf{RWU})_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b\rangle \notin \mathsf{hb}^*$. (Race-III)
We show that $\neg(\mathcal{M}, \mathcal{N}, \Upsilon(\pi) \vdash \mathrm{RaceFree}(\langle \mathrm{Write}, \ell, \_, \mathbf{na}\rangle))$.
We consider the following cases.

(a) $a \in \mathsf{W}^{\mathbf{na}}_{\ell}$. There exists $c \in (\mathsf{WU})_{\ell}$ and $b \in \mathrm{E}^{\pi}$ s.t. $\langle c, b\rangle \in \mathsf{hb}^* \wedge \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} = ts(c)$. We also have $a \neq c$ as that would imply $\langle a, b\rangle \in \mathsf{hb}^*$, contradicting our assumption. We show that there exists t', $(v', V') = \mathcal{M}(\ell)(t')$ s.t. $ts(c) < t'$. We choose $t' := ts(a)$ and $(v', V') := msg(a, G, ts)$.
It suffices to show $ts(c) < ts(a)$. As G is non-racy, we have $\langle a, c\rangle \in \mathsf{hb} \vee \langle c, a\rangle \in \mathsf{hb}$. The former implies, by transitivity, that $\langle a, b\rangle \in \mathsf{hb}^*$, which would contradict our assumption. Thus, $\langle c, a\rangle \in \mathsf{hb}$. As $a \neq b$ we derive $\langle c, a\rangle \in \mathsf{mo}$ from COHERENCE and, thus, $ts(c) < ts(a)$.

(b) $a \in (\mathsf{WU})^{\sqsupseteq \mathbf{rlx}}_{\ell}$. We show that $\mathcal{N}(\ell).\mathrm{aw} \not\sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{aw}$. By way of contradiction, assume that $\mathcal{N}(\ell).\mathrm{aw} \sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{aw}$. We have $a \in \mathcal{N}(\ell).\mathrm{aw}$ and, thus, $a \in \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{aw}$. Hence, there exists $b' \in \mathrm{E}^{\pi}$ s.t. $\langle a, b'\rangle \in \mathsf{hb}^*$, which contradicts our assumption.

(c) $a \in \mathsf{R}_{\ell}$. We show that $\mathcal{N}(\ell).\mathrm{nr} \not\sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr} \vee \mathcal{N}(\ell).\mathrm{ar} \not\sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{ar}$. We have $\mathrm{Inj}(G, a) \in \mathcal{N}(\ell).\mathrm{nr} \vee \mathrm{Inj}(G, a) \in \mathcal{N}(\ell).\mathrm{ar}$. By way of contradiction, assume that $\mathcal{N}(\ell).\mathrm{nr} \sqsubseteq$

$\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr} \wedge \mathcal{N}(\ell).\mathrm{ar} \sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{ar}$. Then $\mathrm{Inj}(G, a) \in \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr} \cup \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{ar}$. Hence, there exists $b' \in \mathrm{E}^{\pi}$ s.t. $\langle a, b'\rangle \in \mathsf{hb}$. This contradicts our assumption.

(4) $\gamma = \mathtt{W}^{o}(\ell, \_) \wedge o \sqsupseteq \mathbf{rlx} \wedge \exists a \in (\mathtt{RW})^{\mathsf{na}}_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b\rangle \notin \mathsf{hb}^{*}$. (Race-IV)
We show $\neg(\mathcal{M}, \mathcal{N}, \Upsilon(\pi) \vdash \mathrm{RaceFree}(\langle \mathrm{Write}, \ell, \_, o\rangle))$. We consider $a \in \mathtt{R}^{\mathsf{na}}$ and $a \in \mathtt{W}^{\mathsf{na}}$ separately.
  (a) $a \in \mathtt{R}^{\mathsf{na}}$. It suffices to show that $\mathcal{N}(\ell).\mathrm{nr} \not\sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr}$.
  By way of contradiction, assume that $\mathcal{N}(\ell).\mathrm{nr} \sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr}$. We have $\mathrm{Inj}(G, a) \in \mathcal{N}(\ell).\mathrm{nr}$ and, thus, $\mathrm{Inj}(G, a) \in \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr}$. This implies that there exists $b' \in \mathrm{E}^{\pi}$ s.t. $\langle a, b'\rangle \in \mathsf{hb}^{*}$, which contradicts our assumption.
  (b) $a \in \mathtt{W}^{\mathsf{na}}$. It suffices to show that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < \mathcal{N}(\ell).\mathrm{w}$.
  Let $a_m \in \mathtt{W}^{\mathsf{na}}_{\ell}$ be the mo-maximal non-atomic write event on ℓ, which implies $ts(a_m) \geq ts(a)$. Then $\mathcal{N}(\ell).\mathrm{w} = ts(a_m)$. It thus suffices to show that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < ts(a_m)$. By way of contradiction, assume that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} \geq ts(a_m)$. Then, there exists $c \in (\mathtt{WU})_{\ell}$ and $b \in \mathrm{E}^{\pi}$ s.t. $\langle c, b\rangle \in \mathsf{hb}^{*} \wedge ts(c) \geq ts(a_m)$. From $\langle a, a_m\rangle \in \mathsf{mo}^{*}$, COHERENCE, and G being non-racy we have that $\langle a, a_m\rangle \in \mathsf{hb}^{*}$. As G is non-racy, we also have $c = a_m \vee \langle a_m, c\rangle \in \mathsf{hb} \vee \langle c, a_m\rangle \in \mathsf{hb}$.
    (i) $c = a_m$. We have $\langle a_m, b\rangle \in \mathsf{hb}^{*}$. Then, by transitivity, we have $\langle a, b\rangle \in \mathsf{hb}^{*}$ which contradicts our initial assumption.
    (ii) $\langle a_m, c\rangle \in \mathsf{hb}$. By transitivity, we have $\langle a_m, b\rangle \in \mathsf{hb}^{*}$, and, thus, $\langle a, b\rangle \in \mathsf{hb}^{*}$. This contradicts our initial assumption.
    (iii) $\langle c, a_m\rangle \in \mathsf{hb} \wedge c \neq a$. By COHERENCE, we have $\langle a_m, c\rangle \notin \mathsf{mo}$ and, thus, $\langle c, a_m\rangle \in \mathsf{mo}$. This contradicts $ts(c) \geq ts(a_m)$.

(5) $\gamma = \mathtt{U}^{o}(\ell, \_, \_) \wedge \exists a \in (\mathtt{RW})^{\mathsf{na}}_{\ell}.\, \forall b \in \mathrm{E}^{\pi}.\, \langle a, b\rangle \notin \mathsf{hb}^{*}$. (Race-V)
We show that performing the "write" part of the update event leads to a race in ORC11, *i.e.*, $\neg(\mathcal{M}, \mathcal{N}, \Upsilon(\pi) \vdash \mathrm{RaceFree}(\langle \mathrm{Write}, \ell, \_, o.w\rangle))$. We consider $a \in \mathtt{R}^{\mathsf{na}}$ and $a \in \mathtt{W}^{\mathsf{na}}$ separately.
  (a) $a \in \mathtt{R}^{\mathsf{na}}$. We show that $\mathcal{N}(\ell).\mathrm{nr} \not\sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr}$.
  By way of contradiction, assume that $\mathcal{N}(\ell).\mathrm{nr} \sqsubseteq \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr}$. We have $\mathrm{Inj}(G, a) \in \mathcal{N}(\ell).\mathrm{nr}$ and, thus, $\mathrm{Inj}(G, a) \in \Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{nr}$. This implies that there exists $b' \in \mathrm{E}^{\pi}$ s.t. $\langle a, b'\rangle \in \mathsf{hb}^{*}$, which contradicts our assumption.
  (b) $a \in \mathtt{W}^{\mathsf{na}}$. It suffices to show that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < \mathcal{N}(\ell).\mathrm{w}$.
  Let $a_m \in \mathtt{W}^{\mathsf{na}}_{\ell}$ be the mo-maximal non-atomic write event on ℓ (which implies $ts(a_m) \geq ts(a)$). Then $\mathcal{N}(\ell).\mathrm{w} = ts(a_m)$. It thus suffices to show that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} < ts(a_m)$. By way of contradiction, assume that $\Upsilon(\pi).\mathrm{cur}(\ell).\mathrm{w} \geq ts(a_m)$. Then, there exists $c \in (\mathtt{WU})_{\ell}$ and $b \in \mathrm{E}^{\pi}$ s.t. $\langle c, b\rangle \in \mathsf{hb}^{*} \wedge ts(c) \geq ts(a_m)$. From $\langle a, a_m\rangle \in \mathsf{mo}^{*}$, COHERENCE, and G being non-racy we have that $\langle a, a_m\rangle \in \mathsf{hb}^{*}$. As G is non-racy, we also have $c = a_m \vee \langle a_m, c\rangle \in \mathsf{hb} \vee \langle c, a_m\rangle \in \mathsf{hb}$.
    (i) $c = a_m$. We have $\langle a_m, b\rangle \in \mathsf{hb}^{*}$. Then, by transitivity, we have $\langle a, b\rangle \in \mathsf{hb}^{*}$ which contradicts our initial assumption.
    (ii) $\langle a_m, c\rangle \in \mathsf{hb}$. By transitivity, we have $\langle a_m, b\rangle \in \mathsf{hb}^{*}$, and, thus, $\langle a, b\rangle \in \mathsf{hb}^{*}$. This contradicts our initial assumption.
    (iii) $\langle c, a_m\rangle \in \mathsf{hb} \wedge c \neq a$. By COHERENCE, we have $\langle a_m, c\rangle \notin \mathsf{mo}$ and, thus, $\langle c, a_m\rangle \in \mathsf{mo}$. This contradicts $ts(c) \geq ts(a_m)$.

□

Lemma 8 Suppose $G \xrightarrow{\gamma_1}{}^{\pi_1} \ldots G_n \xrightarrow{\gamma_n}{}^{\pi_n} \perp_{\mathrm{race}}$, ts a timestamp assignment for G_n, and $G \sim (\mathcal{M}_1, \mathcal{N}_1, \Upsilon_1)$. Then there exist $0 \leq j \leq n$, $\varepsilon_1 \ldots \varepsilon_{j+1}$, $G_2 \sim (\mathcal{M}_2, \mathcal{N}_2, \Upsilon_2) \ldots G_j \sim (\mathcal{M}_j, \mathcal{N}_j, \Upsilon_j)$ such that $(\mathcal{M}_1, \mathcal{N}_1, \Upsilon_1) \xrightarrow{\varepsilon_1}{}^{\pi_1} \ldots\ldots \xrightarrow{\varepsilon_j}{}^{\pi_j} (\mathcal{M}_j, \mathcal{N}_j, \Upsilon_j) \wedge \neg\left((\mathcal{M}_j, \mathcal{N}_j, \Upsilon_j) \xrightarrow{\varepsilon_{j+1}}{}^{\pi_{j+1}} \_\right)$.

Proof. Follows from Lemma 6 and Lemma 7. To invoke Lemma 7, we need Lemma 2 to know that G_j is a non-buggy and consistent execution. □

Definition 21 (Initial State) We define the initial physical state $\mathcal{M}_0$, global race detector state $\mathcal{N}_0$ as well as an initial thread view $\mathcal{V}_0$ as follows.

$$\begin{aligned}
\mathcal{M}_0 &:= \lambda\ell.\,\lambda t.\,\begin{cases}(\dagger, \bot) & \text{if } t = 0\\ \bot & \text{otherwise}\end{cases}\\
V_{\mathrm{aux}} &:= \lambda\ell.\,\{\, \mathsf{w} := 0, \mathsf{aw} := \emptyset, \mathsf{nr} := \emptyset, \mathsf{ar} := \emptyset, \,\}\\
\mathcal{N}_0 &:= V_{\mathrm{aux}}\\
\mathcal{V}_0 &:= \{\mathsf{rel} := \lambda\ell.\,\bot, \mathsf{frel} := \bot, \mathsf{cur} := V_{\mathrm{aux}}, \mathsf{acq} := V_{\mathrm{aux}}, \}
\end{aligned}$$

Intuitively, the initial state only contains allocation events for all locations.

Theorem 1 (ORC11: Racy Programs Get Stuck) Suppose e is buggy. Then e can get stuck in ORC11, *i.e.*, $(\mathcal{M}_0, \mathcal{N}_0) \mid [0 \mapsto (e, \mathcal{V}_0)] \rightarrow^* (\mathcal{M}', \mathcal{N}') \mid \mathcal{TS}'$ such that $\neg((\mathcal{M}', \mathcal{N}') \mid \_)$.

Proof. Follows from Lemma 4 and Lemma 8.

From Lemma 4, we have a trace $G_0 \mid [0 \mapsto e] \rightarrow \ldots G_n \mid \mathcal{TS}_n \rightarrow \bot_{\mathrm{race}}$ for some G_n and $\mathcal{TS}_n$. This, in turn, give us a trace $G_0 \xrightarrow{\gamma_1}{}_{\pi_1} \ldots G_n \xrightarrow{\gamma_n}{}_{\pi_n} \bot_{\mathrm{race}}$. We then can construct the timestamp assignment ts from G_n by following $G_n.\mathsf{mo}_\ell$ for each location ℓ.

Since G_0 only contains allocation events, it is trivially the case that ts is a timestamp assignment for G_0 and $G_0 \sim (\mathcal{M}_0, \mathcal{N}_0, [0 \mapsto \mathcal{V}_0])$. We can then invoke Lemma 8 and get $(\mathcal{M}_0, \mathcal{N}_0, [0 \mapsto \mathcal{V}_0]) \xrightarrow{\varepsilon_0}{}_{\pi_0} \ldots\ldots \xrightarrow{\varepsilon_j}{}_{\pi_j} (\mathcal{M}_j, \mathcal{N}_j, \Upsilon_j) \land \neg\left((\mathcal{M}_j, \mathcal{N}_j, \Upsilon_j) \xrightarrow{\varepsilon_{j+1}}{}_{\pi_{j+1}} \_\right)$. From this we can reconstruct the stuck trace in ORC11. □

3 LIFETIME LOGIC FOR VIEWS

This section gives a full account of the lifetime logic in iRC11. Fortunately, almost all proof rules are sound even after adapting the original lifetime logic from SC to RMM. The only change in the proof rules is in LftL-at-acc, the access rule for atomic borrows, which gives access to the borrowed resource only under the view-join modality. This is to account for the lack of implicit synchronization under RMM.

Other borrows have received modifications to their model by means of synchronized ghost state (in addition to synchronized ghost state used for lifetime tokens) to account for synchronization that always exists but needs to be witnessed explicitly under RMM. Despite these changes, the borrows enjoy the same proof rules as in SC.

To motivate the necessity of synchronized ghost state in the encoding of lifetime tokens, Section 4 presents a counterexample to models of the lifetime logic that use unsynchronized ghost state.

3.1 Proof Rules

Splitting ownership in time. The lifetime logic adds a built-in notion of *lifetimes*, and the notion of "owning P borrowed for lifetime κ", written $\&^{\kappa}_{\mathbf{full}}\, P$.

