Destabilizing Iris (Appendix)

SIMON SPIES, MPI-SWS, Germany
NIKLAS MÜCK, MPI-SWS, Germany
HAOYI ZENG, Saarland University, Germany
MICHAEL SAMMLER, ETH Zurich, Switzerland and ISTA, Austria
ANDREA LATTUADA, MPI-SWS, Germany
PETER MÜLLER, ETH Zurich, Switzerland
DEREK DREYER, MPI-SWS, Germany

Contents

Authors' Contact Information: Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Niklas Mück, MPI-SWS, Saarland Informatics Campus, Germany, mueck@mpi-sws.org; Haoyi Zeng, Saarland University, Saarland Informatics Campus, Germany, haze00001@stud.uni-saarland.de; Michael Sammler, ETH Zurich, Zurich, Switzerland and ISTA, Klosterneuburg, Austria, michael.sammler@ist.ac.at; Andrea Lattuada, MPI-SWS, Saarland Informatics Campus, Germany, andrea@mpi-sws.org; Peter Müller, ETH Zurich, Department of Computer Science, Zurich, Switzerland, peter.mueller@inf.ethz.ch; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

Resource Algebra

An *partially stable resource algebra* $A = (M, \cdot, \varepsilon, |\_|_{\text{core}}, \mathcal{V}, |\_|_{\text{st}}, |\_|_{\text{unst}})$ is a *unital resource algebra*:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad a \cdot b = b \cdot a$$

$$|a|_{\text{core}} \cdot a = a \qquad ||a|_{\text{core}}|_{\text{core}} = |a|_{\text{core}} \qquad a \preccurlyeq b \Rightarrow |a|_{\text{core}} \preccurlyeq |b|_{\text{core}} \qquad (a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V}$$

$$\varepsilon \in \mathcal{V} \qquad \varepsilon \cdot a = a \qquad |\varepsilon|_{\text{core}} = \varepsilon$$

where

$$a \preccurlyeq b \triangleq \exists c.\, a \cdot c = b$$
$$a \rightsquigarrow B \triangleq \forall a_f.\, (a \cdot a_f) \in \mathcal{V} \Rightarrow \exists b \in B.\, (b \cdot a_f) \in \mathcal{V}$$

with Stable and Unstable Elements

$$||a|_{\text{st}}|_{\text{st}} = |a|_{\text{st}} \qquad \text{(RA-STABLE-IDEMP)}$$
$$|a \cdot b|_{\text{st}} = |a|_{\text{st}} \cdot |b|_{\text{st}} \qquad \text{(RA-STABLE-DISTR)}$$
$$||a|_{\text{core}}|_{\text{st}} = |a|_{\text{core}} \qquad \text{(RA-CORE-STABLE)}$$
$$|a|_{\text{st}} \cdot |a|_{\text{unst}} = a \qquad \text{(RA-DECOMPOSE)}$$
$$|a|_{\text{unst}} \cdot a = a \qquad \text{(RA-UNSTABLE-DUPL)}$$
$$||a|_{\text{unst}}|_{\text{unst}} = |a|_{\text{unst}} \qquad \text{(RA-UNSTABLE-IDEMP)}$$
$$a \preccurlyeq b \Rightarrow |a|_{\text{unst}} \preccurlyeq |b|_{\text{unst}} \qquad \text{(RA-UNSTABLE-MONO)}$$
$$|a \cdot |b|_{\text{unst}}|_{\text{unst}} = ||a|_{\text{unst}} \cdot b|_{\text{unst}} \qquad \text{(RA-UNSTABLE-FLIP)}$$
$$a \in \mathcal{V} \Rightarrow |a|_{\text{unst}} \cdot b \in \mathcal{V} \Rightarrow a \cdot |b|_{\text{unst}} \in \mathcal{V} \qquad \text{(RA-UNSTABLE-EXTENSION)}$$

where

$$a \rightsquigarrow_{\text{st}} B \triangleq \forall a_f.\, (a \cdot a_f) \in \mathcal{V} \Rightarrow \exists b \in B.\, (b \cdot |a_f|_{\text{st}}) \in \mathcal{V}$$

Fig. 1. The partially stable resource algebra, additions in violet.

A Resource Algebras

As a foundation for Daenerys, we take the resource algebra model of Iris [5, 6] and generalize it with *unstable resources*. To do so, we extend the definition of a resource algebra (see Fig. 1) with two projections, $|a|_{\text{st}}$ and $|a|_{\text{unst}}$.

The stable projection. Resources can be decomposed in a stable- and unstable-part (RA-DECOMPOSE). The stable-projection $|a|_{\text{st}}$ defines which part of the resource will be preserved by the corresponding updates ($\rightsquigarrow_{\text{st}}$):

$$\frac{b \rightsquigarrow_{\text{st}} b'}{b \cdot a \rightsquigarrow_{\text{st}} b' \cdot |a|_{\text{st}}}$$

As discussed in the paper, the stable projection corresponds to the frame modality $\boxplus$ in the logic.

The stable projection is idempotent (RA-STABLE-IDEMP), distributes over composition (RA-STABLE-DISTR), and preserves the core (RA-CORE-STABLE). Idempotence ensures that the frame modality is idempotent (FRAME-IDEMP in Fig. 6). Distributivity ensures that we can combine the separating

$$ExFrac(X) \quad \ni \quad \varepsilon \mid \mathsf{stab}_q(x) \mid \mathsf{unst}(x)$$

$$\mathsf{stab}_q(x) \in \mathcal{V} \Leftrightarrow q \leq 1 \qquad \mathsf{unst}(x) \in \mathcal{V} \qquad |\mathsf{stab}_q(x)|_{\mathrm{st}} = \mathsf{stab}_q(x) \qquad |\mathsf{unst}(x)|_{\mathrm{st}} = \varepsilon$$

$$|\mathsf{stab}_q(x)|_{\mathrm{unst}} = \mathsf{unst}(x) \qquad |\mathsf{unst}(x)|_{\mathrm{unst}} = \mathsf{unst}(x) \qquad \mathsf{stab}_{q_1}(x) \cdot \mathsf{stab}_{q_2}(x) = \mathsf{stab}_{q_1+q_2}(x)$$

$$\mathsf{stab}_q(x) = \mathsf{stab}_q(x) \cdot \mathsf{unst}(x) \qquad \mathsf{unst}(x) = \mathsf{unst}(x) \cdot \mathsf{unst}(x)$$

$$\mathsf{stab}_{q_1}(x) \cdot \mathsf{stab}_{q_2}(y) \in \mathcal{V} \Leftrightarrow (q_1 + q_2 \leq 1 \land x = y) \qquad \mathsf{unst}(x) \cdot \mathsf{unst}(y) \in \mathcal{V} \Rightarrow x = y$$

$$\mathsf{stab}_1(x) \rightsquigarrow_{\mathrm{st}} \mathsf{stab}_1(y) \qquad \mathsf{UnstableComplete}(\mathsf{stab}_1(x))$$

Fig. 2. The resource algebra $ExFrac(X)$.

conjunction of two frame modalities (FRAME-SEP in Fig. 6). The preservation of the core ensures that persistent assertions are always frameable (FRAME-PERS in Fig. 6).

Note that in resource algebras from standard Iris, all elements are stable (*i.e.,* $|a|_{\mathrm{st}} = a$) such that ($\rightsquigarrow_{\mathrm{st}}$) and ($\rightsquigarrow$) coincide. More specifically, one can trivially turn regular Iris unital resource algebras into partially stable resource algebra by picking $|a|_{\mathrm{st}} = a$ and $|a|_{\mathrm{unst}} = \varepsilon$.

The unstable projection. The unstable-projection $|a|_{\mathrm{unst}}$ defines a potentially temporary part of the resource. As discussed in the paper, the unstable projection corresponds to the unstable modality $\divideontimes$ in the logic. It is idempotent (RA-UNSTABLE-IDEMP) and monotone (RA-UNSTABLE-MONO). Idempotence ensures that the unstable modality is idempotent (UNSTABLE-IDEMP in Fig. 6). Monotonicity ensures (together with RA-UNSTABLE-DUPL) that the unstable modality distributes over separating conjunction (UNSTABLE-SEP in Fig. 6).

In addition, it imposes three key axioms, RA-UNSTABLE-DUPL, RA-UNSTABLE-FLIP, and RA-UNSTABLE-EXTENSION. The axiom RA-UNSTABLE-DUPL guarantees that the unstable part of a resource is *duplicable*, meaning we can create as many copies of it as we would like. It is the basis for duplicating unstable assertions (UNSTABLE-DUPL in Fig. 6). The axiom RA-UNSTABLE-FLIP allows us to flip the unstable projection inside an unstable projection. It is a weaker form of distributitivity of the unstable projection (*i.e.,* weaker than $|a \cdot b|_{\mathrm{unst}} = |a|_{\mathrm{unst}} \cdot |b|_{\mathrm{unst}}$) that still suffices to prove the implication rule for unstable propositions UNSTABLE-IMPL in Fig. 6. The axiom RA-UNSTABLE-EXTENSION ensures that the unstable part $|a|_{\mathrm{unst}}$ is a "complete snapshot" of a in the sense that there cannot be an element b that is valid with $|a|_{\mathrm{unst}}$ but whose unstable part $|b|_{\mathrm{unst}}$ is not valid with a. This rule is needed—in addition to RA-UNSTABLE-FLIP—to prove that the implication between two unstable assertions is unstable (*i.e.,* UNSTABLE-IMPL in Fig. 6). The issue is that Iris's implication $P \Rightarrow Q$ is upclosed with respect to larger resources and, to prove UNSTABLE-IMPL, we end up needing to extend the validity predicate for a resource. However, so long as the unstable projection provides a "complete snapshot" as given by RA-UNSTABLE-EXTENSION, we can show that the extension does not suddenly break validity.

$$Sil(M) \quad \ni \quad \varepsilon \mid \mathsf{orig}(a) \mid \mathsf{sil}(a)$$

$$\mathsf{orig}(a) \in \mathcal{V} \Leftrightarrow a \in \mathcal{V} \qquad \mathsf{sil}(a) \in \mathcal{V} \Leftrightarrow a \in \mathcal{V}$$

$$|\mathsf{orig}(a)|_{\mathrm{st}} = \mathsf{orig}(a) \qquad |\mathsf{sil}(a)|_{\mathrm{st}} = \varepsilon$$

$$|\mathsf{orig}(a)|_{\mathrm{unst}} = \mathsf{sil}(a) \qquad |\mathsf{sil}(a)|_{\mathrm{unst}} = \mathsf{sil}(a)$$

$$\mathsf{orig}(a) = \mathsf{orig}(a) \cdot \mathsf{sil}(a) \qquad \mathsf{orig}(a \cdot b) = \mathsf{orig}(a) \cdot \mathsf{orig}(b) \qquad \mathsf{sil}(a) = \mathsf{sil}(a) \cdot \mathsf{sil}(a)$$

$$\mathsf{sil}(a) \cdot \mathsf{sil}(b) \in \mathcal{V} \Leftrightarrow (\exists c.\, c \in \mathcal{V} \land a \preccurlyeq c \land b \preccurlyeq c) \qquad a \preccurlyeq b \Leftrightarrow \mathsf{sil}(a) \preccurlyeq \mathsf{sil}(b)$$

$$(a \rightsquigarrow b) \Rightarrow (\mathsf{orig}(a) \rightsquigarrow_{\mathrm{st}} \mathsf{orig}(b))$$

$$(a, b) \rightsquigarrow_{\mathsf{L}} (a', b') \Rightarrow (\mathsf{orig}(a), \mathsf{orig}(b)) \rightsquigarrow_{\mathrm{st}}^{\mathsf{L}} (\mathsf{orig}(a'), \mathsf{orig}(b')) \qquad \mathsf{UnstableComplete}(\mathsf{orig}(a))$$

Fig. 3. The resource algebra $Sil(M)$.

A.1 Resource Algebra Combinators

We discuss several interesting instances of our partially stable resource algebra.

Exclusive with fractions. The exclusive resource algebra with fractions $ExFrac(X)$ generalizes the $Ex(\mathbb{N})$ resource algebra discussed in the paper by enabling fractional ownership of the exclusive element. We have two elements $\mathsf{stab}_q(x)$ (the counter part of $\mathsf{ex}(x)$) and $\mathsf{unst}(x)$ (the counter part of $\mathsf{tmp}(x)$). Their key rules are depicted in Fig. 2. The element $\mathsf{stab}_q(x)$ carries fractional ownership, and it is always stable. We can update it for fraction 1. The element $\mathsf{unst}(x)$ is an unstable temporary copy.

Authoritative Resource Algebra. We define an version of the authoritative resource algebra $Auth(A)$ for partially stable resource algebras, written $PSAuth(A)$. Compared to the standard Iris authoritative resource algebra, this version has an additional, unstable element $◒x$, which is an unstable copy of the full element $\bullet x$. That is,

$$\bullet x = \bullet x \cdot ◒x \qquad ◒x = ◒x \cdot ◒x \qquad |\bullet x|_{\mathrm{unst}} = ◒x \qquad ◒x \cdot ◒y \in \mathcal{V} \Rightarrow x = y$$

To fulfill the property RA-UNSTABLE-EXTENSION, we require that full elements $\bullet x$ are maximal in the sense that

$$\mathsf{UnstableComplete}(x) \triangleq \forall y.\, y \in \mathcal{V} \Rightarrow |y|_{\mathrm{unst}} \preccurlyeq |x|_{\mathrm{unst}} \Rightarrow y \preccurlyeq x$$

Silhouette Resource Algebra. We define the $Sil(M)$ resource algebra, which completes a standard unital resource algebra M to one with unstable resources.[1] The resource algebra is depicted in Fig. 3. Its two key elements are $\mathsf{orig}(a)$ which simply embeds elements $a \in M$, and $\mathsf{sil}(a)$, which is an unowned copy of a. When we have two unowned copies $\mathsf{sil}(a) \cdot \mathsf{sil}(b)$, we do not add them together. Instead, there must be some c that includes a and b. The $\mathsf{orig}(a)$ elements are unstable complete, so we can use them in an *PSAuth*-construction.

[1]This resource algebra supports only non-step-indexed (*i.e.,* "discrete") resource algebras as instantiations of M.

