Iron: Managing Obligations in Higher-Order Concurrent Separation Logic

ALEŠ BIZJAK, Aarhus University, Denmark
DANIEL GRATZER, Aarhus University, Denmark
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
LARS BIRKEDAL, Aarhus University, Denmark

Precise management of resources and the obligations they impose, such as the need to dispose of memory, close locks, and release file handles, is hard—especially in the presence of concurrency, when some resources are shared, and different threads operate on them concurrently. We present **Iron**, a novel higher-order concurrent separation logic that allows for precise reasoning about resources that are transferable among dynamically allocated threads. In particular, Iron can be used to show the correctness of challenging examples, where the reclamation of memory is delegated to a forked-off thread. We show soundness of Iron by means of a model of Iron, defined on top of the Iris base logic, and we use this model to prove that memory resources are accounted for precisely and not leaked. We have formalized all of the developments in the Coq proof assistant.

CCS Concepts: • **Theory of computation → Separation logic**; *Program verification*; Programming logic; Operational semantics;

Additional Key Words and Phrases: Separation logic, concurrency, resource management

ACM Reference Format:
Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *Proc. ACM Program. Lang.* 3, POPL, Article 65 (January 2019), 30 pages. https://doi.org/10.1145/3290378

1 INTRODUCTION

To enable reasoning about resources in the presence of concurrency, a plethora of variants of concurrent separation logic (CSL) have been proposed, *e.g.*, [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Feng 2009; Feng et al. 2007; Fu et al. 2010; Hobor et al. 2008; Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a; Mansky et al. 2017; Nanevski et al. 2014; O'Hearn 2007; Svendsen and Birkedal 2014; Turon et al. 2013; Vafeiadis and Parkinson 2007]. Despite their increased expressiveness and increased sophistication to provide modular specifications of program modules, none of these variants of separation logic can both:

(1) reason locally about unstructured *fork-style* concurrency, and,
(2) prove that resources are *necessarily* used, *e.g.,* that a program module is obligated to free all the memory it has allocated, or that it is obligated to released all the locks it has acquired.

Authors' addresses: Aleš Bizjak, Aarhus University, abizjak@cs.au.dk; Daniel Gratzer, Aarhus University, gratzer@cs.au.dk; Robbert Krebbers, Delft University of Technology, mail@robbertkrebbers.nl; Lars Birkedal, Aarhus University, birkedal@cs.au.dk.

2475-1421/2019/1-ART65
https://doi.org/10.1145/3290378

The combination of these features is important in practice, where an often used programming pattern is to transfer resources between dynamically allocated threads. For instance, one could have a background thread that acts as a garbage collector to clear up unused resources, as used in Singularity OS [Fähndrich et al. 2006]. For concreteness, let us consider a very simple example illustrating the transfer of resources from one thread to another (which could be thought of as the garbage collector) that then deallocates the transferred resources:

```
let ℓ = ref (None) in
let rec cleanup() = match !ℓ with
                       None    ⇒ cleanup()
                     | Some ℓ′ ⇒ free(ℓ′); free(ℓ)
                    end in fork {cleanup()} ;
let ℓ′ = ref (0) in ℓ ← Some ℓ′
```

The idea of this example is that the location ℓ acts as a channel. The main thread forks off a new thread, cleanup, which waits until it receives a message with a location ℓ', which it then deallocates, and then it also deallocates the channel ℓ. After forking off the cleanup thread, the main thread allocates a resource, the location ℓ', which is sent to the forked-off thread (so that the forked-off thread may deallocate it).

While existing variants of concurrent separation logic can be used to prove functional correctness of programs that transfer ownership of resources between dynamically allocated threads, none of these logics can be used to establish that such programs correctly dispose of all resources (in the example, to establish that memory resources are not leaked).

Technical problem. For reasoning about (non-disjoint) shared-memory concurrency, most variants of concurrent separation logic employ some form of an *invariant* mechanism to share ownership of resources among threads. When the programming language only includes structured concurrency, it suffices to use CSL-style *scoped* invariants [Nanevski et al. 2014; O'Hearn 2007], where the program syntax is used to delineate how invariants are shared between threads.

In contrast, variants of concurrent separation logic that can reason about *fork*-style concurrency, such as iCap [Svendsen and Birkedal 2014], TaDa [da Rocha Pinto et al. 2014], and Iris [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a], use *shareable* invariants that are not tracked in the logic. Technically, shareable means that such invariants are *persistent*, hence *duplicable*, and hence they need not be explicitly tracked in pre- and postconditions of Hoare triples. One can thus silently throw away resources, such as the "points-to" connective $\ell \mapsto v$ that expresses ownership of a location ℓ, by putting these resources in a shareable invariant, and throwing away that invariant.

Solution and key ideas. In this paper we present a new higher-order concurrent separation logic called **Iron**, which supports precise reasoning about resources in the presence of fork-style concurrency. In order to provide a combination of expressiveness and high-level abstraction, the Iron logic is defined in terms of two layers:

- The core **Iron** logic, which provides a new notion of a *trackable resources.* Concretely, for the case of memory resources, Iron provides the *trackable points-to connective* $\ell \mapsto_\pi v$, which not only denotes ownership of a location ℓ, but also includes a *fraction* π to control sharing. Intuitively, this fraction π expresses the *degree of knowledge* of the heap; if $\pi = 1$, then we know the heap exactly, and if $\pi < 1$, then we only have a partial local view of the heap (other threads may have complementary views of the heap).
  Importantly, trackable resources allow precise tracking of resources even when they are transferred through shareable invariants. In fact, trackable resources allow precise tracking of

resources even when the separation logic is *affine*, *i.e.*, it satisfies the *weakening rule* $P * Q \vdash P$, which allows to forget arbitrary resources. Iron is thus an affine separation logic.

- The high-level **Iron**$^{++}$ logic, which provides a layer on top of the core Iron logic to hide the use of fractions. We prove that this more abstract logic satisfies the standard rules of classical separation logic. In fact, we show that Iron^{++} is *not* affine, but rather linear—in other words, we have defined a linear separation logic (Iron^{++}) on top of an affine one (Iron).
  At the cost of hiding the fractions, Iron^{++} does *not* provide shareable invariants. Instead, it provides a new form of invariants that we call *trackable invariants*. These invariants are tracked in the logic (appear in pre- and postconditions), but are not scoped. Importantly, contrary to shareable invariants, our trackable invariants are not freely duplicable, but instead do support a controlled form of sharing via a so-called splitting rule.

Working in the combination of these logics is crucial. The Iron^{++} logic provides more abstract reasoning principles, which are sufficient for most practical cases, including the challenging *channel module* inspired by Singularity OS (Section 4.3). However, there are some constructs that cannot be verified in the abstract Iron^{++} logic. Notably, to establish a modular specification of a parallel composition operator defined using a reference cell and fork, one needs "drop down" to the core Iron logic (with explicit fractions) and prove that construct there. Partly for this reason, and partly to explain how the reasoning works, we have decided to start with an explicit treatment of Iron using fractions (Section 3-4), and then only later describe the more abstract Iron^{++} (Section 5) rather than the other way round.

In order to establish that Iron indeed enables precise reasoning of resources, we prove an *adequacy* theorem (Section 3.4), which guarantees that given an appropriate Hoare triple, when all threads terminate, all memory resources have indeed been disposed of.

We have modeled Iron on top of the *Iris base logic* [Jung et al. 2018; Krebbers et al. 2017a]. This simplifies the adequacy proof and model construction of Iron significantly (*e.g.*, we do not have to solve recursive domain equations, that is done in the model of Iris). Furthermore, it allows us to reuse Iris's features for shareable invariants and (higher-order) ghost state, and Iris's infrastructure for mechanized proofs in the Coq proof assistant [Krebbers et al. 2018, 2017b].

Contributions. In summary, we make the following contributions:

- We present *trackable resources*, a new construction for precise accounting of resources in separation logics with rules for weakening, either through shareable invariants, or through the weakening rule $P * Q \vdash P$ of affine separation logic (Section 2).
- We show that this construction scales to a rich separation logic for fork-based concurrency by defining **Iron**, a higher-order concurrent separation that allows for precise reasoning about memory resources that are transferable among dynamically allocated threads (Section 3).
- We demonstrate how Iron can be used for verifying challenging examples (Section 4).
- We define a more abstract logic **Iron**$^{++}$, which makes it possible to reason at a higher level of abstraction, similar to Iris, but with fine-grained control over resource usage (Section 5).
- We prove soundness of Iron by means of a model of Iron, defined on top of the Iris base logic. We use this model to prove important adequacy theorems for Iron, which make precise that memory resources are accounted for and not leaked (Section 6).
- We show that the Iron approach can also be used to reason precisely about other user-defined resources, such as locks (Section 8).
- We have formalized all of the theory and examples in the Coq proof assistant. We have used the recent MoSeL framework [Krebbers et al. 2018], which allows us to smoothly mechanize proofs in the combination of both layers of Iron (Section 8). The Coq formalization can be found online at https://iris-project.org/iron/.

$$\textsc{shoare-alloc}\quad \{\mathsf{Emp}\}\ \mathsf{ref}(v)\ \{\ell.\ell \mapsto v\} \qquad \textsc{shoare-free}\quad \{\ell \mapsto -\}\ \mathsf{free}(\ell)\ \{\mathsf{Emp}\} \qquad \textsc{shoare-load}\quad \{\ell \mapsto v\}\ !\,\ell\ \{w.\, w = v \wedge \ell \mapsto v\}$$

$$\textsc{shoare-store}\quad \{\ell \mapsto -\}\ \ell \leftarrow w\ \{\ell \mapsto w\} \qquad \textsc{shoare-seq}\quad \frac{\{P\}\ e_1\ \{Q\} \qquad \{Q\}\ e_2\ \{v.\, R\}}{\{P\}\ e_1; e_2\ \{v.\, R\}} \qquad \textsc{shoare-cons}\quad \frac{P \vdash P' \qquad \{P'\}\ e\ \{v.\, Q'\} \qquad \forall w.\, (Q' \vdash Q)}{\{P\}\ e\ \{w.\, Q\}}$$

Fig. 1. Selected rules of traditional separation logic for a sequential language.

2 THE KEY TECHNICAL CONSTRUCTION—TRACKABLE RESOURCES

Before we describe the rich Iron logic in Section 3, we explain its key notion of *trackable resources*, which allow for precise accounting of resources even if we are allowed to forget resources, either explicitly through weakening, or in the presence of shareable invariant mechanisms as needed for logics for fork-style concurrency. To keep this section self-contained, we will restrict ourselves to memory resources of a simple first-order functional language without concurrency:

$$\begin{aligned} v \in \mathit{Val} &::= () \mid z \mid \mathsf{true} \mid \mathsf{false} \mid \ell && (z \in \mathbb{Z}, \ell \in \mathit{Loc}) \\ e \in \mathit{Exp} &::= v \mid x \mid \mathsf{skip} \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid e_1; e_2 \mid \mathsf{ref}(e) \mid \mathsf{free}(e) \mid\, !\,e \mid e_1 \leftarrow e_2 \mid \ldots \end{aligned}$$

We start with a brief recapitulation of traditional intuitionistic (or *affine*) separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2000] (Section 2.1) and its model (Section 2.2). After that, we explain our new separation logic with trackable resources (Section 2.3) and its model (Section 2.4). Finally, we show an *adequacy* theorem for our separation logic with trackable resources that formally guarantees no memory is leaked (Section 2.5).

2.1 Resources in Traditional Separation Logic

Program specifications in separation logic are Hoare triples $\{P\}\ e\ \{w.\, Q\}$, where e is a program, and P and Q are the pre- and postcondition, respectively. The variable w in the postcondition Q binds the return value of the program. The propositions (typically named P, Q, R) of separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2000, 2002] are formulas of first-order logic extended with separating conjunction $*$ and the Emp connective satisfying the usual commutativity, associativity rules, and the rule stating that Emp is the unit of the separating conjunction. Next to that, separation logic has one more proposition—the *points-to* connective $\ell \mapsto v$, which expresses that the location ℓ in the heap contains the value v. A selection of rules is displayed in Figure 1.

Already in the original separation logic papers, two different models of separation logic were considered: *intuitionistic separation logic* [Reynolds 2000] and *classical separation logic* [Ishtiaq and O'Hearn 2001; Reynolds 2002]. The difference between these two models is that the intuitionistic model enjoys the following "weakening" rule:

$$P * Q \vdash P \qquad (\textsc{affine})$$

Following the nomenclature by Krebbers et al. [2018, 2017b], we call separation logics with the weakening rule *affine*. Affine separation logics are often used to reason about garbage collected languages as the weakening rule allows one to forget about locations that the program no longer cares about. For example, one could prove the following Hoare triple in such logics:

$$\{\ell \mapsto v\}\ \mathsf{skip}\ \{\mathsf{Emp}\} \qquad\qquad \{\mathsf{Emp}\}\ \mathsf{ref}(3); \mathsf{skip}\ \{\mathsf{Emp}\}$$

In the presence of explicit memory deallocation (via the free operation) the weakening rule is usually not desired. We see that skip has the same specification as free, and thus the logic cannot possibly guarantee that resources are freed.

It is important to note that even if a logic does not have the weakening rule, there might be some other mechanism that could be used to leak resources. In the introduction we mentioned one such mechanism, shareable invariants, which appear in slightly different shapes in different logics, but what is common to all of them is that in existing separation logics for fork-style concurrency they all allow us to prove triples as shown above without explicitly using the weakening rule.