The rule LftL-begin is used to create a new lifetime. At this point, we obtain the token $[\kappa]_1$ which asserts that *we own the lifetime* κ: We know that the lifetime is still running, and we can end it any time by applying the view shift we got. Now, it turns out that we may want multiple parties to be able to witness that κ is ongoing, so we need to be able to split this assertion: $[\kappa]_q$ denotes ownership of the fraction q of κ. Lifetimes can be *intersected* using the $\sqcap$ operator.

We also obtain an update to end the new lifetime again. This makes use of the "update that takes a step", defined as follows:

$$P \mathrel{\Rrightarrow\!\!*}^{\mathcal{E}_2}_{\mathcal{E}_1} Q := P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\!\mid\!\!\Rrightarrow^{\mathcal{E}_2} \triangleright\, {}^{\mathcal{E}_2}\!\!\mid\!\!\Rrightarrow^{\mathcal{E}_1} Q$$

The core operation of the lifetime logic is *borrowing* an assertion P at a given lifetime. Using LftL-borrow, P is split into ownership of P during the lifetime κ (the full borrow), and ownership when κ died (a view shift that lets us "inherit" P from κ). In some sense, we are *splitting ownership along the time axis*: The justification for the separating conjunction is the fact that a lifetime is never both ongoing and has already ended at the same time. Thus, the two parts that we split P into can be treated as disjoint resources: They govern the same part of the (logical and physical) state, but they do so at different points in time.

When a lifetime ends, full borrows at that lifetime are not worth anything any more, a fact that is witnessed by LftL-bor-fake.

Borrowed assertions can still be split and merged, as shown by LftL-bor-sep. To get access to a borrowed assertion, we use LftL-bor-acc-strong. The rule is quite a mouthful, so it is worth looking at the following simpler (derived) version:

$$\langle \&^{\kappa}_{\mathbf{full}}\, P * [\kappa]_q \Longleftrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}}} \tag{1}$$

This lets us *open* full borrows ($\&^{\kappa}_{\mathbf{full}}\, P$) if we can prove that the lifetime is still ongoing, which we do by presenting any fraction of the lifetime token. We obtain $\triangleright P$, but lose access to that token for as long as the full borrow is open, which ensures that we do not end the lifetime while the full borrow is open. Once we re-established $\triangleright P$, we can *close* the full borrow again the get our token back.

The full rule LftL-bor-acc-strong actually lets us close not just with $\triangleright P$, but with any $\triangleright Q$ if we can show that Q entails P through a view shift. Furthermore, that view shift is only actually tun when the lifetime ends, which is witnessed by providing the appropriate token ($[\dagger\kappa]$).

Fig. 12. Lifetime logic assertions and proof rules

Notation	Meaning	Timeless	Persistent
$[\kappa]_q$	Fraction q of lifetime token for κ: Witnessing that the lifetime is still ongoing	Yes	No
$[\dagger\kappa]$	Witness confirming that the lifetime κ is dead (*i.e.*, it has ended)	Yes	Yes
$\&^{\kappa}_{\mathrm{full}}\, P$	Ownership of the *full borrow* of P for κ	No	No
$\&^{\kappa}_{i}\, P$	There is an *indexed borrow* named i of P for κ	No	Yes
$[\mathrm{Bor} : i]$	Ownership of the indexed borrow i	Yes	No
$\&^{\kappa/0}_{\mathrm{at}}\, P$	Internal atomic persistent borrow of P for κ	No	Yes

Lifetimes. Lifetimes κ form a cancellable PCM with intersection as the operation ($\sqcap$) and unit ε.

$$\kappa \sqsubseteq \kappa' := \Box \forall q.\, \langle [\kappa]_q \Leftrightarrow q'.\, [\kappa']_{q'} \rangle_{\mathcal{N}_{\mathrm{lft}}}$$

Lifetime creation and end.

LftL-begin
$$\mathsf{True} \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists \kappa.\, [\kappa]_1 * \Box([\kappa]_1 \Rrightarrow\!\!\!*^{\mathcal{N}_{\mathrm{lft}}}_{\emptyset} [\dagger\kappa])$$

LftL-tok-fract
$$[\kappa]_{q+q'} \Leftrightarrow [\kappa]_q * [\kappa]_{q'}$$

LftL-tok-fract-obj
$$[\kappa]_{q+q'} \Rightarrow [\kappa]_q * \langle obj \rangle\, [\kappa]_{q'}$$

LftL-tok-comp
$$[\kappa \sqcap \kappa']_q \Leftrightarrow [\kappa]_q * [\kappa']_q$$

LftL-tok-unit
$$\mathsf{True} \Rightarrow [\varepsilon]_q$$

LftL-not-own-end
$$[\kappa]_q * [\dagger\kappa] \Rightarrow \mathsf{False}$$

LftL-end-comp
$$[\dagger\kappa \sqcap \kappa'] \Leftrightarrow [\dagger\kappa] \vee [\dagger\kappa']$$

LftL-end-unit
$$[\dagger\varepsilon] \Rightarrow \mathsf{False}$$

Creating full borrows and using them.

LftL-borrow
$$\triangleright P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathrm{full}}\, P * ([\dagger\kappa] \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathrm{lft}}} \triangleright P)$$

LftL-bor-sep
$$\&^{\kappa}_{\mathrm{full}}(P * Q) \Lleftarrow\!\!\Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathrm{full}}\, P * \&^{\kappa}_{\mathrm{full}}\, Q$$

LftL-bor-fake
$$\langle subj \rangle\, [\dagger\kappa] \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathrm{full}}\, P$$

LftL-bor-acc-strong
$$\&^{\kappa}_{\mathrm{full}}\, P * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright P * \Big(\forall Q.\, \triangleright(\triangleright Q * \langle subj \rangle\, [\dagger\kappa'] \Rrightarrow\!\!\!*_{\emptyset} \triangleright P) * \triangleright Q \Rrightarrow\!\!\!*_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathrm{full}}\, Q * [\kappa]_q\Big)$$

LftL-bor-acc-atomic-strong
$$\&^{\kappa}_{\mathrm{full}}\, P \;{}^{\mathcal{N}_{\mathrm{lft}}}\!\!\Rrightarrow^{\emptyset} \Big(\exists P' \kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright(\lceil P' \rceil \wedge P) * \Big(\forall Q.\, \triangleright(\triangleright Q * \langle subj \rangle\, [\dagger\kappa'] \Rrightarrow\!\!\!*_{\emptyset} \triangleright P) * \triangleright(\lceil P' \rceil \wedge Q)\; {}^{\emptyset}\!\!\Rrightarrow\!\!\!*^{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathrm{full}}$$
$$\Big(\exists \kappa'.\, \kappa \sqsubseteq \kappa' * \langle subj \rangle\, [\dagger\kappa'] * {}^{\emptyset}\!\!\Rrightarrow^{\mathcal{N}_{\mathrm{lft}}} \mathsf{True}\Big)$$

Finally, the rule LftL-bor-acc-atomic-strong provides a way to access a full borrow *without* having a proof that the lifetime is still ongoing.

A closer look at lifetimes. Before we go on talking about the lifetime logic rules, we have to become more concrete about what a *lifetime* κ is. Lifetimes κ form a partial commutative monoid with unit ε. We will also refer to the composition operation ($\sqcap$) as *intersection* of lifetimes. Moreover, the PCM has to be *cancellable*, which means that the composition function is injective.

Fig. 13. Lifetime logic assertions and proof rules, continued

Indexed borrows.

LftL-bor-idx
$$\&^{\kappa}_{\mathsf{full}}\, P \Leftrightarrow \exists i.\ \&^{\kappa}_{i}\, P * [\mathrm{Bor} : i]$$

LftL-idx-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa}_{i}\, P \Rightarrow \&^{\kappa'}_{i}\, P}$$

LftL-idx-acc
$$\&^{\kappa}_{i}\, P(V_{\mathrm{tok}}) * [\mathrm{Bor} : i](V_{\mathrm{bor}}) * [\kappa]_q(V_{\mathrm{tok}}) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists V.\ V \sqsubseteq V_{\mathrm{tok}} \sqcup V_{\mathrm{bor}} * \triangleright P(V) *$$
$$\left(\forall V'_{\mathrm{tok}}.\ V_{\mathrm{tok}} \sqsubseteq V'_{\mathrm{tok}} * \triangleright P(V'_{\mathrm{tok}} \sqcup V) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} [\mathrm{Bor} : i](V'_{\mathrm{tok}} \sqcup V) * [\kappa]_q(V'_{\mathrm{tok}})\right)$$

LftL-idx-bor-unnest
$$\&^{\kappa}_{i}\, P * \&^{\kappa'}_{\mathsf{full}}([\mathrm{Bor} : i]) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa \sqcap \kappa'}_{\mathsf{full}}\, P$$

LftL-idx-bor-iff
$$\frac{\triangleright \square (P \Leftrightarrow Q)}{\&^{\kappa}_{i}\, P \Rightarrow \&^{\kappa}_{i}\, Q}$$

Internal persistent atomic borrows.

LftL-bor-in-at
$$\&^{\kappa}_{\mathsf{full}}\, P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa/0}_{\mathsf{at}}\, P$$

LftL-in-at-acc
$$\&^{\kappa/0}_{\mathsf{at}}\, P \vdash \langle [\kappa]_q \Lleftarrow\!\!\Rrightarrow V_b.\ \triangleright \lfloor P \rfloor_{\sqcup V_b} \rangle^{\emptyset}_{\mathcal{N}_{\mathrm{lft}}}$$

LftL-in-at-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa/0}_{\mathsf{at}}\, P \Rightarrow \&^{\kappa'/0}_{\mathsf{at}}\, P}$$

LftL-in-at-iff
$$\frac{\triangleright \square (P \Leftrightarrow Q)}{\&^{\kappa/0}_{\mathsf{at}}\, P \Rightarrow \&^{\kappa/0}_{\mathsf{at}}\, Q}$$

Furthermore, we define the following inclusion relation on lifetimes:

$$\kappa \sqsubseteq \kappa' := \square \left(\forall q.\ \langle [\kappa]_q \Lleftarrow\!\!\Rrightarrow q'.\ [\kappa']_{q'} \rangle_{\mathcal{N}_{\mathrm{lft}}}\right)$$

This says that κ is dynamically shorter than κ' if, given any fraction the token for κ, we can produce some fraction of the token for κ'. It is easy to show that this inclusion interacts as expected with lifetime intersection (LftL-incl-isect).

Indexed borrows. While the proof rules given so far bring us pretty far, it turns out that for some of the advanced reasoning we need to do for Rust, they do not suffice. As we start to build more complicated protocols involving full borrows, the fact that $\&^{\kappa}_{\mathsf{full}}\, P$ is neither timeless nor persistent really becomes a problem.

For this reason, the logic provides a way to *decompose* a full borrow into timeless and persistent pieces (the borrow token and the indexed borrow, respectively), which are tied together by an *index* i (LftL-bor-idx). Indexed borrows can be opened using LftL-idx-acc, but they cannot be strengthened, reborrowed or split. Furthermore, indexed borrows can be *shortened* (LftL-idx-shorten) following the dynamic lifetime inclusion $\kappa' \sqsubseteq \kappa$.

Indexed borrows are used to state the rule LftL-idx-bor-unnest, which will be used later to prove two important derived rules: unnesting and reborrowing.

Internal atomic persistent borrows. They are a primitive form of atomic persistent borrow (see the pargraph below about atomic persistent borrows). They have the same opening and closing rules as atomic peristent borrows, but use $\mathcal{N}_{\mathrm{lft}}$ as namespace, which could not be used with atomic persistent borrows.

LftL-incl-isect
$$\kappa \sqcap \kappa' \sqsubseteq \kappa$$

LftL-incl-glb
$$\frac{\kappa \sqsubseteq \kappa' \qquad \kappa \sqsubseteq \kappa''}{\kappa \sqsubseteq \kappa' \sqcap \kappa''}$$

LftL-fract-lincl
$$\frac{\&^{\kappa}_{\mathrm{frac}}\, q'.\, [\kappa']_{q \cdot q'}}{\kappa \sqsubseteq \kappa'}$$

LftL-bor-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa}_{\mathrm{full}}\, P \Rightarrow \&^{\kappa'}_{\mathrm{full}}\, P}$$

LftL-reborrow
$$\kappa' \sqsubseteq \kappa \vdash \&^{\kappa}_{\mathrm{full}}\, P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathrm{full}}\, P * ([\dagger\kappa'] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathrm{full}}\, P)$$

LftL-bor-unnest
$$\&^{\kappa'}_{\mathrm{full}}(\&^{\kappa}_{\mathrm{full}}\, P) \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa \sqcap \kappa'}_{\mathrm{full}}\, P$$

LftL-bor-acc-cons
$$\&^{\kappa}_{\mathrm{full}}\, P * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \triangleright P * \forall Q.\ \triangleright(\triangleright Q \Rrightarrow\!\!\ast_{\emptyset} \triangleright P) * \triangleright Q \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathrm{full}}\, Q * [\kappa]_q$$

LftL-bor-acc
$$\langle [\kappa]_q * \&^{\kappa}_{\mathrm{full}}\, P \Longleftrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}}}$$

LftL-bor-freeze
$$\frac{\tau \text{ inhabited}}{(\&^{\kappa}_{\mathrm{full}}\, \exists x : \tau.\, P) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists x : \tau.\ \&^{\kappa}_{\mathrm{full}}\, P}$$

LftL-bor-iff
$$\frac{\triangleright \Box (P \Leftrightarrow Q)}{\&^{\kappa}_{\mathrm{full}}\, P \Rightarrow \&^{\kappa}_{\mathrm{full}}\, Q}$$

Fig. 14. Lifetime logic derived rules

Internally, they are implemented in a very similar fashion as atomic persistent borrows. The reason we need them is that they are used for implementing fractured borrows, which are in turn used for creating dynamic lifetime inclusion, and this cannot afford using a different mask as $\mathcal{N}_{\mathrm{lft}}$.

3.2 Derived Forms of Borrowing

Fig. 14 shows some rules that can be derived from the basic rules discussed in the previous subsection.

Furthermore, we introduce in Fig. 15 some derived forms of borrowing – that is, assertions that share are somewhat like $\&^{\kappa}_{\mathrm{full}}\, P$, but not exactly.

Reborrowing. Two The rule LftL-reborrow lets us *reborrow* a $\&^{\kappa}_{\mathrm{full}}\, P$, which means that we can pick some statically shorter lifetime $\kappa' \sqsubseteq \kappa$ and obtain P borrowed at κ'. When κ' ends, we can get our original full borrow back.

The rule LftL-bor-unnest is related. It deals with the case that we have a full borrow of a full borrow ($\&^{\kappa'}_{\mathrm{full}}\, \&^{\kappa}_{\mathrm{full}}\, P$). If we have already opened that full borrow and stripped a way the $\triangleright$ added by opening, then we can use LftL-bor-unnest to "unnest" the full borrow in the sense that we end up with a full borrow at the intersected lifetime ($\&^{\kappa' \sqcap \kappa}_{\mathrm{full}}\, P$).