Resource Assertions

$$P, Q ::= \cdots\cdots \mid \ell \mapsto^{\text{basic}}_{q} v \mid \ell \mapsto^{\text{basic}}_{\text{u}} v \mid \mathsf{heap}^{\text{basic}}(h) \mid \mathsf{heap}^{\text{basic}}_{\text{u}}(h)$$

Proof Rules

BASIC-FRAME-PTS
$\ell \mapsto^{\text{basic}}_{q} v \vdash \boxplus \ell \mapsto^{\text{basic}}_{q} v$

BASIC-FRAME-HEAP
$\mathsf{heap}^{\text{basic}}(h) \vdash \boxplus \mathsf{heap}^{\text{basic}}(h)$

BASIC-UNSTABLE-PTS
$\ell \mapsto^{\text{basic}}_{\text{u}} v \vdash \divideontimes\, \ell \mapsto^{\text{basic}}_{\text{u}} v$

BASIC-UNSTABLE-HEAP
$\mathsf{heap}^{\text{basic}}_{\text{u}}(h) \vdash \divideontimes\, \mathsf{heap}^{\text{basic}}_{\text{u}}(h)$

BASIC-GET-PTS-UNSTABLE
$\ell \mapsto^{\text{basic}}_{q} v \vdash \ell \mapsto^{\text{basic}}_{\text{u}} v$

BASIC-GET-HEAP-UNSTABLE
$\mathsf{heap}^{\text{basic}}(h) \vdash \mathsf{heap}^{\text{basic}}_{\text{u}}(h)$

BASIC-OWNED-AGREE
$\ell \mapsto^{\text{basic}}_{q} v * \ell \mapsto^{\text{basic}}_{q'} w \vdash q + q' \leq 1 * v = w$

BASIC-UNSTABLE-AGREE
$\ell \mapsto^{\text{basic}}_{\text{u}} v * \ell \mapsto^{\text{basic}}_{\text{u}} w \vdash v = w$

BASIC-HEAP-AGREE
$\mathsf{heap}^{\text{basic}}_{\text{u}}(h) * \mathsf{heap}^{\text{basic}}_{\text{u}}(h') \vdash h = h'$

BASIC-PTS-SPLIT
$\ell \mapsto^{\text{basic}}_{q+q'} v \dashv\vdash \ell \mapsto^{\text{basic}}_{q} v * \ell \mapsto^{\text{basic}}_{q'} v$

BASIC-UNSTABLE-LOOKUP
$\ell \mapsto^{\text{basic}}_{\text{u}} v * \mathsf{heap}^{\text{basic}}_{\text{u}}(h) \vdash h(\ell) = v$

BASIC-HEAP-LOOKUP
$\ell \mapsto^{\text{basic}}_{q} v * \mathsf{heap}^{\text{basic}}(h) \vdash h(\ell) = v$

BASIC-PTS-UPDATE
$\ell \mapsto^{\text{basic}}_{1} v * \mathsf{heap}^{\text{basic}}(h) \vdash \Rrightarrow_{\mathsf{st}} \ell \mapsto^{\text{basic}}_{1} w * \mathsf{heap}^{\text{basic}}(h[\ell := w])$

BASIC-PTS-ALLOC
$\ell \notin \mathrm{dom}\, h * \mathsf{heap}^{\text{basic}}(h) \vdash \Rrightarrow_{\mathsf{st}} \ell \mapsto^{\text{basic}}_{1} v * \mathsf{heap}^{\text{basic}}(h[\ell := v])$

Fig. 4. The basic theory of heaps obtained by choosing the resource algebra *BHeap*

A.2 Modeling the Unstable Points-To

Equipped with these resource algebra combinators, we can discuss the model of the unstable points-to assertion $\ell \mapsto_{\text{u}} v$. We first discuss a basic version and then how, in Daenerys, we get the full version that supports Iris's discardable fractions (for additional "persistent points-tos" $\ell \mapsto_{\square} v$).

A basic heap. We obtain the basic heap by choosing the resource algebra $\mathit{SHeap} \triangleq \mathit{PSAuth}(\mathit{Loc} \xrightarrow{\text{fin}} \mathit{ExFrac}(\mathit{Val}))$. In Iris, the corresponding heap theory would consist only of two assertions: $\mathsf{heap}^{\text{basic}}(h)$ to keep track of the full heap, and the points-to assertions $\ell \mapsto^{\text{basic}}_{q} v$ for $0 < q \leq 1$ to keep track of individual locations in h. Here, we extend it with two new, unstable assertions: $\mathsf{heap}^{\text{basic}}_{\text{u}}(h)$, an unstable temporary copy of $\mathsf{heap}(h)$ and $\ell \mapsto^{\text{basic}}_{\text{u}} v$, an unstable temporary copy of the points-to.

Let us first discuss their rules, depicted in Fig. 4, before we discuss the definition of the connectives below. As usual, points-tos must agree on their values (BASIC-OWNED-AGREE), we can split-and combine points-to assertions as needed (BASIC-PTS-SPLIT), and a points-to assertion for ℓ determines the value of ℓ in h (BASIC-HEAP-LOOKUP). We can update an entry in the heap with a *stable update* (BASIC-PTS-UPDATE) and allocate a new entry (BASIC-PTS-ALLOC). (In Iris, these rules would be standard frame preserving updates.) The points-to for $q > 0$ and the heap are frameable (BASIC-FRAME-PTS and BASIC-FRAME-HEAP), and their unstable counterparts are unstable (BASIC-UNSTABLE-PTS and BASIC-UNSTABLE-HEAP). We can always get an unstable copy from the owned versions (BASIC-GET-PTS-UNSTABLE and BASIC-GET-HEAP-UNSTABLE), and the unstable versions agree: the unstable points-tos agree on the value in the heap (BASIC-UNSTABLE-AGREE), two unstable heaps agree on the entire heap (BASIC-HEAP-AGREE), and we can look up a location in the unstable heap (BASIC-UNSTABLE-LOOKUP).

To obtain this resource theory, we define

$$\ell \mapsto^{\text{basic}}_{q} v \triangleq \boxed{\circ\{\ell \mapsto \mathsf{stab}_q(v)\}}^{\gamma_{heap}} \qquad \ell \mapsto^{\text{basic}}_{\mathsf{u}} v \triangleq \boxed{\circ\{\ell \mapsto \mathsf{unst}(v)\}}^{\gamma_{heap}}$$

$$\mathsf{heap}^{\text{basic}}(h) \triangleq \boxed{\bullet\{\ell \mapsto \mathsf{stab}_1(v) \mid \ell \mapsto v \in h\}}^{\gamma_{heap}}$$

$$\mathsf{heap}^{\text{basic}}_{\mathsf{u}}(h) \triangleq \boxed{\mathbin{-\!\!\bullet}\{\ell \mapsto \mathsf{stab}_1(v) \mid \ell \mapsto v \in h\}}^{\gamma_{heap}}$$

Here, we write $\boxed{a}^{\gamma}$ for Iris's ownership connective, a version of Own (a) shown in the paper for supporting multiple resource algebras at the same time. The variable γ is the name of the concrete "instance" of the resource; for heaps, we can fix a global name such as γ_{heap}.

This resource algebra is a variation of a standard technique for constructing a resource algebra for heaps in Iris. For the regular points-to $\ell \mapsto^{\text{basic}}_{q} v$, we use a fragment $\circ$ containing a singleton map that maps ℓ to the *stable resource* $\mathsf{stab}_q(v)$ (from the resource algebra in Fig. 2). Analogously, for the unstable points-to $\ell \mapsto^{\text{basic}}_{\mathsf{u}} v$, we use a fragment $\circ$ containing a singleton map that maps ℓ to the *unstable resource* $\mathsf{unst}(v)$ (from the resource algebra in Fig. 2). For the full heap $\mathsf{heap}^{\text{basic}}(h)$, we use the authoritative element $\bullet$ containing the entire heap (where every element has been allocated with fraction 1). For the unstable copy $\mathsf{heap}^{\text{basic}}_{\mathsf{u}}(h)$, we use the new unstable authoritative element $\mathbin{-\!\!\bullet}$ of our *PSAuth* resource algebra.

Supporting discardable fractions. As mentioned above, the version of the heap theory that we use in Daenerys goes beyond the basic heaps by supporting discardable fractions (*i.e.,* the "persistent points-tos" $\ell \mapsto_{\square} v$). The corresponding theory is depicted in Fig. 5.

Similar to the standard Iris points-tos with discardable fractions, its construction in terms of resource algebra combinators is somewhat more involved. It can be found in the accompanying Rocq development. We end up using the resource algebra

$$\mathit{Heap}(\mathit{Loc}, \mathit{Val}) \triangleq \mathit{PSView}((\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}), \mathit{Sil}(\mathit{Loc} \xrightarrow{\text{fin}} \mathit{DFrac} \times \mathit{Ag}(\mathit{Val})))$$

where *PSView* is a version of Iris's view resource algebra *extended with unstable elements* (generalizing the *PSAuth*-resource algebra from §A.1) and *Sil* is the silhouette resource algebra (from §A.1).

Resource Assertions

$$P, Q ::= \cdots\cdots \mid \ell \mapsto_{dq} v \mid \ell \mapsto_{\mathsf{u}} v \mid \mathsf{heap}(h) \mid \mathsf{heap}_{\mathsf{u}}(h) \qquad dq \in \{\square\} \cup \mathbb{Q}^+$$

Proof Rules

FRAME-PTS
$\ell \mapsto_{dq} v \vdash \boxplus \ell \mapsto_{dq} v$

PERSISTENT-PTS
$\ell \mapsto_{\square} v \vdash \square(\ell \mapsto_{\square} v)$

FRAME-HEAP
$\mathsf{heap}(h) \vdash \boxplus \mathsf{heap}(h)$

UNSTABLE-PTS
$\ell \mapsto_{\mathsf{u}} v \vdash \divideontimes\, \ell \mapsto_{\mathsf{u}} v$

UNSTABLE-HEAP
$\mathsf{heap}_{\mathsf{u}}(h) \vdash \divideontimes\, \mathsf{heap}_{\mathsf{u}}(h)$

GET-PTS-UNSTABLE
$\ell \mapsto_{dq} v \vdash \ell \mapsto_{\mathsf{u}} v$

GET-HEAP-UNSTABLE
$\mathsf{heap}(h) \vdash \mathsf{heap}_{\mathsf{u}}(h)$

OWNED-AGREE
$\ell \mapsto_{dq} v * \ell \mapsto_{dq'} w \vdash (dq \cdot dq') \in \mathcal{V} * v = w$

UNSTABLE-AGREE
$\ell \mapsto_{\mathsf{u}} v * \ell \mapsto_{\mathsf{u}} w \vdash v = w$

HEAP-AGREE
$\mathsf{heap}_{\mathsf{u}}(h) * \mathsf{heap}_{\mathsf{u}}(h') \vdash h = h'$

PTS-SPLIT
$\ell \mapsto_{q_1+q_2} v \dashv\vdash \ell \mapsto_{q_1} v * \ell \mapsto_{q_2} v$

UNSTABLE-LOOKUP
$\ell \mapsto_{\mathsf{u}} v * \mathsf{heap}_{\mathsf{u}}(h) \vdash h(\ell) = v$

HEAP-LOOKUP
$\ell \mapsto_{dq} v * \mathsf{heap}(h) \vdash h(\ell) = v$

PTS-UPDATE
$\ell \mapsto_1 v * \mathsf{heap}(h) \vdash \Rrightarrow_{\mathsf{st}} \ell \mapsto_1 w * \mathsf{heap}(h[\ell := w])$

PTS-ALLOC
$\ell \notin \mathrm{dom}\, h * \mathsf{heap}(h) \vdash \Rrightarrow_{\mathsf{st}} \ell \mapsto_1 v * \mathsf{heap}(h[\ell := v])$

PTS-DEALLOC
$\mathsf{heap}(h) * \ell \mapsto_1 v \vdash \Rrightarrow_{\mathsf{st}} \mathsf{heap}(h[\ell := /])$

PTS-PERSIST
$\ell \mapsto_{dq} v \vdash \Rrightarrow_{\mathsf{st}} \ell \mapsto_{\square} v$

Fig. 5. The full theory of heaps with discardable fractions obtained by choosing the resource algebra *Heap*.

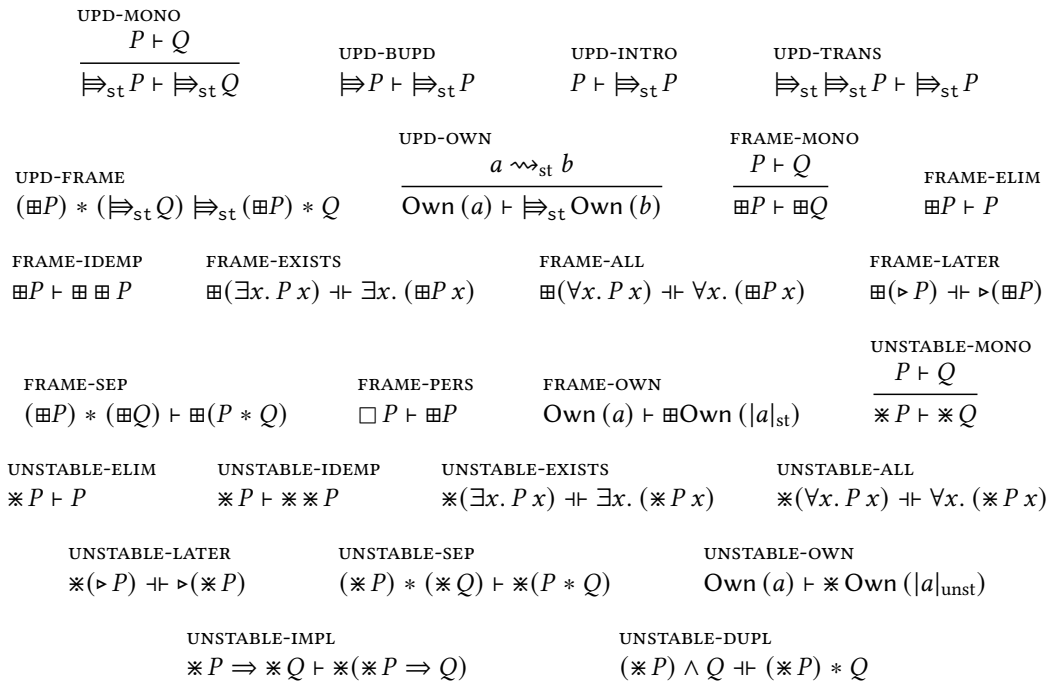

Fig. 6. Base logic rules for the update modality $\Rightarrow\!\!\!\!\models_{\mathsf{st}} P$, frame modality $\boxplus P$, and unstable modality $\divideontimes P$

B Base Logic

The new modalities $\Rightarrow\!\!\!\!\models_{\mathsf{st}}$, $\boxplus$, and $\divideontimes$ satisfy the rules in Fig. 6.

Let us start with the *stable update* $\Rightarrow\!\!\!\!\models_{\mathsf{st}}$. It satisfies several standard rules for Iris updates that make it compositional (a monad to be precise): it is monotone (UPD-MONO), we can always introduce it (UPD-INTRO), and we can always compose updates via transitivity (UPD-TRANS). For the last two rules to hold (which are important for making updates practically usable and integrating them into the weakest precondition), it is important that the underlying update $a \rightsquigarrow_{\mathsf{st}} a$ is *reflexive* (for UPD-INTRO) and *transitive* (for UPD-TRANS). Besides these rules, we can always turn a stable update on resources $a \rightsquigarrow_{\mathsf{st}} a$ into one on the assertion level (UPD-OWN). The standard update $\Rightarrow\!\!\models$ is included in $\Rightarrow\!\!\!\!\models_{\mathsf{st}}$ (UPD-BUPD), which means existing Iris ghost theories (*e.g.,* the theory $\mathsf{Auth}(\mathbb{N}, \max)$) can still be used as is—especially considering that $\Rightarrow\!\!\!\!\models_{\mathsf{st}}$ is used in the definition of the weakest precondition. Lastly, the rule UPD-FRAME allows us to frame assertions R into a stable update *if we can put them underneath a frame modality* $\boxplus$.