2.2 A Model of Traditional Separation Logic

In order to describe the standard model of affine separation logic and to see why it cannot precisely account for resources, we need to fix some notations. We let $\mathcal{H}$ be the set of heap fragments, *i.e.,* partial functions with a finite domain of locations to values. Two such heap fragments can be composed via the partial operation $(\cdot) : \mathcal{H} \times \mathcal{H} \rightharpoonup \mathcal{H}$. The composition of σ_1 and σ_2, denoted $\sigma_1 \cdot \sigma_2$, is only defined when the domains of σ_1 and σ_2 are disjoint, in which case it is defined to be the union of the two heap fragments. This operation has the empty heap fragment $\emptyset$ as the unit. The operation $\cdot$ together with $\emptyset$ makes $\mathcal{H}$ into a partial commutative monoid.

We describe how to model propositions P and program specifications $\{P\}\ e\ \{w.\ Q\}$ in two stages. Propositions P are modeled as upwards closed sets of heap fragments, where upwards closure means that if $\sigma \in [\![P]\!]$, then $\sigma \cdot \sigma_f \in [\![P]\!]$ for any disjoint heap fragment σ_f. The points-to and Emp connectives, and the separating conjunction, are modeled as follows:

$$
\begin{aligned}
[\![\ell \mapsto v]\!] &\triangleq \{\sigma \in \mathcal{H} \mid \mathrm{dom}(\sigma) \supseteq \{\ell\} \land \sigma(\ell) = v\} \\
[\![\mathsf{Emp}]\!] &\triangleq \{\sigma \in \mathcal{H} \mid \mathrm{dom}(\sigma) \supseteq \emptyset\} = \mathcal{H} \\
[\![P * Q]\!] &\triangleq \{\sigma \in \mathcal{H} \mid \exists \sigma_1 \in [\![P]\!], \sigma_2 \in [\![Q]\!] \,.\, \sigma = \sigma_1 \cdot \sigma_2\}
\end{aligned}
$$

It is important to notice that $[\![\ell \mapsto v]\!]$ contains *all* heap fragments h that contain the location ℓ with value v—h might contain other locations. Similarly, $[\![\mathsf{Emp}]\!]$ contains the empty heap fragment, but due to propositions being upwards closed, $[\![\mathsf{Emp}]\!]$ consequently needs to contain all other heap fragments. This argument also shows that there is no proposition in this logic that can express that the heap is empty.

From these definitions we can also see how the upwards closure justifies the weakening rule $P * Q \vdash P$. If $\sigma \in [\![P * Q]\!]$, then $\sigma = \sigma_1 \cdot \sigma_2$ for some $\sigma_1 \in [\![P]\!]$ (and $\sigma_2 \in [\![Q]\!]$). Thus by definition of the upwards closure, $\sigma \in [\![P]\!]$.

In contrast to propositions, Hoare triples are not heap dependent. There are some subtleties in defining their meaning precisely, and the precise definition depends on the precise formulation of the operational semantics. In this section we omit these subtleties since they do not affect the main point (in the model of Iron, as described in Section 6, we fix a concrete operational semantics and define the meaning of Hoare triples precisely). With these caveats, we define the Hoare triple $\{P\}\ e\ \{w.\ Q\}$ to be valid when:

- for any heap fragment $\sigma \in [\![P]\!]$ and any disjoint heap fragment σ_f,
- running the program e in the heap $\sigma \cdot \sigma_f$ is *safe,* and,
- if running e in the heap $\sigma \cdot \sigma_f$ terminates with value w and heap σ_v, then there exists a heap fragment $\sigma' \in [\![Q]\!]$ disjoint from σ_f such that $\sigma_v = \sigma' \cdot \sigma_f$.

We conclude this subsection by showing that the described model of affine separation logic cannot be used to reason precisely about resources. For this, suppose we have proved the triple $\{\mathsf{Emp}\}\ e\ \{\mathsf{Emp}\}$. If we let $\sigma_f = \emptyset$, then the interpretation of Hoare triples implies that if we start in

EMP-SPLIT
$$\mathfrak{e}_{\pi_1} * \mathfrak{e}_{\pi_2} \dashv\vdash \mathfrak{e}_{\pi_1+\pi_2}$$

PT-SPLIT
$$\ell \mapsto_{\pi_1} v * \mathfrak{e}_{\pi_2} \dashv\vdash \ell \mapsto_{\pi_1+\pi_2} v$$

PT-DISJ
$$\ell_1 \mapsto_{\pi_1} - * \ell_2 \mapsto_{\pi_2} - \vdash \ell_1 \neq \ell_2$$

THOARE-ALLOC
$$\{\mathfrak{e}_\pi\}\ \mathsf{ref}(v)\ \{\ell.\, \ell \mapsto_\pi v\}$$

THOARE-FREE
$$\{\ell \mapsto_\pi -\}\ \mathsf{free}(\ell)\ \{\mathfrak{e}_\pi\}$$

THOARE-LOAD
$$\{\ell \mapsto_\pi v\}\ !\,\ell\ \{w.\, w = v \land \ell \mapsto_\pi v\}$$

THOARE-STORE
$$\{\ell \mapsto_\pi -\}\ \ell \leftarrow w\ \{\ell \mapsto_\pi w\}$$

Fig. 2. Selected rules of separation logic with trackable resources for a sequential language.

the empty heap $\emptyset$ and the program terminates with some heap σ_v, then there is a $\sigma' \in [\![\mathsf{Emp}]\!]$ with $\sigma_v = \sigma' \cdot \sigma_f = \sigma'$. However, $[\![\mathsf{Emp}]\!] = \mathcal{H}$, and thus we know nothing about the heap σ_v.

Note that it is impossible to replace Emp by some other postcondition to address this problem. No proposition in this model can give upper bounds on the number of locations, as explained above. Hence in this model it is impossible to guarantee the absence of memory leaks.

2.3 Separation Logic with Trackable Resources

In the previous subsection we saw why traditional affine separation logic cannot guarantee the absence of resource leaks—there is no proposition in the logic that can describe upper bounds on the number of memory locations. We now introduce a new separation logic with *trackable resources* to remedy this shortcoming. This new separation logic still has the weakening rule of affine separation logic, however, contrary to traditional separation logic, the use of weakening will be clearly visible in program specifications.

In order to precisely account for the use of memory resources, it is crucial to be able to express the exact resource footprint of a program. We achieve this by introducing a new kind of points-to connective, *the trackable points-to connective* $\ell \mapsto_\pi v$. Unlike the ordinary points-to connective, the trackable version is equipped with a fraction $\pi \in (0,1]$ to represent the *degree of knowledge* about the heap. The intuitive semantics of this fraction is as follows:

- If we have $\ell_1 \mapsto_{\pi_1} v_1 * \ldots * \ell_n \mapsto_{\pi_n} v_n$ with $\sum \pi_i = 1$, then we not only know the values of these locations, but we also know that the heap contains *exactly* the locations $\ell_1, \ldots, \ell_n$.
- If we have $\ell_1 \mapsto_{\pi_1} v_1 * \ldots * \ell_n \mapsto_{\pi_n} v_n$ with $\sum \pi_i < 1$, then there are complementary parts of the heap not expressed by the given proposition, *i.e.*, we know that the heap contains *at least* the locations $\ell_1, \ldots, \ell_n$.

To ensure that no resources are leaked, we should make sure that the sum of fractions in the pre- and postcondition of each Hoare triple is the same.

The logic with trackable resources needs an additional connective $\mathfrak{e}_\pi$ (with $\pi \in (0,1]$), which can be thought of as "permission to allocate". Its meaning is that the π fraction of the heap is empty. This is best visible in the following specifications for the basic memory operations as displayed in Figure 2. Allocating a reference requires some degree of knowledge about the heap (THOARE-ALLOC), and in turn we trade that knowledge for the points-to connective, with the same fraction π. Freeing a location is exactly dual (THOARE-FREE): we regain the knowledge (with the same fraction π) that the heap is empty. Reading to and writing from a location is exactly as in traditional separation logic, except that there is a fraction π on the points-to connectives (THOARE-LOAD and THOARE-STORE).

There are a number of rules for manipulating $\mathfrak{e}_\pi$ and $\ell \mapsto_\pi v$. First, since both $\mathfrak{e}_\pi$ and $\ell \mapsto_\pi v$ represent some degree of knowledge of the heap, it is always possible to separate both connectives into an empty permission $\mathfrak{e}_{\pi_1}$, along with the original connective at π_2 for any π_1 and π_2 with

$\pi_1 + \pi_2 = \pi$ (rules pt-split and emp-split). These rules are crucial for allocating new locations: the rule for allocation (thoare-alloc) has $\mathfrak{e}_\pi$ as its precondition, which serves as permission to use part of the heap for allocating a new location. So, before allocating a new location, we typically use pt-split or emp-split in right-to-left direction to split off a $\mathfrak{e}_\pi$ permission. Dually, the permission $\mathfrak{e}_\pi$ that is given back after deallocation (thoare-free) can subsequently be merged back into another $\mathfrak{e}_{\pi'}$ or $\ell \mapsto_{\pi'} v$ using pt-split or emp-split in left-to-right direction.

Let us see trackable resources in action on the following program:

$$e \triangleq \mathsf{let}\ \ell_1 = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{let}\ \ell_2 = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{free}(\ell_1); \mathsf{free}(\ell_2).$$

To show that this program has no memory leaks, we shall prove $\{\mathfrak{e}_\pi\}\, e\, \{\mathfrak{e}_\pi\}$ for any π. First, we use emp-split in right-to-left direction to turn the precondition $\mathfrak{e}_\pi$ into $\mathfrak{e}_{\pi/2} * \mathfrak{e}_{\pi/2}$. Then, we use thoare-alloc twice, after which we need to prove $\{\ell_1 \mapsto_{\pi/2} 0 * \ell_2 \mapsto_{\pi/2} 0\}\, \mathsf{free}(\ell_1); \mathsf{free}(\ell_2)\, \{\mathfrak{e}_\pi\}$. This follows from two applications of thoare-free, followed by emp-split in left-to-right direction.

The general pattern that we see here is that if we want to prove that a program does not leak memory, we need to prove a Hoare triple of the form$\{P\}\, e\, \{w.\, Q\}$ where the sum of fractions π in the precondition P and postcondition Q is the same. Note that for composition it is crucial to write specifications that are *parametric* in the fraction π—after all, it is almost impossible to reuse a specification that demands a concrete fraction.

It is important to note that the accounting for resources by means of fractions is precise even if the separation logic enjoys the weakening rule $P * Q \vdash P$. Intuitively, if we would use the weakening rule to forget about a location $\ell \mapsto_{\pi'} v$, we will not be able to reproduce a postcondition with the right sum of fractions.

Finally, to prevent confusion between the well-known fractional permissions [Boyland 2003] and trackable resources, let us point out the key difference between the two. The rule for the store operation (thoare-store) does not require the full fraction. Instead, owning $\ell \mapsto_\pi v$ gives exclusive access to ℓ regardless of the fraction π, so owning both $\ell \mapsto_\pi -$ and $\ell' \mapsto_{\pi'} -$ implies $\ell \neq \ell'$ (rule pt-disj), which is not the case for fractional permissions.

2.4 A Model of Separation Logic with Trackable Resources

We will now construct a model for the separation logic with trackable resources as described in the previous subsection. The construction is very similar to the model of affine separation logic as described in Section 2.2, but we will use a different partial commutative monoid. For this, let $\mathcal{M}$ be the partial commutative monoid with carrier the set $(0, 1] \times \mathcal{H}$ together with an additional element ε, which is defined to be the unit. Furthermore, we define:

$$(\pi_1, \sigma_1) \cdot (\pi_2, \sigma_2) \triangleq (\pi_1 + \pi_2, \sigma_1 \cdot \sigma_2)$$

whenever $\pi_1 + \pi_2 \leq 1$ and $\sigma_1 \cdot \sigma_2$ is defined; otherwise the operation is undefined. We need to add a separate unit ε for modeling Emp. Moreover, note that it is crucial that the fractions π are nonzero (we use the half-closed interval $(0, 1]$ instead of $[0, 1]$), see descriptions of $\mathfrak{e}_\pi$ and $\ell \mapsto_\pi v$ below for a detailed explanation.

Propositions are modeled as upwards closed sets of elements of $\mathcal{M}$. Upwards closure means the same as before: if $m \in [\![P]\!]$ then $m \cdot m_f \in [\![P]\!]$, for any m_f for which $m \cdot m_f$ is defined.

The Emp connective and separating conjunction are modeled as in the model in Section 2.2, replacing the partial commutative monoid of heap fragments with the partial commutative monoid $\mathcal{M}$. The interesting part is how the points-to and the $\mathfrak{e}_\pi$ connectives are modeled:

$$[\![\ell \mapsto_\pi v]\!] \triangleq \uparrow \{(\pi, \sigma) \mid \mathrm{dom}(\sigma) = \{\ell\} \land \sigma(\ell) = v\}$$
$$[\![\mathfrak{e}_\pi]\!] \triangleq \uparrow \{(\pi, \sigma) \mid \mathrm{dom}(\sigma) = \emptyset\}$$

Here, $\uparrow M$ denotes the least upwards closed set containing the set M.

Let us spell out the definitions more concretely. If $\pi = 1$, then both $[\![\ell \mapsto_1 v]\!]$ and $[\![\mathfrak{e}_1]\!]$ *are exactly the singleton sets containing* $(1, \sigma)$ for, respectively, σ the singleton heap fragment, and the empty heap fragment ε. This is because the only element compatible with $(1, \sigma)$ is the unit ε (here it is crucial that the fraction π is nonzero).

