Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris*

Technical report MPI-SWS-2017-###

Jan-Oliver Kaiser
janno@mpi-sws.org

Hoang-Hai Dang†
haidang@mpi-sws.org

Derek Dreyer
dreyer@mpi-sws.org

Ori Lahav
orilahav@mpi-sws.org

Viktor Vafeiadis
viktor@mpi-sws.org

November 20, 2017

DRAFT IN PROGRESS

This technical report fleshes out the encoding of two weak memory program logics —GPS and RSL—in Iris. We discuss many technical problems that closely follow the Coq development but were not mentioned in the original paper [1]. Specifically, we present the full RA+NA operational semantics, the model and soundness proof of the base logic, and models and proofs of both iRSL and iGPS. We also present how extensions to GPS and RSL are built upon those models. Finally, we show how these extensions can simplify proofs of many examples. This report also aims to provide a helpful reference for ones who wish to use Iris to construct their logics.

*This report accompanies our original paper published in ECOOP'17, available at [1].

†Hai is responsible for the wording of this report, as a writing exercise. Please direct any complaint or suggestion to him.

Contents

1 Introduction

Mechanizing formalization is hard, and trying to understand other people's mechanized formalizations is even harder, as it is almost impossible to quickly get a high-level intuition by diving into thousands of lines of code written in a possibly different style from your own. It is a myth that one can successfully go through those developments without the original authors' help. This report aims to fill this gap for our paper [1] on the encodings of weak memory program logics RSL and GPS in Iris. Due to lack of space, many interesting details only live in the Coq development and were not mentioned in the paper. These include:

- the full Release-Acquire operational semantics with allocation and deallocation
- the full model and soundness proof of the base logic
- the models and soundness proofs of both iRSL and iGPS
- extensions of iRSL and iGPS and their proofs
- proofs of examples, which demonstrate the power of the extensions, in comparison with the origin proofs.

This report provides an intuitive documentation for all of these points, as well as for all the techniques employed to formalize them in Iris in Coq. It may therefore be beneficial to those who wish to use Iris to construct their logics. We explain the full operational semantics in §2, and how to instantiate Iris with it to get the base logic in §3. In §4, we present *view-monotone predicates*, which are the basic building blocks for all iRSL and iGPS assertions. In §5, we diverge a bit to present *persistor* and *fractor*—two useful constructs that we employ to derive different instances for some assertions. Next, in §6, we show how the points-to assertions and rules for non-atomic accesses can be built in a direct way from the base logic. We also demonstrate the application of the fractor construct to create fractional points-to assertions. In §7 and §8, we define the iRSL and iGPS assertions, respectively, and prove soundness of all the rules. These includes all extensions to RSL and GPS, which depend on the persistor and fractor constructs. The most important extension is the single-writer protocols. Finally in §9, we show the proofs of all the examples we have formalized.

2 RA+NA operational semantics

2.1 RA+NA machine

The intuition of the operational semantics comes from the observation that different threads have different view of the memory. We thus have to keep track of pass write events so that some threads can still access them. Moreover, write events to the same location must follow a total order enforced by C11 call *modification order* (*mo* for short). Finally, we need to keep track of each thread's progress on each location's *mo*, which decides which writes to the location the thread can read, and where its new writes may end up.

For the *mo* order, the RA+NA machine manages for each location a totally ordered set of timestamps $t \in \text{Time} \triangleq \mathbb{N}$. Each write of some value ν to a location ℓ gets assigned a new timestamp (unique for ℓ), resulting in a write event $\omega \in \text{Event} \triangleq \text{Loc} \times \text{ADVal} \times \text{Time}$, where $\nu \in \text{ADVal} \triangleq \mathbb{Z} \cup \{A, D\}$. The values A and D are used to mark allocation and deallocation events respectively. A keen reader may have noticed that our machine only accept simple integers, not tuples or functions, as values. Using timestamps, the thread's "progress" is represented by a *view*, $V \in \text{View} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Time}$, which records the timestamp of the most recent write event observed by the thread for every location. To enable communication between threads, every write event is augmented with the writing thread's view (including the timestamp of the write), yielding a message $m \in \text{Msg} \triangleq \text{Event} \times \text{View}$.