Let us now turn to the *frame modality* $\boxplus$. The frame modality is a co-monad (*e.g.,* like Iris's persistency modality): it is monotone (FRAME-MONO), we can always eliminate it (FRAME-ELIM), and it is idempotent (FRAME-IDEMP). The rule FRAME-ELIM means we never have to worry about "getting rid" of a frame modality (from our assumptions). The rule FRAME-IDEMP means we can always "add another" frame modality, which can be useful for proof rules that remove a frame modality. Moreover, the frame modality distributes over existential quantification (FRAME-EXISTS), universal quantification (FRAME-ALL), and the later modality (FRAME-LATER). It *does not* distribute over the separating conjunction in both directions. More specifically, we can combine two frameable assertions (FRAME-SEP), but the opposite direction does not hold (*i.e.,* turning $\boxplus(P * Q)$ into $(\boxplus P) *$

$(\boxplus Q)$). For example, it would be unsound to turn $\boxplus(\ell \mapsto v * \ell \mapsto_{\mathsf{u}} v)$ into $(\boxplus \ell \mapsto v) * (\boxplus \ell \mapsto_{\mathsf{u}} v)$, because $(\boxplus \ell \mapsto v)$ could be used to update ℓ invalidating $(\boxplus \ell \mapsto_{\mathsf{u}} v)$. Furthermore, to make sure that persistent propositions $\Box P$ keep their usual meaning (*i.e.,* once created, they persist across updates and state changes), we make sure that persistent assertions are frameable (FRAME-PERS). Finally, when we own a resource a, we always frameably own its stable part (FRAME-OWN).

Let us now turn to the *unstable modality* $\divideontimes$. Like the frame modality, it is a co-monad: it is monotone (UNSTABLE-MONO), we can always eliminate it (UNSTABLE-ELIM), and it is idempotent (UNSTABLE-IDEMP). Thus, also for the unstable modality, we never have to worry about "getting rid" of it (from our assumptions), and we can always "add another". Moreover, the unstable modality distributes over existential quantification (UNSTABLE-EXISTS), universal quantification (UNSTABLE-ALL), and the later modality (UNSTABLE-LATER). Importantly, the unstable modality makes separating conjunction and ordinary conjunction coincide (UNSTABLE-DUPL). This rule ensures that we do not have to give up ownership when we prove an unstable assertions. From it, we can derive, *e.g.,* $\divideontimes P \vdash \divideontimes P * \divideontimes P$. The unstable modality *does* distribute over separation conjunction in both directions: UNSTABLE-SEP gives one direction, and the other direction can be derived from UNSTABLE-MONO and UNSTABLE-DUPL. Furthermore, we have the rule UNSTABLE-IMPL, which effectively means that the implication between two unstable assertions $\divideontimes P$ and $\divideontimes Q$ is itself an unstable proposition. We use it to justify adding $F_1 \Rightarrow F_2$ to the *hProp*-fragment. Finally, when we own a resource a, we always unstably own its unstable part (UNSTABLE-OWN).

$$
\begin{array}{lrcll}
\text{Terms} & t, s & ::= & \mathsf{x} \mid \mathsf{f}\,\vec{t} & (\mathsf{f} \in \mathcal{C}) \\
\text{Formulas} & \pi, \chi & ::= & \mathsf{True} \mid \mathsf{False} \mid t \doteq_{\mathsf{S}} s \mid \mathsf{p}\,\vec{t} \mid \pi_1 \mathbin{\dot\wedge} \pi_2 \mid \pi_1 \mathbin{\dot\vee} \pi_2 & (\mathsf{p} \in \mathcal{P}) \\
& & \mid & \pi_1 \dot\Rightarrow \pi_2 \mid \dot\exists \mathsf{x} : \mathsf{S}.\, \pi \mid \dot\forall \mathsf{x} : \mathsf{S}.\, \pi &
\end{array}
$$

Typing

$$\textsc{type-var}\ \frac{\mathsf{x} : \mathsf{S} \in \Sigma}{\Sigma \vdash_{\mathsf{term}} \mathsf{x} : \mathsf{S}} \qquad \textsc{type-func}\ \frac{\vdash_{\mathsf{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \qquad \Sigma \vdash_{\mathsf{term}} \vec{t} : \vec{\mathsf{S}}}{\Sigma \vdash_{\mathsf{term}} \mathsf{f}\,\vec{t} : \mathsf{T}}$$

$$\textsc{type-true}\ \Sigma \vdash_{\mathsf{form}} \mathsf{True} \qquad \textsc{type-false}\ \Sigma \vdash_{\mathsf{form}} \mathsf{False}$$

$$\textsc{type-and}\ \frac{\Sigma \vdash_{\mathsf{form}} \pi_1 \qquad \Sigma \vdash_{\mathsf{form}} \pi_2}{\Sigma \vdash_{\mathsf{form}} \pi_1 \mathbin{\dot\wedge} \pi_2} \qquad \textsc{type-or}\ \frac{\Sigma \vdash_{\mathsf{form}} \pi_1 \qquad \Sigma \vdash_{\mathsf{form}} \pi_2}{\Sigma \vdash_{\mathsf{form}} \pi_1 \mathbin{\dot\vee} \pi_2} \qquad \textsc{type-impl}\ \frac{\Sigma \vdash_{\mathsf{form}} \pi_1 \qquad \Sigma \vdash_{\mathsf{form}} \pi_2}{\Sigma \vdash_{\mathsf{form}} \pi_1 \dot\Rightarrow \pi_2}$$

$$\textsc{type-all}\ \frac{\Sigma, \mathsf{x} : \mathsf{S} \vdash_{\mathsf{form}} \pi}{\Sigma \vdash_{\mathsf{form}} \dot\forall \mathsf{x} : \mathsf{S}.\, \pi} \qquad \textsc{type-exists}\ \frac{\Sigma, \mathsf{x} : \mathsf{S} \vdash_{\mathsf{form}} \pi}{\Sigma \vdash_{\mathsf{form}} \dot\exists \mathsf{x} : \mathsf{S}.\, \pi} \qquad \textsc{type-eq}\ \frac{\Sigma \vdash_{\mathsf{term}} t : \mathsf{S} \qquad \Sigma \vdash_{\mathsf{term}} s : \mathsf{S}}{\Sigma \vdash_{\mathsf{form}} t \doteq_{\mathsf{S}} s}$$

$$\textsc{type-pred}\ \frac{\vdash_{\mathsf{pred}} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \qquad \Sigma \vdash_{\mathsf{term}} \vec{t} : \vec{\mathsf{S}}}{\Sigma \vdash_{\mathsf{form}} \mathsf{p}\,\vec{t}}$$

Fig. 7. Terms and formulas of first-order logic.

C Many-Sorted First-Order Logic

In this section, we describe the first-order logic that we consider.

C.1 General First-Order Logic

First-order terms and formulas. We define first-order terms t and formulas π in Fig. 7. We define them parametrically over a signature $\Omega = (\mathcal{S}, \mathcal{C}, \mathcal{P}, \vdash_{\mathsf{func}}, \vdash_{\mathsf{pred}})$, where $\mathcal{S}$ is a collection of first-order sort symbols; $\mathcal{C}$ a collection of first-order function symbols; $\mathcal{P}$ a collection of first-order predicate symbols; $\vdash_{\mathsf{func}}$ assigns argument sorts and a result sort to every function symbol $\mathsf{f} \in \mathcal{C}$; and $\vdash_{\mathsf{func}}$ assigns argument sorts to every predicate symbol $\mathsf{p} \in \mathcal{P}$. We write $\vdash_{\mathsf{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \in \mathcal{C}$ to mean $\mathsf{f} \in \mathcal{C}$ and $\vdash_{\mathsf{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T}$. We write $\vdash_{\mathsf{pred}} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \in \mathcal{P}$ to mean $\mathsf{p} \in \mathcal{P}$ and $\vdash_{\mathsf{pred}} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop}$.

Definition C.1. We say a signature $\Omega_1 = (\mathcal{S}_1, \mathcal{C}_1, \mathcal{P}_1, \vdash_{\mathsf{func}_1}, \vdash_{\mathsf{pred}_1})$ extends a signature $\Omega_2 = (\mathcal{S}_2, \mathcal{C}_2, \mathcal{P}_2, \vdash_{\mathsf{func}_2}, \vdash_{\mathsf{pred}_2})$, written $\Omega_1 \sqsupseteq_{\mathsf{sig}} \Omega_2$, iff. (1) $\mathcal{S}_1 \supseteq \mathcal{S}_2$, (2) $\mathcal{C}_1 \supseteq \mathcal{C}_2$, and (3) $\mathcal{P}_1 \supseteq \mathcal{P}_2$ and the typing judgments agree:

$$\vdash_{\mathsf{pred}_2} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \Rightarrow \vdash_{\mathsf{pred}_1} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \qquad \vdash_{\mathsf{func}_2} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \Rightarrow \vdash_{\mathsf{func}_1} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T}$$

$$
\begin{aligned}
[\![\mathsf{x}]\!]^M_{\mathrm{term}}(\epsilon) &= \epsilon(\mathsf{x}) \\
[\![\mathsf{f}\,\vec{t}]\!]^M_{\mathrm{term}}(\epsilon) &= [\![\mathsf{f}]\!]_{\mathrm{func}}([\![\vec{t}]\!]^M_{\mathrm{term}}(\epsilon)) \\
[\![\mathsf{True}]\!]^M_{\mathrm{form}}(\epsilon) &= \mathsf{True} \\
[\![\mathsf{False}]\!]^M_{\mathrm{form}}(\epsilon) &= \mathsf{False} \\
[\![\pi_1 \mathbin{\dot\wedge} \pi_2]\!]^M_{\mathrm{form}}(\epsilon) &= [\![\pi_1]\!]^M_{\mathrm{form}}(\epsilon) \wedge [\![\pi_2]\!]^M_{\mathrm{form}}(\epsilon) \\
[\![\pi_1 \mathbin{\dot\vee} \pi_2]\!]^M_{\mathrm{form}}(\epsilon) &= [\![\pi_1]\!]^M_{\mathrm{form}}(\epsilon) \vee [\![\pi_2]\!]^M_{\mathrm{form}}(\epsilon) \\
[\![\pi_1 \mathbin{\dot\Rightarrow} \pi_2]\!]^M_{\mathrm{form}}(\epsilon) &= [\![\pi_1]\!]^M_{\mathrm{form}}(\epsilon) \Rightarrow [\![\pi_2]\!]^M_{\mathrm{form}}(\epsilon) \\
[\![\dot\forall \mathsf{x} : \mathsf{S}.\ \pi]\!]^M_{\mathrm{form}}(\epsilon) &= \forall d : D_{\mathsf{S}}.\ [\![\pi]\!]^M_{\mathrm{form}}(\epsilon, \mathsf{x} \mapsto d) \\
[\![\dot\exists \mathsf{x} : \mathsf{S}.\ \pi]\!]^M_{\mathrm{form}}(\epsilon) &= \exists d : D_{\mathsf{S}}.\ [\![\pi]\!]^M_{\mathrm{form}}(\epsilon, \mathsf{x} \mapsto d) \\
[\![t \doteq_{\mathsf{S}} s]\!]^M_{\mathrm{form}}(\epsilon) &= [\![t]\!]^M_{\mathrm{term}}(\epsilon) = [\![s]\!]^M_{\mathrm{term}}(\epsilon) \\
[\![\mathsf{p}\,\vec{t}]\!]^M_{\mathrm{form}}(\epsilon) &= [\![\mathsf{p}]\!]_{\mathrm{pred}}([\![\vec{t}]\!]^M_{\mathrm{term}}(\epsilon))
\end{aligned}
$$

Fig. 8. The first-order Tarski semantics for a given model $M = (D_{\_}, [\![\_]\!]_{\mathrm{func}}, [\![\_]\!]_{\mathrm{pred}})$

Tarski semantics. We define the semantics of first-order terms and formulas, a standard Tarski-semantics, in Fig. 8. To define the semantics, we assume a model $M = (D_{\_}, [\![\_]\!]_{\mathrm{func}}, [\![\_]\!]_{\mathrm{pred}})$ where D maps each $\mathsf{S} \in \mathcal{S}$ to a non-empty set D_{S}, the *domain* for S; $[\![\_]\!]_{\mathrm{func}}$ maps each function $\vdash_{\mathrm{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \in \mathcal{C}$ to a meta-level function $[\![\mathsf{f}]\!]_{\mathrm{func}} : \vec{D_{\mathsf{S}}} \to D_{\mathsf{T}}$; and $[\![\_]\!]_{\mathrm{pred}}$ maps each predicate $\vdash_{\mathrm{pred}} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \in \mathcal{P}$ to a meta-level predicate $[\![\mathsf{p}]\!]_{\mathrm{pred}} : \vec{D_{\mathsf{S}}} \to \mathit{Prop}$.

Definition C.2. We say a well-formed formula $\vdash_{\mathrm{form}} \pi$ holds in a model $M = (D_{\_}, [\![\_]\!]_{\mathrm{func}}, [\![\_]\!]_{\mathrm{pred}})$, written $M \vDash \pi$, iff. $[\![\pi]\!]^M_{\mathrm{form}}(\emptyset)$ holds.

Note the distinction between the meta-level logic and the object language: The terms and formulas in Fig. 7 are just syntax. We give them meaning via the Tarski-semantics in Fig. 8, which interprets every symbol using the corresponding connective from the ambient meta-logic.

Definition C.3. Let $\Omega_1 = (\mathcal{S}_1, \mathcal{C}_1, \mathcal{P}_1, \vdash_{\mathrm{func}1}, \vdash_{\mathrm{pred}_1})$ and $\Omega_2 = (\mathcal{S}_2, \mathcal{C}_2, \mathcal{P}_2, \vdash_{\mathrm{func}2}, \vdash_{\mathrm{pred}_2})$ such that $\Omega_1 \sqsupseteq_{\mathrm{sig}} \Omega_2$. We say a model $M^1 = (D^1_{\_}, [\![\_]\!]^1_{\mathrm{func}}, [\![\_]\!]^1_{\mathrm{pred}})$ for signature Ω_1 extends a model $M^2 = (D^2_{\_}, [\![\_]\!]^2_{\mathrm{func}}, [\![\_]\!]^2_{\mathrm{pred}})$ for signature Ω_2, written $M^1 \sqsupseteq_{\mathrm{model}} M^2$, iff.