In contrast, if $\pi < 1$, then the set $[\![\ell \mapsto_\pi v]\!]$ additionally contains pairs (π', σ) with $\pi' > \pi$ and $\mathrm{dom}(\sigma) \supseteq \{\ell\}$ and $\sigma(\ell) = v$. Analogously, the set $[\![\mathfrak{e}_\pi]\!]$ additionally contains pairs (π', σ) with $\pi' > \pi$ and arbitrary $\sigma \in \mathcal{H}$. Note that it is crucial that all other pairs (π', σ) have the component π' strictly greater than π (here it is again crucial that the fractions are nonzero). This allows us to conclude that if, *e.g.*, $(\pi, \sigma) \in [\![\mathfrak{e}_\pi]\!]$ (note *the same* π), then σ is the empty heap fragment.

Let us now turn to Hoare triples, again with the same caveats about the exact way the operational semantics is phrased as before. The Hoare triple $\{P\}\ e\ \{w.\, Q\}$ is valid when:

- for any $(\pi, \sigma) \in [\![P]\!]$ and any disjoint element $m_f \in \mathcal{M}$ (*i.e.*, for which $(\pi, \sigma) \cdot m_f$ is defined),
- suppose $(\pi, \sigma) \cdot m_f = (\pi_1, \sigma_1)$,
- running the program e in the heap σ_1 is *safe*, and,
- if running e in the heap σ_1 terminates with value v and heap σ_2, then there exists an element $m' \in [\![Q(w)]\!]$ disjoint from m_f such that $(\pi_1, \sigma_2) = m' \cdot m_f$.

2.5 Adequacy of Separation Logic with Trackable Resources

We now show that the model of separation logic with trackable resources as constructed in the previous subsection guarantees the following adequacy theorem:

Theorem 2.1 (Adequacy for correct usage of resources). *Suppose the Hoare triple* $\{\mathfrak{e}_1\}\ e\ \{\mathfrak{e}_1\}$ *is derivable. If running* e *in the empty heap* $\emptyset$ *terminates with the heap* σ *then* $\sigma = \emptyset$.

Proof. Assume that we are given $\{\mathfrak{e}_1\}\ e\ \{\mathfrak{e}_1\}$, and we know that running e in the heap $\emptyset$ terminates with the heap σ. If we let $(1, \emptyset) \in [\![\mathfrak{e}_1]\!]$ and $m_f = \varepsilon$, then by the interpretation of Hoare triples:

(1) there must exist an element $(\pi', \sigma') \in [\![\mathfrak{e}_1]\!]$ such that $(1, \sigma) = (\pi', \sigma') \cdot m_f = (\pi', \sigma')$,
(2) from the definition of $(\pi', \sigma') \in [\![\mathfrak{e}_1]\!]$ we have that $\pi' = 1$ and $\sigma' = \emptyset$,
(3) which implies that $\sigma = \emptyset$, *i.e.*, the program terminated in the empty heap, cleaning up all the memory that has been allocated during its run. □

In order to make use of the adequacy theorem, it is crucial to make sure that the sum of fractions in the pre- and postcondition of each Hoare triple is the same—which only can be done if each location is eventually deallocated. Notably, we cannot prove specifications such as $\{\mathfrak{e}_\pi\}$ ref(3); skip $\{\mathfrak{e}_\pi\}$, since the only way to regain $\mathfrak{e}_\pi$ with the full fraction π is to use free. The best we could prove is $\{\mathfrak{e}_\pi\}$ ref(3); skip $\{\mathfrak{e}_{\pi'}\}$ for some (in fact all) $\pi' < \pi$, but then we could not use the adequacy theorem to conclude anything about the heap the program terminates in.

To summarize, we have shown how to enable precise reasoning using trackable resources even when we are allowed to throw away resources. The key idea is to require positive evidence, instead of relying on the absence of resources as evidence in itself. This is important because even if we did not allow explicit weakening in the logic, there might be some other way in which the resources might be leaked, such as invariants. With the approach outlined here, the user of the logic is able to and required to show that they have not forgotten to account for some resources.

As we already saw in the simple example in Section 2.3, keeping precise track of resources using fractions results in additional bookkeeping. However, as we will illustrate through numerous examples in the full Iron logic (Section 4), this bookkeeping is principled, and can in fact be hidden in most cases using the more abstract Iron^{++} logic (Section 5). In the rest of this paper we will show

$$
\begin{array}{rll}
P, Q, R \in \mathsf{Prop} ::= & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q & \\
& \mid \forall x.\, P \mid \exists x.\, P \mid t = u & \text{(Higher-order logic with equality)} \\
& \mid P * Q \mid P \mathrel{-\!\!*} P \mid \{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} & \text{(The BI connectives and Hoare triples)} \\
& \mid \triangleright P \mid \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} \mid P \Rrightarrow_{\mathcal{E}} Q \mid \dots & \text{(The Iris connectives)} \\
& \mid \mathfrak{e}_{\pi} \mid \ell \mapsto_{\pi} v & \text{(Iron's trackable heap connectives)} \\
& \mid \boxed{\pi.P}^{\mathcal{N},\gamma} \mid \mathsf{OPerm}_{\gamma}\,(p) \mid \mathsf{DPerm}_{\gamma}\,(\pi) & \text{(Iron's trackable invariants)}
\end{array}
$$

Fig. 3. Grammar of the Iron propositions (the novel connectives of Iron are highlighted in blue).

how trackable resources form a very powerful mechanism that enables reasoning about intricate concurrent code with memory transfer between dynamically allocated threads.

3 THE IRON LOGIC

In this section we describe the rich core Iron logic. Contrary to the simple logic with trackable resources that we introduced in Section 2, Iron features support for fork-based concurrency and higher-order functions. To make reasoning about said features possible, Iron not only provides the trackable points-to connectives, but also the full feature set of Iris [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a], along with a new notion of *trackable invariants*.

Similar to Iris, Iron is parameterized by the programming language that one wishes to reason about. For the purpose of this paper we instantiate Iron with $\lambda_{\text{ref,conc}}$—an ML-like language with higher-order store, explicit deallocation, the fork primitive, and compare-and-set (cas), as given below (the language includes the usual operations on pairs and sums, but we have elided them):

$$
\begin{array}{ll}
v \in \mathit{Val} ::= () \mid z \mid \mathsf{true} \mid \mathsf{false} \mid \ell \mid \lambda x.\, e \mid \dots & (z \in \mathbb{Z}, \ell \in \mathit{Loc}) \\
e \in \mathit{Exp} ::= v \mid x \mid e_1(e_2) \mid \mathsf{fork}\ \{e\} \mid \mathsf{ref}(e) \mid \mathsf{free}(e) \mid {!}\, e \mid e_1 \leftarrow e_2 \mid \mathsf{cas}(e, e_1, e_2) \mid \dots &
\end{array}
$$

The language is fairly standard, but has fork {} in contrast to simply having a parallel composition operation. The presence of fork makes reasoning considerably more challenging, since the newly created threads are not scoped, *i.e.,* they can run after the parent thread has terminated.

Figure 3 displays the grammar of Iron propositions, and Figure 4 displays a selection of the rules. (We use $P \dashv\vdash Q$ as notation for bidirectional entailment.) For reasons of space, we do not include a detailed description of the connectives and rules Iron inherits from Iris; we refer to Jung et al. [2018] for an extensive formal description, and Birkedal and Bizjak [2017] for a tutorial-style introduction. There are a couple of things that we need to point out though.

Like Iris, but unlike the simple logic from Section 2, Hoare triples $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$ are annotated with a *mask* $\mathcal{E}$ to keep track of which invariants can be used. We will discuss masks in more detail in Section 3.2, but ignore them for the first part of this section. Moreover, like Iris, many of the Iron rules involve the "later" modality ($\triangleright$). This modality is necessary to prevent logical inconsistencies in the presence of impredicative invariants, but it is orthogonal to the novel features of Iron. Thus, it is safe to ignore the modality on the first reading; more details can be found in the previous Iris literature [Jung et al. 2018].

Throughout this section we will explain Iron's rules for fork-based concurrency (Section 3.1) and trackable invariants (Section 3.2), as well as how Iris's machinery for ghost state is embedded into Iron (Section 3.3). We finally describe Iron's adequacy theorem (Section 3.4) that formally ensures no resources are leaked.

Ordinary separation logic:

HOARE-FRAME
$$\frac{\{P\}\, e\, \{w.\, Q\}_{\mathcal{E}}}{\{P * R\}\, e\, \{w.\, Q * R\}_{\mathcal{E}}}$$

HOARE-VAL
$$\{\mathsf{True}\}\, v\, \{w.\, w = v\}_{\mathcal{E}}$$

HOARE-λ
$$\frac{\{P\}\, e[v/x]\, \{w.\, Q\}_{\mathcal{E}}}{\{\triangleright P\}\, (\lambda x.\, e)\, v\, \{w.\, Q\}_{\mathcal{E}}}$$

HOARE-BIND
$$\frac{\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \qquad \forall v.\, \{Q\}\, K[\,v\,]\, \{w.\, R\}_{\mathcal{E}}}{\{P\}\, K[\,e\,]\, \{w.\, R\}_{\mathcal{E}}}\ K \text{ a call-by-value evaluation context}$$

Heap manipulation:

EMP-SPLIT
$$\mathfrak{e}_{\pi_1} * \mathfrak{e}_{\pi_2} \dashv\vdash \mathfrak{e}_{\pi_1+\pi_2}$$

PT-SPLIT
$$\ell \mapsto_{\pi_1} v * \mathfrak{e}_{\pi_2} \dashv\vdash \ell \mapsto_{\pi_1+\pi_2} v$$

PT-DISJ
$$\ell_1 \mapsto_{\pi_1} - * \ell_2 \mapsto_{\pi_2} - \vdash \ell_1 \neq \ell_2$$

HOARE-ALLOC
$$\{\triangleright \mathfrak{e}_{\pi}\}\, \mathsf{ref}(v)\, \{\ell.\, \ell \mapsto_{\pi} v\}_{\mathcal{E}}$$

HOARE-FREE
$$\{\triangleright \ell \mapsto_{\pi} -\}\, \mathsf{free}(\ell)\, \{\mathfrak{e}_{\pi}\}_{\mathcal{E}}$$

HOARE-LOAD
$$\{\triangleright \ell \mapsto_{\pi} v\}\, !\,\ell\, \{w.\, w = v \wedge \ell \mapsto_{\pi} v\}_{\mathcal{E}}$$

HOARE-STORE
$$\{\triangleright \ell \mapsto_{\pi} -\}\, \ell \leftarrow w\, \{\ell \mapsto_{\pi} w\}_{\mathcal{E}}$$

Fork-based concurrency:

HOARE-FORK-TRUE
$$\frac{\{P\}\, e\, \{\mathsf{True}\}}{\{\triangleright P\}\, \mathsf{fork}\ \{e\}\ \{w.\, w = ()\}_{\mathcal{E}}}$$

HOARE-FORK-EMP
$$\frac{\{P\}\, e\, \{\mathfrak{e}_{\pi}\}}{\{\triangleright P\}\, \mathsf{fork}\ \{e\}\ \{w.\, w = () \wedge \mathfrak{e}_{\pi}\}_{\mathcal{E}}}$$

Shareable (Iris) invariants:

INV-DUP
$$\boxed{I}^{\mathcal{N}} * \boxed{I}^{\mathcal{N}} \dashv\vdash \boxed{I}^{\mathcal{N}}$$

INV-ALLOC
$$\frac{\boxed{I}^{\mathcal{N}} \vdash \{P\}\, e\, \{w.\, Q\}_{\mathcal{E}}}{\{P * \triangleright I\}\, e\, \{w.\, Q\}_{\mathcal{E}}}$$

INV-OPEN
$$\frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathrm{atomic}(e) \qquad \{P * \triangleright I\}\, e\, \{w.\, Q * \triangleright I\}_{\mathcal{E}\setminus\mathcal{N}}}{\boxed{I}^{\mathcal{N}} \vdash \{P\}\, e\, \{w.\, Q\}}$$

Trackable (Iron) invariants:

TINV-SPLIT
$$\mathsf{OPerm}_{\gamma}\,(p_1 + p_2) \dashv\vdash \mathsf{OPerm}_{\gamma}\,(p_1) * \mathsf{OPerm}_{\gamma}\,(p_2)$$

TINV-DUP
$$\boxed{\pi.I}^{\mathcal{N},\gamma} * \boxed{\pi.I}^{\mathcal{N},\gamma} \dashv\vdash \boxed{\pi.I}^{\mathcal{N},\gamma}$$

TINV-ALLOC
$$\frac{\left\{\exists \gamma.\, P * \boxed{\pi.I(\pi)}^{\mathcal{N},\gamma} * \mathsf{OPerm}_{\gamma}\,(1) * \mathsf{DPerm}_{\gamma}\,(\pi_1)\right\}\, e\, \{w.\, Q\}_{\mathcal{E}}}{\{P * (\forall \gamma.\, \triangleright I(\pi_1))\}\, e\, \{w.\, Q\}_{\mathcal{E}}}$$

TINV-OPEN
$$\frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathrm{atomic}(e) \qquad \mathrm{uniform}(I) \qquad \{P * \triangleright I(\pi_1) * \mathsf{OPerm}_{\gamma}\,(p)\}\, e\, \{w.\, \exists \pi_2.\, Q * \triangleright I(\pi_2)\}_{\mathcal{E}\setminus\mathcal{N}}}{\boxed{\pi.I(\pi)}^{\mathcal{N},\gamma} \vdash \{P * \mathfrak{e}_{\pi_1} * \mathsf{OPerm}_{\gamma}\,(p)\}\, e\, \{w.\, \exists \pi_2.\, Q * \mathfrak{e}_{\pi_2}\}}$$

TINV-DEALLOC
$$\frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathrm{atomic}(e) \qquad \{P * \triangleright I(\pi_1) * \mathsf{OPerm}_{\gamma}\,(p)\}\, e\, \{w.\, Q * \triangleright \mathsf{OPerm}_{\gamma}\,(1)\}_{\mathcal{E}\setminus\mathcal{N}}}{\boxed{\pi.I(\pi)}^{\mathcal{N},\gamma} \vdash \{P * \mathsf{OPerm}_{\gamma}\,(p) * \mathsf{DPerm}_{\gamma}\,(\pi_1)\}\, e\, \{w.\, Q\}}$$

Fig. 4. Selected rules of the Iron logic.