Both of these rules are *derived* from LftL-idx-bor-unnest.

Persistent borrows. Persistent borrows are a persistent version of borrows. This means that many parties are allowed to get access to its content. In order to avoid reentrant accesses, we can use *two* different mechanisms, giving rise to two flavors of persistent borrows.

Similarly to invariants in Iris, the first possible mechanism is to force only atomic accesses. We then get *atomic persistent borrows*, which are essentially like invariant in Iris with the additional quirk that the invariant is only maintained for the duration of the lifetime of the borrow. They can be defined as follows:

$$\&^{\kappa/\mathcal{N}}_{\mathrm{at}}\, P := \exists i.\ \&^{\kappa}_{i}\, P * \mathcal{N} \# \mathcal{N}_{\mathrm{lft}} * \boxed{[\mathrm{Bor} : i]}^{\mathcal{N}}$$

The other possible mechanism is to restrict the persistent borrow to be used in a threaded manner, by using the mechanism of *non-atomic invariants* described in the Iris documentation (and can be adapted to the iRC11 logic with the same rules). The persistent borrows of this other flavor are called *non-atomic persistent borrows*. They can be defined by:

$$\&^{\kappa/p.\mathcal{N}}_{\mathrm{na}}\, P := \exists i.\ \&^{\kappa}_{i}\, P * \mathsf{NaInv}^{p.\mathcal{N}}([\mathrm{Bor} : i])$$

Fractured borrows. A *fractured borrow* is a borrow of a permission $\Phi(q)$ that can be *fractured*, *i.e.*, decomposed according to a fraction:

$$\Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)$$

Notation	Meaning	Timeless	Persistent
$\&^{\kappa/\mathcal{N}}_{\mathrm{at}}\, P$	There is a *atomic persistent borrow* of P for κ in namespace $\mathcal{N}$	No	Yes
$\&^{\kappa}_{\mathrm{frac}}\, \lambda q.\, P$	There is a *fractured borrow* of $\lambda q.\, P$ for κ	No	Yes
$\&^{\kappa/p.\mathcal{N}}_{\mathrm{na}}\, P$	There is a *non-atomic persistent borrow* of P for κ in non-atomic invariant pool p, namespace $\mathcal{N}$	No	Yes

Atomic persistent borrows.

LftL-bor-at
$$\mathcal{N} \# \mathcal{N}_{\mathrm{lft}} \vdash \&^{\kappa}_{\mathrm{full}}\, P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa/\mathcal{N}}_{\mathrm{at}}\, P$$

LftL-at-acc
$$\&^{\kappa/\mathcal{N}}_{\mathrm{at}}\, P \vdash \langle [\kappa]_q \Lleftarrow\!\!\Rrightarrow V_b.\ \triangleright \lfloor P \rfloor_{\sqcup V_b} \rangle^{\mathcal{N}_{\mathrm{lft}}}_{\mathcal{N}_{\mathrm{lft}},\mathcal{N}}$$

LftL-at-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa/\mathcal{N}}_{\mathrm{at}}\, P \Rightarrow \&^{\kappa'/\mathcal{N}}_{\mathrm{at}}\, P}$$

LftL-at-iff
$$\frac{\triangleright \Box (P \Leftrightarrow Q)}{\&^{\kappa/\mathcal{N}}_{\mathrm{at}}\, P \Rightarrow \&^{\kappa/\mathcal{N}}_{\mathrm{at}}\, Q}$$

Non-atomic persistent borrows.

LftL-bor-na
$$\&^{\kappa}_{\mathrm{full}}\, P \Rrightarrow_{\mathcal{N}} \&^{\kappa/p.\mathcal{N}}_{\mathrm{na}}\, P$$

LftL-na-acc
$$\&^{\kappa/p.\mathcal{N}}_{\mathrm{na}}\, P \vdash \langle [\kappa]_q * [\mathrm{Na} : p.\mathcal{N}] \Lleftarrow\!\!\Rrightarrow \triangleright P \rangle_{\mathcal{N}_{\mathrm{lft}},\mathcal{N}}$$

LftL-na-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa/p.\mathcal{N}}_{\mathrm{na}}\, P \Rightarrow \&^{\kappa'/p.\mathcal{N}}_{\mathrm{na}}\, P}$$

LftL-na-iff
$$\frac{\triangleright \Box (P \Leftrightarrow Q)}{\&^{\kappa/\mathcal{N}.P}_{\mathrm{na}} \Rightarrow \&^{\kappa/\mathcal{N}.Q}_{\mathrm{na}}}$$

Fractured borrows.

LftL-bor-fracture
$$\frac{\forall q_1, q_2.\, \Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)}{\&^{\kappa}_{\mathrm{full}}\, \Phi(1) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathrm{frac}}\, \Phi}$$

LftL-fract-acc
$$\&^{\kappa}_{\mathrm{frac}}\, \Phi \vdash \langle [\kappa]_q \Lleftarrow\!\!\Rrightarrow q'.\ \triangleright \Phi(q') \rangle_{\mathcal{N}_{\mathrm{lft}}}$$

LftL-fract-shorten
$$\frac{\kappa' \sqsubseteq \kappa}{\&^{\kappa}_{\mathrm{frac}}\, \Phi \Rightarrow \&^{\kappa'}_{\mathrm{frac}}\, \Phi}$$

LftL-fract-iff
$$\frac{\triangleright \Box (\forall q.\, \Phi(q) \Leftrightarrow \Psi(q))}{\&^{\kappa}_{\mathrm{frac}}\, \Phi \Rightarrow \&^{\kappa}_{\mathrm{frac}}\, \Phi}$$

Fig. 15. Lifetime logic derived forms

Intuitively, it should be possible to share such a borrow, and still obtain some fraction of Φ via a non-atomic accessor, *i.e.*, $\Phi(q)$ can actually be kept around for non-atomic expressions. This is because even if other threads are concurrently accessing the borrow, they will always leave *some* fraction of Φ in the borrow.

Fractured borrows are particularly interesting for giving rise to dynamic lifetime inclusion (LftL-fract-lincl).

4 COUNTEREXAMPLE: LIFETIME LOGIC WITH UNSYCHRONIZED GHOST STATE

If in RMM we model lifetime tokens as view-agnostic ghost state, then by using the Ghost-Mod rule we can provide a *spurious* verification of the buggy MP example given in Fig. 16.

We create a lifetime κ and a borrow for X, and instantiate SendRecv for Y before giving them to the two threads. In thread 1 (Fig. 16b), we access the borrow and write to X. Then, to send $[\kappa]_{1/2}$ (via a **rlx** write to Y), we use Ghost-Mod to obtain $\Delta[\kappa]_{1/2}$. *Note that this proof step is only possible because we assume view-agnostic lifetime tokens.*

In thread 2 (Fig. 16c), after receiving $\nabla[\kappa]_{1/2}$, we apply Ghost-Mod again to strip off the acquire modality, thus obtaining the missing half of the token. Combining both halves, we kill κ and apply the inheritance to obtain $X \mapsto -$. This, in turn, licenses the following non-atomic write to X, which is *not* happens-after thread 1's write to X and thus constitutes a data race.

As we can see from this scenario, our hypothetical lifetime logic for relaxed memory violates a key safety guarantee: that a lifetime κ's inheritance must happen-after all accesses to all borrows of κ. The root of the problem is that we are able to move view-agnostic lifetime tokens in and out of the fence modalities.

$$X := 0; Y := 0;$$
$$\begin{array}{r||l} X := 42; & \textbf{if}\ {}^{*\mathtt{rlx}}Y \mathrel{!=} 0 \\ Y :=_{\mathtt{rlx}} 1 & \textbf{then}\ X := 57; \end{array}$$

(a) Buggy Message-Passing.

$\left\{[\kappa]_{1/2} * \&^{\kappa}_{\mathsf{full}}(X \mapsto -) * \mathsf{Send}_Y([\kappa]_{1/2})\right\}$
$X := 42;\ \left\{[\kappa]_{1/2} * \mathsf{Send}_Y([\kappa]_{1/2})\right\}$
$\left\{\Delta[\kappa]_{1/2} * \mathsf{Send}_Y([\kappa]_{1/2})\right\}$ *Unsound!*
$Y :=_{\mathtt{rlx}} 1;\ \{\mathsf{True}\}$

(b) Buggy proof of thread 1.

$\left\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, X \mapsto -) * \mathsf{Recv}_Y([\kappa]_{1/2})\right\}$
$\textbf{if}\,({}^{*\mathtt{rlx}}Y \mathrel{!=} 0)$
$\quad\left\{[\kappa]_{1/2} * \ldots * \nabla[\kappa]_{1/2}\right\}$
$\quad\left\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \ldots * [\kappa]_{1/2}\right\}$ *Unsound!*
$\quad\{[\dagger\kappa] * \mathsf{Inh}(\kappa, X \mapsto -)\}\ \{X \mapsto -\}\ X := 57;\ \{X \mapsto 57\}$

(c) Buggy proof of thread 2.

Fig. 16. Buggy MP spuriously verified with view-agnostic lifetime tokens.

5 IRC11

iRC11 is an extension of iGPS ([Kaiser et al. 2017]) that adopts the fence modalities from FSL ([Doko and Vafeiadis 2016, 2017]). Fig. 17 lists the rules for traditional points-to assertions (non-atomics). Fig. 18 lists the rules for fork and fences.

iRC11 combines GPS single-location protocols and iGPS single-write protocols with atomic borrows (Fig. 19, Fig. 20, Fig. 22, Fig. 23, Fig. 24, Fig. 25). These protocols are used to verify `Mutex`, `RwLock`. Note that atomic borrows are slightly different from raw cancellable invariants, as its cancellation depends on lifetimes.

iRC11 provides cancellable single-location protocols based on raw cancellable invariants. Some of them are given in Fig. 26 and Fig. 27. These protocols are used to verify `Arc<T>`, `thread::spawn`, and `rayon::join`.

NA-FRAC-AGREE
$$\ell \overset{q}{\mapsto} v * \ell \overset{q'}{\mapsto} v' \Leftrightarrow \ell \xmapsto{q+q'} v * v = v'$$

NA-FREEABLE-COMBINE
$$\dagger^{m}_{q}\ \ell * \dagger^{m'}_{q'}\ \ell + m \Leftrightarrow \dagger^{m+m'}_{q+q'}\ \ell$$

NA-ALLOC
$$\{\mathsf{True}\}\ \mathbf{alloc}(n)\ \{\ell.\ \exists \overline{v}.\ \ell \mapsto \overline{v} * |\overline{v}| = n * \dagger^{n}_{1}\ \ell\}$$

NA-FREE
$$\left\{\ell \mapsto \overline{v} * \dagger^{|\overline{v}|}_{1}\ \ell\right\}\ \mathbf{free}(|\overline{v}|, v)\ \{\mathsf{True}\}$$

NA-READ
$$\left\{\ell \overset{q}{\mapsto} v\right\} {}^{*}\ell \left\{v'.\ v' = v * \ell \overset{q}{\mapsto} v\right\}$$

NA-WRITE
$$\{\ell \mapsto v\}\ \ell := w\ \{\ell \mapsto w\}$$

NA-MEMCPY
$$\frac{|\overline{v}_1| = |\overline{v}_2| = n}{\left\{\ell_1 \mapsto \overline{v}_1 * \ell_2 \overset{q}{\mapsto} \overline{v}_2\right\}\ \ell_1 :=_n {}^{*}\ell_2 \left\{\ell_1 \mapsto \overline{v}_2 * \ell_2 \overset{q}{\mapsto} \overline{v}_2\right\}}$$

Fig. 17. Non-atomics rules.

FORK
$$\frac{\forall \rho.\ \{P\}\ e\ \mathsf{in}\ \rho\ \{\mathsf{True}\}}{\{P\}\ \mathbf{fork}\ \{\ e\ \}\ \{\mathsf{True}\}}$$

REL-FENCE
$$\{P\}\ \mathbf{fence_{rel}}\ \mathsf{in}\ \pi\ \{\Delta_\pi\ P\}$$

ACQ-FENCE
$$\{\nabla_\pi\ P\}\ \mathbf{fence_{acq}}\ \mathsf{in}\ \pi\ \{P\}$$

Fig. 18. Fork and fences rules.

AtBor-N-persistent

$$\&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}} \Rightarrow \square\, \&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}}$$

AtBor-N-local

$$\&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}} \Rightarrow \mathcal{R}(\ell, t, s, v, \mathcal{I})$$

AtBor-N-local-join

$$\mathcal{R}(\ell, t, s, v, \mathcal{I}) * \&^{\kappa}\boxed{\ell : (t', s', v') \mid \mathcal{I}} \Rightarrow \&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}}$$

AtBor-N-init

$$\begin{array}{l}[\kappa]_q \twoheadrightarrow \&^{\kappa}_{\mathsf{full}}\,(\exists v.\, \ell \mapsto v * P(v)) \twoheadrightarrow (\forall t, v.\, \triangleright P(v) \twoheadrightarrow \triangleright \mathcal{I}_w(\ell, t, s, v)) \twoheadrightarrow \\ \qquad (\square\, \forall t, s, v.\, \triangleright \mathcal{I}_w(\ell, t, s, v) \Rrightarrow \triangleright P(v)) \Rrightarrow\!\!\ast \\ \qquad\qquad [\kappa]_q * \exists t, v.\, \&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}}\end{array}$$

AtBor-N-rlx-read

$$\frac{\begin{array}{c}\forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_r(\ell, t', s', v') \Rrightarrow \mathcal{I}_r(\ell, t', s', v') * P(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_w(\ell, t', s', v') \Rrightarrow \mathcal{I}_w(\ell, t', s', v') * P(t', s', v')\end{array}}{\left\{[\kappa]_q * \&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}}\right\} *^{\mathtt{rlx}}\ell \text{ in } \pi \left\{v'.\, [\kappa]_q * \exists t' \sqsupseteq t, s' \sqsupseteq s.\, \&^{\kappa}\boxed{\ell : (t', s', v') \mid \mathcal{I}} * \nabla_{\pi}\, P(t', s', v')\right\}}$$

AtBor-N-acq-read

$$\frac{\begin{array}{c}\forall t', s', v'.\, \mathcal{I}_r(\ell, t', s', v') \Rrightarrow \mathcal{I}_r(\ell, t', s', v') * P(t', s', v') \\ \forall t', s', v'.\, \mathcal{I}_w(\ell, t', s', v') \Rrightarrow \mathcal{I}_w(\ell, t', s', v') * P(t', s', v')\end{array}}{\left\{[\kappa]_q * \&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}}\right\} *^{\mathtt{acq}}\ell \left\{v'.\, [\kappa]_q * \exists t' \sqsupseteq t, s' \sqsupseteq s.\, \&^{\kappa}\boxed{\ell : (t', s', v') \mid \mathcal{I}} * P(t', s', v')\right\}}$$

AtBor-N-rlx-write

$$\begin{array}{l}\left\{[\kappa]_q * \&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}} * \Delta_{\pi}\, (\forall t' > t.\, \mathcal{R}(\ell, t', s', v', \mathcal{I}) \Rrightarrow \mathcal{I}_w(\ell, t', s', v'))\right\} \\ \quad \ell :=_{\mathtt{rlx}} v' \text{ in } \pi \\ \left\{[\kappa]_q * \&^{\kappa}\boxed{\ell : (t', s', v') \mid \mathcal{I}}\right\}\end{array}$$