The machine state σ comprises a message pool M (called *memory*) and a thread-view map T that tracks the view of each thread. To detect data races during the execution of a program, we add an additional component to the physical state: the *non-atomic view* N, which tracks the timestamp of the most recent non-atomic write to every location.

For the reduction steps of the semantics, we need to track the operation's memory event $\varepsilon \in \mathcal{E}$, which can be read, write, update or fork. Updates are always release write and acquire read, while a normal read or write can have access mode $\alpha \in \text{Access}$, which can be non-atomic or atomic. An atomic read is always an acquire event, while an atomic write a release one.

The types of all components are summarized in Figure 1.

Variable name	Type	Definition
$\ell \in$	Loc	$\triangleq \mathbb{N}$
$t \in$	Time	$\triangleq \mathbb{N}$
$\nu \in$	ADVal	$\triangleq A \mid D \mid z \in \mathbb{Z}$
$V \in$	View	$\triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Time}$
$m \in$	Msg	$\triangleq \text{Loc} \times \text{ADVal} \times \text{Time} \times \text{View}$
$\pi, \rho \in$	ThreadId	$\triangleq \mathbb{N}$
$\sigma, (M, T, N) \in$	Σ	$\triangleq \mathcal{P}(\text{Msg}) \times (\text{ThreadId} \xrightarrow{\text{fin}} \text{View}) \times \text{View}$
$\alpha \in$	Access	$\triangleq \text{na} \mid \text{at}$
$\varepsilon \in$	$\mathcal{E}$	$\triangleq \langle \text{Read}_\alpha, \ell, \nu \rangle \mid \langle \text{Write}_\alpha, \ell, \nu \rangle \mid \langle \text{Update}, \ell, \nu_o, \nu_n \rangle \mid \langle \text{Fork}, \rho \rangle$

Figure 1: Types of RA+NA machine's components (`lang/types.v`).

2.1.1 Machine state wellformedness

We only consider wellformed states. The reduction steps (see below) always preserve wellformedness. Wellformedness enforces the following restrictions on the components (M, T, N) of the machine state:

- `nats_ok`: all timestamps of the non-atomic view N must be justified by a message in M.
- `threads_ok`: similarly, all timestamps of a thread-view in T must be justified by a message in M.
- `msgs_ok`: any message in M must include its write timestamp in its view, whose all timestamps also must be justified by M.
- `alloc_inv` and `dealloc_inv`: any message must be compatible with the location's history of allocations and deallocations before it. This means that (1) an allocation must be the first event, or must immediately follow a deallocation; (2) a normal-value event (*i.e.,* a write with $v \in \mathbb{Z}$) is only possible if the previously closest non-normal-value event is an allocation; and (3) a deallocation is only possible if the previously closest non-normal-value event is an allocation. Note that although this condition allows the reuse of deallocated locations by re-allocating them, we do not exploit this ability. For the sake of simplicity, we maintain that an allocation will always pick a fresh location. These two invariants are just relics from the initial development, the kind of which the reader needs not pay to much attention.
- `pairwise_disj`: the memory only contains *pairwise disjoint* messages, which requires that two messages that have the same location and timestamp must be exactly the same.