3.1 Fork-based Concurrency

In order to support concurrency, $\lambda_{\text{ref,conc}}$ has the expression $\mathsf{fork}\ \{e\}$, which spawns a thread e that is executed in the background. Iron includes two rules for proving Hoare triples involving fork (HOARE-FORK-TRUE and HOARE-FORK-EMP), displayed in Figure 4.

The two rules deal with two different uses for $\mathsf{fork}\ \{e\}$. The rule HOARE-FORK-TRUE, which is like the fork rule of Iris, is sufficient if e either does not make use of memory at all, or if all memory it uses is joined at the end of its execution by means of explicit synchronization (see Section 4.5, where we use this rule to prove the correctness of the parallel composition operator, implemented using a synchronization mechanism). The rule HOARE-FORK-TRUE is insufficient for proving more interesting programs, however. For example, it cannot be used to verify the example from the introduction (Section 1), where there is a cleanup thread, which acts as a "garbage collector" that continually monitors a data structure to see if it is still in use, and otherwise deallocates it. In order to verify such programs, Iron has the additional rule HOARE-FORK-EMP, which allows the forked-off thread to return a permission $\mathfrak{e}_\pi$ in its postcondition, while the main thread continues to have the permission $\mathfrak{e}_\pi$ too. Before taking a look at the actual example in Section 4.3, let us show how this rule can be used to a prove that the program below is free of memory leaks:

$$e \triangleq \mathsf{let}\,\ell_1 = \mathsf{ref}\,(0)\ \mathsf{in}\ \mathsf{let}\,\ell_2 = \mathsf{ref}\,(0)\ \mathsf{in}\ \mathsf{fork}\ \{\mathsf{free}(\ell_1)\}\,; \mathsf{free}(\ell_2).$$

This program is much like the example from Section 2.3, but the location ℓ_1 is now deallocated by a forked-off thread instead of by the main thread. In order to establish $\{\mathfrak{e}_\pi\}\ e\ \{\mathfrak{e}_\pi\}$ (for any π), we first use EMP-SPLIT in right-to-left direction to turn our precondition into $\mathfrak{e}_{\pi/2} * \mathfrak{e}_{\pi/2}$, and use HOARE-ALLOC twice, after which it remains to prove $\{\ell_1 \mapsto_{\pi/2} 0 * \ell_2 \mapsto_{\pi/2} 0\}\ \mathsf{fork}\ \{\mathsf{free}(\ell_1)\}\,; \mathsf{free}(\ell_2)\ \{\mathfrak{e}_\pi\}$. To do so, we use HOARE-FORK-EMP, after which it suffices to prove the following Hoare triples:

$$\{\ell_1 \mapsto_{\pi/2} 0\}\ \mathsf{free}(\ell_1)\ \{\mathfrak{e}_{\pi/2}\} \qquad \{\ell_2 \mapsto_{\pi/2} 0 * \mathfrak{e}_{\pi/2}\}\ \mathsf{free}(\ell_2)\ \{\mathfrak{e}_\pi\}$$

The first of these triples follows from HOARE-FREE and the second follows from a combination of HOARE-FREE and EMP-SPLIT.

Suppose we had instead used HOARE-FORK-TRUE, then we would have had to show:

$$\{\ell_1 \mapsto_{\pi/2} 0\}\ \mathsf{free}(\ell_1)\ \{\mathsf{True}\} \qquad \{\ell_2 \mapsto_{\pi/2} 0\}\ \mathsf{free}(\ell_2)\ \{\mathfrak{e}_\pi\}$$

The second goal is the issue: without the $\mathfrak{e}_{\pi/2}$ from the forked-off thread we cannot reconstruct the full $\mathfrak{e}_\pi$, needed to satisfy the postcondition.

3.2 Shared and Trackable Invariants

Similar to Iris, Iron supports *shared invariants* $\boxed{I}^{\mathcal{N}}$, which can be used to share resources between any number of (forked-off) threads. Shared invariants are *impredicative*, because the resources that are guarded by the invariant are described by an arbitrary Iron proposition I, which may contain Hoare triples, or even nested invariants themselves. In this section, we will briefly recap how shared invariants are used in Iris, followed by a discussion showing that shared invariants do not provide the appropriate level of abstraction to reason about correct disposal of resources. We then introduce Iron's novel solution to this: *trackable invariants*.

Shared (Iris) invariants work as follows: Using the rule inv-alloc, one may transfer any proposition I into an invariant $\boxed{I}^{\mathcal{N}}$.[1] This assertion is *duplicable*[2] (inv-dup), and can thus be freely shared among any number of threads. Converting resources I into an invariant that may be shared among threads comes at a price—using the rule inv-open, one can only get *temporary* access to I for the duration of an instruction e that is *physically atomic* (denoted $\mathsf{atomic}(e)$). The restriction to physically atomic instruction e is essential for soundness: within the verification of e, the invariant I might be temporarily broken, but since the execution of physically atomic instructions is limited to a single step, it is ensured that no other thread can observe that I was broken. In $\lambda_{\text{ref,conc}}$, reading and writing from memory ($!$ and $\leftarrow$), allocation and deallocation (ref and free), and compare-and-set (cas) are all physically atomic operations, and can thus be used by the rule inv-open.

Though powerful, shared (Iris) invariants have two limitations:

(1) once a proposition I is put into an invariant $\boxed{I}^{\mathcal{N}}$ it can never be taken out, and,
(2) once a proposition I is put into an invariant $\boxed{I}^{\mathcal{N}}$ it can never be changed.

The first limitation means that if we simply put $\ell \mapsto_\pi v$ into an invariant, we can never get it back to free the location (using the free instruction). The second limitation means that once we put $\ell \mapsto_\pi v$ into an invariant, we can never change the fraction π. Both of these limitations could be worked around by more explicit accounting of fractions, and by making use of ghost state. However, doing so would lead to excessive bookkeeping of fractions and overly rigid specifications, which is problematic when building hierarchies of specifications, *e.g.*, when specifying a module in terms of other modules. Related to that, as we will show in Section 5.2, shared invariants cannot be integrated into the more abstract logic Iron^{++}, which hides the fractions.

Trackable (Iron) invariants capture a common pattern of use when reasoning about resources, and thereby solve both of these limitations. They also form the key ingredient to enable exact resource accounting in the more abstract logic Iron^{++} in Section 5. Although trackable invariants are in fact encoded in terms of ordinary shared (Iris) invariants, we will not discuss this encoding here, but focus on the connectives and reasoning principles they provide.

The proof pattern supported by trackable invariants is as follows. When we use the rule tinv-alloc to allocate the invariant, we obtain the following three resources:

- The *trackable invariant assertion* $\boxed{\pi . I}^{\mathcal{N},\gamma}$, which expresses the knowledge that the invariant exists.[3] Like shared invariants, the trackable invariant assertion is duplicable (tinv-dup), but unlike shared invariants, it does not provide immediate access to I.
- The *opening token* $\mathsf{OPerm}_\gamma\,(p)$ where $p \in (0, 1]$, which provides the permission to *open* the invariant using the rule tinv-open, *i.e.,* it provides the permission to temporarily access the resources I. The fraction p should not be confused with the fractions π that are used to express the degree of knowledge of the heap. The fraction p is a fractional permission [Boyland 2003]

[1] Every invariant has a namespace $\mathcal{N}$ [Jung et al. 2018], which appears in the invariant assertion $\boxed{I}^{\mathcal{N}}$. Namespaces are needed to avoid *reentrancy*, which in the case of invariants means avoiding "opening" the same invariant twice in a nested fashion. Reentrancy is avoided by annotating Hoare triples $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$ with a mask $\mathcal{E}$, representing the names of invariants which can be used. When the mask $\mathcal{E}$ is omitted it is assumed to be the set of all masks. In practice the use of masks results in some additional bookkeeping orthogonal to our focus, and thus we mostly ignore namespaces in this paper.

[2] Technically, the invariant proposition $\boxed{I}^{\mathcal{N}}$ is *persistent*, which is stronger property than being duplicable, *e.g.*, it allows one to move the proposition in and out of the precondition of a Hoare triple. For this paper, the exact difference between duplicable and persistent does not matter, and we refer to Jung et al. [2018] for further details.

[3] Due to technical reasons related to the encoding in Iris, trackable invariants $\boxed{\pi . I}^{\mathcal{N},\gamma}$ have both a namespace $\mathcal{N}$, which is chosen by the user that allocates the invariant, and an *invariant name* γ, which is dynamically chosen upon allocation of the invariant. The namespace $\mathcal{N}$ is used to prevent reentrancy, exactly the same as for shared invariants. The invariant name γ is used to connect the knowledge of the invariant assertion with the opening and deallocation tokens.

for the specific invariant: $p = 1$ provides unique ownership of the invariant, while $p < 1$ provides shared ownership of the invariant. The fraction can be split using the rule tinv-split.

- The *deallocation token* $\mathsf{DPerm}_{\gamma}\,(\pi)$, which together with the full opening token $\mathsf{OPerm}_{\gamma}\,(1)$ provides the permission to deallocate the invariant using the rule tinv-dealloc, *i.e.,* it allows to permanently take out the resources I of the invariant.

Trackable invariants solve limitation (1); due to the fact that we have the opening and deallocation tokens, we can keep exact track of the number of threads that may access the invariant. Whenever there is just one thread left (*i.e.,* we own $\mathsf{OPerm}_{\gamma}\,(1)$ and $\mathsf{DPerm}_{\gamma}\,(\pi)$), it can be deallocated using the rule tinv-dealloc. Before describing the details of these tokens (in particular, why there is a separate open and deallocation token), let us see how trackable invariants address limitation (2).

To address limitation (2), the proposition I in $\boxed{\pi.I}^{\mathcal{N},\gamma}$ is parameterized by a fraction $\pi \in (0, 1]$. (In the notation, π is used as a binder to provide syntactical convenience.) Parameterizing I by a fraction makes the invariant easier to use since we do not have to reestablish it at the same fraction we opened it. To see how this works, let us take a look at the rule tinv-open for opening trackable invariants. This rule requires $\mathfrak{e}_{\pi_1}$ in order to open the invariant, and in turn, provides the resources I at the fraction π_1 for the duration of the physically atomic instruction. After the verification of the atomic instruction has been concluded, I needs to be reestablished, but this may be done at a different fraction π_2. After closing the invariant, we thus get back $\mathfrak{e}_{\pi_2}$ in return.

For the rule tinv-open to be sound, the proposition I must be *uniform w.r.t. fractions*:

$$\mathsf{uniform}(I) \triangleq \forall \pi_1, \pi_2.\ I(\pi_1 + \pi_2) \dashv\vdash I(\pi_1) * \mathfrak{e}_{\pi_2}.$$

Conceptually, this condition means that the fraction π in I is only used by connectives $\mathfrak{e}_{\pi}$ and $\ell \mapsto_{\pi} v$ appearing in I. A way to think about the use of the permission $\mathfrak{e}_{\pi}$ in the rule tinv-open is that we temporarily trade the resources $\mathfrak{e}_{\pi}$ for the resources $I(\pi)$; uniformity allows exactly this.

When allocating or deallocating a trackable invariant $\boxed{\pi.I}^{\mathcal{N},\gamma}$ (using the rules tinv-alloc and tinv-dealloc, respectively) we also need to take the fraction π into account. To make this possible, the deallocation token $\mathsf{DPerm}_{\gamma}\,(\pi)$ records the fraction π at which the invariant I was initially established. As such, when allocating a trackable invariant, one needs to establish the invariant I at the same fraction π as the one recorded in the deallocation token $\mathsf{DPerm}_{\gamma}\,(\pi)$. Dually, upon deallocation, the invariant I is returned at the fraction π recorded in the deallocation token $\mathsf{DPerm}_{\gamma}\,(\pi)$, making sure no resources have gotten lost in action.

As will be shown in Section 4.2, it is often useful to put some fraction p of the opening token in the invariant resource I. To facilitate this, the rules for trackable invariants feature some interesting bells and whistles. Firstly, since the invariant name γ is dynamically chosen upon allocation (as witnessed by the existential quantifier $\exists\gamma$ in the rule tinv-alloc), the invariant I needs to be initially established for any γ (*i.e.,* one needs to prove $\forall\gamma.\ \triangleright I(\pi_1)$ in the rule tinv-alloc). Secondly, in case a fraction of the token $\mathsf{OPerm}_{\gamma}\,(p)$ resides in the invariant I, it may be the case that the full permission $\mathsf{OPerm}_{\gamma}\,(1)$ is not present up front when deallocating an invariant. As such, the deallocation rule tinv-dealloc allows one to first obtain the contents I of the invariant, and then also use I to account for the full permission $\mathsf{OPerm}_{\gamma}\,(1)$ to justify the deallocation of the invariant.