AtBor-N-rel-write

$$\begin{array}{l}\left\{[\kappa]_q * \&^{\kappa}\boxed{\ell : (t, s, v) \mid \mathcal{I}} * (\forall t' > t.\, \mathcal{R}(\ell, t', s', v', \mathcal{I}) \Rrightarrow \mathcal{I}_w(\ell, t', s', v'))\right\} \\ \quad \ell :=_{\mathtt{rel}} v' \text{ in } \pi \\ \left\{[\kappa]_q * \&^{\kappa}\boxed{\ell : (t', s', v') \mid \mathcal{I}}\right\}\end{array}$$

Fig. 19. Atomic-borrow-based normal iRC11 prototocols.

$$\Delta_\pi^{?o_w} P := (\texttt{if}\ o_w = \texttt{rel}\ \texttt{then}\ P\ \texttt{else}\ \Delta_\pi P)$$

$$\nabla_\pi^{?o_r} P := (\texttt{if}\ o_r = \texttt{acq}\ \texttt{then}\ P\ \texttt{else}\ \nabla_\pi P)$$

AtBor-N-cas

$$\frac{\begin{array}{c} o_f, o_r \in \{\texttt{rlx}, \texttt{acq}\} \qquad o_w \in \{\texttt{rlx}, \texttt{rel}\} \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_w(\ell, t', s', v') \lor \mathcal{I}_r(\ell, t', s', v') \Rightarrow (\vdash v_r =^? v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, (\vdash v' \neq v_r) \Rightarrow \mathcal{I}_r(\ell, t', s', v') \Rrightarrow \mathcal{I}_r(\ell, t', s', v') * R(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, (\vdash v' \neq v_r) \Rightarrow \mathcal{I}_w(\ell, t', s', v') \Rrightarrow \mathcal{I}_w(\ell, t', s', v') * R(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s.\, \triangleright \mathcal{I}_w(\ell, t', s', v_r) \Rrightarrow \triangleright Q_1(t', s') * \triangleright Q_2(t', s') \\ \Delta_\pi^{?o_w} \left(\begin{array}{l} \forall t' \sqsupseteq t, s' \sqsupseteq s.\, P \twoheadrightarrow \triangleright Q_2(t', s') \Rrightarrow \exists s'' \sqsupseteq s'.\, \forall t'' > t.\, \triangleright \mathcal{R}(\ell, t'', s'', v_w, \mathcal{I}) \\ \quad \Rrightarrow (\langle obj \rangle (\triangleright Q_1(t', s') \Rrightarrow \triangleright \mathcal{I}_m(\ell, t', s', v_r))) * \Rrightarrow\!\!\ast\, (Q(t'', s'') * \mathcal{I}_w(\ell, t'', s'', v_w)) \end{array} \right) \end{array}}{\begin{array}{l} \left\{ [\kappa]_q * \&^\kappa \boxed{\ell : (t, s, v)}\boxed{\mathcal{I}} * \Delta_\pi^{?o_w} P \right\} \\ \quad \texttt{CAS}(\ell, v_r, v_w, o_f, o_r, o_w)\ \texttt{in}\ \pi \\ \left\{ b.\, [\kappa]_q * \exists s' \sqsupseteq s.\, \begin{array}{l} b = 1 * \exists t' > t.\, \&^\kappa \boxed{\ell : (t', s', v_w)}\boxed{\mathcal{I}} * \nabla_\pi^{?o_r} Q(t'', s'') \\ \lor b = 0 * \Delta_\pi^{?o_w} P * \exists t' \geq t, v'.\, (\vdash v' \neq v_r) * \&^\kappa \boxed{\ell : (t', s', v')}\boxed{\mathcal{I}} * \nabla_\pi^{?o_f} R(t', s', v') \end{array} \right\} \end{array}}$$

Fig. 20. CAS rule for atomic-borrow-based normal iRC11 prototocols.

sw-writer-local-exclusive
$$\mathcal{W}(\ell, t, s, v, \mathcal{I}) * \mathcal{W}(\ell, t, s, v, \mathcal{I}) \Rightarrow \text{False}$$

sw-local-writer-reader
$$\mathcal{W}(\ell, t, s, v, \mathcal{I}) \Rightarrow \mathcal{R}(\ell, t, s, v, \mathcal{I})$$

sw-creaders-local-join
$$\mathcal{R}^q_{\text{shr}}(\ell, t, s, v, \mathcal{I}) * \mathcal{R}^{q'}_{\text{shr}}(\ell, t', s', v', \mathcal{I}) \Rightarrow \mathcal{R}^{q+q'}_{\text{shr}}(\ell, t, s, v, \mathcal{I})$$

sw-creaders-local-split
$$\mathcal{R}^{q+q'}_{\text{shr}}(\ell, t, s, v, \mathcal{I}) \Rightarrow \mathcal{R}^q_{\text{shr}}(\ell, t, s, v, \mathcal{I}) * \mathcal{R}^{q'}_{\text{shr}}(\ell, t, s, v, \mathcal{I})$$

sw-cwriter-local-exclusive
$$\mathcal{W}_{\text{shr}}(\ell, t, s, v, \mathcal{I}) * \mathcal{W}_{\text{shr}}(\ell, t', s', v', \mathcal{I}) \Rightarrow \text{False}$$

sw-share-local-cwriter
$$\mathcal{W}(\ell, t, s, v, \mathcal{I}) \Rightarrow \mathcal{W}_{\text{shr}}(\ell, t, s, v, \mathcal{I}) * \mathcal{R}^1_{\text{shr}}(\ell, t, s, v, \mathcal{I})$$

sw-reader-creader-local
$$\mathcal{R}^q_{\text{shr}}(\ell, t, s, v, \mathcal{I}) \Rightarrow \mathcal{R}(\ell, t, s, v, \mathcal{I})$$

sw-cw-local-exclusive
$$\mathcal{W}(\ell, t, s, v, \mathcal{I}) * \mathcal{W}_{\text{shr}}(\ell, t', s', v', \mathcal{I}) \Rightarrow \text{False}$$

sw-cr-local-exclusive
$$\mathcal{W}(\ell, t, s, v, \mathcal{I}) * \mathcal{R}^q_{\text{shr}}(\ell, t', s', v', \mathcal{I}) \Rightarrow \text{False}$$

Fig. 21. Local assertions of single-writer iRC11 prototocols.

AtBor-sw-reader-persistent
$$\&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{R} \Rightarrow \square \&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{R}$$

AtBor-sw-reader-local
$$\&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{R} \Rightarrow \mathcal{R}(\ell, t, s, v, \mathcal{I})$$

AtBor-sw-reader-local-join
$$\mathcal{R}(\ell, t, s, v, \mathcal{I}) * \&^{\kappa} \boxed{\ell : (t', s', v') \mid \mathcal{I}}_{R} \Rightarrow \&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{R}$$

AtBor-sw-writer-local
$$\&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{W} \Rightarrow \mathcal{W}(\ell, t, s, v, \mathcal{I})$$

AtBor-sw-writer-local-join
$$\mathcal{W}(\ell, t, s, v, \mathcal{I}) * \&^{\kappa} \boxed{\ell : (t', s', v') \mid \mathcal{I}}_{R} \Rightarrow \&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{W}$$

AtBor-sw-creader-local
$$\&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{CR}^{q} \Rightarrow \mathcal{R}_{\text{shr}}^{q}(\ell, t, s, v, \mathcal{I})$$

AtBor-sw-creader-local-join
$$\mathcal{R}_{\text{shr}}^{q}(\ell, t, s, v, \mathcal{I}) * \&^{\kappa} \boxed{\ell : (t', s', v') \mid \mathcal{I}}_{R} \Rightarrow \&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{CR}^{q}$$

AtBor-sw-cwriter-local
$$\&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{CW} \Rightarrow \mathcal{W}_{\text{shr}}(\ell, t, s, v, \mathcal{I})$$

AtBor-sw-cwriter-local-join
$$\mathcal{W}_{\text{shr}}(\ell, t, s, v, \mathcal{I}) * \&^{\kappa} \boxed{\ell : (t', s', v') \mid \mathcal{I}}_{R} \Rightarrow \&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{CW}$$

AtBor-sw-unshare-local-cwriter
$$[\kappa]_q * \mathcal{W}_{\text{shr}}(\ell, t, s, v, \mathcal{I}) * \&^{\kappa} \boxed{\ell : (t', s', v') \mid \mathcal{I}}_{CR}^{1} \Rrightarrow [\kappa]_q * \&^{\kappa} \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{W}$$

Fig. 22. Atomic-borrow-based single-writer iRC11 prototocols (1).

AtBor-sw-init

$$[\kappa]_q * \&^{\kappa}_{\mathbf{full}} (\exists v.\, \ell \mapsto v * P(v)) \mathrel{-\!\!*} (\forall t, v.\, \triangleright P(v) \mathrel{-\!\!*} \mathcal{W}(\ell,t,s,v,\mathcal{I}) \Rrightarrow \triangleright \mathcal{I}_w(\ell,t,s,v) * Q(t,v)) \mathrel{-\!\!*}$$
$$(\Box\, \forall t,s,v.\, \triangleright \mathcal{I}_w(\ell,t,s,v) \Rrightarrow \triangleright P(v)) \Rrightarrow\!\!\!\ast$$
$$[\kappa]_q * \exists t,v.\, \&^{\kappa} \boxed{\ell : (t,s,v) \mid \mathcal{I}}_R * Q(t,v)$$

AtBor-sw-read

$$\frac{\begin{array}{c} o \in \{\mathbf{rlx}, \mathbf{acq}\} \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_r(\ell,t',s',v') \Rrightarrow \mathcal{I}_r(\ell,t',s',v') * P(t',s',v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_w(\ell,t',s',v') \Rrightarrow \mathcal{I}_w(\ell,t',s',v') * P(t',s',v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_m(\ell,t',s',v') \Rrightarrow \mathcal{I}_m(\ell,t',s',v') * P(t',s',v') \end{array}}{\begin{array}{l} \left\{[\kappa]_q * \&^{\kappa} \boxed{\ell : (t,s,v) \mid \mathcal{I}}_R\right\} \\ \quad {*^{o}}\ell \text{ in } \pi \\ \left\{v'.\, [\kappa]_q * \exists t' \sqsupseteq t, s' \sqsupseteq s.\ \&^{\kappa} \boxed{\ell : (t',s',v') \mid \mathcal{I}}_R * \nabla^{?o}_{\pi} P(t',s',v')\right\} \end{array}}$$

AtBor-sw-exclusive-read

$$\frac{o \in \{\mathbf{rlx}, \mathbf{acq}\} \qquad \mathcal{I}_w(\ell,t,s,v) \Rrightarrow \mathcal{I}_w(\ell,t,s,v) * P}{\left\{[\kappa]_q * \&^{\kappa} \boxed{\ell : (t,s,v) \mid \mathcal{I}}_W *\right\} {*^{o}}\ell \text{ in } \pi \left\{v.\, [\kappa]_q * \&^{\kappa} \boxed{\ell : (t,s,v) \mid \mathcal{I}}_W * \nabla^{?o}_{\pi} P\right\}}$$

AtBor-sw-creader-read

$$\frac{\begin{array}{c} o \in \{\mathbf{rlx}, \mathbf{acq}\} \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_r(\ell,t',s',v') \Rrightarrow \mathcal{I}_r(\ell,t,s,v) * P(t',s',v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_w(\ell,t',s',v') \Rrightarrow \mathcal{I}_w(\ell,t',s',v') * P(t',s',v') \end{array}}{\begin{array}{l} \left\{[\kappa]_{q_0} * \&^{\kappa} \boxed{\ell : (t,s,v) \mid \mathcal{I}}^{q}_{CR}\right\} \\ \quad {*^{o}}\ell \text{ in } \pi \\ \left\{v'.\, [\kappa]_{q_0} * \exists t' \sqsupseteq t, s' \sqsupseteq s.\ \&^{\kappa} \boxed{\ell : (t',s',v') \mid \mathcal{I}}^{q}_{CR} * \nabla^{?o}_{\pi} P(t',s',v')\right\} \end{array}}$$

Fig. 23. Atomic-borrow-based single-writer iRC11 prototocols (2).

AtBor-sw-write

$$\frac{o \in \{\texttt{rlx}, \texttt{rel}\} \qquad s \sqsubseteq s' \qquad \triangleright \langle obj \rangle\, (\mathcal{I}_w(\ell, t, s, v) \Rrightarrow \mathcal{I}_m(\ell, t, s, v) * Q)}{\begin{array}{l}\left\{[\kappa]_q * \&^\kappa \boxed{\ell : (t, s, v) \mid \mathcal{I}}_W * \Delta^{?o}_\pi\, (\forall t' > t.\, \mathcal{R}(\ell, t', s', v', \mathcal{I}) \Rrightarrow \mathcal{I}_w(\ell, t', s', v'))\right\} \\ \quad \ell :=_o v' \text{ in } \pi \\ \left\{[\kappa]_q * \&^\kappa \boxed{\ell : (t', s', v') \mid \mathcal{I}}_W * Q\right\}\end{array}}$$

AtBor-sw-rel-write

$$\frac{s \sqsubseteq s' \qquad \triangleright \langle obj \rangle\, (\mathcal{I}_w(\ell, t, s, v) \Rrightarrow Q_1 * Q_2)}{\begin{array}{l}\left\{\begin{array}{l}[\kappa]_q * \&^\kappa \boxed{\ell : (t, s, v) \mid \mathcal{I}}_W * \\ \triangleright \left(\forall t' > t.\, \mathcal{W}(\ell, t', s', v', \mathcal{I}) \twoheadrightarrow Q_2 \Rrightarrow\!\!\ast \left(\langle obj \rangle\, (Q_1 \Rrightarrow\!\!\ast \mathcal{I}_m(\ell, t, s, v)) * \Rrightarrow\!(\mathcal{I}_w(\ell, t', s', v') * Q(t'))\right)\right)\end{array}\right\} \\ \quad \ell :=_{\texttt{rel}} v' \\ \left\{[\kappa]_q * \exists t' > t.\ \&^\kappa \boxed{\ell : (t', s', v') \mid \mathcal{I}}_R * Q(t')\right\}\end{array}}$$

Fig. 24. Atomic-borrow-based single-writer iRC11 protototocols (3).

$$b \mathbin{?} P \coloneqq \textbf{if } b \textbf{ then } P \textbf{ else } \mathsf{True} \qquad b \mathbin{?} P \,:\, Q \coloneqq \textbf{if } b \textbf{ then } P \textbf{ else } Q$$