Machine state wellformedness (`lang/machine.v`) $\boxed{\texttt{phys\_inv}(\sigma)}$

$$
\begin{aligned}
\texttt{view\_ok}(M, V) &\triangleq \forall \ell, t.\ V(\ell) = t \Rightarrow \exists m.\ m \in M \land m.\mathrm{loc} = \ell \land m.\mathrm{time} = t \land m.\mathrm{view} \sqsubseteq V \\
\texttt{nats\_ok}(M, N) &\triangleq \forall \ell, t.\ N(\ell) = t \Rightarrow \exists m.\ m \in M \land m.\mathrm{loc} = \ell \land m.\mathrm{time} = t \\
\texttt{threads\_ok}(\sigma) &\triangleq \forall \pi, V.\ \sigma.T(\pi) = V \Rightarrow \texttt{view\_ok}(\sigma.M, V) \\
\texttt{msg\_ok}(m) &\triangleq m.\mathrm{view}(m.\mathrm{loc}) = m.\mathrm{time} \\
\texttt{msgs\_ok}(M) &\triangleq \forall m \in M.\ \texttt{msg\_ok}(m) \land \texttt{view\_ok}(M, m.\mathrm{view}) \\
\texttt{phys\_inv}(\sigma) &\triangleq \texttt{nats\_ok}(\sigma.M, \sigma.N) \land \texttt{threads\_ok}(\sigma) \land \texttt{msgs\_ok}(\sigma.M) \\
&\quad \land \texttt{alloc\_inv}(\sigma.M) \land \texttt{dealloc\_inv}(\sigma.M) \land \texttt{pairwise\_disj}(\sigma.M)
\end{aligned}
$$

2.1.2 Per-thread reduction

The λ_{RN} language's reduction relation is factored into the *expression reduction*, concerned with the evaluation of the language's expressions, and the *machine reduction*, concerted with how the execution of an expression affects the machine state. We will define the complete reduction relation later, after defining the expression reduction. In this sub-section we define the *per-thread* machine reduction relation (Figure 2).

The key difference between the definition of the operational semantics actually used to instantiate Iris and the definition given in the paper [1] is the *bad states*: $\bot_{\mathrm{race}}$ and $\bot_{\mathrm{uninit}}$. In the current Coq development, we do not have bad states in the operational semantics: we simply model that, if the operational semantics is going to step to a bad state, it just gets stuck *i.e.,*

Basic thread reduction (`thread_red`) $\boxed{(M, V) \xrightarrow{\varepsilon} (M', V')}$

$$\textsc{Thread-Read}\ \frac{(\ell, \nu, t, V_o) \in M \qquad V(\ell) \le t}{(M, V) \xrightarrow{\langle \mathrm{Read}_\alpha, \ell, \nu\rangle} (M, V \sqcup V_o)} \qquad \textsc{Thread-Write}\ \frac{\neg\exists \nu_x, V_x.\ (\ell, \nu_x, t, V_x) \in M \qquad V(\ell) < t \qquad V' = V[\ell \mapsto t]}{(M, V) \xrightarrow{\langle \mathrm{Write}_\alpha, \ell, \nu\rangle} (M \cup \{(\ell, \nu, t, V')\}, V')}$$

$$\textsc{Thread-Update}\ \frac{(\ell, \nu_o, t, V_o) \in M \qquad V(\ell) \le t \qquad \neg\exists \nu_x, V_x.\ (\ell, \nu_x, t+1, V_x) \in M \qquad V' = V[\ell \mapsto t+1] \sqcup V_o}{(M, V) \xrightarrow{\langle \mathrm{Update}, \ell, \nu_o, \nu_n\rangle} (M \cup \{(\ell, \nu_n, t+1, V')\}, V')}$$

Data-race-freedom reduction (`drf_red`) $\boxed{(M, N) \xrightarrow{\varepsilon}{}^{V} (M', N')}$

$$\textsc{DRF-Non-NA}\ \frac{\varepsilon \in \{\langle \mathrm{Read}_\alpha, \ell, \_\rangle, \langle \mathrm{Write}_{\mathrm{at}}, \ell, \_\rangle, \langle \mathrm{Update}, \ell, \_, \_\rangle\} \qquad N(\ell) \le V(\ell) \\ \varepsilon = \langle \mathrm{Read}_{\mathrm{na}}, \_, \_\rangle \Rightarrow \forall \nu', t', V'.(\ell, \nu', t', V') \in M \Rightarrow t' \le V(\ell)}{(M, N) \xrightarrow{\varepsilon}{}^{V} (M', N)}$$