The fact that we may store the opening token $\mathsf{OPerm}_{\gamma}\,(p)$ in the invariant itself, is also the reason Iron has a separate deallocation token $\mathsf{DPerm}_{\gamma}\,(\pi)$. The token $\mathsf{DPerm}_{\gamma}\,(\pi)$ is not uniform with respect to the fraction π, and thus cannot be put into the invariant.

3.3 Ghost State

Iron inherits Iris's sophisticated mechanism for *ghost state*, which can be used to keep track of additional verification information that is not present in the source code of the program itself. For

Thread-local head reduction:

$$((\lambda x.\, e)v, \sigma) \rightarrow_{\mathsf{h}} (e[v/x], \sigma) \qquad (\mathsf{ref}(v), \sigma) \rightarrow_{\mathsf{h}} (\ell, \sigma[\ell \mapsto v]) \qquad \text{if } \ell \notin \mathrm{dom}(\sigma)$$

$$(\mathsf{fork}\ \{e\}, \sigma) \rightarrow_{\mathsf{h}} ((), \sigma, e) \qquad (\ell \leftarrow v, \sigma[\ell \mapsto w]) \rightarrow_{\mathsf{h}} ((), \sigma[\ell \mapsto v]) \qquad \text{if } \ell \notin \mathrm{dom}(\sigma)$$

Threadpool reduction:

$$\frac{(e_1, \sigma_1) \rightarrow_{\mathsf{h}} (e_2, \sigma_2, \vec{e}_f)}{(K[\, e_1 \,], \sigma_1) \rightarrow_{\mathsf{t}} (K[\, e_2 \,], \sigma_2, \vec{e}_f)} \qquad \frac{(e_1, \sigma_1) \rightarrow_{\mathsf{t}} (e_2, \sigma_2, \vec{e}_f)}{(T; e_1; T', \sigma) \rightarrow_{\mathsf{tp}} (T; e_2; T'; \vec{e}_f, \sigma_2)}$$

Fig. 5. Selected rules of the operational semantics of $\lambda_{\text{ref,conc}}$.

the purpose of this paper, it suffices to know that ghost state can be used to encode transition systems, which can be used to control the transitions made by different threads. Some of these transitions are expressed in terms of a *view shift* $P \Rrightarrow_{\mathcal{E}} P'$, which says that, potentially through a ghost state transition, the resource P can be turned into P'. The generalized rule of consequence below says that we can apply view shifts in the pre- and postconditions of triples:

$$\frac{\overset{\text{HOARE-CONS}}{P \Rrightarrow_{\mathcal{E}} P'} \qquad \{P'\}\, e\, \{w.\, Q'\}_{\mathcal{E}} \qquad \forall w.\, (Q' \Rrightarrow_{\mathcal{E}} Q)}{\{P\}\, e\, \{w.\, Q\}_{\mathcal{E}}}$$

We will see an example of ghost state in Section 4.1.

3.4 Adequacy

To formally establish that Iron ensures that resources are correctly disposed of, we show an adequacy statement with respect to a standard call-by-value operational semantics of $\lambda_{\text{ref,conc}}$.

Operational semantics. The operational semantics of $\lambda_{\text{ref,conc}}$ is given by means of small-step operational semantics; we show selected rules in Figure 5. It is defined in terms of configurations (T, σ), which consist of a *threadpool* T (a list of expressions) and a heap σ (a finite partial function from locations to values). The main part of the semantics is the *threadpool reduction* $(T, \sigma) \rightarrow_{\mathsf{tp}} (T', \sigma')$, and its reflexive-transitive closure $(T, \sigma) \twoheadrightarrow_{\mathsf{tp}} (T', \sigma')$.

Configurations (T, σ_1) are reduced by non-deterministically choosing a thread $e_1 \in T$ from the threadpool T, and letting this thread make a *thread-local reduction* $(e_1, \sigma_1) \rightarrow_{\mathsf{t}} (e_2, \sigma_2, \vec{e}_f)$. Following the conventions in Iris [Jung et al. 2018], the thread-local reduction relation includes a list of newly forked-off threads $\vec{e}_f$. As usual, thread-local reduction is defined in terms of a *thread-local head reduction* relation $(e_1, \sigma_1) \rightarrow_{\mathsf{h}} (e_2, \sigma_2, \vec{e}_f)$, which is lifted by means of standard call-by-value evaluation contexts K to thread-local reductions $(K[\, e_1 \,], \sigma_1) \rightarrow_{\mathsf{t}} (K[\, e_2 \,], \sigma_2, \vec{e}_f)$.

Adequacy. The adequacy theorem is crucial for inferring properties of the operational behavior of programs from their logical specifications because these are often very abstract and involve higher-order quantification, ghost state, invariants, *etc.* While these features are necessary for specifying open programs and modules, in the end, we typically compose individual modules into a closed program and wish to conclude, *e.g.*, that its result is a particular number, or that it does not leak memory when executed. That is what Iron's adequacy theorems allow us to conclude.

Theorem 3.1 (Basic adequacy). *Let ϕ be a first-order predicate over values and suppose the Hoare triple $\{\mathfrak{e}_1\}\, e\, \{w.\, \phi(w)\}$ is derivable in Iron. If we have:*

$$(e, \emptyset) \twoheadrightarrow_{\mathsf{tp}} ((e_1, e_2, \ldots, e_n), \sigma)$$

then the following properties hold:

(1) ***Postcondition validity:*** *If e_1 is a value, then $\phi(e_1)$ holds at the meta-level.*
(2) ***Safety:*** *Each e_i that is not a value can make a thread-local reduction step.*

This theorem provides the normal adequacy guarantees of Iris-like logics: safety, which ensures that threads cannot get stuck, and it ensures that the postcondition holds for the resulting value.

The novel part of Iron is the next adequacy theorem, which guarantees that once all threads have terminated, all resources have been disposed of properly.

THEOREM 3.2 (ADEQUACY FOR CORRECT USAGE OF RESOURCES). *Suppose the Hoare triple $\{\mathfrak{e}_1\}\ e\ \{\mathfrak{e}_1\}$ is derivable in Iron and we have: $(e, \emptyset) \twoheadrightarrow_{\mathbf{tp}} ((e_1, e_2, \ldots, e_n), \sigma)$. If all expressions e_i are* values, *then $\sigma = \emptyset$.*

Note that the adequacy theorem for correct usage of resources requires *all* threads to have terminated, whereas the basic adequacy theorem for postconditions only requires the main thread to have terminated. This is due to our strong fork rule HOARE-FORK-EMP, which allows one to transfer resources $\mathfrak{e}_\pi$ to the forked-off thread. These resources are only ensured to be correctly disposed of once the forked-off thread terminates (*e.g.*, the forked-off thread could just loop, and never dispose of the resources that were transferred to it).

The adequacy theorems presented in this section are special cases of more generic adequacy statements, which allow one to start and end in an arbitrary heap, instead of the empty heap $\emptyset$. The proofs of the adequacy theorems are discussed in Section 6.

4 EXAMPLES

In this section, we will specify and verify a channel module inspired by the one in Singularity OS (Section 4.3), as well as a client of that channel module (Section 4.4). Before doing that, we verify a simpler example—first in the setting of scoped concurrency, using the parallel composition operator (Section 4.1), and then in the setting of unscoped concurrency, using fork (Section 4.2).

Parallel composition. The parallel composition operator is not primitive in our language, but it is definable via fork in the usual way (Section 4.5). Parallel composition can be given the following specification (where P_i and Q_i are arbitrary Iron propositions):

HOARE-PAR
$$\frac{\forall i \in \{1, 2\}.\ \{P_i\}\ e_i\ \{w_i.\ Q_i\}}{\{P_1 * P_2 * \mathfrak{e}_\pi\}\ e_1 \mid\mid e_2\ \{(w_1, w_2).\ Q_1 * Q_2 * \mathfrak{e}_\pi\}}$$

This rule is almost the same as the usual rule for parallel composition of CSL, but for the $\mathfrak{e}_\pi$ in the pre- and postconditions. The reason $\mathfrak{e}_\pi$ is needed is that the implementation of parallel composition uses a location to signal between the forked-off thread, which runs e_2, and the main thread, which runs e_1. (In Section 5.3 we show how to hide this fraction.) We will prove this rule in Section 4.5, since its proof illustrates an important feature of Iron. For now, we will assume this rule to hold.

4.1 Resource Transfer Using Parallel Composition

Consider the following example program:

```
e_par ≜ let ℓ = ref(None) in
        let ℓ′ = ref(0) in  ||  (rec cleanup() = match !ℓ with
        ℓ ← Some ℓ′         ||                     None    ⇒ cleanup()
                            ||                   | Some ℓ′ ⇒ free(ℓ′)
                            ||                   end) ()
        free(ℓ)
```

$$\begin{array}{lr}
\mathsf{True} \Rrightarrow_{\mathcal{E}} \exists \gamma_{\mathsf{sts}}.\, t_1(\gamma_{\mathsf{sts}}) * s_1(\gamma_{\mathsf{sts}}) * t_2(\gamma_{\mathsf{sts}}) & t_1(\gamma_{\mathsf{sts}}) * s_j(\gamma_{\mathsf{sts}}) \vdash j \in \{1\} \\
t_1(\gamma_{\mathsf{sts}}) * s_1(\gamma_{\mathsf{sts}}) \vdash s_2(\gamma_{\mathsf{sts}}) & t_2(\gamma_{\mathsf{sts}}) * s_j(\gamma_{\mathsf{sts}}) \vdash j \in \{1, 2\} \\
t_2(\gamma_{\mathsf{sts}}) * s_2(\gamma_{\mathsf{sts}}) \vdash t_3(\gamma_{\mathsf{sts}}) * s_3(\gamma_{\mathsf{sts}}) & t_3(\gamma_{\mathsf{sts}}) * s_j(\gamma_{\mathsf{sts}}) \vdash j \in \{3\}
\end{array}$$

Fig. 6. The rules for the transition system used in Sections 4.1, 4.2 and 4.5.

The idea is that the location ℓ acts as a channel. The left thread sends a message (the location ℓ'), while the right thread waits until it receives the message (using a busy loop), and then deallocates the location ℓ'. After both threads finish, we dispose of the location ℓ.

To show that this program does not leak memory, we will prove $\{\mathfrak{e}_{\pi}\}\ e_{\mathsf{par}}\ \{\mathfrak{e}_{\pi}\}$. Since the location ℓ is shared among two threads, the proof will become slightly more complicated than the examples we have seen so far—we will need a trackable invariant to account for the sharing that occurs. This invariant contains a disjunction of the possible states in which the program may be in:

$$\begin{array}{llr}
I(\pi) \triangleq & \big(s_1(\gamma_{\mathsf{sts}}) * \ell \mapsto_{\pi} \mathsf{None}\big) \vee & \text{(initial state)} \\
& \big(s_2(\gamma_{\mathsf{sts}}) * \exists \ell'\, \pi_1\, \pi_2.\, (\pi = \pi_1 + \pi_2) * \ell \mapsto_{\pi_1} \mathsf{Some}\,\ell' * \ell' \mapsto_{\pi_2} -\big) \vee & \text{(message sent)} \\
& \big(s_3(\gamma_{\mathsf{sts}}) * \ell \mapsto_{\pi} -\big) & \text{(message received)}
\end{array}$$

In order to keep track of the state of the invariant, we use a transition system consisting of tokens $s_j(\gamma_{\mathsf{sts}})$ and $t_j(\gamma_{\mathsf{sts}})$ for $j \in \{1, 2, 3\}$. The tokens $s_j(\gamma_{\mathsf{sts}})$ appear in the different states of the invariant I, while the tokens $t_j(\gamma_{\mathsf{sts}})$ are carried around through the pre- and postconditions of the Hoare triples so as to ensure that the invariant is in the expected state. The transition system is modeled using ghost state, following the usual approach in Iris, and gives rise to the rules in Figure 6.