AtBor-sw-creader-cas

$$\frac{\begin{array}{c} o_f, o_r \in \{\mathbf{rlx}, \mathbf{acq}\} \qquad o_w \in \{\mathbf{rlx}, \mathbf{rel}\} \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_w(\ell, t', s', v') \lor \mathcal{I}_r(\ell, t', s', v') \Rightarrow (\vdash v_r =^? v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, (\vdash v' \neq v_r) \Rightarrow \mathcal{I}_r(\ell, t', s', v') \Rrightarrow \mathcal{I}_r(\ell, t', s', v') * R(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, (\vdash v' \neq v_r) \Rightarrow \mathcal{I}_w(\ell, t', s', v') \Rrightarrow \mathcal{I}_w(\ell, t', s', v') * R(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s.\, \triangleright \mathcal{I}_w(\ell, t', s', v_r) \Rrightarrow \triangleright Q_1(t', s') * \triangleright Q_2(t', s') \\ \Delta_\pi^{?o_w} \left(\begin{array}{l} \forall t' \sqsupseteq t, s' \sqsupseteq s.\, P \mathrel{-\!\!*} \triangleright Q_2(t', s') \Rrightarrow \triangleright \mathcal{W}_{\mathrm{shr}}(\ell, t', s', v_r) * \exists s'' \sqsupseteq s'. \\ \quad \forall t'' > t.\, \triangleright \mathcal{W}_{\mathrm{shr}}(\ell, t'', s'', v_w, \mathcal{I}) \mathrel{-\!\!*} b_{\mathrm{drop}} \mathbin{?} \triangleright \mathcal{R}^q_{\mathrm{shr}}(\ell, t'', s'', v_w, \mathcal{I}) \\ \quad \Rrightarrow (\langle obj \rangle\, (\triangleright Q_1(t', s') \Rrightarrow \triangleright \mathcal{I}_m(\ell, t', s', v_r))) * \Rrightarrow\!\!\!\ast\; (Q(t'', s'') * \mathcal{I}_w(\ell, t'', s'', v_w)) \end{array} \right) \end{array}}{\begin{array}{l} \left\{ [\kappa]_{q_0} * \&^\kappa \boxed{\ell : (t, s, v) \mid \mathcal{I}}^{q}_{CR} * \Delta_\pi^{?o_w} P \right\} \\ \quad \mathbf{CAS}(\ell, v_r, v_w, o_f, o_r, o_w) \text{ in } \pi \\ \left\{ b.\, [\kappa]_{q_0} * \exists s' \sqsupseteq s. \begin{array}{l} b = 1 * \exists t' > t.\, \left(b_{\mathrm{drop}} \mathbin{?} \&^\kappa \boxed{\ell : (t', s', v_w) \mid \mathcal{I}}_{R} \,:\, \&^\kappa \boxed{\ell : (t', s', v_w) \mid \mathcal{I}}^{q}_{CR} \right) * \\ \quad \nabla_\pi^{?o_r} Q(t', s') \\ \lor\, b = 0 * \Delta_\pi^{?o_w} P * \exists t' \geq t, v'.\, (\vdash v' \neq v_r) * \&^\kappa \boxed{\ell : (t', s', v') \mid \mathcal{I}}^{q}_{CR} * \\ \quad \nabla_\pi^{?o_f} R(t', s', v') \end{array} \right\} \end{array}}$$

AtBor-sw-reader-cas

$$\frac{\begin{array}{c} o_f, o_r \in \{\mathbf{rlx}, \mathbf{acq}\} \qquad o_w \in \{\mathbf{rlx}, \mathbf{rel}\} \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, \mathcal{I}_w(\ell, t', s', v') \lor \mathcal{I}_r(\ell, t', s', v') \lor \mathcal{I}_m(\ell, t', s', v') \Rightarrow (\vdash v_r =^? v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, (\vdash v' \neq v_r) \Rightarrow \mathcal{I}_r(\ell, t', s', v') \Rrightarrow \mathcal{I}_r(\ell, t', s', v') * R(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, (\vdash v' \neq v_r) \Rightarrow \mathcal{I}_w(\ell, t', s', v') \Rrightarrow \mathcal{I}_w(\ell, t', s', v') * R(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s, v'.\, (\vdash v' \neq v_r) \Rightarrow \mathcal{I}_m(\ell, t', s', v') \Rrightarrow \mathcal{I}_m(\ell, t', s', v') * R(t', s', v') \\ \forall t' \sqsupseteq t, s' \sqsupseteq s.\, \triangleright(P \mathrel{-\!\!*} \mathcal{I}_m(\ell, t', s', v_r)) \Rrightarrow \mathsf{False} \\ \forall t' \sqsupseteq t, s' \sqsupseteq s.\, \triangleright \mathcal{I}_w(\ell, t', s', v_r) \Rrightarrow \triangleright Q_1(t', s') * \triangleright Q_2(t', s') \\ \Delta_\pi^{?o_w} \left(\begin{array}{l} \forall t' \sqsupseteq t, s' \sqsupseteq s.\, P \mathrel{-\!\!*} \triangleright Q_2(t', s') \Rrightarrow \triangleright \mathcal{W}_{\mathrm{shr}}(\ell, t', s', v_r) * \exists s'' \sqsupseteq s'.\, \forall t'' > t.\, \triangleright \mathcal{W}_{\mathrm{shr}}(\ell, t'', s'', v_w, \mathcal{I}) \\ \quad \Rrightarrow (\langle obj \rangle\, (\triangleright Q_1(t', s') \Rrightarrow \triangleright \mathcal{I}_m(\ell, t', s', v_r))) * \Rrightarrow\!\!\!\ast\; (Q(t'', s'') * \mathcal{I}_w(\ell, t'', s'', v_w)) \end{array} \right) \end{array}}{\begin{array}{l} \left\{ [\kappa]_q * \&^\kappa \boxed{\ell : (t, s, v) \mid \mathcal{I}}_{R} * \Delta_\pi^{?o_w} P \right\} \\ \quad \mathbf{CAS}(\ell, v_r, v_w, o_f, o_r, o_w) \text{ in } \pi \\ \left\{ b.\, [\kappa]_q * \exists s' \sqsupseteq s. \begin{array}{l} b = 1 * \exists t' > t.\, \&^\kappa \boxed{\ell : (t', s', v_w) \mid \mathcal{I}}_{R} * \nabla_\pi^{?o_r} Q(t', s') \\ \lor\, b = 0 * \Delta_\pi^{?o_w} P * \exists t' \geq t, v'.\, (\vdash v' \neq v_r) * \&^\kappa \boxed{\ell : (t', s', v') \mid \mathcal{I}}_{R} * \\ \quad \nabla_\pi^{?o_f} R(t', s', v') \end{array} \right\} \end{array}}$$

Fig. 25. Atomic-borrow-based single-writer iRC11 prototocols (4).

ViewInv-sw-reader-persistent

$${}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_R \Rightarrow \square\, {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_R$$

ViewInv-sw-reader-local

$${}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_R \Rightarrow \mathcal{R}(\ell, t, s, v, \mathcal{I})$$

ViewInv-sw-reader-local-join

$$\mathcal{R}(\ell, t, s, v, \mathcal{I}) * {}^{\tau}\boxed{\ell : (t', s', v') \mid \mathcal{I}}_R \Rightarrow {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_R$$

ViewInv-sw-writer-local

$${}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_W \Rightarrow \mathcal{W}(\ell, t, s, v, \mathcal{I})$$

ViewInv-sw-writer-local-join

$$\mathcal{W}(\ell, t, s, v, \mathcal{I}) * {}^{\tau}\boxed{\ell : (t', s', v') \mid \mathcal{I}}_R \Rightarrow {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_W$$

ViewInv-sw-creader-local

$${}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}^{q}_{CR} \Rightarrow \mathcal{R}^{q}_{\mathrm{shr}}(\ell, t, s, v, \mathcal{I})$$

ViewInv-sw-creader-local-join

$$\mathcal{R}^{q}_{\mathrm{shr}}(\ell, t, s, v, \mathcal{I}) * {}^{\tau}\boxed{\ell : (t', s', v') \mid \mathcal{I}}_R \Rightarrow {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}^{q}_{CR}$$

ViewInv-sw-cwriter-local

$${}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_{CW} \Rightarrow \mathcal{W}_{\mathrm{shr}}(\ell, t, s, v, \mathcal{I})$$

ViewInv-sw-cwriter-local-join

$$\mathcal{W}_{\mathrm{shr}}(\ell, t, s, v, \mathcal{I}) * {}^{\tau}\boxed{\ell : (t', s', v') \mid \mathcal{I}}_R \Rightarrow {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_{CW}$$

ViewInv-sw-unshare-local-cwriter

$$[\tau]_q * \mathcal{W}_{\mathrm{shr}}(\ell, t, s, v, \mathcal{I}) * {}^{\tau}\boxed{\ell : (t', s', v') \mid \mathcal{I}}^{1}_{CR} \Rrightarrow [\tau]_q * {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_W$$

Fig. 26. View-invariant-based single-writer iRC11 prototocols (1).

ViewInv-sw-init

$$\ell \mapsto v \twoheadrightarrow (\forall \tau, t, v.\ \triangleright \mathcal{I}_w(\ell, t, s, v)) \Rrightarrow\!\!\ast\ \exists \tau, t, v.\ [\tau]_1 * {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_W$$

ViewInv-sw-rel-write

$$\frac{s \sqsubseteq s' \qquad \triangleright \langle obj \rangle\, (\mathcal{I}_w(\ell, t, s, v) \Rrightarrow Q_1 * Q_2)}{\left\{\begin{array}{l}[\tau]_q * {}^{\tau}\boxed{\ell : (t, s, v) \mid \mathcal{I}}_W * \\ \triangleright \Big(\forall t' > t.\ \mathcal{W}(\ell, t', s', v', \mathcal{I}) \twoheadrightarrow Q_2 \twoheadrightarrow [\tau]_q \Rrightarrow\!\!\ast\ \big(\langle obj \rangle\, (Q_1 \Rrightarrow\!\!\ast\ \mathcal{I}_m(\ell, t, s, v)) * \Rrightarrow(\mathcal{I}_w(\ell, t', s', v') * Q(t'))\big)\Big)\end{array}\right\}\ \ell :=_{\mathtt{rel}} v'\ \left\{\exists t' > t.\ {}^{\tau}\boxed{\ell : (t', s', v') \mid \mathcal{I}}_R * Q(t')\right\}}$$

Fig. 27. View-invariant-based single-writer iRC11 prototocols (2).

$$\mathcal{R}(\ell, t, s, v, \mathcal{I}) \Rrightarrow \square\, \mathcal{R}(\ell, t, s, v, \mathcal{I})$$

$$\dfrac{\begin{array}{c}\langle obj\rangle\, \forall v', t' \geq t, s' \sqsupseteq s.\, \mathcal{I}_r(\ell, t', s', v') \Rrightarrow \mathcal{I}_r(\ell, t', s', v') * P(v') \\ \langle obj\rangle\, \forall v', t' \geq t, s' \sqsupseteq s.\, \mathcal{I}_w(\ell, t', s', v') \Rrightarrow \mathcal{I}_w(\ell, t', s', v') * P(v') \\ \langle obj\rangle\, \forall v', t' \geq t, s' \sqsupseteq s.\, \mathcal{I}_m(\ell, t', s', v') \Rrightarrow \mathcal{I}_m(\ell, t', s', v') * P(v')\end{array}}{\begin{array}{l}\{\mathcal{R}(\ell, t, s, v, \mathcal{I}) * \lfloor \triangleright \mathsf{ATOM}(\ell, \mathcal{I}) \rfloor_V\} \\ \quad *^{\texttt{rlx}}\ell \text{ in } \pi \\ \{v'.\, \nabla_\pi\, P(v') * t \leq t' * s \sqsubseteq s' * \mathcal{R}(\ell, t', s', v', \mathcal{I}_r) * \lfloor \triangleright \mathsf{ATOM}(\ell, \mathcal{I}) \rfloor_V\}\end{array}}$$

$$\dfrac{\begin{array}{c}\langle obj\rangle\, \forall v', t' \geq t, s' \sqsupseteq s.\, \mathcal{I}_r(\ell, t', s', v') \Rrightarrow \mathcal{I}_r(\ell, t', s', v') * P(v') \\ \langle obj\rangle\, \forall v', t' \geq t, s' \sqsupseteq s.\, \mathcal{I}_w(\ell, t', s', v') \Rrightarrow \mathcal{I}_w(\ell, t', s', v') * P(v') \\ \langle obj\rangle\, \forall v', t' \geq t, s' \sqsupseteq s.\, \mathcal{I}_m(\ell, t', s', v') \Rrightarrow \mathcal{I}_m(\ell, t', s', v') * P(v')\end{array}}{\begin{array}{l}\{\mathcal{R}(\ell, t, s, v, \mathcal{I}) * \lfloor \triangleright \mathsf{ATOM}(\ell, \mathcal{I}) \rfloor_V\} \\ \quad *^{\texttt{acq}}\ell \\ \{v'.\, P(v') * t \leq t' * s \sqsubseteq s' * \mathcal{R}(\ell, t', s', v', \mathcal{I}_r) * \lfloor \triangleright \mathsf{ATOM}(\ell, \mathcal{I}) \rfloor_V\}\end{array}}$$

$$\mathcal{W}(\ell, \mathcal{I}) * \mathcal{W}(\ell, \mathcal{I}) \Rightarrow \mathsf{False} \qquad \{\mathcal{W}(\ell)\}\, \ell :=_{\texttt{rlx}} w\, \{\mathsf{True}\} \qquad \{\mathcal{W}(\ell)\}\, \ell :=_{\texttt{rel}} w\, \{\mathsf{True}\}$$

$$\{\mathcal{R}(\ell)\}\, \mathbf{CAS}(\ell, v_1, v_2, o_f, o_r, o_w)\, \{\mathsf{True}\} \qquad \left\{\mathcal{R}^q_{\text{shr}}(\ell)\right\} *^{\texttt{rlx}}\ell\, \{\mathsf{True}\} \qquad \left\{\mathcal{R}^q_{\text{shr}}(\ell)\right\} *^{\texttt{acq}}\ell\, \{\mathsf{True}\}$$

$$\left\{\mathcal{R}^q_{\text{shr}}(\ell)\right\} \mathbf{CAS}(\ell, v_1, v_2, o_f, o_r, o_w) \left\{\mathcal{R}^q_{\text{shr}}(\ell)\right\}$$

Fig. 28. Intermediate-level rules for GPS single-writers.

6 CASE STUDY: ARC

The verification of the Arc library is by far the most challenging library verification in $\mathbf{RB}_{\texttt{rlx}}$. To make the verification go through, we needed to strengthen two atomic reads from **rlx** to **acq** in the implementations of Arc::get_mut and Arc::make_mut. We conjecture that the relaxed access in Arc::make_mut is indeed sound but verifying this would have required a significantly more complex invariant. The relaxed access in Arc::get_mut turned out to be a bug. We provide more details about this bug in §6.7.

6.1 The Core Arc library

A selection of iRC11 *cancellable single-location invariants* is given in Fig. 29. We explain these rules with the verification of Core Arc.