$$\textsc{DRF-Write-NA}\ \frac{N(\ell) \le V(\ell) \qquad \forall \nu', t', V'.\ (\ell, \nu', t', V') \in M' \Rightarrow t' \le t}{(M, N) \xrightarrow{\langle \mathrm{Write}_{\mathrm{na}}, \ell, \nu\rangle}{}^{V} (M', N[\ell \mapsto t])}$$

Allocation reduction $\boxed{\texttt{alloc\_red}(M, V, \varepsilon)}$

$$\textsc{Alloc-Read}\ \frac{\mathrm{initialized}(M, \ell, V(\ell)) \qquad v \in \mathbb{Z}}{\texttt{alloc\_red}(M, V, \langle \mathrm{Read}_\alpha, \ell, v\rangle)} \qquad \textsc{Alloc-Write}\ \frac{\mathrm{allocated}(M, \ell, V(\ell)) \qquad v \in \mathbb{Z}}{\texttt{alloc\_red}(M, V, \langle \mathrm{Write}_\alpha, \ell, v\rangle)}$$

$$\textsc{Alloc-Update}\ \frac{\mathrm{initialized}(M, \ell, V(\ell)) \qquad v_o, v_n \in \mathbb{Z}}{\texttt{alloc\_red}(M, V, \langle \mathrm{Update}, \ell, v_o, v_n\rangle)}$$

$$\textsc{Alloc-Alloc}\ \frac{\ell \text{ is fresh in } M}{\texttt{alloc\_red}(M, V, \langle \mathrm{Write}_{\mathrm{na}}, \ell, A\rangle)} \qquad \textsc{Alloc-Dealloc}\ \frac{\mathrm{allocated}(M, \ell, V(\ell))}{\texttt{alloc\_red}(M, V, \langle \mathrm{Write}_{\mathrm{na}}, \ell, D\rangle)}$$

Per-thread machine reduction (`machine_red`) $\boxed{(M, T, N) \xrightarrow{\varepsilon}{}^{\pi} (M', T', N')}$

$$\textsc{Machine-Fork}\ \frac{\rho \notin \mathrm{dom}(T)}{(M, T, N) \xrightarrow{\langle \mathrm{Fork}, \rho\rangle}{}^{\pi} (M, T[\rho \mapsto T(\pi)], N)} \qquad \textsc{Machine-Thread}\ \frac{\varepsilon \neq \langle \mathrm{Fork}, \_\rangle \qquad (M, T(\pi)) \xrightarrow{\varepsilon} (M', V') \\ (M, N) \xrightarrow{\varepsilon}{}^{T(\pi)} (M', N') \qquad \texttt{alloc\_red}(M, T(\pi), \varepsilon)}{(M, T, N) \xrightarrow{\varepsilon}{}^{\pi} (M', T[\pi \mapsto V'], N')}$$

Figure 2: Per-thread reductions for the RA+NA machine (`lang/machine.v`).

it cannot make a step. The bad states are only used in the correspondence proof between the axiomatic and operational semantics.

Another difference is that, in the Coq development, the per-thread reduction is factored into 3 smaller component reductions: the *basic thread* reduction, the *data-race-freedom* reduction, and the *allocation* reduction.

Basic thread reduction The basic thread reduction maintains the following:

- Thread-Read: a read of a location ℓ always reads from a message in the memory M that is *mo*-later than the thread's view V for the location, and update the thread's view with the view of the message.
- Thread-Write: a write of ℓ always picks an unused timestamp t that is greater from the thread's view V, and updates the thread's view with t.
- Thread-Update: an update combines a read and a write, with the write being adjacent to the read by picking the new timestamp to be $t + 1$.