Let us sketch the proof of $\{\mathfrak{e}_{\pi}\}\ e_{\mathsf{par}}\ \{\mathfrak{e}_{\pi}\}$. Starting with the precondition $\mathfrak{e}_{\pi}$, we first split it into 4 parts $\mathfrak{e}_{\pi/4}$. We use one part $\mathfrak{e}_{\pi/4}$ to allocate the location $\ell \mapsto_{\pi/4} \mathsf{None}$ (using HOARE-ALLOC), one part for the precondition of HOARE-PAR, and the other parts for both threads. We allocate the tokens $t_1(\gamma_{\mathsf{sts}})$, $s_1(\gamma_{\mathsf{sts}})$ and $t_2(\gamma_{\mathsf{sts}})$ using the first rule in Figure 6, which sets us up with all the resources needed to establish the initial state of the invariant I. So, using TINV-ALLOC, we obtain $\boxed{\pi.I(\pi)}^{\mathcal{N},\gamma}$, thereby giving up $s_1(\gamma_{\mathsf{sts}})$ and $\ell \mapsto_{\pi/4} \mathsf{None}$ (*i.e.*, the left disjunct of $I(\pi/4)$), while getting $\mathsf{OPerm}_{\gamma}\,(1)$ and $\mathsf{DPerm}_{\gamma}\,(\pi/4)$ in return. To proceed, we rearrange the resources as follows:

$$\underbrace{\mathsf{DPerm}_{\gamma}\,(\pi/4)}_{\text{deallocation token}} \;*\; \underbrace{\mathfrak{e}_{\pi/4}}_{\text{for par}} \;*\; \underbrace{t_1(\gamma_{\mathsf{sts}}) * \mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,(1/2)}_{\text{precondition of the left thread}} \;*\; \underbrace{t_2(\gamma_{\mathsf{sts}}) * \mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,(1/2)}_{\text{precondition of the right thread}}$$

This rearrangement allows us to use the rule HOARE-PAR, which in turn, requires us to prove the following Hoare triples for the left and right thread, respectively:

$$\{t_1(\gamma_{\mathsf{sts}}) * \mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,(1/2)\}\ e_{\mathtt{left}}\ \{\mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,(1/2)\}$$
$$\{t_2(\gamma_{\mathsf{sts}}) * \mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,(1/2)\}\ e_{\mathtt{right}}\ \{t_3(\gamma_{\mathsf{sts}}) * \mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,(1/2)\}$$

To verify the left thread, we split $\mathfrak{e}_{\pi/4}$ up into $\mathfrak{e}_{\pi/8}$ and $\mathfrak{e}_{\pi/8}$. We use the first $\mathfrak{e}_{\pi/8}$ to allocate $\ell' \mapsto_{\pi/8} 0$ (using HOARE-ALLOC). We then open the invariant (using TINV-OPEN) using the other permission $\mathfrak{e}_{\pi/8}$. Since we own the token $t_1(\gamma_{\mathsf{sts}})$, we know the invariant is in the initial state. We thus obtain $\ell \mapsto_{\pi/8} \mathsf{None}$, and by using the assignment rule HOARE-STORE, we then obtain $\ell \mapsto_{\pi/8} \mathsf{Some}\,\ell'$. Combining this with $\ell' \mapsto_{\pi/8} 0$, we close the invariant in the second state (after updating the tokens of the transition system using the rule $t_1(\gamma_{\mathsf{sts}}) * s_1(\gamma_{\mathsf{sts}}) \vdash s_2(\gamma_{\mathsf{sts}})$, as shown Figure 6). Subsequently, we close the invariant with fraction $\pi_{\pi/4}$ (this fraction corresponds to the sum of the fractions of the two points-to propositions). This concludes the proof of the left thread.

For the right thread we wait until the invariant is in the second state (technically this is achieved by using Löb induction and opening and closing the invariant using $\mathfrak{e}_{\pi/4}$ for each iteration of the busy loop). Once it is in the second state (recall that we have the token $t_2(\gamma)$, which guarantees it can never be in the third state), we obtain fractions π_{41} and π_{42} with $\ell \mapsto_{\pi_{41}} \mathsf{Some}\,\ell'$, and $\ell' \mapsto_{\pi_{42}} -$, and ${}^{\pi}\!/4 = \pi_{41} + \pi_{42}$. We then update the transition system to $t_3(\gamma_{\mathsf{sts}})$, and close the invariant using $\ell \mapsto_{\pi_{41}} -$, obtaining $\mathfrak{e}_{\pi_{41}}$. After freeing the location ℓ', we conclude the proof of the right thread.

After having verified both threads, we have the following resources left:

$$\underbrace{\mathsf{DPerm}_{\gamma}\,({}^{\pi}\!/4)}_{\text{deallocation token}} \;*\; \underbrace{\mathfrak{e}_{\pi/4}}_{\text{for par}} \;*\; \underbrace{\mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,({}^{1}\!/2)}_{\text{postcondition of the left thread}} \;*\; \underbrace{t_3(\gamma_{\mathsf{sts}}) * \mathfrak{e}_{\pi/4} * \mathsf{OPerm}_{\gamma}\,({}^{1}\!/2)}_{\text{postcondition of the right thread}}$$

We conclude the entire proof by combining the opening token $\mathsf{OPerm}_{\gamma}\,(1)$, together with the deallocation token $\mathsf{DPerm}_{\gamma}\,({}^{\pi}\!/4)$, so we can use the rule TINV-DEALLOC to deallocate the invariant. Using the token $t_3(\gamma_{\mathsf{sts}})$, we know that the invariant is in the third state, giving us $\ell \mapsto_{\pi/4} -$, which allows us to free the location ℓ. We finally compose all the $\mathfrak{e}$ connectives to obtain $\mathfrak{e}_{\pi}$ as needed.

It is worth pointing out that the accounting for fractions followed a consistent pattern. As such, using the more abstract Iron^{++} logic (Section 5), we can hide the fractions completely.

4.2 Resource Transfer Using Fork

We now consider a slight modification of the previous example. This example illustrates the utility of transferring the token $\mathsf{OPerm}_{\gamma}\,(p)$ to open the invariant through the invariant itself:

```
e_fork ≜ let ℓ = ref(None) in
         let rec cleanup() = match !ℓ with
                               None    ⇒ cleanup()
                             | Some ℓ′ ⇒ free(ℓ′); free(ℓ)
                             end in
         fork {cleanup()} ;
         let ℓ′ = ref(0) in ℓ ← Some ℓ′
```

The modification is in the fact that now the main thread is sending the location ℓ' to an independent thread. Thus instead of parallel composition we use fork. This also means that the receiving thread must deallocate the channel once it is done with receiving the message—after all, the main thread does not wait for the receiving thread to terminate.

Note that even though this example is contrived, it reflects a common pattern. Instead of the cleanup function in the forked-off thread, we could imagine a runtime system that would reclaim the memory, and then the specification of a method would indicate that either the method itself dispose of resources, or it has passed them to the runtime system.

In Iron, we again prove $\{\mathfrak{e}_{\pi}\}\; e_{\mathsf{fork}}\; \{\mathfrak{e}_{\pi}\}$ to show that the program does not leak memory resources. The verification of the program is much the same as it was before—we use a trackable invariant, and put the location ℓ into it so it can be shared between both threads. The invariant we use is almost the same as before, but since there is no "join" after the forked-off cleanup thread is finished, the forked-off thread will be in charge of deallocating the invariant. To achieve that, we will transfer the token $\mathsf{OPerm}_{\gamma}\,({}^{1}\!/2)$ from the main thread to the forked-off thread. This is done by slightly augmenting the invariant that we use (the change from the previous one is highlighted in blue):

$$\begin{aligned} I(\pi) \triangleq\ & \big(s_1(\gamma_{\mathsf{sts}}) * \ell \mapsto_\pi \mathsf{None}\big) \vee \\ & \big(s_2(\gamma_{\mathsf{sts}}) * \exists \ell'\, \pi_1\, \pi_2.\, (\pi = \pi_1 + \pi_2) * \ell \mapsto_{\pi_1} \mathsf{Some}\, \ell' * \ell' \mapsto_{\pi_2} - * \mathsf{OPerm}_\gamma\, (1/2)\big) \vee \\ & \big(s_3(\gamma_{\mathsf{sts}}) * \ell \mapsto_\pi -\big) \end{aligned}$$

The proof then proceeds almost the same as before, except that the main thread transfers its $\mathsf{OPerm}_\gamma\,(1/2)$ token into the invariant together with the location ℓ'. Subsequently, the forked-off thread takes out the token $\mathsf{OPerm}_\gamma\,(1/2)$ when the invariant is in the second state, combines it with its own token $\mathsf{OPerm}_\gamma\,(1/2)$, so it can deallocate the invariant.

4.3 The Channel Module

We now present our main example of memory management, a core implementation of a channel module for communication between two threads. This example is inspired by the implementation of inter-process communication in Singularity OS [Fähndrich et al. 2006].

A channel consists of two endpoints which both support three operations: send (send a message), receive (receive a message), and close (close the endpoint). The idea is that each thread gets an endpoint, and a message is sent from one thread to another if the sending thread calls send on its endpoint, and the receiving thread calls receive on its endpoint.

There are several intricacies in the verification of this module:

- A channel is alive as long as either of its endpoints is alive (*i.e.,* not closed). In particular, one can send messages over the channel even if the receiving endpoint is closed. This is to reduce the need for inter-thread signaling.
- To reduce overhead, only primitive types (*e.g.,* integers and pointers) can be send over the channel. However, one can send pointers to compound data structures (*e.g.,* linked lists) over the channel, and thus transfer the ownership of compound data structures too.
- Each channel endpoint has a queue of messages it has received. Adding and removing to this queue uses no locking, or other fine-grained synchronization mechanisms, since it is a single consumer/single producer queue.

There are thus several challenging aspects from the memory management perspective. For example, we can allocate a linked list in one thread and send a pointer to it through the channel. But it may turn out that the other endpoint (owned by some other thread) has already closed at this point, so who should be in charge of disposing of the linked list?

In Singularity OS, the runtime system keeps track of channels, and when both endpoints of a channel are closed, the runtime system disposes of the memory still owned by the message queues, as well as the auxiliary data structures of the channel. Here we model the runtime system by a background thread which is responsible for said disposal. This background thread waits until both endpoints of the channel are closed, at which point it disposes of the memory still in the message queues, the queues themselves, as well as the auxiliary locations used to keep track of the liveness of channels. This is best shown by means of the constructor of the channel module:

```
newchannel(d) ≜ let q_x = qNew () in let q_y = qNew () in
                let x_a = ref (true) in let y_a = ref (true) in
                let rec cleanup() = if ! x_a then cleanup()
                                    else if ! y_a then cleanup()
                                    else qDealloc(d, q_x); qDealloc(d, q_y); free(x_a); free(y_a)
                fork {cleanup()} ; ((q_x, q_y, x_a), (q_y, q_x, y_a))
```

Two messages queues (which have operations qNew for creating a queue, qInsert and qRemove for inserting and removing an element, and qDealloc for deallocation) q_x and q_y are created, as well as two references to Boolean flags x_a and y_a to keep track of whether the channel is still alive (true) or not (false). In addition, newchannel forks off a background thread cleanup, which is responsible for cleaning up any remaining memory left over when both of the endpoints are closed. The method is parameterized by a destructor d, which depends on what data structures are sent through the channel. The destructor d is passed to qDealloc to deallocate all elements in the queue.

The code of the send, receive, and close methods is as follows:

$$
\begin{aligned}
\mathsf{send}(\mathsf{ep}, w) &\triangleq \mathsf{let}\,(q_{\mathsf{recv}}, q_{\mathsf{send}}, x) = \mathsf{ep}\ \ \mathsf{in}\ \mathsf{qInsert}(q_{\mathsf{send}}, w) \\
\mathsf{receive}(\mathsf{ep}) &\triangleq \mathsf{let}\,(q_{\mathsf{recv}}, q_{\mathsf{send}}, x) = \mathsf{ep}\ \ \mathsf{in} \\
&\qquad \mathsf{let\ rec}\ \mathsf{recv}() = \ \begin{array}[t]{l} \mathsf{match}\ \mathsf{qRemove}(q_{\mathsf{recv}})\ \mathsf{with} \\ \quad \mathsf{None} \quad\ \ \Rightarrow \mathsf{recv}() \\ \mid \mathsf{Some}\, w \Rightarrow w \\ \mathsf{end}\ \mathsf{in}\ \mathsf{recv}() \end{array} \\
\mathsf{close}(\mathsf{ep}) &\triangleq \mathsf{let}\,(q_{\mathsf{recv}}, q_{\mathsf{send}}, x) = \mathsf{ep}\ \ \mathsf{in}\ x \leftarrow \mathsf{false}
\end{aligned}
$$

The send method has two arguments. The first is the endpoint, the second is the value to be sent. Sending a message means inserting it into the message queue of the receiving endpoint. The receive method waits until there is a message on the endpoint, *i.e.,* it is blocking. The fact that it is blocking is inessential, however it simplifies its use. The close method simply sets the flag of the endpoint to false (meaning the endpoint is no longer alive).

Specification of the channel module. The specifications of the channel module is parameterized by an Iron predicate $\Phi(w, \pi)$ on values and fractions that describes the invariant that each message that is sent over the channel should satisfy. For instance, $\Phi(w, \pi)$ could be $\exists n.\, w \mapsto_\pi n * \mathsf{even}(n)$, specifying that only even numbers are sent over the channel, but it could also be something more sophisticated like the "list predicate". To specify the methods of the channel module, we use two abstract predicates $\mathsf{Endpoint}_1(\mathsf{ep}, \gamma, \pi)$ and $\mathsf{Endpoint}_2(\mathsf{ep}, \gamma, \pi)$, corresponding to the two endpoints. The specifications we prove are as follows (where $i \in \{1, 2\}$, and d is a destructor function satisfying $\{\Phi(w, \pi)\}\ d(w)\ \{\mathfrak{e}_\pi\}$ for each value w and fraction π):

$$
\{\mathfrak{e}_\pi\}\ \mathsf{newchannel}(d) \left\{ (\mathsf{ep}_1, \mathsf{ep}_2).\ \exists \gamma, \pi_1, \pi_2.\ \begin{array}{l} (\pi = \pi_1 + \pi_2) * \\ \mathsf{Endpoint}_1(\mathsf{ep}_1, \gamma, \pi_1) * \\ \mathsf{Endpoint}_2(\mathsf{ep}_2, \gamma, \pi_2) \end{array} \right\}
$$

$$
\left\{ \mathsf{Endpoint}_i(\mathsf{ep}, \gamma, \pi_1) * \Phi(w, \pi_2) \right\}\ \mathsf{send}(\mathsf{ep}, w) \left\{ \mathsf{Endpoint}_i(\mathsf{ep}, \gamma, \pi_1 + \pi_2) \right\}
$$

$$
\left\{ \mathsf{Endpoint}_i(\mathsf{ep}, \gamma, \pi) \right\}\ \mathsf{receive}(\mathsf{ep}) \left\{ w.\ \exists \pi_1, \pi_2.\ \begin{array}{l} (\pi = \pi_1 + \pi_2) * \\ \mathsf{Endpoint}_i(\mathsf{ep}, \gamma, \pi_1) * \Phi(w, \pi_2) \end{array} \right\}
$$

$$
\left\{ \mathsf{Endpoint}_i(\mathsf{ep}, \gamma, \pi) \right\}\ \mathsf{close}(\mathsf{ep}) \left\{ \mathfrak{e}_\pi \right\}
$$

Both of the channel endpoints have the same specifications: in case we send a message, it needs to satisfy Φ, and when we receive a message, we know it satisfies Φ. We could give a stronger specification to the channel module, *e.g.,* using TaDa style logical atomic triples [da Rocha Pinto et al. 2014] or HOCAP-style specification [Svendsen et al. 2013] that provides tighter guarantees about the messages being sent. However, that is an orthogonal consideration from memory management, so here our protocol is simply that all messages satisfy Φ.