Arc<T>, short for *Atomically Reference Counted*, is used to share atomically an object of type T, whose mutation is disabled by default. To mutate T, one needs T to support thread-safe mutability, for example with T being an atomic type, or with T wrapped inside a lock (*e.g.*, Mutex<T>). The following Rust example instantiates Arc with an atomic integer AtomicUsize and demonstrates how Arc is typically used:

```
1  let arc1 = Arc::new(AtomicUsize::new(5)); // create the first Arc pointer
2  let arc2 = Arc::clone(&arc1);             // clone for the second pointer
3  thread::spawn(move || {                   // give arc2 to child thread
4    println!("child: {:?}", arc2.fetch_add(1, Ordering::Relaxed)); // drop(arc2);
5  });
6  println!("main: {:?}", arc1.fetch_add(2, Ordering::Relaxed));     // drop(arc1);
```

In line 1 in the main thread, a new Arc pointer arc1 is created to govern an atomic integer allocated in shared memory. The Arc's internal counter field for the number of references to the content is set to 1. An Arc pointer acts almost like its underlying content, so in line 6 we can call fetch_add on arc1 as if on the atomic integer itself. To share the content with the child thread, we create another arc2 by clone-ing arc1 (line 2), which effectively increments the internal counter

Ghost-Mod
$$\boxed{a}^{\gamma} \Leftrightarrow \Delta\boxed{a}^{\gamma} \Leftrightarrow \nabla\boxed{a}^{\gamma}$$

iRC11-CInv-New
$$\frac{q + q' = 1 \qquad \forall \tau.\, [\tau]_q * P \Rrightarrow\!\!\ast\ \mathcal{I}(v) * Q}{\ell \mapsto v * P \Rrightarrow\!\!\ast\ \exists \tau.\, [\tau]_{q'} * {}^{\tau}\boxed{\ell \mid \mathcal{I}} * Q}$$

iRC11-CInv-Tok
$$[\tau]_{q+q'} \Leftrightarrow [\tau]_q * [\tau]_{q'}$$

iRC11-CInv-FAA-Rlx
$$\frac{\forall v.\, P * \mathcal{I}(v) \Rrightarrow\!\!\ast\ \mathcal{I}(v+n) * Q(v)}{{}^{\tau}\boxed{\ell \mid \mathcal{I}} \vdash \{[\tau]_q * \Delta P\}\ \mathsf{FAA}_{\texttt{rlx}}(\ell, n)\ \{v.\, [\tau]_q * \nabla Q(v)\}}$$

iRC11-CInv-FAA-SRel
$$\frac{\forall v.\, [\tau]_q * P * \mathcal{I}(v) \Rrightarrow\!\!\ast\ \mathcal{I}(v+n) * Q(v)}{{}^{\tau}\boxed{\ell \mid \mathcal{I}} \vdash \{[\tau]_q * P\}\ \mathsf{FAA}_{\texttt{rel}}(\ell, n)\ \{v.\, \nabla Q(v)\}}$$

iRC11-CInv-Cancel
$${}^{\tau}\boxed{\ell \mid \mathcal{I}} \vdash [\tau]_1 \Rrightarrow \exists v.\, \ell \mapsto v * \mathcal{I}(v)$$

Dealloc
$$\{X \mapsto -\}\ \mathtt{free}(X)\ \{\mathsf{True}\}$$

Fig. 29. Selected iRC11 rules.

```
  new(v) := let a = alloc(2) in            drop(a) := if FAA_rel(a.counter, −1) == 1
              a.counter := 1;                            fence_acq;
              a.data := v;                               free(a, 2)
              a                            clone(a) := FAA_rlx(a.counter, 1);
deref(a) := *^na a.data                                  a
```

Fig. 30. Implementation of Core Arc.

to 2: there are now 2 pointers sharing the atomic integer. Unsurprisingly, to allow concurrent updates by multiple threads, the internal counter field is implemented with an atomic integer.

When the Arc pointers go out of scope (after lines 4 and 6), their destructors—the drop function—are called and the counter field is decremented accordingly. The last call of drop will deallocate the underlying content *and* the counter field.

The core functions of Arc are given in Fig. 30. The new function allocates two locations, one for the counter field and one for the data field, then initializes them. The deref function provides access to the data field, effectively allowing an Arc<T> to behave like its content T. The clone function does a relaxed (**rlx**) *fetch-and-add* (**FAA**) by 1 to increment counter and then return a copy of a.

Finally, the drop function does a *release* (**rel**) fetch-and-add by -1 to decrement counter. If the value of counter was 1 before the decrement (*i.e.*, this is the last drop), drop additionally does an acquire (**acq**) fence before deallocating both the counter and data fields. The acquire fence is needed because the release **FAA**, although being a release write, is only a relaxed read.

Correctness. Intuitively, the main correctness guarantee of Core Arc is that the deallocation of its data and counter fields is synchronized with (happens-after) *all* accesses to those fields. Those accesses happen between (and including) the construction of an Arc pointer, either by new or clone, and its destruction by drop. Therefore, the correctness guarantee translates to making sure that the deallocation done by the last drop is synchronized with all previous drop's. In this case, that synchronization is established between the release **FAA**'s of all previous drop's and the acquire fence of the last drop.

6.2 Setting Up the Cancellable Single-Location Invariant for Core Arc

We demonstrate the verification of the most important functions of Core Arc: new, clone and drop. For clone, we need to guarantee that any newly-created pointer arc to an object a can *non-atomically read* its data field a.data (so that the deref function can be called on arc), and perform *atomic* **FAA**'s on its counter field a.counter (so that clone and drop can be called on arc). This means that both fields must be *shared* for concurrent accesses by multiple threads.

For drop, we instead show that this sharing of the fields must have been finished before the deallocation is called. The rule DEALLOC (Fig. 29) states the requirement for dealllocating a single location X: we need to have the full ownership of X, represented by its points-to assertion $X \mapsto -$. To deallocate a block a of two locations using **free**(a, 2), the general deallocation rule requires us to have the full ownership of the whole block *i.e.*, both a.data $\mapsto -$ and a.counter $\mapsto -$.

In short, we start out with the full ownership a.data $\mapsto v$ and a.counter $\mapsto 1$ in the new function, then we share both a.data and a.counter for concurrent accesses, and at the end reclaim both a.data $\mapsto -$ and a.counter $\mapsto -$ in the last drop for deallocation. Our job is to set up the sharing to satisfy this scheme. Because the data field only needs concurrent *read* accesses, we employ *fractional* ownership [Boyland 2003] on the points-to assertion a.data $\mapsto v$. That is, we start out with the full fraction $\texttt{a.data} \mapsto v = \texttt{a.data} \xmapsto{1} v$ and for every newly-created pointer we give it a small fraction $\texttt{a.data} \xmapsto{q} v$, where $q \in (0, 1)$. Each fractional points-to assertion $\texttt{a.data} \xmapsto{q} v$ is sufficient to perform concurrent reads. When a pointer goes out of scope, its small fraction $\texttt{a.data} \xmapsto{q} v$ is recollected. Before the very end, we recollect all the small fractions into the full fraction $\texttt{a.data} \xmapsto{1} v = \texttt{a.data} \mapsto v$. Then we are ready for deallocating a.data.

The `counter` field, on the other hand, needs concurrent **FAA** accesses, so we will use iRC11 cancellable single-location invariants to share it. The cancellable invariant is also used for recollecting the small fractions of the `data` field. And now we need to understand what a iRC11 cancellable single-location invariant *is*.

Cancellable single-location invariants. The freely-duplicable assertion ${}^{\tau}\boxed{\ell \mid \mathcal{I}}$ says that the location ℓ is governed by the invariant $\mathcal{I}$ protected by the token τ. That is, $\mathcal{I}$ is only governing ℓ when the token piece $[\tau]_q$ of τ is available. A piece $[\tau]_q$, for some $q \in (0, 1]$, is called an *access* token for the invariant. As seen in some of the access rules iRC11-CInv-FAA-Rlx and iRC11-CInv-FAA-SRel—we will explain more below), a token is needed for every access to the invariant.

The predicate $\mathcal{I}$, also called the *interpretation*, is a user-defined predicate on values: If the current value of ℓ is v, $\mathcal{I}(v)$ defines what the invariant means at that value. As such, $\mathcal{I}(v)$ is a requirement that every write of value v to ℓ must provide. In reverse, a read of value v from ℓ can make use of the interpretation $\mathcal{I}(v)$. Thus, the interpretation acts as a logical communication channel between writes and reads.

The invariant for Core `Arc`*.* Using fractional ownership for the `data` field and cancellable invariant for the `counter` field of Core `Arc`, we want to prove the following simple specification:

$$\{\mathsf{True}\}\ \texttt{new}(v)\ \{\mathsf{a}.\ \exists\tau,\gamma.\ \mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\} \qquad \text{(iRC11-ARC-New)}$$

$$\{\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\}\ \texttt{clone}(\mathsf{a})\ \{\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I}) * \mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\} \qquad \text{(iRC11-ARC-Clone)}$$

$$\{\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})\}\ \texttt{drop}(\mathsf{a})\ \{\mathsf{True}\} \qquad \text{(iRC11-ARC-Drop)}$$

Here, we define an abstract predicate $\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})$ to represent the logical ownership of an `Arc` pointer:

$$\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I}) ::= \exists q.\ \texttt{a.data} \xmapsto{q} v * [\tau]_q * {}^{\tau}\boxed{\texttt{a.counter} \mid \mathcal{I}^{\gamma,v}} * \boxed{\mathsf{Count}(q)}^{\gamma} \qquad \text{(iRC11-ARC)}$$

Owning an `Arc` pointer $\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})$ means that we own: (1) some small fraction q of the `data` field $\texttt{a.data} \xmapsto{q} v$ at the value v, which allows us to safely read a.data for the value v; (2) the fact ${}^{\tau}\boxed{\texttt{a.counter} \mid \mathcal{I}^{\gamma,v}}$ that the `counter` field is governed by an invariant $\mathcal{I}$ protected by τ, as well as the access token $[\tau]_q$—with the *same* fraction q—to access the invariant, which allows us to concurrently access a.counter; and finally (3) an *unsynchronized* ghost element $\boxed{\mathsf{Count}(q)}^{\gamma}$ that represent the 1 *single* count of this pointer in the *total* count (see below).

The invariant for a.counter is defined as follows:

$$\mathcal{I}^{\gamma,v_{\mathtt{data}}}(n) ::= \begin{cases} \mathsf{False} & n < 0 \\ \boxed{\mathsf{TotalCount}(0,0)}^{\gamma} & n = 0 \\ \exists q_{\mathrm{in}}, q_{\mathrm{out}} \in (0,1).\ \texttt{a.data} \xmapsto{q_{\mathrm{in}}} v_{\mathtt{data}} * [\tau]_{q_{\mathrm{in}}} & \\ \quad * q_{\mathrm{in}} + q_{\mathrm{out}} = 1 * \boxed{\mathsf{TotalCount}(n, q_{\mathrm{out}})}^{\gamma} & n > 0 \end{cases} \qquad \text{(iRC11-ARC-Inv)}$$

First, $\mathcal{I}$ requires that the value v of the `counter` field to be non-negative. When it is positive *i.e.*, when there is some `Arc` pointers, the number of pointers is v and the invariant owns the

$$\textsc{Counting-Start}\quad q \in (0,1] \Rrightarrow\!\!\ast\; \exists\gamma.\, \boxed{\mathrm{TotalCount}(1,q)}^{\gamma} * \boxed{\mathrm{Count}(q)}^{\gamma}$$

$$\textsc{Counting-New}\quad q + q' \leq 1 \vdash \boxed{\mathrm{TotalCount}(n,q)}^{\gamma} \Rrightarrow\!\!\ast\; \boxed{\mathrm{TotalCount}(n+1,q+q')}^{\gamma} * \boxed{\mathrm{Count}(q')}^{\gamma}$$

$$\textsc{Counting-Agree}\quad \boxed{\mathrm{TotalCount}(n,q)}^{\gamma} * \boxed{\mathrm{Count}(q')}^{\gamma} \Rightarrow n \geq 1 \land 1 \geq q \geq q'$$

$$\textsc{Counting-Drop}\quad \boxed{\mathrm{TotalCount}(n+1,q+q')}^{\gamma} * \boxed{\mathrm{Count}(q')}^{\gamma} \Rrightarrow\!\!\ast\; \boxed{\mathrm{TotalCount}(n,q)}^{\gamma} \land (n = 0 \Rightarrow q = 0)$$

Fig. 31. Counting permissions for Core `Arc`.

unsynchronized ghost element $\boxed{\mathrm{TotalCount}(v, q_{\mathrm{out}})}^{\gamma}$. The element $\boxed{\mathrm{TotalCount}(v, q_{\mathrm{out}})}^{\gamma}$ tracks the globally-consistent knowledge that there are currently v pointers and the *sum* of all fractional permissions owned by those pointers is q_{out}.[1] The invariant further requires that the remaining fractions $q_{\mathrm{in}} = 1 - q_{\mathrm{out}}$ must be owned by the invariant. This includes the fractional ownership of `a.data` and the access token $[\tau]_{q_{\mathrm{in}}}$ of `a.counter`. The fraction q_{in} is in fact the *used* fraction that has been recollected by $\mathcal{I}$ from the pointers that have been `drop`-ped. Thus the invariant makes sure that any fractions of the `a.data` and τ are all accounted for. Finally, when the `counter` reaches 0, the invariant is simply trivial.

The ghost elements $\boxed{\mathrm{TotalCount}(n, q)}^{\gamma}$ and $\boxed{\mathrm{Count}(q)}^{\gamma}$ is an instance of *counting permissions* [Bornat et al. 2005], used here to track the outside fractions associated with each single count. They satisfy the axioms in Fig. 31. Counting-Start creates a ghost location γ for the first count and gives us the total count $\boxed{\mathrm{TotalCount}(1, q)}^{\gamma}$ as well as a single count $\boxed{\mathrm{Count}(q)}^{\gamma}$. With Counting-New we can increase the total count and produce more single counts. With Counting-Drop we can decrease the total count by consuming single counts. Counting-Agree ensures that every single count is always included in the total count. How this ghost construction comes into play will be revealed next section.

After this long setup, we are finally ready to demonstrate the rules of iRC11 in Fig. 29 through the verification of Core `Arc`.

6.3 Verifying `new`

In the proof of iRC11-ARC-New, we elide the standard allocation and initialization of the `data` and `counter` fields. Our main obligation here is to transform the two full ownership `a.data` $\mapsto v$ and `a.counter` $\mapsto 1$ to the abstract permission $\mathrm{ARC}^{\gamma}(\mathtt{a}, v, \tau, \mathcal{I})$ for some τ and γ. That is, turning our unique ownership into sharing mode.

To do so, we have planned to initialize iRC11 cancellable invariant for `a.counter`. The rule iRC11-CInv-New (Fig. 29) creates for the location ℓ a new cancellable invariant protected by some token τ. As a result, we get the full token $[\tau]_1$ which can be split using iRC11-CInv-Tok so that the pieces can be given to multiple threads for sharing. What we need to provide are the points-to $\ell \mapsto v$ and the interpretation $\mathcal{I}(v)$.