(1) for every $\mathsf{S} \in \mathcal{S}_2$, there is a function $i_{\mathsf{S}} : D^2_{\mathsf{S}} \to D^1_{\mathsf{S}}$,
(2) for every $\mathsf{S} \in \mathcal{S}_2$, there is a function $r_{\mathsf{S}} : D^1_{\mathsf{S}} \to D^2_{\mathsf{S}}$,
(3) for every $\mathsf{S} \in \mathcal{S}_2$, i_{S} and r_{S} form a bijection, *i.e.*, $\forall (a \in D^2_{\mathsf{S}}), (b \in D^1_{\mathsf{S}}).\ i_{\mathsf{S}}(a) = b$ *iff.* $a = r_{\mathsf{S}}(b)$,
(4) for every $\vdash_{\mathrm{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \in \mathcal{C}_2$, we have $\forall \vec{a} \in \vec{D_{\mathsf{S}}}.\ r_{\mathsf{T}}([\![\mathsf{f}]\!]^1_{\mathrm{func}}(i_{\vec{\mathsf{S}}}(\vec{a}))) = [\![\mathsf{f}]\!]^2_{\mathrm{func}}(\vec{a})$,
(5) for every $\vdash_{\mathrm{pred}_2} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \in \mathcal{P}_2$, we have $\forall \vec{a} \in \vec{D_{\mathsf{S}}}.\ [\![\mathsf{p}]\!]^1_{\mathrm{pred}}(i_{\vec{\mathsf{S}}}(\vec{a}))$ *iff.* $[\![\mathsf{p}]\!]^2_{\mathrm{pred}}(\vec{a})$.

The model extension $M^1 \sqsupseteq_{\mathrm{model}} M^2$ ensures that, for the part described by Ω_2, the domains in are in bijection, the function interpretations are the same (up to the bijection), and the predicates are equivalent (up to the bijection).

Sort Interpretations

$$D_{\mathsf{unit}} \triangleq \{()\} \qquad D_{\mathsf{bool}} \triangleq \mathbb{B} \qquad D_{\mathsf{int}} \triangleq \mathbb{Z} \qquad D_{\mathsf{bv}\,n} \triangleq \{m \in \mathbb{Z} \mid 0 \le m < 2^n\} \qquad \cdots$$

Function Interpretations

$$[\![()]\!]_{\mathrm{func}}() \triangleq () \qquad [\![\mathsf{true}]\!]_{\mathrm{func}}() \triangleq \mathsf{true} \qquad [\![\mathsf{false}]\!]_{\mathrm{func}}() \triangleq \mathsf{true} \qquad [\![n]\!]_{\mathrm{func}}() \triangleq n$$

$$[\![+_{\mathsf{int}}]\!]_{\mathrm{func}}(n,m) \triangleq n+m \qquad [\![\cdot_{\mathsf{int}}]\!]_{\mathrm{func}}(n,m) \triangleq n \cdot m \qquad [\![-_{\mathsf{int}}]\!]_{\mathrm{func}}(n,m) \triangleq n-m$$

$$[\![\mathsf{neg}]\!]_{\mathrm{func}}(a) \triangleq \mathit{if}\ a\ \mathit{then}\ \mathsf{false}\ \mathit{else}\ \mathsf{true} \qquad [\![\mathsf{xor}_{\mathsf{bv}\,n}]\!]_{\mathrm{func}}(u_1,u_2) \triangleq u_1 \oplus u_2$$

$$[\![==_{\mathsf{bv}\,n}]\!]_{\mathrm{func}}(u_1,u_2) \triangleq \mathit{if}\ u_1 = u_2\ \mathit{then}\ \mathsf{true}\ \mathit{else}\ \mathsf{false} \qquad [\![\mathsf{if}_S]\!]_{\mathrm{func}}(a,d_1,d_2) \triangleq \mathit{if}\ a\ \mathit{then}\ d_1\ \mathit{else}\ d_2$$

$$\cdots$$

Predicate Interpretations

$$[\![\le_{\mathsf{int}}]\!]_{\mathrm{pred}}(n,m) \triangleq n \le m \qquad [\![<_{\mathsf{int}}]\!]_{\mathrm{pred}}(n,m) \triangleq n < m \qquad \cdots$$

Fig. 9. The base model M_{base}.

C.2 Interpreted Theories

An SMT-solver has built-in knowledge for several mathematical theories such as integers and bitvectors. To be sound, we have to make sure that we agree with the SMT-solver on these theories. To this end, we define the following *base signature* Ω_{base} (and *base model* M_{base} below):

$$\begin{array}{rrcl} \text{Sorts} & S, T & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{bv}\,n \mid \cdots \\ \text{Functions} & f, g & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid +_{\mathsf{int}} \mid \cdot_{\mathsf{int}} \mid -_{\mathsf{int}} \mid \mathsf{neg} \mid \mathsf{xor}_{\mathsf{bv}\,n} \mid ==_{\mathsf{bv}\,n} \mid \mathsf{if}_S \mid \cdots \\ \text{Predicates} & p, q & ::= & \le_{\mathsf{int}} \mid <_{\mathsf{int}} \mid \cdots \end{array}$$

The signature introduces several standard base sorts (such as integers int and bit vectors $\mathsf{bv}\,n$), along with canonical functions and predicates on them (*e.g.,* addition on integers, or comparison on integers, *etc.*). We then define a base model M_{base}, a standard interpretation for these sorts, functions, and predicates, depicted in Fig. 9.

With the base model in hand, we can then define the main validity judgment $\vDash \pi$:

Definition C.4. Let Ω_{smt} be a signature extending the base signature Ω_{base}, *i.e.,* $\Omega_{\mathrm{smt}} \sqsupseteq_{\mathrm{sig}} \Omega_{\mathrm{base}}$.

$$\vDash \pi \triangleq (\vdash_{\mathrm{form}} \pi) \Rightarrow \forall (M : \Omega_{\mathrm{smt}}).\ M \sqsupseteq_{\mathrm{model}} M_{\mathrm{base}} \Rightarrow M \vDash \pi$$

That is, π holds true *in all models extending the base model* M_{base}. Since the Tarski semantics is only defined for well-formed formulas, we assume the formula is well-formed in this definition.

$$\langle \mathsf{x} \rangle_{\mathsf{T}}^{\gamma} = \gamma(\mathsf{x}) \qquad \langle \mathsf{f}\, \vec{t} \rangle_{\mathsf{T}}^{\gamma} = \langle \mathsf{f} \rangle_{\mathsf{C}}\, \langle \vec{t} \rangle_{\mathsf{T}}^{\gamma} \qquad \langle \mathsf{True} \rangle_{\mathsf{F}}^{\gamma} \triangleq \mathsf{True} \qquad \langle \mathsf{False} \rangle_{\mathsf{F}}^{\gamma} \triangleq \mathsf{False}$$

$$\langle t \doteq_{\mathsf{S}} s \rangle_{\mathsf{F}}^{\gamma} \triangleq \langle t \rangle_{\mathsf{T}}^{\gamma} \equiv \langle s \rangle_{\mathsf{T}}^{\gamma} \qquad \langle \mathsf{p}\, \vec{t} \rangle_{\mathsf{F}}^{\gamma} \triangleq \langle \mathsf{p} \rangle_{\mathsf{P}}(\langle \vec{t} \rangle_{\mathsf{T}}^{\gamma}) \qquad \langle \pi_1 \mathbin{\dot\wedge} \pi_2 \rangle_{\mathsf{F}}^{\gamma} \triangleq \langle \pi_1 \rangle_{\mathsf{F}}^{\gamma} \wedge \langle \pi_2 \rangle_{\mathsf{F}}^{\gamma}$$

$$\langle \pi_1 \mathbin{\dot\vee} \pi_2 \rangle_{\mathsf{F}}^{\gamma} \triangleq \langle \pi_1 \rangle_{\mathsf{F}}^{\gamma} \vee \langle \pi_2 \rangle_{\mathsf{F}}^{\gamma} \qquad \langle \pi_1 \mathbin{\dot\Rightarrow} \pi_2 \rangle_{\mathsf{F}}^{\gamma} \triangleq \langle \pi_1 \rangle_{\mathsf{F}}^{\gamma} \Rightarrow \langle \pi_2 \rangle_{\mathsf{F}}^{\gamma}$$

$$\langle \dot\forall \mathsf{x} : \mathsf{S}.\, \pi \rangle_{\mathsf{F}}^{\gamma} \triangleq \forall v.\, v \in \mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!] \Rightarrow \langle \pi \rangle_{\mathsf{F}}^{\gamma, \mathsf{x} \mapsto v} \qquad \langle \dot\exists \mathsf{x} : \mathsf{S}.\, \pi \rangle_{\mathsf{F}}^{\gamma} \triangleq \exists v.\, v \in \mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!] \wedge \langle \pi \rangle_{\mathsf{F}}^{\gamma, \mathsf{x} \mapsto v}$$

Fig. 10. Translations of terms to expressions $\langle t \rangle_{\mathsf{T}}^{\gamma}$ and formulas to *hProp*-assertions $\langle \pi \rangle_{\mathsf{F}}^{\gamma}$, given a translation of sorts to types $\langle \mathsf{S} \rangle_{\mathsf{S}}$, function constants to values $\langle \mathsf{f} \rangle_{\mathsf{C}}$, and predicates to *hProp*-predicates $\langle \mathsf{p} \rangle_{\mathsf{P}}$.

Sort Translations

$$\langle \mathsf{unit} \rangle_{\mathsf{S}} \triangleq \mathsf{unit} \qquad \langle \mathsf{bool} \rangle_{\mathsf{S}} \triangleq \mathsf{bool} \qquad \langle \mathsf{int} \rangle_{\mathsf{S}} \triangleq \mathsf{int} \qquad \langle \mathsf{bv}\, n \rangle_{\mathsf{S}} \triangleq \mathsf{bv}\, n \qquad \cdots$$

Function Translations

$$\langle () \rangle_{\mathsf{C}} \triangleq () \qquad \langle \mathsf{true} \rangle_{\mathsf{C}} \triangleq \mathsf{true} \qquad \langle \mathsf{false} \rangle_{\mathsf{C}} \triangleq \mathsf{true} \qquad \langle n \rangle_{\mathsf{C}} \triangleq n \qquad \langle +_{\mathsf{int}} \rangle_{\mathsf{C}} \triangleq \lambda(x, y).\, x + y$$

$$\langle \cdot_{\mathsf{int}} \rangle_{\mathsf{C}} \triangleq \lambda(x, y).\, x * y \qquad \langle -_{\mathsf{int}} \rangle_{\mathsf{C}} \triangleq \lambda(x, y).\, x - y \qquad \langle \mathsf{neg} \rangle_{\mathsf{C}} \triangleq \lambda x.\, \sim x$$

$$\langle \mathsf{xor}_{\mathsf{bv}\, n} \rangle_{\mathsf{C}} \triangleq \lambda(x, y).\, x \ \mathsf{xor}\ y \qquad \langle ==_{\mathsf{bv}\, n} \rangle_{\mathsf{C}} \triangleq \lambda(x, y).\, x == y$$

$$\langle \mathsf{if}_{\mathsf{S}} \rangle_{\mathsf{C}} \triangleq \lambda(x, y_1, y_2).\, \mathbf{if}\ x\ \mathbf{then}\ y_1\ \mathbf{else}\ y_2 \qquad \cdots$$

Predicate Translations

$$\langle \leq_{\mathsf{int}} \rangle_{\mathsf{P}}(v_1, v_2) \triangleq \exists n_1, n_2.\, v_1 = n_1 \wedge v_2 = n_2 \wedge n_1 \leq n_2$$

$$\langle <_{\mathsf{int}} \rangle_{\mathsf{P}}(v_1, v_2) \triangleq \exists n_1, n_2.\, v_1 = n_1 \wedge v_2 = n_2 \wedge n_1 \leq n_2 \qquad \cdots$$

Fig. 11. Translations of sorts to semantic types $\langle \mathsf{S} \rangle_{\mathsf{S}}$, function constants to λ_{dyn}-values $\langle \mathsf{f} \rangle_{\mathsf{C}}$, and predicates to *hProp*-predicates $\langle \mathsf{p} \rangle_{\mathsf{P}}$.

D From First-Order Logic to Iris

The translation from first-order logic to *hProp*-assertions $\langle \pi \rangle_{\mathsf{F}}^{\gamma}$ is depicted in Fig. 10. We complete it with translations for sorts, functions, and predicates, depicted in Fig. 11. Note that the translation maps (1) *sorts* S to *semantic types* τ in our logical relation, (2) *functions* f to λ_{dyn}*-values* v in our programming language, and (3) *predicates* Φ to *hProp-predicates* $F(v)$ in the fragment of almost-pure assertions.[2] The translation can be extended, as needed, with uninterpreted sorts (*e.g.,* mapping buffer to buf bv 64) or uninterpreted functions (*e.g.,* mapping checksum to checksum).

We will now use it to show the following correspondence:

Let π be well-formed. If $\vDash \pi$ holds, then $(\langle \pi \rangle_{\mathsf{F}}^{\emptyset} \Rightarrow \mathsf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathsf{wp}\ e\ \{v.\, Q(v)\}$ holds in Iris.

As an aside, note that we assume that π is well-formed (*i.e.,* $\vdash_{\mathsf{form}} \pi$ according to the typing judgment in Fig. 7) here. Well-formedness ensures that the semantics defined in Fig. 8 is meaningful (*e.g.,* we do not apply a function that expects Boolean arguments to an integer). To ease the presentation, we have glossed over well-formedness in the paper.

[2]For better usability, in Rocq, the translation is actually aware of, *e.g.,* binary operations such that $\langle 41 +_{\mathsf{int}} 1 \rangle_{\mathsf{T}}$ is translated to the λ_{dyn}-expression $41 + 1$ instead of the λ_{dyn}-expression $(\lambda(x, y).\, x + y)\ (41, 1)$. We gloss over this detail in the following.

$$\lfloor \phi \rfloor(h) \triangleq \phi \qquad \lfloor e \Downarrow v \rfloor(h) \triangleq (e, h) \rightsquigarrow^*_{\text{det}} (v, h) \qquad \lfloor \ell \mapsto_{\text{u}} v \rfloor(h) \triangleq h(\ell) = v$$

$$\lfloor F \wedge G \rfloor(h) \triangleq \lfloor F \rfloor(h) \wedge \lfloor G \rfloor(h) \qquad \lfloor F \vee G \rfloor(h) \triangleq \lfloor F \rfloor(h) \vee \lfloor G \rfloor(h)$$

$$\lfloor F \Rightarrow G \rfloor(h) \triangleq \lfloor F \rfloor(h) \Rightarrow \lfloor G \rfloor(h) \qquad \lfloor \forall x.\, F\, x \rfloor(h) \triangleq \forall x.\, \lfloor F\, x \rfloor(h) \qquad \lfloor \exists x.\, F\, x \rfloor(h) \triangleq \exists x.\, \lfloor F\, x \rfloor(h)$$

Fig. 12. Translation $\lfloor \_ \rfloor(\_) : \mathit{hProp} \to \mathit{Heap} \to \mathit{Prop}$ between *hProp*-assertions F and pure assertions $\lfloor F \rfloor(h)$.