Data-race-freedom reduction The data-race-freedom reduction (DRF-Non-NA and DRF-Write-NA) maintains that, to perform any access to a location ℓ, the thread's view V have observed the most recent non-atomic write to ℓ, *i.e.,* $N(\ell) \leq V(\ell)$. Moreover, if it is a non-atomic read, then the thread's view must have observed the latest write. If it is a non-atomic write, then the write must pick a timestamp greater than all existing timestamps of messages of the same location, and update the non-atomic view. Intuitively, these restrictions enforce that each non-atomic write to ℓ starts a new "era" in ℓ's timestamps, after which any attempt to access writes from a previous era (or to write with a timestamp from a previous era) constitutes a race.

Allocation reduction The allocation reduction maintains the following:

- an allocation is a non-atomic write to a fresh location (Alloc-Alloc), while a deallocation is a non-atomic write that is only applicable for allocated locations (Alloc-Dealloc).
- a normal-value write only requires that the thread's view V has observed the location to be allocated (Alloc-Write), while a normal-value read or update also requires the thread's view to have observed that the location is initialized (Alloc-Read and Alloc-Update). Note that `alloc_red` restricts that reads, updates and atomic writes can only work with normal values ($v \in \mathbb{Z}$).

Per-thread machine reduction The per-thread machine reduction combines all the 3 reductions for the non-fork case (Machine-Thread), and simply adds a new thread with the same view as the parent thread in the fork case (Machine-Fork).

2.2 The λ_{RN} language

We now define the expressions and expression reduction for λ_{RN}. Then we will define the combined reduction reduction for the language.

λ_{RN} is a standard lambda calculus with recursive functions, forks, and first-order references with atomic and non-atomic accesses:

$$
\begin{aligned}
v \in \mathrm{Val} &\triangleq \mid () \mid z \in \mathbb{Z} \mid \ell \in \mathrm{Loc} \mid \textbf{fix}\ (f, x).\ e \\
e \in \mathrm{Expr} &\triangleq \mid v \mid e\ e \mid \ \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid \textbf{fork}\ e \\
&\quad \mid e_{[\alpha]} \mid e_{[\alpha]} := e \mid \textbf{cas}(e, e, e) \mid \textbf{fai}(e, n) \\
&\quad \mid \textbf{alloc} \mid \textbf{free}\ e \\
&\quad \mid (\textbf{int})\ e \mid (\textbf{loc})\ e \mid \ -e \mid \neg e \\
&\quad \mid e + e \mid e - e \mid e \bmod e \\
&\quad \mid e \le e \mid e < e \mid e = e
\end{aligned}
$$

In addition to standard expressions, we also have casting operations (**loc**) and (**int**), to cast positive integers to locations and vice versa, since only integers can be stored in the machine state. The fetch-and-increment primitive $\textbf{fai}(\ell, n)$, which is needed to model fixed width integer types[1], updates the value z of ℓ to $z + 1 \bmod n$, where n is the maximum unsigned value of such a type. Boolean values are mapped by the common way to integer values *i.e.,* non-zeros for True and zero for False. We also support additions between locations and integers—see `bin_op_eval` in `lang/lang.v` for more details.

The expression reduction relation is given in Figure 3. The reduction is split into a *non-stateful reduction* $(e \to e')$, which does not change the physical machine state, and a *stateful reduction* $(e \xrightarrow{\varepsilon} e', \overline{(e_f, \pi_f)})$, which does change the state, and thus is labeled with an event ε that represents the change. The stateful reduction also has an extra component $\overline{(e_f, \pi_f)}$, which is a list of newly forked threads. This list is only populated by a **fork** expression.

The expression reduction and the machine reduction are then combined into the *head reduction*, given by the Combined-* rules in Figure 3. The head reduction couples the machine state σ with the executing thread id π and its expression e. Non-stateful reduction (Combined-Pure) simply defers to the expression reduction and maintains the state. Stateful reduction (Combined-Mem) binds together the expression and machine reduction using the memory event ε.