For `a.counter`, we do have its points-to assertion as `a.counter` $\mapsto 1$, so we only need to provide $\mathcal{I}^{\gamma,v}(1)$ for some γ. First, for γ, we use Counting-Start to create the total count and the first single count with $q ::= 1/2$. That is, we get $\boxed{\mathrm{TotalCount}(1, 1/2)}^{\gamma}$ and $\boxed{\mathrm{Count}(1/2)}^{\gamma}$. We use

[1]Here, *out* means ownership of the fractions outside of the invariant.

$\boxed{\mathsf{TotalCount}(1, 1/2)}^{\gamma}$ for $\mathcal{I}$ and $\boxed{\mathsf{Count}(1/2)}^{\gamma}$ for ARC. Similarly, we split $\mathtt{a.data} \mapsto v$ into two halves $\mathtt{a.data} \xmapsto{1/2} v$'s and use each for $\mathcal{I}$ and ARC.

With $\mathtt{a.data} \xmapsto{1/2} v$ and $\boxed{\mathsf{TotalCount}(1, 1/2)}^{\gamma}$, we only need $[\tau]_{1/2}$ for $\mathcal{I}^{\gamma,v}(1)$. Fortunately, iRC11-CInv-New allows us to use some of the token to establish $\mathcal{I}$. In our case, we need $[\tau]_{1/2}$ to complete $\mathcal{I}^{\gamma,v}(1)$. We pick $q = q' ::= 1/2$ and $P ::= \mathtt{a.data} \xmapsto{1/2} v * \boxed{\mathsf{TotalCount}(1, 1/2)}^{\gamma}$, and thus establish the cancellable invariant for $\mathtt{a.counter}$. We get as a result $[\tau]_{1/2} * {}^{\tau}\boxed{\mathtt{a.counter} \mid \mathcal{I}^{\gamma,v}}$. We combine this with the remaining $\mathtt{a.data} \xmapsto{1/2} v$ and $\boxed{\mathsf{Count}(1/2)}^{\gamma}$ to complete the first $\mathrm{ARC}^{\gamma}(\mathtt{a}, v, \tau, \mathcal{I})$ permission. □

6.4 Verifying `clone`

In proving iRC11-ARC-Clone, we need to duplicate one permision $\mathrm{ARC}^{\gamma}(\mathtt{a}, v, \tau, \mathcal{I})$ to two permissions. Unfolding the definition of $\mathrm{ARC}^{\gamma}(\mathtt{a}, v, \tau, \mathcal{I})$ (see iRC11-ARC), we see that the fractions $\mathtt{a.data} \xmapsto{q} v * [\tau]_q$ (for some q) can be split into halves *i.e.*, $\mathtt{a.data} \xmapsto{q/2} v * [\tau]_{q/2}$, each for one new ARC. The invariant assertion ${}^{\tau}\boxed{\mathtt{a.counter} \mid \mathcal{I}^{\gamma,v}}$ is freely duplicable. So we only need to transform one single count $\boxed{\mathsf{Count}(q)}^{\gamma}$ into two. To match the fraction $q/2$, we actually need two single counts of the form $\boxed{\mathsf{Count}(q/2)}^{\gamma}$. Unfortunately, $\boxed{\mathsf{Count}(q)}^{\gamma}$ is *not* splittable into two $\boxed{\mathsf{Count}(q/2)}^{\gamma}$'s. So we can only get two $\boxed{\mathsf{Count}(q/2)}^{\gamma}$'s with the help of the total count $\boxed{\mathsf{TotalCount}(-, -)}^{\gamma}$, which is inside the invariant. To do so, at the relaxed **FAA** made by `clone` (Fig. 30), we invoke the rule iRC11-CInv-FAA-Rlx.

First, the key novelty of our logic compared to previous logics is the ability to cancel a single-location invariant. Here, the iRC11-CInv-FAA-Rlx (Fig. 29) is an *access* rule to a cancellable invariant on a location ℓ. In order to safeguard the access, the rule must know that the invariant has *not* been canceled. Thus it requires such a proof from us (the invoker of the rule) in the form of the access token $[\tau]_q$ (see the precondition of the Hoare triple in iRC11-CInv-FAA-Rlx). The token $[\tau]_q$ proves that no one has used the full token $[\tau]_1$ to cancel the invariant. The rule additionally withholds the token during the access and only returns it afterwards (see the postcondition of the Hoare triple).

Second, a **FAA** is a read-modify-write (RMW) operation that has the effect of both a read and a write, and thus can make use of the interpretation $\mathcal{I}$ of the value it read for the interpretation of the value it is going to write. This is demonstrated in the premise of iRC11-CInv-FAA-Rlx. Here, v is the value read and $v + n$ is the value to be written. The rule allows us to use some of our local resource P and the interpretation of the read $\mathcal{I}(v)$ to establish the interpretation of the write $\mathcal{I}(v + n)$, and we can additionally take out any remaining resource Q. This is the standard way in RMM logics for RMW operations to communicate with other reads or RMWs. In our case, it is the way for `clone`'s to communicate about the total count (see below).

Third, in `clone`, we use a *relaxed* **FAA** which is a relaxed read and a relaxed write. Therefore in order to use our local resource P for the interpretation, P needs to be protected by a release modality: ΔP (see the precondition of the Hoare triple in iRC11-CInv-FAA-Rlx). A resource can be put under a release modality if that resource is available at the last release fence, as required by the rule Rel-fence. On the other hand, a relaxed read gives us a resource Q under the acquire modality: ∇Q (see the postcondition in iRC11-CInv-FAA-Rlx). The acquire modality can be removed by an acquire fence, as shown in the rule Acq-fence. Together the two fence rules establish the synchronization pattern of the chain "release fence → relaxed write → relaxed read → acquire fence".

Finally, if our resource is, however, view-agnostic—for example, if they are unsynchronized ghost state—then the fence modalities can be bypassed. In particular, the Ghost-Mod rule allows unsynchronized ghost states to move freely between fence modalities without using physical fences. We exploit this in our invocation of iRC11-CInv-FAA-Rlx for `clone`.

In particular, as we have $\boxed{\mathsf{Count}(q)}^{\gamma}$, we use Ghost-Mod to get $\Delta\boxed{\mathsf{Count}(q)}^{\gamma}$. Then, using our token $[\tau]_q$, we invoke iRC11-CInv-FAA-Rlx with

$P ::= \boxed{\mathsf{Count}(q)}^{\gamma}$ and $Q ::= \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$.

We now have to show that $\mathcal{I}^{\gamma,v}(v') * \boxed{\mathsf{Count}(q)}^{\gamma} \Rrightarrow\!\!\ast\ \mathcal{I}^{\gamma,v}(v'+1) * \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$, where v' is the value the **FAA** reads from `a.counter`. That is, we need to transform the resource $\mathcal{I}^{\gamma,v}(v') * \boxed{\mathsf{Count}(q)}^{\gamma}$ into $\mathcal{I}^{\gamma,v}(v'+1) * \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$.

First, by the definition of $\mathcal{I}^{\gamma,v}(v')$ (see iRC11-ARC-Inv), we know that $v' \geq 0$. By owning $\boxed{\mathsf{Count}(q)}^{\gamma}$, we also know that v' cannot be 0, because if $v' = 0$, we can combine $\boxed{\mathsf{TotalCount}(0,0)}^{\gamma}$ with $\boxed{\mathsf{Count}(q)}^{\gamma}$ and use the rule Counting-Agree to derive the contradiction that $0 \geq 1$. Thus $v' > 0$.

Now, we are not going to change the fractions ($q_{\mathrm{in/out}}$) and the fractional ownerships: we will keep them the same (*i.e.*, *framing*) for $\mathcal{I}^{\gamma,v}(v'+1)$. Therefore our job is simply transform $\boxed{\mathsf{TotalCount}(v', q_{\mathrm{out}})}^{\gamma} * \boxed{\mathsf{Count}(q)}^{\gamma}$ to $\boxed{\mathsf{TotalCount}(v'+1, q_{\mathrm{out}})}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$. This is simple: We first use Counting-Drop to drop the single count $\boxed{\mathsf{Count}(q)}^{\gamma}$ associated with q and get $\boxed{\mathsf{TotalCount}(v'-1, q_{\mathrm{out}}-q)}^{\gamma}$. We then call Counting-New twice on $\boxed{\mathsf{TotalCount}(v'-1, q_{\mathrm{out}}-q)}^{\gamma}$, each time creating a new single count $\boxed{\mathsf{Count}(q/2)}^{\gamma}$ and in the end we get back $\boxed{\mathsf{TotalCount}(v'+1, q_{\mathrm{out}})}^{\gamma}$. Note that we always satisfy the side condition of Counting-New because $q_{\mathrm{out}} \leq 1$.

Finally, after the access, we get back the access token $[\tau]_q$ and two single counts:

$$\nabla Q = \nabla\left(\boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}\right)$$

Since the single counts are unsynchronized ghost state, we use Ghost-Mod to get $\boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$. Now we can split the token $[\tau]_q$ and the fraction ownership $\texttt{a.data} \overset{q}{\mapsto} v$ into two halves and gain two $\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})$'s. □

6.5 Verifying `drop`

The first intuition in the proof of `drop` is that, if the `drop` is not the last drop, we will return all the resources of the current pointer $\mathrm{ARC}^{\gamma}(\mathsf{a}, v, \tau, \mathcal{I})$ to the invariant. This includes the fractional ownership $\texttt{a.data} \overset{q}{\mapsto} v$, the access token $[\tau]_q$ and the single count element $\boxed{\mathsf{Count}(q)}^{\gamma}$. The former two will be stored in the invariant and will be transferred to the last `drop` for deallocation. The single count element will be used to decrease the total count by 1.

The second intuition is that, in the case of the last `drop`, we know from the ARC permission and the invariant that the local fraction and the fractions stored in the invariant sum up to 1, so we can recollect the full fraction for dealloction.

In both cases, we need a stronger rule for *release* **FAA** that allow us to use the token $[\tau]_q$ to access the invariant and *simultaneously* use the token to establish the interpretation of the invariant. This is supported in the rule iRC11-CInv-FAA-SRel. The difference with iRC11-CInv-FAA-Rlx is that in the premise we can additionaly use $[\tau]_q$ to reestablish $\mathcal{I}(v+n)$. Consequently, we would not regain $[\tau]_q$ in the postcondition of the rule. Note that this rule is only sound for a release **FAA**, and thus we can use our local resource P without using a release fence.

Now, at the release **FAA** of `drop`, using $[\tau]_q$, we invoke iRC11-CInv-FAA-SRel with the following P and Q.

$$P ::= \mathtt{a.data} \overset{q}{\mapsto} v * \boxed{\mathsf{Count}(q)}^{\gamma}$$

$$Q(v') ::= \begin{cases} \mathsf{True} & v' \neq 1 \\ \mathtt{a.data} \mapsto v * [\tau]_1 & v' = 1 \end{cases}$$

We then have to prove that $[\tau]_q * P * \mathcal{I}^{\gamma,v}(v') \Rrightarrow \mathcal{I}^{\gamma,v}(v'-1) * Q(v')$ where v' is the old value of `a.counter`. Similarly to the reasoning in `clone`, with $\boxed{\mathsf{Count}(q)}^{\gamma}$ from P, we know that $v' > 0$ and the invariant has some fractional permissions $[\tau]_{q_{\text{in}}}$ and $\mathtt{a.data} \overset{q_{\text{in}}}{\longmapsto} v$ for some q_{in} (see iRC11-ARC-Inv).

Now, if this is not the last `drop` *i.e.*, $v'-1 > 0$, we need to re-establish $\mathcal{I}^{\gamma,v}(v'-1)$ with some new fractions $q'_{\text{in/out}}$. We pick them as follows: $q'_{\text{in}} ::= q_{\text{in}} + q$ and $q'_{\text{out}} ::= q_{\text{out}} - q$. From $[\tau]_q * P * \mathcal{I}^{\gamma,v}(v')$, we can easily get $[\tau]_{q'_{\text{in}}} = [\tau]_{q_{\text{in}}} * [\tau]_q$ and $\mathtt{a.data} \overset{q'_{\text{in}}}{\longmapsto} v = \mathtt{a.data} \overset{q_{\text{in}}}{\longmapsto} v * \mathtt{a.data} \overset{q}{\mapsto} v$, which are needed for $\mathcal{I}^{\gamma,v}(v'-1)$. Our remaining work is to transform $\boxed{\mathsf{TotalCount}(v', q_{\text{out}})}^{\gamma} * \boxed{\mathsf{Count}(q)}^{\gamma}$ to $\boxed{\mathsf{TotalCount}(v'-1, q'_{\text{out}})}^{\gamma}$. Fortunately, this is but a simple application of Counting-Drop. Then we are done because $v' \neq 1$.

In the case where this is the last `drop`'s **FAA**, we have $v' = 1$ and we must prove $[\tau]_q * P * \mathcal{I}^{\gamma,v}(1) \Rrightarrow \mathcal{I}^{\gamma,v}(0) * Q(1)$. From $\mathcal{I}^{\gamma,v}(1)$ we have $\boxed{\mathsf{TotalCount}(1, q_{\text{out}})}^{\gamma}$ and from P we have $\boxed{\mathsf{Count}(q)}^{\gamma}$. By an application of Counting-Drop, we have $\boxed{\mathsf{TotalCount}(0,0)}^{\gamma}$, which is exactly $\mathcal{I}^{\gamma,v}(0)$, and additionaly the fact that $q_{\text{out}} = q$. From $\mathcal{I}^{\gamma,v}(1)$ we also know that $q_{\text{in}} + q_{\text{out}} = q_{\text{in}} + q = 1$. Thus combining what we have left from our assumption $[\tau]_q * P * \mathcal{I}^{\gamma,v}(1)$, we have $Q(1) = \mathtt{a.data} \mapsto v * [\tau]_1$. So we finish the last `drop`'s **FAA** and gain $\nabla Q(1)$.

As the return value is $v' = 1$, we perform an acquire fence (see the code of `drop` in Fig. 30). Thanks to the acquire fence rule Acq-fence, we remove the modality and regain $Q(1) = \mathtt{a.data} \mapsto v * [\tau]_1$. We are almost done: We only need to get back the points-to ownership of `a.counter`. For this we *cancel* the invariant for `a.counter` using the cancellation rule iRC11-CInv-Cancel. The rule requires the full token $[\tau]_1$, which we do have, to ensure that the cancellation happens after all accesses to the invariant. At long last, after the cancellation we now have the full ownership of both fields and can safely use Dealloc to free them. □

6.6 The Full APIs of `Arc`

We discuss the verification of an extended version of `Arc`, which is also the version we have verified in $\mathbf{RB}_{\mathtt{rlx}}$. Its most interesting APIs are given in Fig. 32. Here we need to tackle two extra sets of behaviors, presented as two following challenges.