Implementation of the channel module. The channel uses two queues to transfer messages between the endpoints; one queue for each endpoint. Since there is exactly one producer and one consumer, the queue need not use any locking or any other synchronization mechanism, such as compare-and-set (cas), to work correctly. To verify the channel module, it suffices to know that the queue behaves as a bag storing elements satisfying an Iron predicate $\Phi(w, \pi)$. Its specification is as follows (where d is a destructor function satisfying $\{\Phi(w, \pi)\}\ d(w)\ \{\mathfrak{e}_\pi\}$ for each value w and fraction π):[4]

$$\{\mathfrak{e}_\pi\}\ \mathsf{qNew}()\ \left\{q.\ \exists\gamma, \pi_1, \pi_2, \pi_3.\ \begin{array}{l}\pi = (\pi_1 + \pi_2 + \pi_3) * \mathsf{iHandle}(q, \gamma, \pi_1) *\\ \mathsf{rHandle}(q, \gamma, \pi_2) * \mathsf{dHandle}(q, \gamma, \pi_3)\end{array}\right\}$$

$$\{\Phi(w, \pi_1) * \mathsf{iHandle}(q, \gamma, \pi_2)\}\ \mathsf{qInsert}(q, w)\ \{\mathsf{iHandle}(q, \gamma, \pi_1 + \pi_2)\}$$

$$\{\mathsf{rHandle}(q, \gamma, \pi)\}\ \mathsf{qRemove}(q)\ \left\{w.\ \begin{array}{l}(w = \mathsf{None} * \mathsf{rHandle}(q, \gamma, \pi)) \lor\\ \left(\begin{array}{l}\exists w', \pi_1, \pi_2.\ (\pi = \pi_1 + \pi_2) * w = \mathsf{Some}\, w' *\\ \qquad \mathsf{rHandle}(q, \gamma, \pi_1) * \Phi(w', \pi_2)\end{array}\right)\end{array}\right\}$$

$$\left\{\begin{array}{l}\mathsf{iHandle}(q, \gamma, \pi_1) *\\ \mathsf{rHandle}(q, \gamma, \pi_2) *\\ \mathsf{dHandle}(q, \gamma, \pi_3)\end{array}\right\}\ \mathsf{qDealloc}(d, q)\ \{\mathfrak{e}_{\pi_1 + \pi_2 + \pi_3}\}$$

There are three abstract predicates: iHandle (the handle to insert elements), rHandle (the handle to remove elements), and dHandle (the handle to deallocate the queue). None of these predicates are duplicable, hence the queue can be used by at most two threads. However, the insert and remove handles $\mathsf{iHandle}(q, \gamma, \pi)$ and $\mathsf{rHandle}(q, \gamma, \pi)$ are uniform with respect to the fraction π. This means in particular that the handles can be transferred through Iron's trackable invariants—something which is essential to verify the channel module.

Verification of the channel module. Using this queue specification, we can quite easily verify the channel module. The predicate Endpoint is defined roughly as follows:

$$\mathsf{Endpoint}_i((q_1, q_2, x), \gamma, \pi) \triangleq \exists\pi_1, \pi_2.\ (\pi = \pi_1 + \pi_2) * \\ \mathsf{rHandle}(q_1, \gamma, \pi_1) * \mathsf{iHandle}(q_2, \gamma, \pi_2) * \mathsf{Alive}_i * \mathfrak{CI}$$

Each endpoint has two handles, one for sending, and one for receiving messages. An endpoint also contains Alive_i, a piece of ghost state which ensures that the endpoint is open when $\mathsf{Endpoint}_i$ is owned. We use a piece of ghost state rather than explicitly holding $x \mapsto_{\pi'} \mathsf{true}$ because the background thread must also be able to read this reference to see when it is safe to dispose of the endpoint. Finally, the proposition $\mathfrak{CI}$ is the invariant used to communicate between the endpoints and the cleanup thread. It is again a trackable invariant, which encodes a 4-state transition system, whose states correspond to:

(1) both channel endpoints are alive,
(2) the first endpoint is closed, but the second still alive,
(3) the second endpoint is closed, but the first still alive, and,
(4) both of the channel endpoints are closed.

The close methods transition from the first to the second or third state, or from the second or third to the fourth state, depending on which of the two endpoints is closed first. In this case, it transfers the proposition rHandle, iHandle and Alive_i into the invariant $\mathfrak{CI}$. Note, to transfer rHandle and iHandle into the trackable invariant it is crucial that they are uniform w.r.t. fractions. Finally, the background thread gets to run when the invariant is in the fourth state, at which point it retrieves all the handles from the invariant, and disposes of all the memory.

[4] In the accompanying Coq formalization we have implemented and verified such a queue, with a very precise HOCAP-style specification. From that, we have derived the bag-like specification that is given here. The verification of the queue is intricate. However, none of the intricacies are to do with the fraction accounting, but rather with the fact that the implementation is delicate (since it does not use any synchronization primitives). We thus do not include the verification in this paper.

4.4 Client of the Channel Module

To illustrate that the specification of the channel is indeed useful for ensuring safe disposal of memory, we will prove a specification for the following simple module:

$$
\begin{array}{l}
e_{\mathsf{msg}} \triangleq \mathsf{let}\,(\mathsf{ep}_1, \mathsf{ep}_2) = \mathsf{newchannel}(\mathsf{disposeList})\ \ \mathsf{in} \\
\qquad\qquad \left(\begin{array}{l}
\mathsf{let}\,\ell_1 = \mathsf{mkList}\,(5)\ \mathsf{in} \\
\mathsf{let}\,\ell_2 = \mathsf{mkList}\,(10)\ \mathsf{in} \\
\mathsf{send}(\mathsf{ep}_1, \ell_1); \\
\mathsf{send}(\mathsf{ep}_1, \ell_2); \\
\mathsf{close}(\mathsf{ep}_1)
\end{array}
\;\middle|\middle|\;
\begin{array}{l}
\mathsf{let}\,\ell_3 = \mathsf{mkList}\,(5)\ \mathsf{in} \\
\mathsf{send}(\mathsf{ep}_2, \ell_3); \\
\mathsf{let}\,\ell_4 = \mathsf{receive}\,(\mathsf{ep}_2)\ \mathsf{in} \\
\mathsf{disposeList}(\ell_4); \\
\mathsf{close}(\mathsf{ep}_2)
\end{array}\right.
\end{array}
$$

The method $\mathsf{mkList}(n)$ creates a linked list of integers $n, n-1, \ldots, 1$, and $\mathsf{disposeList}$ is the destructor for lists. The left thread creates two linked lists, and sends (*references to*) them over the channel, and finally closes its endpoint ep_1. It does not clean up any memory on its own. The right thread creates another list ℓ_3, and sends it on the other endpoint ep_2. However, since no thread ever receives on ep_1, the ℓ_3 list will never be removed from the channel's message queue. The right thread receives only one (reference to a) list—note that ℓ_4 will be equal to ℓ_1—which it then disposes of. Finally, after both endpoints are closed, the channel's $\mathsf{cleanup}$ thread deallocates the lists ℓ_2 and ℓ_3, which at that point, are still in the message queues of the endpoints ep_2 and ep_1, respectively.

Using the channel module specification, it is straightforward to show that the example program satisfies the specification $\{\mathfrak{e}_\pi\}\ e_{\mathsf{msg}}\ \{\mathfrak{e}_\pi\}$. By adequacy of Iron (Theorem 3.2), we can thus conclude that all the allocated memory has been disposed of when the program terminates.

4.5 The Parallel Composition Operator

We conclude this section by showing how to prove the Hoare triple of the parallel composition operator HOARE-PAR using Iron. This verification cannot be carried out in the same way as the examples we have seen before—namely, using trackable invariants. The reason is essentially that trackable invariants require the proposition stored in the invariant to be uniform w.r.t. fractions. As a consequence, if we were to use them to prove the Hoare triple of the parallel composition operator, we would need to impose that the postconditions of one of the two threads is uniform, while that is not the case for any postcondition. We thus would like our rule for parallel composition to be more flexible and allow arbitrary postconditions for both operands.

Implementation of parallel composition. As usual, parallel composition is implemented via two auxiliary methods spawn and join. The method spawn takes a method and runs it in another thread, but it also allocates a reference cell, which is used to signal that the thread is finished:

$$
\begin{array}{ll}
\mathsf{spawn}(f) \triangleq & \mathsf{let}\,c = \mathsf{ref}\,(\mathsf{None})\ \mathsf{in} \\
& \mathsf{fork}\ \{c \leftarrow \mathsf{Some}(f\,())\}\,; \\
& c
\end{array}
\qquad\qquad
\begin{array}{ll}
\mathsf{join}(c) \triangleq & \mathsf{match}\ !\,c\ \mathsf{with} \\
& \quad \mathsf{Some}\,x \Rightarrow \mathsf{free}(c); x \\
& \mid \mathsf{None} \quad\ \Rightarrow \mathsf{join}(c) \\
& \mathsf{end}
\end{array}
$$

Using spawn and join, we let the parallel composition operator be syntactic sugar:

$$
\begin{array}{rl}
e_1 \,||\, e_2 \triangleq & \mathsf{let}\,h = \mathsf{spawn}\,(\lambda\,\_.\,e_1)\ \mathsf{in} \\
& \mathsf{let}\,v_2 = e_2\ \ \mathsf{in} \\
& \mathsf{let}\,v_1 = \mathsf{join}\,(h)\ \mathsf{in}\ (v_1, v_2)
\end{array}
$$

Verification of parallel composition. The crucial part of verifying the Hoare triple HOARE-PAR is proving the Hoare triples for spawn and join—from these, the correctness of HOARE-PAR follows immediately. We can give the methods spawn and join the following specifications (where P is an arbitrary Iron proposition, and $\Phi : \mathit{Val} \to \mathsf{Prop}$ an arbitrary Iron predicate):

$$\{\mathfrak{e}_\pi * P * \{P\}\, f()\, \{w.\, \Phi(w)\}\}\ \mathsf{spawn}(f)\ \{c.\ \mathsf{joinHandle}(c, \Phi, \pi)\}$$

$$\{\mathsf{joinHandle}(c, \Phi, \pi)\}\ \mathsf{join}(c)\ \{w.\, \mathfrak{e}_\pi * \Phi(w)\}$$

The predicate $\mathsf{joinHandle}(c, \Phi, \pi)$ is defined to be:

$$\exists \gamma_{\mathsf{sts}}.\, t_2(\gamma_{\mathsf{sts}}) * \boxed{(s_1(\gamma_{\mathsf{sts}}) * c \mapsto_\pi \mathsf{None}) \vee (s_2(\gamma_{\mathsf{sts}}) * (\exists w.\, c \mapsto_\pi \mathsf{Some}\, w * \Phi(w))) \vee s_3(\gamma_{\mathsf{sts}})}^{\mathcal{N}}$$

Note that we are using an ordinary *shared* invariant, and are thereby fixing a specific fraction π inside the invariant. As the result of using a shared invariant, P and Φ can be arbitrary propositions. The invariant again makes use of the three state transition system from Figure 6. The forked-off thread (in spawn) transitions from the first to the second state, and the join method transitions from the second to the third state (once the invariant is in the second state). Note that in the third state the invariant no longer owns any memory resources, *i.e.,* it is essentially deallocated.

5 A MORE ABSTRACT VIEW—HOW TO HIDE THE FRACTIONS

Thus far we have reasoned explicitly about the fractions in each proof. However, we took care that the fraction accounting was principled and modular—every example, apart from the derivation of the parallel composition operator specification, treated all fractions parametrically. In this section we show how we can exploit that fact using **Iron**$^{++}$: a more abstract logic build on top of the Iron logic that hides the fractions. As it turns out, all examples but the parallel composition operator can be entirely specified and proven in Iron^{++}.