The head reduction is then lifted to evaluation contexts in a standard way[2], reflected in EvalCtx-Thread in Figure 4. Finally, Iris[3] helps us lift all of these reductions to the full thread-pool reduction (Threadpool-Red-No-Fork and Threadpool-Red-Fork), where threadpools are finite partial functions from thread ids to expressions: $\mathcal{TS} \in \mathrm{ThreadId} \xrightarrow{\mathrm{fin}} \mathrm{Expr}$. This concludes the presentation of the λ_{RN} language.

[1] see the bounded ticket lock example.
[2] Iris supports this style, see `iris.program_logic.ectx_language.prim_step`. We skip the standard definition of evaluation contexts here.
[3] Also see `iris.program_logic.language.step`.

Non-stateful expression reduction (`base.head_step`) $\boxed{e \to e'}$

$$
\begin{array}{rcll}
(\textbf{fix}\ (f,x).\ e)\ v & \to & e[(\textbf{fix}\ (f,x).\ e)/f][v/x] & \\
\textbf{if}\, z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \to & e_1 & \text{if } z \neq 0 \\
\textbf{if}\, z\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \to & e_2 & \text{if } z = 0 \\
(\textbf{int})\ \ell & \to & z & \text{where } z \text{ is } \ell \text{ as } \mathbb{Z} \\
(\textbf{loc})\ z & \to & \ell & \text{if } z \in \mathrm{Loc} \text{ and } \ell \text{ is } z \text{ as Loc} \\
 & & \ldots &
\end{array}
$$

Stateful expression reduction (`ra_lang.step`) $\boxed{e \xrightarrow{\varepsilon} e', \overline{(e_f, \pi_f)}}$

$$
\begin{array}{rcll}
\ell_{[\alpha]} & \xrightarrow{\langle \mathrm{Read}_\alpha, \ell, z\rangle} & z, \mathsf{nil} & \\
\ell_{[\alpha]} := z & \xrightarrow{\langle \mathrm{Write}_\alpha, \ell, z\rangle} & (), \mathsf{nil} & \\
\textbf{cas}(\ell, z_o, z_n) & \xrightarrow{\langle \mathrm{Update}, \ell, z_o, z_n\rangle} & 1, \mathsf{nil} & \\
\textbf{cas}(\ell, z_o, z_n) & \xrightarrow{\langle \mathrm{Read}_{\mathrm{at}}, \ell, z\rangle} & 0, \mathsf{nil} & \text{if update fails} \\
\textbf{fai}(\ell, n) & \xrightarrow{\langle \mathrm{Update}, \ell, z, z+1 \bmod n\rangle} & z, \mathsf{nil} & \\
\textbf{fork}\ e & \xrightarrow{\langle \mathrm{Fork}, \rho\rangle} & (), [(e, \rho)] & \\
\textbf{alloc} & \xrightarrow{\langle \mathrm{Write}_{\mathrm{na}}, \ell, A\rangle} & \ell, \mathsf{nil} & \\
\textbf{free}\ \ell & \xrightarrow{\langle \mathrm{Write}_{\mathrm{na}}, \ell, D\rangle} & (), \mathsf{nil} &
\end{array}
$$

Combined head reduction (`ra_lang.head_step`) $\boxed{\sigma; (e,\pi) \to_{\mathsf{h}} \sigma'; (e',\pi), \overline{(e_f,\pi_f)}}$

$$
\frac{e \to e'}{\sigma; (e,\pi) \to_{\mathsf{h}} \sigma; (e',\pi), \mathsf{nil}}\ \textsc{Combined-Pure}
\qquad
\frac{e \xrightarrow{\varepsilon} e', \overline{(e_f,\pi_f)} \qquad \sigma \xrightarrow{\varepsilon}{}^{\pi} \sigma'}{\sigma; (e,\pi) \to_{\mathsf{h}} \sigma'; (e',\pi), \overline{(e_f,\pi_f)}}\ \textsc{Combined-Mem}
$$

Figure 3: λ_{RN} reductions (`lang/lang.v`).