`Arc<T>` *has a subordinate type* `Weak<T>`. The first challenge involves a type called `Weak<T>`. `Weak` itself is a variant of `Arc`: it has a counter to count how many `Weak` pointers are in existence, and also has the similar `clone` and `drop` functions (Fig. 32). However, `Weak` does not guarantee access to the underlying object of type `T`: while owning an `Arc` guarantees that the object is still available, owning a `Weak` does not prevent the object to be reclaimed. In order to access the object with a `Weak` pointer, one first calls `Weak::upgrade` to obtain an `Arc` pointer. `upgrade` can fail when the object has already been reclaimed, that is when there is no `Arc` pointer left. A `Weak` pointer are typically created by calling `Arc::downgrade` on a shared reference of `Arc`.

The challenge for verifying `Arc` and `Weak` in the relaxed memory setting is that they involve two tightly coupled atomic locations—one for each counter. As multi-location invariants are in general

Arc	Weak
`new: fn(T) -> Arc<T>`	`new: fn() -> Weak<T>`
`deref: fn(&Arc<T>) -> &T`	
`clone: fn(&Arc<T>) -> Arc<T>`	`clone: fn(&Weak<T>) -> Weak<T>`
`downgrade: fn(&Arc<T>) -> Weak<T>`	`upgrade: fn(&Weak<T>) -> Option<Arc<T>>`
`drop: fn(Arc<T>) -> ()`	`drop: fn(Weak<T>) -> ()`
`get_mut: fn(&mut Arc<T>) -> Option<&mut T>`	
`make_mut: fn(&mut Arc<T>) -> &mut T`	

Fig. 32. An excerpt of Rust's `Arc<T>` and `Weak<T>` APIs.

unsound for RMM, we need to use separate iRC11 protocols for each counter and at the same time maintain their relation. This is a known challenge, as has been observed by GPS [Turon et al. 2014]. The general solution is to construct ghost state to encode the relation between the locations and prevent their protocols from breaking the relation. We were able to set up several unsynchronized ghost state constructions to encode the relation, but those, unfortunately, are not enough.

`Arc<T>` *supports temporary borrows of the underlying content.* The second challenge involves the support to temporarily reclaim full ownership of the underlying content when the thread knows it owns the last unique `Arc` and `Weak` pointers. The functions `Arc::get_mut` and `Arc::make_mut` provide these capabilities: they return a mutable reference `&mut T` to the underlying content. The reclamation is temporary because when the reference goes out of scope (when the lifetime of the mutable reference ends), the content is returned and the original `Arc` pointer can be used again.

The challenge here is to guarantee that if the temporary reclamation is successful, it is synchronized with *all* accesses to the content of type `T`. Again, note that those accesses can only happen between the construction and the destruction of an `Arc` pointer. How an `Arc` pointer can be constructed is now more complicated than that of Core `Arc`: an `Arc` pointer can now additionally be created by `upgrade`-ing from a `Weak` pointer. Therefore, to establish the synchronization guarantee, we now need to handle the intertwined life-cycles of `Arc` and `Weak` pointers.

To be more concrete, let us look at the implementation of `get_mut` (Fig. 33). To return temporary full ownership of the `data` field, the function checks that the thread owns the unique `Arc` and `Weak` pointers in two steps, using `is_unique`.

First, it acquires a "lock" on the `Weak` counter—`a.weak`—to make sure that there is no other `Weak` pointers. This is done by an acquire compare-and-swap (**CAS**) from 1 to −1. The function uses −1 as the "locked" value to resolve conflicts with other contentious `Arc::get_mut` or `Arc::downgrade` calls. If the **CAS** succeeds, the thread knows that there is no `Weak` pointers left, but there may exist still some `Arc` pointers. This comes from the agreed contract between the counters: the `Weak` counter *implicitly* counts 1 for *all* `Arc` pointers. So when the thread still owns an `Arc` pointer, and the value of the `Weak` counter is exactly 1, that 1 must be accountable for the remaining `Arc` pointers, and there is no `Weak` pointers left.

Second, it does an acquire read on the `Arc` counter—`a.strong`—and then checks if the value read is 1. If that value is 1, `is_unique` succeeds and `get_mut` concludes that thread owns the unique `Arc` pointer, and gives the thread temporary full access to the underlying content with type `&mut T`.[2]

[2]The `Arc::make_mut` function also follows the similar pattern, but the targets are reversed: it first acquires a "lock" on the `Arc` counter and then reads the `Weak` counter.

```
fn is_unique(&mut self) -> bool {
  // lock the weak pointer count if we appear to be the sole weak pointer holder.
  if self.inner().weak.compare_exchange(1, usize::MAX, Acquire, Relaxed).is_ok() {
    let unique = self.inner().strong.load(Relaxed) == 1;

    self.inner().weak.store(1, Release); // release the lock
    unique
  } else { false }
}
fn get_mut(this: &mut Self) -> Option<&mut T> {
  if this.is_unique() {
    unsafe { Some(&mut this.ptr.as_mut().data) }
  } else { None }
}
fn drop(&mut self) {
  if self.inner().strong.fetch_sub(1, Release) != 1 {
    return;
  } ...
}
```

Fig. 33. Rust's implementation (excerpt) of `Arc::get_mut` and `Arc::drop`.

No matter if the second check fails or not, `is_unique` will release the lock on the `Weak` counter with a release write of value 1.

Correctness. The two checks by `is_unique` ensure the synchronization guarantee for temporary reclamation. The second check ensures that the thread is synchronized with all *other* `Arc::drop` calls. This means that it is synchronized with all accesses to the content made by all other `Arc` pointers. The thread, of course, must have synchronized with all accesses made by the current `Arc` pointer that it owns. Consequently, the thread must have synchronized with all accesses to the underlying content.

The problem, however, is that the second check uses an acquire *read*, instead of a **CAS**. If it were a **CAS**, then we are guaranteed to read the latest value of the `Arc` counter, and thus synchronizing with all other `Arc::drop`'s. However, an acquire read does not guarantee reading the latest value: it can read a stale one. Consider a truncated history of the `Arc` counter in Fig. 34, where our call to `get_mut` was initiated somewhere before the latest write 1(c) to the counter. Since we do not know exactly when `get_mut` was initiated, the second check by `is_unique` may read 1 from any events 1(a), 1(b) or 1(c). Had it read from 1(a), we would not have synchronized with the `Arc::drop`'s or `downgrade`'s after that. Our obligation here is to show that if the second check read 1, it must have read from 1(c).

By contradiction, we show that it is impossible to read 1 from 1(a), 1(b) or any stale 1 values. Put it another way, we show that the thread has observed all updates to the `Arc` counter from a stale 1 to 2, denoted as stale($1 \rightsquigarrow 2$), and therefore cannot read those stale 1's again. This is where the first check comes into play: it gives us the guarantee that the thread has observed all stale($1 \rightsquigarrow 2$) updates. Note that these updates come either from an `Arc::clone` or from a `Weak::upgrade`. If the update is from an `Arc::clone`, like in 1(a), the thread must have observed it because that update must have been performed by some `Arc` pointer—unique at that time—of which the current `Arc` pointer (which this thread owns) is a *descendant*.

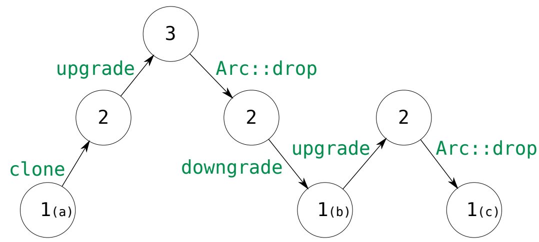

Fig. 34. A truncated history of the Arc counter.

The remaining case is when the update is from a Weak::upgrade, like in 1(b). By the first check the thread is synchronized with all Weak::drop's by *all* Weak pointers. Note that Weak::drop, similar to Core Arc::drop (Fig. 30), does a release **FAA** to decrement the Weak counter. However, unlike in Core Arc, the last Weak::drop decrements the counter to 1 (instead of 0). Therefore, when the first check did a successful acquire **CAS** for value 1 on the Weak counter, it knows that there is no Weak pointers left and it is synchronized with all Weak::drop's.

If an update stale$(1 \rightsquigarrow 2)$ is from an Weak::upgrade, it must happen-before the Weak::drop of the same Weak pointer. Thus, by synchronizing with all Weak::drop's, the thread is guaranteed to synchronize with all stale$(1 \rightsquigarrow 2)$ updates from Weak::upgrade's. It follows that the thread must have read the latest write to the Arc counter.

Another instance of synchronized ghost state. Thus, our challenge here pins down to formalizing the observations of stale$(1 \rightsquigarrow 2)$ and the two sources of those observations. Furthermore, the observations are tied to the ownership of some Arc or Weak pointer, and when such ownership is transferred the observations must also be transferred in a *synchronized* way.

For this purpose, we use an instance of synchronized ghost state for those observations. Similar to the ghost state for raw cancellable invariants, we use the ghost state of form $\boxed{\circ\,(q, O)}^{\gamma}$ where O is a set of observations. In the particular case of Arc, an observation is simply a unique identifier *id* for each stale$(1 \rightsquigarrow 2)$ update event on the Arc counter. The iRC11 logic therefore must additionally provide unique identifiers for update events. In the implementation of the logic we simply expose the *timestamp* of an update/write event as its identifier in the protocol assertion. Using the timestamps as identifiers, we thus tie the logical ghost state with the write events through the protocol assertion, making the observations actually *physical* and therefore can only be transferred with physical synchronization.

In the verification of Arc, we use two different constructions: one, $\boxed{\circ\,(q, O_u)}^{\mu}$, to track the observations coming from Weak::upgrade, and another, $\boxed{\circ\,(q, O_c)}^{\gamma}$, to track those coming from Arc::clone. The former construction $\boxed{\circ\,(q, O_u)}^{\mu}$ enjoys similar properties to that of raw cancellable invariants. That is, the observations can be joined (using set union), and if we own the full fraction $\boxed{\circ\,(1, O_u)}^{\mu}$, then we are guaranteed that O_u contains all possible Weak::upgrade's stale$(1 \rightsquigarrow 2)$ events and we have *physically* seen them all. Additionally, each owner of each fraction q can concurrently add observations to its local set O. This is to reflect the fact that any Weak pointer can always perform a stale$(1 \rightsquigarrow 2)$ event.

The latter construction $\boxed{\circ\,(q, O_c)}^{\gamma}$ is a bit different. Even if we only own a fraction $\boxed{\circ\,(q, O_c)}^{\gamma}$, we need to know that O_c contains all possible Arc::clone's stale$(1 \rightsquigarrow 2)$ events and we have *physically* seen all of them. Furthermore, we can only add observations to O_c if we have the full fraction $\boxed{\circ\,(1, O_c)}^{\gamma}$. This reflects the fact that any Arc pointer must have seen all Arc::clone's

$1 \rightsquigarrow 2$ updates, and that any `Arc::clone`'s $1 \rightsquigarrow 2$ update can only be done by the one `Arc` pointer that was unique and should own the full fraction at the time of the update.

We then set up that the abstract predicate ARC for ownership of `Arc` pointers also contains a fraction $\boxed{\circ(q, O_c)}^{\gamma}$ for some q (the same q in $[\tau]_q$ and $\texttt{a.data} \overset{q}{\mapsto} -$, see IRC11-ARC) and O_c (because only `Arc` pointers can do `Arc::clone`), and that the abstract predicate WEAK for ownership of `Weak` pointers contains a fraction $\boxed{\circ(q, O_u)}^{\mu}$ for some q and O_u (because only `Weak` pointers can do `Weak::upgrade`). We further require that `Arc::drop` also releases the fraction $\boxed{\circ(q, O_c)}^{\gamma}$ like releasing the other fractions, and similarly that `Weak::drop` releases $\boxed{\circ(q, O_u)}^{\mu}$.

With that setup, we are ready to show that when the two checks of `is_unique` succeed, the thread must have observed all stale($1 \rightsquigarrow 2$) updates. First, when acquiring the "lock" on the `Weak` counter, the thread also acquire the full fraction $\boxed{\circ(1, O_1)}^{\mu}$ from the `Weak` counter protocol. The full fraction is available in the protocol because all `Weak` pointers have been `drop`-ped. With $\boxed{\circ(1, O_1)}^{\mu}$, the thread is guaranteed to have seen all `Weak::upgrade`'s stale($1 \rightsquigarrow 2$) updates. Second, since the thread owns an `Arc` pointer, it owns a fraction $\boxed{\circ(q, O_2)}^{\gamma}$, which guarantees that the thread has seen all `Arc::clone`'s stale($1 \rightsquigarrow 2$) updates. Consequently, the thread must have read 1 from the latest write to the `Arc` counter, and thus is synchronized with all previous accesses to the underlying content `T`. □

6.7 Insufficient Synchronization in `get_mut`

Unfortunately, our setup was not strong enough to verify `Arc` and `Weak` without change. The two reads of the counters in the second check of `get_mut` and `make_mut` were **`rlx`** in the original code (line 4, Fig. 33), and we had to strengthen them both to **`acq`** in order to make the verification go through. The reason is that, while we managed to temporarily get the full resources out by a read, the **`rlx`** reads do not give us those resources in the current view (they are under a ∇ modality). While we conjecture that a **`rlx`** read in `make_mut` is in fact sufficient, a **`rlx`** read in `get_mut` turned out to be insufficient and we have reported the bug and the fix has been merged into Rust codebase. The following example invokes a data race when using `get_mut`:

```
1  let mut arc1 = Arc::new(0);
2  let     arc2 = Arc::clone(&arc1);
3  thread::spawn( move || { let _ : u32 = *arc2; /* drop(arc2); */ } );
4  loop { match Arc::get_mut(&mut arc1) {
5           None => {}
6           Some(m) => { *m = 1u32; return; }}}
```

In this example there are two non-atomic operations: the read of the underlying integer in line 3 (child thread) and the write to the same integer in line 6 (parent thread). The read should be safe because the child thread owns `arc2`, thus the underlying integer is shared and *immutable*. The write should be safe because `get_mut` guarantees that the parent thread owns the unique `Arc` pointer (`arc1`) and should temporarily gain full access to the non-atomic integer. This can only happens after the child thread finishes and `arc2` has been dropped. However, the two non-atomic operations constitute a data race by C11 standard, because neither one happens-before the other. More specifically, in line 3 of the child thread, when `arc2` goes out of scope, it will be destructed by `Arc::drop`, which uses a release (**`rel`**) RMW (see the code at line 16, Fig. 33). This release RMW will be read by `get_mut` (line 4, Fig. 33) in the parent thread (line 4). If this read had been **`acq`**, then there would have been a release-acquire synchronization between the release RMW of `drop` and the acquire read of `get_mut`, and the non-atomic read of the child thread would have been guaranteed to happen-before the non-atomic write of the parent thread. However, the read was **`rlx`**, thus no happen-before relationship can be established between the two non-atomic operations.

REFERENCES

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. (2005), 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking interference with fractional permissions. In *SAS (LNCS)*. https://doi.org/10.1007/3-540-44898-5_4

Marko Doko and Viktor Vafeiadis. 2016. A program logic for C11 memory fences. In *VMCAI (LNCS)*. Springer, 413–430.

Marko Doko and Viktor Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *ESOP*.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language – Technical appendix and Coq development. https://plv.mpi-sws.org/rustbelt/popl18/.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP (LIPIcs)*. 17:1–17:29.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, 691–707.