Proof overview. We will proceed in three steps: First, we connect *hProp*-assertions to meta-level, pure assertions (§D.1). Then, we connect the Tarski-semantics from Fig. 8 to *hProp*-assertions (§D.2). Finally, we put everything together to derive the correspondence above in Theorem D.8 (§D.3).

D.1 Connecting *hProp*-Assertions and Meta-Level Assertions

As the first step, we define a translation—*for a fixed heap h*—from *hProp*-assertions to Rocq assertions. The translation, $\lfloor \_ \rfloor(h)$, is depicted in Fig. 12. We then show the following correspondence:

Lemma D.1. *For any heap h, we have* $\mathsf{heap}_{\text{u}}(h) * (\ast_{\ell \mapsto v \in h} \ell \mapsto_{\text{u}} v) \vdash (F \Leftrightarrow \lfloor F \rfloor(h))$.

Proof. By induction on F.[3] For most connectives the proof is straightforward. (For implication $F_1 \Rightarrow F_2$, it is important that we prove an equivalence instead of merely an implication between the two.) An interesting case is evaluation $e \Downarrow v$. Here, we must show:

$$\mathsf{heap}_{\text{u}}(h) * (\ast_{\ell \mapsto v \in h} \ell \mapsto_{\text{u}} v) \vdash (e \Downarrow v \Leftrightarrow \lfloor e \Downarrow v \rfloor(h))$$

Considering the definition of $e \Downarrow v \triangleq \exists h'.\, (e, h') \rightsquigarrow^*_{\text{det}} (v, h') * (\ast_{\ell \mapsto w \in h'} \ell \mapsto_{\text{u}} w)$, the forward direction "$\Rightarrow$" follows from heap_{u} including all $\ell \mapsto_{\text{u}} v$ assertions concealed under the definition of $e \Downarrow v$. The backward direction "$\Rightarrow$" follows, because we have unstable points-to assertions $\ell \mapsto_{\text{u}} v$ in our assumption for every entry in h. □

D.2 Constructing a First-Order Model for λ_{dyn}

Equipped with the translation $\lfloor \_ \rfloor(h)$, we can now construct a first-order logic model using λ_{dyn}. We will use the model to instantiate $\vDash \pi$ (in §D.3).

The model M^h_{dyn}. We construct the model for an arbitrary, but fixed heap h, and write M^h_{dyn} to indicate the dependence on the heap h. First, we must instantiate the domains:

$$D_{\mathsf{S}} \triangleq \left\{ v \mid \lfloor v \in \mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!] \rfloor(h) \right\}$$

We instantiate the domains with sets of values v, where—using our translation $\lfloor \_ \rfloor(h)$—we know the value v is in the logical relation at type $\langle \mathsf{S} \rangle_{\mathsf{S}}$. Since first-order domains must be non-empty, we only do so for sorts where the corresponding value relation $\mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!]$ is non-empty.

For the predicates p, we simply choose:

$$[\![\mathsf{p}]\!]_{\text{pred}} \triangleq \lfloor \langle \mathsf{p} \rangle_{\mathsf{P}} \rfloor(h)$$

The first-order functions f are more interesting. We must instantiate the functions with meta-level functions for the interpretation $[\![\_]\!]_{\text{func}}$. To this end, we prove the following lemma, which turns program functions f into meta-level functions f:

[3] To obtain impredicative quantification in *hProp*, in Rocq, we actually make Lemma D.1 a *defining property* of *hProp*-assertions, together with $F \vdash \text{⋇} F$, and then prove that the *hProp*-connectives (*e.g.*, $e \Downarrow v$, $\ell \mapsto_{\text{u}} v$, $\forall x.\, F(x), \ldots$) satisfy these properties.

Lemma D.2. *Let* $\vdash \mathtt{f} \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. *For any* h, *there exists a function* $f : Val \to Val$ *such that for all values* v *where* $\lfloor v \in \tau_1 \rfloor(h)$, *we have* $(\mathtt{f}\, v, h) \rightsquigarrow^*_{det} (f(v), h)$ *and* $\lfloor f(v) \in \tau_2 \rfloor(h)$.

Proof. Given the definition of $\lfloor \_ \rfloor(h)$ and the equivalence in Lemma D.1, this lemma follows by first proving $\lfloor \mathtt{f} \in \mathcal{V}[\![\tau_1 \to \tau_2]\!] \rfloor(h)$, which unfolds to

$$\forall v.\ \lfloor v \in \tau_1 \rfloor(h) \Rightarrow \exists w.\ (\mathtt{f}\, v, h) \rightsquigarrow^*_{\mathrm{det}} (w, h) \wedge \lfloor w \in \tau_2 \rfloor(h)$$

and then applying the axiom of choice to extract w. □

To define $[\![\mathtt{f}]\!]_{\mathrm{func}}$, we apply Lemma D.2 (generalized to arbitrary function arities) to $\mathtt{f} \triangleq \langle \mathtt{f} \rangle_{\mathsf{C}}$.

Proving the extension. Having defined the model, we must then show that it is an extension of the base model M_{base}:

Lemma D.3. $M^h_{\mathrm{dyn}} \sqsupseteq_{\mathrm{model}} M_{base}$ *for any heap* h

Proof Sketch. Recall that the signature of M_{base} is Ω_{base}. For the sorts S in Ω_{base}, it is straightforward to construct a bijection between the domains D_{S} of the base model M_{base} and well-typed values in our semantic types $\mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!]$. The remaining equivalences of $M^h_{\mathrm{dyn}} \sqsupseteq_{\mathrm{model}} M_{\mathrm{base}}$ then follow by case analysis on our interpreted sorts, functions, and predicates. Here, we must make sure that the implementation of, *e.g.*, multiplication $\langle \cdot_{\mathsf{int}} \rangle_{\mathsf{C}} \triangleq \lambda xy.\ x * y$ (a *value* in λ_{dyn}) matches (up to the bijection) the semantics of multiplication, *i.e.*, $[\![\cdot_{\mathsf{int}}]\!]_{\mathrm{func}}(n, m) \triangleq n \cdot m$ (in ordinary mathematics on integers). □

Proving the equivalence. Equipped with the model, we can now show that formulas π interpreted in the first-order logic model M^h_{dyn} are equivalent to their translation $\langle \pi \rangle_{\mathsf{F}}$ at the meta-level if we apply $\lfloor \_ \rfloor(h)$. To state the relationship formally, we connect a first-order variable mapping ϵ (to elements of D_{S} in M^h_{dyn}) to a variable substitution γ (from variables to values) with:

$$\mathrm{agree}(\Sigma, \gamma, \epsilon) \triangleq \forall \mathsf{x} : \mathsf{S} \in \Sigma.\ \exists v.\ \gamma(\mathsf{x}) = v \wedge \epsilon(\mathsf{x}) = v$$

We first show for terms that their semantics corresponds to the evaluation in λ_{dyn}:

Lemma D.4. *Let* $\Sigma \vdash_{\mathsf{term}} t : \mathsf{S}$. *If* $\mathrm{agree}(\Sigma, \gamma, \epsilon)$, *then for every heap* h,

$$[\![t]\!]^{M^h_{\mathrm{dyn}}}_{\mathrm{term}}(\epsilon) = v \quad \textit{iff.} \quad (\langle t \rangle^{\gamma}_{\mathsf{T}}, h) \rightsquigarrow^*_{det} (v, h)$$

Proof. By induction on the typing $\Sigma \vdash_{\mathsf{term}} t : \mathsf{S}$. □

Then we use this result to show the desired equivalence:

Lemma D.5. *Let* $\Sigma \vdash_{\mathsf{form}} \pi$. *If* $\mathrm{agree}(\Sigma, \gamma, \epsilon)$, *then for every heap* h,

$$[\![\pi]\!]^{M^h_{\mathrm{dyn}}}_{\mathrm{form}}(\epsilon) \quad \textit{iff.} \quad \lfloor \langle \pi \rangle^{\gamma}_{\mathsf{F}} \rfloor(h)$$

Proof. By induction on the typing $\Sigma \vdash_{\mathsf{form}} \pi$, using Lemma D.4. □

As a corollary, we can connect valid formulas and the meta-level version of their translation:

Corollary D.6. *Let* π *be well-formed. If* $\vDash \pi$, *then* $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ *for any heap* h.

Proof. Fix a heap h. We instantiate $\vDash \pi$ with M^h_{dyn}, which extends the M_{base} (Lemma D.3). Thus, we obtain $[\![\pi]\!]^{M^h_{\mathrm{dyn}}}_{\mathrm{form}}(\emptyset)$. With Lemma D.5 it follows that $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ holds. □

D.3 Putting Everything Together with the Ambient Heap

To put everything together, as the last step, we need access to the ambient heap. That is, note that the equivalence in Lemma D.1 imposes a precondition, namely $\mathsf{heap}_{\mathsf{u}}(h) * (*_{\ell\mapsto v\in h}\, \ell \mapsto_{\mathsf{u}} v)$, requiring effectively unstable resources for the entire current heap. We call this precondition the *ambient heap* and define:

$$\mathsf{ambient\_heap\_is}(h) \triangleq \mathsf{heap}_{\mathsf{u}}(h) * (*_{\ell\mapsto v\in h}\, \ell \mapsto_{\mathsf{u}} v)$$
$$\mathsf{ambient\_heap} \triangleq \exists h.\, \mathsf{ambient\_heap\_is}(h)$$

Let us now first show that given an ambient heap, we can bring π from first-order logic to Iris:

Lemma D.7. *Let π be well-formed. If $\vDash \pi$ holds, then* $\mathsf{ambient\_heap} \vdash \langle\pi\rangle_{\mathsf{F}}^{\emptyset}$

Proof. Let $\vDash \pi$. It suffices to prove for all heaps h

$$\mathsf{ambient\_heap\_is}(h) \vdash \langle\pi\rangle_{\mathsf{F}}^{\emptyset}$$

By Lemma D.1, it then suffices to prove

$$(\langle\pi\rangle_{\mathsf{F}}^{\emptyset} \Leftrightarrow \lfloor\langle\pi\rangle_{\mathsf{F}}^{\emptyset}\rfloor(h)) \vdash \langle\pi\rangle_{\mathsf{F}}^{\emptyset}$$

By Lemma D.6 and $\vDash \pi$, we know $\lfloor\langle\pi\rangle_{\mathsf{F}}^{\emptyset}\rfloor(h)$ holds true for any heap h. Thus, we can use the equivalence $\langle\pi\rangle_{\mathsf{F}}^{\emptyset} \Leftrightarrow \lfloor\langle\pi\rangle_{\mathsf{F}}^{\emptyset}\rfloor(h)$ to establish $\langle\pi\rangle_{\mathsf{F}}^{\emptyset}$. □

We can obtain an ambient heap at any point when proving a weakest precondition for λ_{dyn}:

GET-AMBIENT-HEAP
$$(\forall h.\, \mathsf{ambient\_heap\_is}\, h \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathsf{wp}\ e\ \{v.\, Q(v)\}$$

Thus, putting both together, we obtain:

Theorem D.8.
Let π be well-formed. If $\vDash \pi$ holds, then $(\langle\pi\rangle_{\mathsf{F}}^{\emptyset} \Rrightarrow \mathsf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathsf{wp}\ e\ \{v.\, Q(v)\}$ holds in Iris.

Proof. We acquire the ambient heap via GET-AMBIENT-HEAP. Then, we apply Lemma D.7 on the ambient heap to obtain $\langle\pi\rangle_{\mathsf{F}}^{\emptyset}$. Finally, we use $\langle\pi\rangle_{\mathsf{F}}^{\emptyset} \Rrightarrow \mathsf{wp}\ e\ \{v.\, Q(v)\}$ to obtain $\mathsf{wp}\ e\ \{v.\, Q(v)\}$. □

Group	Case Study	Iris [5]	ViperCore [3]	Viper [7]	Daenerys
#1	Channel Library	●	○	○	●
	Checksum Exchange	○	○	●	●
#2	Popcount 32-bit Integer	◐	●	●	●
	Popcount Buffer *à la* Redis [8]	◐	●	●	●
	Priority Bit Map *à la* RefinedC [9]	●	●	●	●
#3	Iterative Linked-List	○	○	●	●
#4	Polymorphic Hashmap	○	○	●	●
#5	Iris Concurrent Logical Relation [11]	●	○	○	●
	Barrier [4], Reader-Writer Lock, Spinlock	●	○	○	●
Foundational Model		●	●	○	●

Fig. 13. Evaluation of Daenerys. We compare with the other approaches Iris, ViperCore, and Viper-based verifiers and mark whether they support the case study. We write ● for *yes*, ○ for *no*, and ◐ for case studies that *could conceivably be done* but require significant manual effort in Iris.

Implementation

```
chan() ≜ ref(None)
recv(c) ≜ let v = !c in
          match v with
           | None ⇒ recv(c)
           | Some(l) ⇒ if CAS(c, Some(l), None) then !l else recv(c)
send(c, x) ≜ let l = ref(x) in
             if CAS(c, None, Some(l)) then () else send(c, x)
```

Specification

$\{\mathsf{True}\}\ \mathsf{chan}()\ \{c.\ \mathsf{ischan}(c, \Phi)\}$ $\quad$ $\mathsf{persistent}(\mathsf{ischan}(c, \Phi))$

$\{\mathsf{ischan}(c, \Phi) * \boxplus\Phi(v)\}\ \mathsf{send}(c, v)\ \{\_.\ \mathsf{True}\}$ $\quad$ $\{\mathsf{ischan}(c, \Phi)\}\ \mathsf{recv}(c)\ \{v.\ \boxplus\Phi(v)\}$

Fig. 14. The channel implementation and specification.

E Case Studies

Below, we describe in more detail the case studies from the main paper, depicted in Fig. 13.

E.1 Best of Both Worlds (#1)

This example consists of two parts: verifying the channel implementation and verifying the actual exchange between client and worker thread. We first discuss the channel implementation and then proceed with how to verify the exchange.

Channel Library. The implementation of the channel library is depicted in Fig. 14. Each channel is represented as a reference to an option. The option is either None if there is currently no value being transferred over the channel, or Some(ℓ) where ℓ is a reference storing the value that is

currently being transferred from the sender to the receiver. (We add the additional indirection via a reference wrapping the value, because HeapLang and by extension λ_{dyn} forbid "compare-and-set" operations on references that store larger, composite values such as a pair.)

The three operations for channels are chan, send, and recv.