Thread-local reduction with evaluation contexts $\boxed{\sigma; (e,\pi) \to_{\mathsf{t}} \sigma'; (e',\pi), \overline{(e_f,\pi_f)}}$

$$
\frac{\sigma; (e,\pi) \to_{\mathsf{h}} \sigma'; (e',\pi), \overline{(e_f,\pi_f)}}{\sigma; (K[e],\pi) \to_{\mathsf{t}} \sigma'; (K[e'],\pi), \overline{(e_f,\pi_f)}}\ \textsc{EvalCtx-Thread}
$$

Threadpool reduction $\boxed{\sigma; \mathcal{TS} \to_{\mathsf{tp}} \sigma'; \mathcal{TS}'}$

$$
\frac{\sigma; (\mathcal{TS}(\pi),\pi) \to_{\mathsf{t}} \sigma'; (e',\pi), \mathsf{nil}}{\sigma; \mathcal{TS} \to_{\mathsf{tp}} \sigma'; \mathcal{TS}[\pi \mapsto e']}\ \textsc{Threadpool-Red-No-Fork}
\qquad
\frac{\sigma; (\mathcal{TS}(\pi),\pi) \to_{\mathsf{t}} \sigma'; (e',\pi), [(e_f,\pi_f)]}{\sigma; \mathcal{TS} \to_{\mathsf{tp}} \sigma'; \mathcal{TS}[\pi \mapsto e'] \uplus [\pi_f \mapsto e_f]}\ \textsc{Threadpool-Red-Fork}
$$

Figure 4: Threadpool reduction (derived from Iris).

3 Instantiating Iris: the base logic

The key pattern one repeatedly runs into in Iris is called *fictional separation*. Its idea was explained in the paper [1], but the term "fictional separation" was not mentioned, and its ubiquitous application also was not appreciated properly there. In the paper, we presented the case where one should get a physical state assertion $\mathsf{Phys}(\sigma)$ after instantiating Iris with some language, and one would want to split this ownership into local assertions. To do this, one simply creates a ghost copy of that ownership, and splits the ghost copy. That is, one does not achieve separation directly on some resource r, but *fictionally* and indirectly through a splittable ghost copy of r.

We only showed how the pattern worked for a sequentially consistent language λ_{SC} and the λ_{RN} language in the paper. But the pattern is more applicative than that: we also use fictional separation of the history assertion $\text{Hist}(\ell, h)$ to create fractional non-atomics, fractional iRSL assertions or fractional iGPS protocols. Therefore it might be helpful to summarize the pattern here in Figure 5. The pattern's specific instance for fractional assertions was implemented in the *fractor* (see §5.2). We would need to ask the reader to refer to the paper [1] for a short, intuitive explanation of the authoritative PCM Auth, or the original Iris paper [2] for more technical details.

We can now explain the application of this pattern for the base logic of λ_{RN}. However, let us first look at the interface of the base logic, which mainly concerns with views.

3.1 The logic of views

3.2 Model

To fictionally separate a monolithic, non-splittable resource r:

1. Make a splittable ghost copy of r, by designing a PCM $\mathcal{M}$ with the right separation structure *i.e.,* an appropriate monoid composition.
2. Keep the copy in sync with the original resource r, by using the Auth construction on $\mathcal{M}$ and then an invariant I to tie the authoritative part $\bullet\, r$ with r.
3. Derive rules to update the splittable non-authoritative parts $\circ$ in conjunction with $\bullet\, r$.
4. Use the non-authoritative parts $\circ$ in conjunction with the invariant I to build local assertions.

Figure 5: The pattern of fictional separation in Iris.

BASE-NA
$\max(h).\text{view} \sqsubseteq V \dashv\vdash \text{na}(h, V)$

BASE-NA-MONO
$V \sqsubseteq V' \vdash \text{na}(h, V) \Rightarrow \text{na}(h, V')$

BASE-NA-PERSISTENT
$\text{na}(h, V) \vdash \Box\text{na}(h, V)$

BASE-INIT
$\exists (v, V_0) \in h.\ v \in \mathbb{Z} \land V_0 \sqsubseteq V \dashv\vdash \text{init}(h, V)$

BASE-INIT-MONO
$V \sqsubseteq V' \vdash \text{init}(h, V) \Rightarrow \text{init}(h, V')$

BASE-INIT-PERSISTENT
$\text{init}(h, V) \vdash \Box\text{init}(h, V)$

BASE-ALLOC
$\exists (v, V_0) \in h.\ v \neq D \land V \sqsubseteq V \dashv\vdash \text{alloc}(h, V)$

BASE-ALLOC-MONO
$V \sqsubseteq V' \vdash \text{alloc}(h, V) \Rightarrow \text{alloc}(h, V')$

BASE-ALLOC-PERSISTENT
$\text{alloc}(h, V) \vdash \Box\text{alloc}(h, V)$

BASE-INIT-ALLOC
$\text{init}(h, V) \vdash \text{alloc}(h, V)$

BASE-SEEN-EXCL
$\text{Seen}(\pi, V) * \text{Seen}(\pi, V') \vdash \mathsf{False}$

BASE-HIST-EXCL
$\text{Hist}(\ell, h) * \text{Hist}(\ell, h') \vdash \mathsf{False}$

BASE-INFO-EXCL
$\text{Info}^1(\ell, n) * \text{Info}^1(\ell, n') \vdash \mathsf{False}$

BASE-HIST-TIMESTAMP-DISJOINT
$\texttt{PSCtx} \vdash \text{Hist}(\ell, h) \Rrightarrow_{\mathcal{N}_{\mathsf{phys}}} \Box\forall (v_1, V_1) \in h, (v_2, V_2) \in h.\ V_1(\ell) = V_2(\ell) \Rightarrow (v_1, V_1) = (v_2, V_2)$

BASE-INFO-FRAC
$$\frac{q_1, q_2, q_1 + q_2 \in (0, 1]}{\text{Info}^{q_1}(\ell, n) * \text{Info}^{q_2}(\ell, n) \Leftrightarrow \text{Info}^{q_1 + q_2}(\ell, n)}$$

BASE-INFO-AGREE
$\text{Info}^{q_1}(\ell, n) * \text{Info}^{q_2}(\ell, n') \Rightarrow n = n'$

Figure 6: Properties of the base logic assertions.

4 View-monotone predicates

5 Persistor and Fractor

5.1 Persistor

5.2 Fractor

6 Non-Atomics

6.1 Proof rules

6.2 Model

7 iRSL

7.1 Proof rules

7.2 Model for atomic reads and writes

7.3 Adding CAS

8 iGPS

8.1 Proof rules

8.1.1 Plain protocols

8.1.2 Single-Writer protocols

8.1.3 Fractional protocols

8.1.4 Fractional Single-Writer protocols with life cycles

8.1.5 Exchanges and Escrows

8.2 Proof setup

8.3 Raw protocols

8.4 Plain protocols

8.5 Single-Writer protocols

8.6 Fractional protocols

8.7 Fractional Single-Writer protocols with life cycles

8.8 Exchanges and Escrows

9 Examples

9.1 Message passing in the base logic

9.2 Message passing in iGPS

9.3 Spin lock

9.4 Treiber stack

9.5 Circular buffer

9.6 Michael-Scott queue

9.7 Bounded ticket lock

9.8 Read-Copy-Update

References

[1] The original paper and the Coq development are available at the following URL: http://plv.mpi-sws.org/igps/.

[2] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.