(1) The operation chan initializes an new channel by creating the channel reference. The reference initially stores None (*i.e.,* no value is currently being transferred).
(2) The operation send wraps the value it wants to send x in a reference l, and then tries to store Some(l) in the channel reference c. It does so via an atomic "compare-and-set" operation that tests whether the channel is currently empty (*i.e.,* currently storing None) and, if so, updates it to Some(l). If the update succeeds, send returns, and if it fails, send loops to wait for a point where the channel is empty again.
(3) The operation recv reads the contents v of the channel reference c. If they are None, then no value is currently being transferred and the recv operation loops to wait for a value. If there is some value currently being transferred (wrapped in a reference l) then recv tries to update c to None with a "compare-and-set" to claim the value. If the update succeeds, then the recv can return the contents of l (*i.e.,* the value being transferred). If the update fails, another call to recv must have been successful at receiving the value. In this case, recv loops to wait for the next value.

To verify the channel operations (depicted in Fig. 14), we use an Iris invariant:

$$\mathsf{ischan}(c, \Phi) \triangleq \exists \ell.\, c = \ell * \boxed{\exists v.\, \ell \mapsto v * (v = \mathsf{None} \lor \exists \ell', w.\, v = \mathsf{Some}(\ell') * \ell' \mapsto w * \boxplus \Phi(w))}^{\mathcal{N}}$$

It captures the two possible cases the channel reference ℓ can be in: either (1) no value is currently being transferred and the reference is storing None, or (2) some value w is being transferred, wrapped in some reference ℓ'. In the latter case, the invariant stores that Φ holds for the value being currently transferred. It does so underneath a frame modality to ensure that the ownership being transferred via the channel is stable. (Formally, this modality is needed to make sure that the contents of the invariant are frameable, a side condition of invariants in Daenerys.) With this invariant, verifying the specifications in Fig. 14 is a straightforward Iris proof. Moreover, the representation predicate $\mathsf{ischan}(c, \Phi)$ is persistent—and can be shared freely between threads and different program parts—because invariants are persistent.

What is interesting about the invariant above is that it stores an arbitrary Iris predicate Φ, which can include Hoare-triples (as we will see below) or even other invariants. The reason why such generality is allowed in Iris is that its invariants are *impredicative* [10] (*i.e.,* they that can contain arbitrary Iris propositions). Impredicativity allows us to state a general and modular specification for the channels in Fig. 14 (*e.g.,* for every channel, a new predicate Φ can be chosen). It does, however, also come at a cost: to justify soundness of its invariants, Iris uses step-indexing [1, 2], which adds significant complexity to its underlying model. Fortunately, in Daenerys, we effectively inherit the support for step-indexing from Iris and can, hence, use impredicative invariants to verify the channel implementation.

Checksum Exchange. Let us now turn to the exchange between the worker thread and the client (that uses the channel implementation to communicate between both). Recall the implementation:

$$\begin{aligned}
\mathsf{wrk}(\mathsf{i}, \mathsf{o}) &\triangleq \textbf{let}\ (\mathsf{p}, \mathsf{c}) = \mathsf{recv}(\mathsf{i})\ \textbf{in let}\ \mathsf{b} = \mathsf{p}()\ \textbf{in let}\ \mathsf{s} = c(\mathsf{b})\ \textbf{in}\ \mathsf{send}(\mathsf{o}, (\mathsf{b}, \mathsf{s}));\, \mathsf{wrk}(\mathsf{i}, \mathsf{o}) \\
\mathsf{client}() &\triangleq \textbf{let}\ (\mathsf{i}, \mathsf{o}) = (\mathsf{chan}(), \mathsf{chan}())\ \textbf{in} \quad \mathsf{fork}\,\{\mathsf{wrk}(\mathsf{i}, \mathsf{o})\}\,; \\
&\qquad \mathsf{send}(\mathsf{i}, (\mathtt{produceA}, \mathtt{checksumA}));\, \textbf{let}\ (\mathsf{b}, \mathsf{s}) = \mathsf{recv}(\mathsf{o})\ \textbf{in}\ \mathsf{assert}(\mathsf{s} == \mathtt{checksumA}(\mathsf{b})); \\
&\qquad \mathsf{send}(\mathsf{i}, (\mathtt{produceB}, \mathtt{checksumB}));\, \textbf{let}\ (\mathsf{b}, \mathsf{s}) = \mathsf{recv}(\mathsf{o})\ \textbf{in}\ \mathsf{assert}(\mathsf{s} == \mathtt{checksumB}(\mathsf{b}))
\end{aligned}$$

The worker wrk (1) receives on the input channel i a workload p (*i.e.,* a function that will produce a buffer) and a checksum function c, (2) produces the buffer b by executing p, (3) computes the checksum s of b, (4) sends both b and s back via the output channel o, and (5) repeats the entire process. The client client creates an input and an output channel, spawns the worker thread, and then sends the worker two different workloads: First, it sends `produceA` with checksum function `checksumA`, receives the result b and s, and ensures that s matches `checksumA` of b. Then, it repeats the same process with a *different workload and checksum* implementation.

Let us now discuss how we can verify this implementation given the channel specifications in Fig. 14. We start by introducing the two channel predicates, one for the input channel and one for the output channel:

$$\Phi_{\text{inp}}(p, c) \triangleq \{\mathsf{True}\}\ p()\ \{v.\ \exists b, \vec{u}.\ v = b * b \mapsto \vec{u}\} * \square(c \in \mathcal{V}[\![\mathsf{buf\ u64} \to \mathsf{u64}]\!]) * \gamma \rightrightarrows_{1/2}(p, c)$$
$$\Phi_{\text{outp}}(b, s) \triangleq \exists \vec{u}, p, c.\ b \mapsto \vec{u} * c(b) \Downarrow s * \gamma \rightrightarrows_{1/2}(p, c)$$

On the input channel (Φ_{inp}), the client sends a pair of *two functions* p and c. For p, it sends a Hoare-triple that p will compute a buffer. For c, it sends the fact that c is a well-typed function. In addition, it sends a custom piece of ghost state, a "fractional ghost variable" $\gamma \rightrightarrows_{1/2}(p, c)$, to track which workload the worker is currently working on. The fractional ghost variables $\gamma \rightrightarrows_q z$ behave like fractional points-to assertions $\ell \mapsto_q v$, but are not physically part of the program. As we will see below, the client keeps one half $\gamma \rightrightarrows_{1/2}(p, c)$ and sends the other half to the worker (via Φ_{inp}).

On the output channel (Φ_{outp}), the worker sends a pair of buffer b and checksum s. For b, it sends the ownership of the buffer $b \mapsto \vec{u}$. For s, it sends $c(b) \Downarrow s$, meaning s is the result of computing the checksum c on the buffer b. Along with them, it returns the ghost variable $\gamma \rightrightarrows_{1/2}(p, c)$.

Given these two predicates, the specifications that we have proven are:

$$\{\mathsf{True}\}\ \mathsf{client}()\ \{\_.\ \mathsf{True}\} \qquad \left\{\mathsf{ischan}(i, \Phi_{\text{inp}}) * \mathsf{ischan}(o, \Phi_{\text{outp}})\right\}\ \mathsf{wrk}(i, o)\ \{\_.\ \mathsf{True}\}$$

The client is safe to execute (*i.e.,* pre- and postcondition True), which means the asserts inside must succeed. The worker thread is safe to execute given an input channel adhering to Φ_{inp} and an output channel adhering to Φ_{outp}.

To illustrate the basic structure of the proof, we give a proof outline of the client in Fig. 15. Initially, we allocate the two channels (using the specification of chan in Fig. 14). We additionally allocate a new ghost variable γ initially storing an arbitrary pair of values. Next, we fork-off the worker thread. To do so, we give it a copy of the representation predicate of the input/output channel $\mathsf{ischan}(\mathsf{i}, \Phi_{\text{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\text{outp}})$ and keep another copy to ourselves.

Next, we start with the first checksum-and-produce combination. We update the ghost variable and split it into two halves: $\gamma \rightrightarrows_{1/2}(\mathtt{produceA}, \mathtt{checksumA}) * \gamma \rightrightarrows_{1/2}(\mathtt{produceA}, \mathtt{checksumA})$. Moreover, we prove the following specifications for `produceA` and `checksumA`:

$$\{\mathsf{True}\}\ \mathtt{produceA}()\ \{v.\ \exists b, \vec{u}.\ v = b * b \mapsto \vec{u}\} \qquad \vDash \mathtt{checksumA} : \mathsf{buf\ u64} \to \mathsf{u64}$$

Together, we can then send $\Phi_{\text{inp}}(\mathtt{produceA}, \mathtt{checksumA})$ to the worker thread via the input channel. (The assertion $\Phi_{\text{inp}}(\mathtt{produceA}, \mathtt{checksumA})$ is frameable, meaning $\Phi_{\text{inp}}(\mathtt{produceA}, \mathtt{checksumA}) \vdash \boxplus\Phi_{\text{inp}}(\mathtt{produceA}, \mathtt{checksumA})$, so we can cross the frame modality in the specification of send.)

Next, we receive on the output channel the resulting combination of buffer b and checksum s, meaning $\Phi_{\text{outp}}(\mathsf{b}, \mathsf{s})$. We have two haves of the ghost variable γ: $\gamma \rightrightarrows_{1/2}(\mathtt{produceA}, \mathtt{checksumA})$ and $\gamma \rightrightarrows_{1/2}(p, c)$ at this point. Just like for a fractional points-to, this means the two must agree (so $p = \mathtt{produceA}$ and $c = \mathtt{checksumA}$). Thus, the client has just received the knowledge of $\mathtt{checksumA}(\mathsf{b}) \Downarrow \mathsf{s}$. It can use it to justify that the assert succeeds.

Finally, we can combine the ghost variable again into one, and proceed with the second checksum-and-produce combination. It is completely analogous to the first one.

$\{\mathsf{True}\}$

let $(\mathsf{i}, \mathsf{o}) = (\mathsf{chan}(), \mathsf{chan}())$ **in**

$\{\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_1 (\_, \_)\}$

$\mathsf{fork}\ \{\mathsf{wrk}(\mathsf{i}, \mathsf{o})\}$;

$\{\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_1 (\_, \_)\}$

$\{\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceA}, \mathsf{checksumA}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceA}, \mathsf{checksumA})\}$

$\mathsf{send}(\mathsf{i}, (\mathsf{produceA}, \mathsf{checksumA}))$;

$\{\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceA}, \mathsf{checksumA})\}$

let $(\mathsf{b}, \mathsf{s}) = \mathsf{recv}(\mathsf{o})$ **in**

$\left\{\begin{array}{l}\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceA}, \mathsf{checksumA}) * \\ \boxplus\ (\exists \vec{u}, p, c.\ \mathsf{b} \mapsto \vec{u} * c(\mathsf{b}) \Downarrow \mathsf{s} * \gamma \rightrightarrows_{1/2} (p, c))\end{array}\right\}$

$\left\{\begin{array}{l}\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceA}, \mathsf{checksumA}) * \\ (\exists \vec{u}.\ \mathsf{b} \mapsto \vec{u} * \mathsf{checksumA}(\mathsf{b}) \Downarrow \mathsf{s} * \gamma \rightrightarrows_{1/2} (\mathsf{produceA}, \mathsf{checksumA}))\end{array}\right\}$

$\mathsf{assert}(\mathsf{s} == \mathsf{checksumA}(\mathsf{b}))$;

$\{\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_1 (\_, \_)\}$

$\{\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceB}, \mathsf{checksumB}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceB}, \mathsf{checksumB})\}$

$\mathsf{send}(\mathsf{i}, (\mathsf{produceB}, \mathsf{checksumB}))$;

$\{\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceB}, \mathsf{checksumB})\}$

let $(\mathsf{b}, \mathsf{s}) = \mathsf{recv}(\mathsf{o})$ **in**

$\left\{\begin{array}{l}\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceB}, \mathsf{checksumB}) * \\ \boxplus\ (\exists \vec{u}, p, c.\ \mathsf{b} \mapsto \vec{u} * c(\mathsf{b}) \Downarrow \mathsf{s} * \gamma \rightrightarrows_{1/2} (p, c))\end{array}\right\}$

$\left\{\begin{array}{l}\mathsf{ischan}(\mathsf{i}, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(\mathsf{o}, \Phi_{\mathrm{outp}}) * \gamma \rightrightarrows_{1/2} (\mathsf{produceB}, \mathsf{checksumB}) * \\ (\exists \vec{u}.\ \mathsf{b} \mapsto \vec{u} * \mathsf{checksumB}(\mathsf{b}) \Downarrow \mathsf{s} * \gamma \rightrightarrows_{1/2} (\mathsf{produceB}, \mathsf{checksumB}))\end{array}\right\}$

$\mathsf{assert}(\mathsf{s} == \mathsf{checksumB}(\mathsf{b}))$

$\{\mathsf{True}\}$

Fig. 15. Proof outline of the client.

Summary. In summary, in this example, we get the best of both worlds: Iris and IDF. We use Iris's impredicative invariants to verify the channel implementation above, and then we use the evaluation assertion $e \Downarrow v$ of Daenerys to send information about the current result of the checksum function over the output channel. The latter allows us to avoid proving functional correctness of *two different* checksum implementations. Instead, we can simply send the assertion $c(b) \Downarrow s$ from worker to client.

E.2 Leveraging SMT-Solvers (#2)

We consider three different examples:

Popcount 32-bit Integer. For the Popcount 32-bit Integer-example, we consider the following implementation to compute the number of ones in a 32-bit integer:

$$\begin{aligned}\mathsf{pc32}(\mathsf{u}) \triangleq\ & \textbf{let}\ \mathsf{v} = \mathsf{u} - (\mathsf{u} \gg 1\ \&\ \texttt{0x55555555})\ \textbf{in}\\ & \textbf{let}\ \mathsf{w} = (\mathsf{v}\ \&\ \texttt{0x33333333}) + ((\mathsf{v} \gg 2)\ \&\ \texttt{0x33333333})\ \textbf{in}\\ & (((\mathsf{w} + (\mathsf{w} \gg 4))\ \&\ \texttt{0x0F0F0F0F}) * \texttt{0x01010101}) \gg 24\end{aligned}$$

We show that it implements the following, naïve implementation:

$$\mathsf{ones}(\mathsf{u}) \triangleq (\mathsf{u} \gg 31)\ \&\ \texttt{0x1} + \cdots + (\mathsf{u} \gg 0)\ \&\ \texttt{0x1}$$

More specifically, for a bitvector u where $0 \le u < 2^{32}$, the specification that we prove is

$$\{\mathsf{True}\}\ \mathsf{pc32}(u)\ \{v.\ v \equiv \mathsf{ones}(u)\}$$

For that, we give the SMT solver an unfolding of the definitions of pc32 and ones and ask the following query:

$$\begin{aligned}\pi_{\mathrm{pc32}} \triangleq\ & (\forall x.\ \mathsf{aux1}(x) = x - ((x \gg 1)\ \&\ \texttt{0x55555555})) \Rightarrow\\ & (\forall x.\ \mathsf{aux2}(x) = (x\ \&\ \texttt{0x33333333}) + ((x \gg 2)\ \&\ \texttt{0x33333333})) \Rightarrow\\ & (\forall x.\ \mathsf{aux3}(x) = ((x + (x \gg 4))\ \&\ \texttt{0x0F0F0F0F})) \Rightarrow\\ & (\forall x.\ \mathsf{aux}(x) = \mathsf{aux3}(\mathsf{aux2}(\mathsf{aux1}(x)))) \Rightarrow\\ & (\forall x.\ \mathsf{pc32}(x) = (\mathsf{aux}(x)\ \&\ \texttt{0x01010101}) \gg 24) \Rightarrow\\ & (\forall x.\ \mathsf{ones}(x) = (x \gg 31)\ \&\ \texttt{0x1} + \cdots + (x \gg 0)\ \&\ \texttt{0x1}) \Rightarrow\\ & \quad \forall x.\ \mathsf{True} \Rightarrow \forall res.\ \mathsf{pc32}(x) = res \Rightarrow res = \mathsf{ones}(x)\end{aligned}$$

In Rocq we assume $\vDash \pi_{\mathrm{pc32}}$, instantiate the premises of the implication by manually proving the unfoldings (which is straightforward), and then we automatically derive the Hoare triple from the conclusion of the query.

Popcount Buffer. In the Popcount Buffer example (inspired by a similar implementation in Redis [8]), the implementation counts the ones in a buffer[4] by considering seven 32-bit integers at a time (in a loop over the buffer). Afterwards, for the remaining 0-6 integers, it looks up the corresponding ones in a table.

$$\begin{aligned}\mathsf{aux1}(x) &\triangleq x - ((x \gg 1)\ \&\ \texttt{0x55555555})\\ \mathsf{aux2}(x) &\triangleq (x\ \&\ \texttt{0x33333333}) + ((x \gg 2)\ \&\ \texttt{0x33333333})\\ \mathsf{aux3}(x) &\triangleq ((x + (x \gg 4))\ \&\ \texttt{0x0F0F0F0F})\\ \mathsf{aux}(x) &\triangleq \mathsf{aux3}(\mathsf{aux2}(\mathsf{aux1}(x)))\\ \mathsf{pc7}(x_1, x_v, x_3, x_4, x_5, x_6, x_7) &\triangleq ((\mathsf{aux}(x_1) + \mathsf{aux}(x_2) + \mathsf{aux}(x_3) + \mathsf{aux}(x_4) + \mathsf{aux}(x_5) + \mathsf{aux}(x_6) + \mathsf{aux}(x_7))\\ &\qquad * \texttt{0x01010101}) \gg 24\\ \mathsf{bitsinbyte} &\triangleq \#[0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 1, 2, 2,\\ &\qquad 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 1, 2, 2, 3, 2, 3, 3, 4,\\ &\qquad 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4,\end{aligned}$$

[4] To access the elements of the buffer, we use a function $\mathsf{buf\_get}(:, \mathsf{buffer}())\,\tau \times \mathsf{int} \to \tau$, which returns a default value in case the index is out of bounds.

```
                5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3,
                3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5,
                6, 4, 5, 5, 6, 5, 6, 6, 7, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6,
                5, 6, 6, 7, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8]
    lookup(x) ≜ bitsinbyte[int((x ≫ 0) & 0xFF)]
                + bitsinbyte[int((x ≫ 8) & 0xFF)]
                + bitsinbyte[int((x ≫ 16) & 0xFF)]
                + bitsinbyte[int((x ≫ 32) & 0xFF)]
  while p f ≜ rec while(f) := if p () then f (); while p f else ()
pc_redis(buf) ≜ let count = ref(length(buf)) in
                let i = ref(0) in
                let bits = ref(0) in
                while (λ_. 7 ≤ ! count) (
                  bits := ! bits + pc7(
                    buf_get(buf, ! i + 1),
                    buf_get(buf, ! i + 2),
                    buf_get(buf, ! i + 3),
                    buf_get(buf, ! i + 4),
                    buf_get(buf, ! i + 5),
                    buf_get(buf, ! i + 6)
                  );
                  count := ! count − 7;
                  i := ! i + 7;
                );
                while (λ_. 0 < ! count) (
                  bits := ! bits + lookup(buf_get(buf, ! i))
                  count := ! count − 1;
                  i := ! i + 1;
                );
                ! bits
```

We give unfoldings of all definitions excluding while and pc_redis to the SMT solver and ask whether they imply $\mathsf{pc7}(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = \mathsf{ones}(x_1) + \mathsf{ones}(x_2) + \mathsf{ones}(x_3) + \mathsf{ones}(x_4) + \mathsf{ones}(x_5) + \mathsf{ones}(x_6) + \mathsf{ones}(x_7)$ and $\mathsf{lookup}(x) = \mathsf{ones}(x)$. Then we do a manual proof by induction on the value stored in count to prove the following Hoare triple:

$$\{\mathsf{buffer}(\mathsf{buf})\}$$
$$\quad \mathsf{pc\_redis}(\mathsf{buf})$$
$$\{v.\ \ \mathsf{iter}(0,\ \mathsf{length}(\mathsf{buf}),\ 0,\ \lambda\, \mathsf{i}\, \mathsf{bits}.\, \mathsf{bits} + \mathsf{ones}(\mathsf{buf\_get}(\mathsf{buf},\ \mathsf{i}))) \Downarrow v\ \}$$

Priority Bit Map. Here we verify a version of a bit map implementation previously verified in RefinedC[5]. A bitmap $[0; 255) \mapsto [0, 1]$ containing 256 bits is efficiently represented as a vector of 4 64-bit integers and verified against the naïve meta logic representation as a vector of 256 Booleans. The oparations are:

- init initializes the bitmap to all 0
- highest returns the smallest index of a set bit (*i.e.,* of a bit that is 1)
- set_prio sets a bit to 1
- clear_prio sets a bit to 0

We consider the following implementation:

$$\begin{aligned}
\mathsf{getBit}(\mathsf{x}, \mathsf{i}) &\triangleq (\mathsf{nth}(\mathsf{x}, \mathsf{int}(\mathsf{i} \gg 6)) \gg (\mathsf{i} \,\&\, 63)) \,\&\, 1 \\
\mathsf{ffs}(\mathsf{x}) &\triangleq \begin{array}{lllll}
\textbf{if} & \mathsf{x} \,\&\, (1 \ll 0) \neq 0 & \textbf{then} & 1 & \textbf{else} \\
\textbf{if} & \mathsf{x} \,\&\, (1 \ll 1) \neq 0 & \textbf{then} & 2 & \textbf{else} \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
\textbf{if} & \mathsf{x} \,\&\, (1 \ll 63) \neq 0 & \textbf{then} & 64 & \textbf{else} \\
 & & & 0 &
\end{array} \\
\mathsf{init} &\triangleq \#[0, 0, 0, 0] \\
\mathsf{highest}(\mathsf{xs}) &\triangleq \begin{array}{llllll}
\textbf{if} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 0)) \neq 0 & \textbf{then} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 0)) + -1 & \textbf{else} \\
\textbf{if} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 1)) \neq 0 & \textbf{then} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 1)) + 63 & \textbf{else} \\
\textbf{if} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 2)) \neq 0 & \textbf{then} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 2)) + 127 & \textbf{else} \\
\textbf{if} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 3)) \neq 0 & \textbf{then} & \mathsf{ffs}(\mathsf{nth}(\mathsf{xs}, 3)) + 191 & \textbf{else} \\
 & & & -1 &
\end{array} \\
\mathsf{set_prio}(\mathsf{xs}, \mathsf{p}) &\triangleq \mathsf{set}(\mathsf{xs}, \mathsf{int}(\mathsf{p} \gg 6), (\mathsf{nth}(\mathsf{xs}, \mathsf{int}(\mathsf{p} \gg 6))) \mid (1 \ll (\mathsf{p} \,\&\, 63))) \\
\mathsf{clear_prio}(\mathsf{xs}, \mathsf{p}) &\triangleq \mathsf{set}(\mathsf{xs}, \mathsf{int}(\mathsf{p} \gg 6), (\mathsf{nth}(\mathsf{xs}, \mathsf{int}(\mathsf{p} \gg 6))) \,\&\, \sim(1 \ll (\mathsf{p} \,\&\, 63)))
\end{aligned}$$

We define a representation predicate to relate the operation to vectors of bool in the specs:

$$\mathsf{rep}(\mathsf{xs}, \mathsf{bs}) \triangleq \mathsf{len}(\mathsf{xs}) = 4 \land \mathsf{len}(\mathsf{bs}) = 256 \land \forall i.\, 0 \leq i < 256 \Rightarrow \mathsf{nth}(\mathsf{bs}, i) = (\mathsf{getBit}(\mathsf{xs}, i) \neq 0)$$

Then we prove the following Hoare triples by asking the SMT solver similar queries as above, i.e., whether the unfolding of all definitions implies the first-order version of the Hoare triple. We include the exact SMT queries in the accompanying Rocq development.

$$\{\mathsf{True}\}\ \mathsf{init}\ \{\mathsf{xs}.\ \mathsf{rep}(\mathsf{xs}, \#[\mathsf{false}, \ldots, \mathsf{false}])\}$$

$$\{\mathsf{rep}(\mathsf{xs}, \mathsf{bs})\}\ \mathsf{highest}(\mathsf{xs})\ \left\{p.\ \begin{array}{c}(p = -1 \land \forall i < 256.\ \mathsf{bs}[\mathsf{int}(i)] = \mathsf{false}) \\ \lor\ (p \neq -1 \land \mathsf{bs}[\mathsf{int}(p)] = \mathsf{true} \land \forall i < p.\ \mathsf{bs}[\mathsf{int}(i)] = \mathsf{false})\end{array}\right\}$$

$$\{\mathsf{rep}(\mathsf{xs}, \mathsf{ys})\}\ \mathsf{set\_prio}(\mathsf{xs}, \mathsf{p})\ \{\mathsf{xs}'.\ \mathsf{rep}(\mathsf{xs}', \mathsf{set}(\mathsf{bv}, \mathsf{int}(\mathsf{p}), \mathsf{true}))\}$$

$$\{\mathsf{rep}(\mathsf{xs}, \mathsf{ys})\}\ \mathsf{clear\_prio}(\mathsf{xs}, \mathsf{p})\ \{\mathsf{xs}'.\ \mathsf{rep}(\mathsf{xs}', \mathsf{set}(\mathsf{bv}, \mathsf{int}(\mathsf{p}), \mathsf{false}))\}$$

[5]The original code can be found here: https://gitlab.mpi-sws.org/iris/refinedc/-/blob/fecd1e6d396dde55e464a71d62e908d60fe920b9/examples/scheduler/include/fdsched/priority.h

```
new() ≜ None
len(l) ≜ match l with None ⇒ 0 | Some(x) ⇒ 1 + len(snd(! x))
nth(l, n) ≜ match l with
              | None ⇒ None
              | Some(x) ⇒ if n == 0 then Some(fst(! l)) else nth(snd(! l), n − 1)
push(l, v) ≜ let newl = ref(v, l) in Some(newl)
set(l, i, v) ≜ match l with
              | None ⇒ ()
              | Some(x) ⇒ if i == 0 then x ≔ (v, snd(! x)) else set(snd(! x), i − 1, v)
```

Fig. 16. Linked list implementation for the iterative verification.

E.3 Iterative Verification (#3)

We iteratively verify the functions new, set, and push of a standard linked-list implementation, depicted in Fig. 16. Each list is either None for the empty list or Some(ℓ) where ℓ is a reference to a pair of the the head and tail of the list. We define the following abstract predicate for the ownership of a list:

$$\mathsf{list}(l) \triangleq l = \mathsf{None} \lor \exists \ell, n, l'.\, l = \mathsf{Some}(\ell) * \ell \mapsto (n, l') * \mathsf{list}(l')$$

We verify three different kinds of specifications: (1) with respect to ownership of the list for memory safety, (2) with respect to their effect on the length of the list and, (3) with respect to the contents of the list for functional correctness. The resulting specifications are depicted in Fig. 17.

Old expressions. The specifications use so-called "old expressions" **old** $\{e\}$, which refer to the value of e in the precondition. To encode them in Daenerys, we define a new Hoare triple

$$\{P\}\ e\ \{v, \mathbf{old}.\ Q(v, \mathbf{old})\}_{\mathrm{old}}$$

It gets access in the postcondition to **old** : *Expr* $\to$ *Val*, a function from expressions to their resulting values if executed in the heap of the precondition. We define it using the following lemma:

Lemma E.1. *There exists a function* eval : *Heap* $\to$ *Expr* $\to$ *Val such that if* $(e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h)$, *then* eval $h\ e = v$.

Proof. The proof uses classical reasoning. For any heap h and expression e, classically either $\exists v.\ (e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h)$ or $\neg(\exists v.\ (e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h))$. In the first case, we can use the axiom of choice to return the value v. In the second case, it does not matter which value eval returns. □

Given the eval function, we define the Hoare-triple using the *ambient heap* (see §D.3) as:

$$\{P\}\ e\ \{v, \mathbf{old}.\ Q(v, \mathbf{old})\}_{\mathrm{old}} \triangleq \Box(\forall h.\ \boxplus P \mathrel{-\!\!*} \mathsf{ambient\_heap\_is}\ h \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ \boxplus Q(v, \mathsf{eval}\ h)\})$$

Compared to the regular Hoare-triples $\{P\}\ e\ \{v.\ Q(v)\}$, this version additionally assumes an ambient heap h, the current heap at the point where we prove the weakest precondition. In the postcondition, we then use eval partially instantiated with the heap h.

For postconditions that do not care about old expressions, this Hoare triple is the same as the regular Hoare-triple $\{P\}\ e\ \{v.\ Q(v)\}$, since the weakest precondition can always get access to the ambient heap (see get-ambient-heap in §D.3).

Ownership Specification

$$\{\mathsf{True}\}\ \mathsf{new}()\ \{l, \_.\ \mathsf{list}(l)\}_{\mathrm{old}} \qquad \{\mathsf{list}(l)\}\ \mathsf{push}(l, n)\ \{l', \_.\ \mathsf{list}(l')\}_{\mathrm{old}}$$

$$\{\mathsf{list}(l)\}\ \mathsf{set}(l, i, n)\ \{\_, \_.\ \mathsf{list}(l)\}_{\mathrm{old}}$$

Length Specification

$$\{\mathsf{True}\}\ \mathsf{new}()\ \{l, \_.\ \mathsf{list}(l) * \mathsf{len}(l) \Downarrow 0\}_{\mathrm{old}}$$

$$\{\mathsf{list}(l)\}\ \mathsf{push}(l, n)\ \{l', \mathbf{old}.\ \mathsf{list}(l') * \mathsf{len}(l') - 1 \Downarrow \mathbf{old}\ \{\mathsf{len}(l)\}\}_{\mathrm{old}}$$

$$\{\mathsf{list}(l)\}\ \mathsf{set}(l, i, n)\ \{\_, \mathbf{old}.\ \mathsf{list}(l) * \mathsf{len}(l) \Downarrow \mathbf{old}\ \{\mathsf{len}(l)\}\}_{\mathrm{old}}$$

Functional Correctness Specification

$$\{\mathsf{True}\}\ \mathsf{new}()\ \{l, \_.\ \mathsf{list}(l) * \mathsf{len}(l) \Downarrow 0 * \forall(i : \mathbb{Z}).\ \mathsf{nth}(l, i) \Downarrow \mathsf{None}\}_{\mathrm{old}}$$

$$\{\mathsf{list}(l)\}\ \mathsf{push}(l, n)\ \left\{l', \mathbf{old}.\ \begin{array}{l}\mathsf{list}(l') * \mathsf{len}(l') - 1 \Downarrow \mathbf{old}\ \{\mathsf{len}(l)\} * \mathsf{nth}(l', 0) \Downarrow \mathsf{Some}(n)\\ * \forall(j : \mathbb{Z}).\ 0 \leq_{\mathsf{hp}} j \Rightarrow \mathsf{nth}(l', j+1) \Downarrow \mathbf{old}\ \{\mathsf{nth}(l, j)\}\end{array}\right\}_{\mathrm{old}}$$

$$\{\mathsf{list}(l)\}\ \mathsf{set}(l, i, n)\ \left\{\_, \mathbf{old}.\ \begin{array}{l}\mathsf{list}(l) * \mathsf{len}(l) \Downarrow \mathbf{old}\ \{\mathsf{len}(l)\}\\ * (0 \leq_{\mathsf{hp}} i \wedge i <_{\mathsf{hp}} \mathsf{len}(l) \Rightarrow \mathsf{nth}(l, i) \Downarrow \mathsf{Some}(n))\\ * (\forall(j : \mathbb{Z}).\ 0 \leq_{\mathsf{hp}} j \wedge j <_{\mathsf{hp}} \mathsf{len}(l) \wedge i \neq_{\mathsf{hp}} j \Rightarrow \mathsf{nth}(l, j) \Downarrow \mathbf{old}\ \{\mathsf{nth}(l, j)\})\end{array}\right\}_{\mathrm{old}}$$

Fig. 17. Three different types of linked-list function specifications, ordered by strength, where we abbreviate $e_1 \leq_{\mathsf{hp}} e_2 \triangleq \exists n_1, n_2.\ e_1 \Downarrow n_1 \wedge e_2 \Downarrow n_2 \wedge n_1 \leq n_2$ and $e_1 <_{\mathsf{hp}} e_2 \triangleq \exists n_1, n_2.\ e_1 \Downarrow n_1 \wedge e_2 \Downarrow n_2 \wedge n_1 < n_2$ and $e_1 \neq_{\mathsf{hp}} e_2 \triangleq \exists v_1, v_2.\ e_1 \Downarrow v_1 \wedge e_2 \Downarrow v_2 \wedge v_1 \neq v_2$.

Specification

```
list_empty() ≜ None
list_cons(val, l) ≜ Some(ref(val, l))
list_find(eq, key, l) ≜ match l with
                        | None ⇒ None
                        | Some(p) ⇒
                            let (hd, tl) = ! p in
                            let (k, val) = hd in
                            if eq(key, k) then Some(val) else list_find(eq, key, tl)
list_update(eq, key, val, l) ≜ match l with
                        | None ⇒ (list_cons((key, val), None), false)
                        | Some(p) ⇒
                            let (hd, tl) = ! p in
                            let (k, val) = hd in
                            if eq(key, k) then p := ((k, val), tl); (Some(p, true)) else
                                let (tl′, updated) = list_update(eq, key, val, tl) in
                                p := (hd, tl′); (Some(p), updated)
```

Representation Predicate

$$\mathsf{islist}(x, []) \triangleq x = \mathsf{None}$$
$$\mathsf{islist}(x, (k, v) :: vs) \triangleq \exists l, w.\ x = \mathsf{Some}(l) \land l \mapsto ((k, v), w) \land \mathsf{islist}(w, vs)$$

Fig. 18. The linked list implementation parameterized by a custom eq.

E.4 Polymorphic Hashmap (#4)

In this example, we consider a polymorphic hashmap implementation. The hashmap is parameterized by a custom equality function eq and a hash function hash. The implementation is depicted in Fig. 19. It uses a linked-list implementation depicted in Fig. 18. The hashmap is implemented as an array of "buckets" (*i.e.,* linked-lists storing key-value pairs).

Specifications. The specifications that we prove are depicted in Fig. 20. They are parametric over a key type $K : Val \to hProp$ and a persistent predicate $K_o : Val \to iProp$ for ownership of the keys, where we require the following two properties:

$$\forall k.\ K_o(k) \vdash K(k) \qquad\qquad \forall k.\ \mathsf{persistent}(K_o(k))$$

For an equality function eq and a hash function hash, the function hashmap_create allows us to create a new hashmap (of non-zero size). We will define its precondition iseqhash below. It returns an empty hashmap. To track ownership and contents of the hashmap, we use the abstract predicate ishashmap(eq, h, m). It tracks the physical value corresponding to the hashmap h, the equality function eq that was used to create the hashmap, and the current contents m.

Once created, we can resize a hashmap with hashmap_resize, which will return a new hashmap (of increased capacity).

```
hashmap_create(eq, hash, cap) ≜ (eq, hash, cap, allocN(cap, list_empty()))[6]
hashmap_get(hm, key) ≜ let (eq, hash, cap, arr) = hm in
                       let idx = hash(key) % cap in
                       let b = ! (arr + idx) in
                       list_find(eq, key, b)
hashmap_put(hm, key, val) ≜ let (eq, hash, cap, arr) = hm in
                            let idx = hash(key) % cap in
                            let b = ! (arr + idx) in
                            let (b′, updated) = list_update(eq, key, val, b) in
                            arr + idx := b′; updated
iter_bucket(bucket, f) ≜ match bucket with
                         | None ⇒ ()
                         | Some(b) ⇒
                             let (hd, tl) = ! b in
                             let (k, v) = hd in
                             iter_bucket(tl, f); f k v
iter_buckets(arr, i, cap, f) ≜ if i < cap then
                                 let bucket = ! (arr + i) in
                                 iter_buckets(arr, i + 1, cap, f);
                                 iter_bucket(bucket, f) else ()
hashmap_resize(hm) ≜ let (eq, hash, cap, arr) = hm in
                     let map = hashmap_create(eq, hash, 2 ∗ cap) in
                     iter_buckets(arr, 0, cap, λ k v. hashmap_put(map, k, v));
                     map
```

Fig. 19. The hashmap implementation.

We can get an element for key k in the hashmap with hashmap_get. It returns either an element from the hashmap such that comparison with the equality function eq results in true, or it returns no element and we know that every key in the hashmap is not equal to k according to eq.

We can set an element for key k with hashmap_put. It returns true if the key was already contained (up to eq) in the hashmap and an update has been performed. It returns false if the key was not present (up to eq) and instead the new key-value pair has been added to the map.

[6]The operation allocN(n, v) allocates an array of length n, where each element is initialized with v.

$$\{0 < c * \mathsf{iseqhash}(\mathsf{eq}, \mathsf{hash})\}\quad \mathsf{hashmap\_create}(\mathsf{eq}, \mathsf{hash}, c)\quad \{u.\ \mathsf{ishashmap}(\mathsf{eq}, u, [])\}$$

$$\{\mathsf{ishashmap}(\mathsf{eq}, h, m)\}\quad \mathsf{hashmap\_resize}(h)\quad \{u.\ \mathsf{ishashmap}(\mathsf{eq}, u, m)\}$$

$$\{\mathsf{ishashmap}(\mathsf{eq}, h, m) * K_o(k)\}\quad \mathsf{hashmap\_get}(h, k)$$
$$\left\{u.\ \mathsf{ishashmap}(\mathsf{eq}, h, m) * \begin{array}{l}(\exists k, w.\ u = \mathsf{Some}(w) * (k', w) \in m * \square\, \mathsf{eq}(k, k') \Downarrow \mathsf{true}) \\ \qquad\qquad \vee \\ (u = \mathsf{None} * \square\, \forall (k', \_) \in m.\ \mathsf{eq}(k, k') \Downarrow \mathsf{false})\end{array}\right\}$$

$$\{\mathsf{ishashmap}(\mathsf{eq}, h, m) * K_o(k)\}\quad \mathsf{hashmap\_put}(h, k, v)$$
$$\left\{u.\ \exists b.\ u = b * \textbf{if } b \begin{array}{l}\textbf{then } \exists k', i.\ \mathsf{ishashmap}(\mathsf{eq}, h, m[i \mapsto (k', v)]) * m[i] = (k', \_) * \square\, \mathsf{eq}(k, k') \Downarrow \mathsf{true} \\ \textbf{else } \mathsf{ishashmap}(\mathsf{eq}, h, (k, v) :: m) * \square\, \forall (k', \_) \in m.\ \mathsf{eq}(k, k') \Downarrow \mathsf{false}\end{array}\right\}$$

Fig. 20. The hashmap specification.

The equality and hash function. Let us now focus on the properties that we require of the functions eq and hash when creating a hashmap:

$$\begin{aligned}
\mathsf{refl}(\mathsf{eq}) &\triangleq \mathsf{smt}\ (\forall x.\ K(x) \Rightarrow \mathsf{eq}(x, x) \Downarrow \mathsf{true}) \\
\mathsf{sym}(\mathsf{eq}) &\triangleq \mathsf{smt}\ (\forall x, y.\ K(x) \Rightarrow K(y) \Rightarrow \mathsf{eq}(x, y) \Downarrow \mathsf{true} \Rightarrow \mathsf{eq}(y, x) \Downarrow \mathsf{true}) \\
\mathsf{trans}(\mathsf{eq}) &\triangleq \mathsf{smt} \begin{pmatrix}\forall x, y, z.\ K(x) \Rightarrow K(y) \Rightarrow K(z) \Rightarrow \\ \qquad \mathsf{eq}(x, z) \Downarrow \mathsf{true} \Rightarrow \mathsf{eq}(x, y) \Downarrow \mathsf{true} \Rightarrow \mathsf{eq}(z, y) \Downarrow \mathsf{true}\end{pmatrix} \\
\mathsf{eqhash}(\mathsf{eq}, \mathsf{hash}) &\triangleq \mathsf{smt}\ (\forall x, y.\ K(x) \Rightarrow K(y) \Rightarrow \mathsf{eq}(x, y) \Downarrow \mathsf{true} \Rightarrow \mathsf{hash}(x) \equiv \mathsf{hash}(y)) \\
\mathsf{eqty}(\mathsf{eq}) &\triangleq \square(\mathsf{eq} \in \mathcal{V}[\![K \times K \to \mathsf{bool}]\!]) \\
\mathsf{hashty}(\mathsf{hash}) &\triangleq \square(\mathsf{hash} \in \mathcal{V}[\![K \to \mathsf{int}]\!]) \\
\mathsf{iseq}(\mathsf{eq}) &\triangleq \mathsf{sym}(\mathsf{eq}) * \mathsf{trans}(\mathsf{eq}) * \mathsf{refl}(\mathsf{eq}) * \mathsf{eqty}(\mathsf{eq}) \\
\mathsf{iseqhash}(\mathsf{eq}, \mathsf{hash}) &\triangleq \mathsf{iseq}(\mathsf{eq}) * \mathsf{hashty}(\mathsf{hash}) * \mathsf{eqhash}(\mathsf{eq}, \mathsf{hash}) \\
\mathsf{smt}\ (P) &\triangleq \square(\mathsf{ambient_heap} \mathrel{-\!\!*} P)
\end{aligned}$$

The function eq must be of type $K \times K \to \mathsf{bool}$, and the hash function hash of type $K \to \mathsf{int}$. The equality function must be reflexive (refl), symmetric (sym), and transitive (trans). Moreover, equal elements must have the same hash (eqhash). Since we wish to be able to establish reflexivity, symmetry, transitivity, and the eq-hash relation with an SMT solver, we wrap them with smt (_). It adds the ambient heap (see §D.3) as a condition to the properties, because this is what the first-order correspondence gives us when we come from first-order logic (see Lemma D.7 in §D.3).

The hashmap predicate. Let us now turn to the hashmap predicate. Each bucket in the hashmap is represented as a linked list, with each element being a pair of a key and a value. Given an eq and

hash function, we define the representation predicate for a bucket as follows:

$$\mathsf{bucketeq}(b) \triangleq \forall i, j, k, k', v, v'.\ b_i = (k, v) \Rightarrow b_j = (k', v') \Rightarrow i \neq j \Rightarrow \mathsf{eq}(k, k') \Downarrow \mathsf{false}$$

$$\mathsf{buckethash}(b, c, i) \triangleq \forall (k, \_) \in b.\ (\mathsf{hash}(k)\ \%\ c) \Downarrow i$$

$$\mathsf{isbucket}(b, c, i, l) \triangleq \Box\, \mathsf{bucketeq}(b) * \Box\, \mathsf{buckethash}(b, c, i) * \exists v.\ l + i \mapsto v * \mathsf{islist}(v, b) * \mathop{\Large *}_{(k,\_) \in b} K_o(k)$$

A hashmap is represented as a tuple of the equality function, the hash function, the capacity, and an array of buckets. The representation predicate for the hashmap is defined as follows:

$$\mathsf{ishashmap}(\mathsf{eq}, h, m) \triangleq \exists\, \mathsf{hash}, c, l, C.\ h = (\mathsf{eq}, \mathsf{hash}, c, l) * |C| = c * c > 0 * \mathsf{concat}(C) \equiv_p m$$
$$* \ \mathsf{iseqhash}(\mathsf{eq}, \mathsf{hash}) * \mathop{\Large *}_{i=1,\ldots,c} \mathsf{isbucket}(C_i, c, i, l)$$

where "$\mathsf{concat}(C) \equiv_p m$" means m is a permutation of the list obtained by concatenating the list of lists C that tracks the contents of the individual buckets in the hashmap.

E.5 Iris Examples (#5)

The modified code and the original version are provided as part of the accompanying Rocq development.

References

[1] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.* 32, 3 (2010), 7:1–7:67. https://doi.org/10.1145/1709093.1709094

[2] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

[3] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 2024. Formal Foundations for Translational Separation Logic Verifiers (extended version). arXiv:2407.20002 [cs.PL] https://arxiv.org/abs/2407.20002

[4] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. ACM, 256–269. https://doi.org/10.1145/2951913.2951943

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

[6] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[7] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[8] Redis. 2024. Redis Pocount Implementation for Potentially Large Buffers. https://github.com/redis/redis/blob/3fac869f02657d94dc89fab23acb8ef188889c96/src/bitops.c#L40.

[9] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036

[10] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS, Vol. 8410)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[11] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* (July 2024). https://doi.org/10.1145/3676954