The key observation that motivates the setup of Iron^{++} is that most propositions (that appeared as pre- and postconditions and invariants) throughout this paper are of the shape:

$$(\pi = \pi_1 + \pi_2 + \ldots + \pi_n) * P_1(\pi_1) * P_2(\pi_2) * \ldots * P_n(\pi_n) \tag{1}$$

That is, the fraction is split among the separating conjuncts and the precise partition chosen is unimportant. In order to hide this splitting of fractions, we will consider predicates over fractions and "lift" the separating conjunction to such predicates P and Q as follows:

$$(P * Q)(\pi) \triangleq \exists \pi_1, \pi_2.\, (\pi = \pi_1 + \pi_2) * P(\pi_1) * Q(\pi_2)$$

With the lifted separation conjunction in hand, we are now able to write propositions like 1 simply as $P_1 * P_2 * \ldots * P_n$. This idea turns out to generalize to all connectives of separation logic, leading to the logic Iron^{++}, whose type of propositions Prop^{++} and its entailment relation ($\vdash$) are defined as (recall that Prop is the type of Iris propositions):

$$\mathsf{Prop}^{++} \triangleq [0,1] \to \mathsf{Prop} \qquad\qquad P \vdash Q \triangleq \forall \pi.\, P(\pi) \vdash Q(\pi)$$

By means of lifting, we can define all the connectives of higher-order logic with equality (True, False, $\Rightarrow$, $\wedge$, $\vee$, $\exists$, $\forall$, $=$), together with the usual BI connectives ($*$, $-\!\!*$, Emp), and the lifted points-to connective ($\mathrel{\widehat{\mapsto}}$). Below we give the definitions of some of these connectives:

$$(P \wedge Q)(\pi) \triangleq P(\pi) \wedge Q(\pi) \qquad\qquad \mathsf{Emp}(\pi) \triangleq \pi = 0$$

$$(P \Rightarrow Q)(\pi) \triangleq P(\pi) \Rightarrow Q(\pi) \qquad\qquad (P -\!\!* Q)(\pi) \triangleq \forall \pi'.\, P(\pi') -\!\!* Q(\pi + \pi')$$

$$(t = s)(\pi) \triangleq (t = s) \qquad\qquad (\ell \mathrel{\widehat{\mapsto}} v)(\pi) \triangleq \pi > 0 \wedge \ell \mapsto_\pi v$$

An important feature of Iron^{++} is that, unlike Iris and Iron, it *does not* satisfy the weakening rule $P * Q \vdash P$. In particular, $\ell \mathrel{\widehat{\mapsto}} v * \ell' \mathrel{\widehat{\mapsto}} v' \vdash \ell \mathrel{\widehat{\mapsto}} v$ is not valid in Iron^{++}, which means that we cannot leak memory resources. We have thus built a linear separation logic over an affine one.

Ordinary separation logic:

LHOARE-FRAME
$$\frac{\{P\}\,e\,\{w.\,Q\}_{\mathcal{E}}}{\{P * R\}\,e\,\{w.\,Q * R\}_{\mathcal{E}}}$$

LHOARE-VAL
$$\{\mathsf{Emp}\}\,v\,\{w.\,w = v \land \mathsf{Emp}\}_{\mathcal{E}}$$

LHOARE-λ
$$\frac{\{P\}\,e[v/x]\,\{w.\,Q\}_{\mathcal{E}}}{\{\triangleright P\}\,(\lambda x.\,e)\,v\,\{w.\,Q\}_{\mathcal{E}}}$$

LHOARE-BIND
$$\frac{\{P\}\,e\,\{v.\,Q\}_{\mathcal{E}} \qquad \forall v.\,\{Q\}\,K[\,v\,]\,\{w.\,R\}_{\mathcal{E}}}{\{P\}\,K[\,e\,]\,\{w.\,R\}_{\mathcal{E}}}\ K \text{ a call-by-value evaluation context}$$

Heap manipulation:

LPT-DISJ
$$\ell_1 \mathrel{\widehat{\mapsto}} - * \ell_2 \mathrel{\widehat{\mapsto}} - \vdash \ell_1 \neq \ell_2$$

LHOARE-ALLOC
$$\{\mathsf{Emp}\}\,\mathsf{ref}(v)\,\{\ell.\,\ell \mathrel{\widehat{\mapsto}} v\}_{\mathcal{E}}$$

LHOARE-FREE
$$\{\triangleright \ell \mathrel{\widehat{\mapsto}} -\}\,\mathsf{free}(\ell)\,\{\mathsf{Emp}\}_{\mathcal{E}}$$

LHOARE-LOAD
$$\{\triangleright \ell \mathrel{\widehat{\mapsto}} v\}\,!\,\ell\,\{w.\,w = v \land \ell \mathrel{\widehat{\mapsto}} v\}_{\mathcal{E}}$$

LHOARE-STORE
$$\{\triangleright \ell \mathrel{\widehat{\mapsto}} -\}\,\ell \leftarrow w\,\{\ell \mathrel{\widehat{\mapsto}} w\}_{\mathcal{E}}$$

Fork-based concurrency:

LHOARE-FORK
$$\frac{\{P\}\,e\,\{\mathsf{Emp}\}}{\{\triangleright P\}\,\mathsf{fork}\ \{e\}\ \{w.\,w = () \land \mathsf{Emp}\}_{\mathcal{E}}}$$

Trackable (Iron) invariants:

LTINV-SPLIT
$$\widehat{\mathsf{OPerm}}_{\gamma}\,(p_1 + p_2) \dashv\vdash \widehat{\mathsf{OPerm}}_{\gamma}\,(p_1) * \widehat{\mathsf{OPerm}}_{\gamma}\,(p_2)$$

LTINV-DUP
$$\boxed{I}^{\mathcal{N},\gamma} * \boxed{I}^{\mathcal{N},\gamma} \dashv\vdash \boxed{I}^{\mathcal{N},\gamma}$$

LTINV-ALLOC
$$\frac{\left\{\exists\gamma.\,P * \boxed{I}^{\mathcal{N},\gamma} * \widehat{\mathsf{OPerm}}_{\gamma}\,(1) * \widehat{\mathsf{DPerm}}_{\gamma}\right\}\,e\,\{w.\,Q\}_{\mathcal{E}}}{\{P * (\forall\gamma.\,\triangleright I)\}\,e\,\{w.\,Q\}_{\mathcal{E}}}$$

LTINV-OPEN
$$\frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathrm{atomic}(e) \qquad \mathrm{uniform}(I) \qquad \left\{P * \triangleright I * \widehat{\mathsf{OPerm}}_{\gamma}\,(p)\right\}\,e\,\{w.\,Q * \triangleright I\}_{\mathcal{E}\setminus\mathcal{N}}}{\boxed{I}^{\mathcal{N},\gamma} \vdash \left\{P * \widehat{\mathsf{OPerm}}_{\gamma}\,(p)\right\}\,e\,\{w.\,Q\}}$$

LTINV-DEALLOC
$$\frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathrm{atomic}(e) \qquad \left\{P * \triangleright I * \widehat{\mathsf{OPerm}}_{\gamma}\,(p)\right\}\,e\,\left\{w.\,Q * \widehat{\mathsf{OPerm}}_{\gamma}\,(1)\right\}_{\mathcal{E}\setminus\mathcal{N}}}{\boxed{I}^{\mathcal{N},\gamma} \vdash \left\{P * \widehat{\mathsf{OPerm}}_{\gamma}\,(p) * \widehat{\mathsf{DPerm}}_{\gamma}\right\}\,e\,\{w.\,Q\}}$$

Fig. 7. Selected rules of the Iron^{++} logic.

The weakening rule $P * Q \vdash P$ does, however, hold for the class of propositions Q that are *affine* [Krebbers et al. 2018]. Intuitively, affine propositions Q are those that do not hold resources that should be accounted for precisely—which formally means $Q \vdash \mathsf{Emp}$. Clearly $\ell \mathrel{\widehat{\mapsto}} v$ is not affine, but throughout this section we will see examples of other connectives that are affine.

In the remainder of this section we will see how Hoare triples are incorporated into Iron^{++} and how that leads to the corresponding adequacy theorems for Iron^{++} (Section 5.1). We then show how trackable invariants are integrated into Iron^{++} (Section 5.2), and finally conclude with some examples (Section 5.3). A selection of Iron^{++} rules is shown in Figure 7.

5.1 Hoare Triples and Adequacy

Some of the Iron rules for Hoare triples contained permissions $\mathfrak{e}_\pi$ in the pre- or postcondition (HOARE-ALLOC and HOARE-FREE). In Iron^{++} we can hide these permissions by essentially threading through a permission $\mathfrak{e}_\pi$ in the pre- and postcondition:[5]

$$(\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}})(\pi) \triangleq (\pi = 0) \land \forall \pi_1 > 0, \pi_2 \geq 0.$$
$$\{\mathfrak{e}_{\pi_1} * P(\pi_2)\}\, e\, \{v.\, \exists \pi_1'\, \pi_2'.\, (\pi_1 + \pi_2 = \pi_1' + \pi_2') * \mathfrak{e}_{\pi_1'} * Q(\pi_2')\}_{\mathcal{E}}$$

To ensure no resources are lost, the fractions in the postcondition should sum up to the same value as those in the precondition. Using this definition, we can prove the usual rules for heap manipulation of classical separation logic: LHOARE-ALLOC, LHOARE-FREE, LHOARE-LOAD, and LHOARE-STORE—instead of pre- and postconditions containing $\mathfrak{e}_\pi$, these now contain Emp (without a fraction). Hoare triples themselves are affine and duplicable, which is crucial for verifying higher-order functions.

An important feature of Iron^{++} is that we inherit Iron's machinery for handling concurrency. In particular, the rule LHOARE-FORK simply follows from HOARE-FORK-EMP. It is also worth noting that this rule mirrors the rule for fork in existing separation logics such as Iris, except that we require the forked-off thread to not leak any resources—*i.e.,* the postcondition is Emp as opposed to True.

Adequacy. We will state versions of Iron's adequacy statements (Theorem 3.1 and 3.2) for Iron^{++}. We take the liberty of stating these theorems in a more generic way, so that the initial and final heap can be arbitrary (instead of the empty heap $\emptyset$).

THEOREM 5.1 (LIFTED BASIC ADEQUACY). *Given a first-order predicate over values ϕ, and suppose the Hoare triple $\{\ast_{(\ell,v)\in\sigma}\, \ell \mathrel{\widehat{\mapsto}} v\}\, e\, \{w.\, \phi(w)\}$ is derivable in Iron^{++}. Now, if we have:*

$$(e, \sigma) \twoheadrightarrow_{\mathbf{tp}} ((e_1, e_2, \ldots, e_n), \sigma')$$

then the following properties hold:

1. ***Postcondition validity:*** *If e_1 is a value, then $\phi(e_1)$ holds at the meta-level.*
2. ***Safety:*** *Each e_i that is not a value can make a thread-local reduction step.*

THEOREM 5.2 (LIFTED ADEQUACY FOR CORRECT USAGE OF RESOURCES). *Suppose the Hoare triple $\{\ast_{(\ell,v)\in\sigma}\, \ell \mathrel{\widehat{\mapsto}} v\}\, e\, \{w.\, \ast_{(\ell,v)\in\sigma'}\, \ell \mathrel{\widehat{\mapsto}} v\}$ is derivable. Now, if we have:*

$$(e, \sigma) \twoheadrightarrow_{\mathbf{tp}} ((e_1, e_2, \ldots, e_n), \sigma''),$$

and all expressions e_i are values, then $\sigma'' = \sigma'$.

5.2 Trackable Invariants

The most difficult concept to incorporate into Iron^{++} is invariants. One may attempt to lift shared (Iris) invariants like we have lifted the other connectives—*i.e.,* to define the invariant assertion $\boxed{I}^{\mathcal{N}}(\pi)$ as $\boxed{I(\pi)}^{\mathcal{N}}$. However, this does not work: this invariant assertion is generally not duplicable—it is only duplicable if I is independent of the fraction (*i.e.,* if I is constant). As such, one cannot put points-to connectives in the invariant, rendering them essentially useless. [6]

[5] We distinguish the Hoare triples of Iron and Iron^{++} by coloring the pre- and postconditions in red and blue, respectively.

[6] We could also define lifted invariants which could only hold affine assertions, with no other restrictions, however, again, this would not be useful since the points-to connective is not affine.

Trackable invariants on the other hand, can be lifted into Iron^{++}, and when lifted, they enjoy good abstract rules. We lift trackable invariants as follows (where $p \in (0, 1]$):

$$\begin{aligned}
\boxed{I}^{\mathcal{N},\gamma}(\pi) &\triangleq \pi = 0 \land \boxed{\pi.\exists \pi_1 \geq 0, \pi_2 > 0.\, (\pi = \pi_1 + \pi_2) * I(\pi_1) * \mathfrak{e}_{\pi_2}}^{\mathcal{N},\gamma} \\
\widehat{\mathsf{OPerm}_\gamma}\,(p)\,(\pi) &\triangleq \pi = 0 \land \mathsf{OPerm}_\gamma\,(p) \\
\widehat{\mathsf{DPerm}_\gamma}(\pi) &\triangleq \pi > 0 \land \mathsf{DPerm}_\gamma\,(\pi)
\end{aligned}$$

The invariant assertion $\boxed{I}^{\mathcal{N},\gamma}$ is affine (due to $\pi = 0$) and duplicable. The opening token $\widehat{\mathsf{OPerm}_\gamma}\,(p)$ is also affine, and can be split through the fraction p. In contrast, the deallocation token $\widehat{\mathsf{DPerm}_\gamma}$ is *not affine*, which is crucial—it ensures that we cannot forget to deallocate an invariant.

Similar to the opening rule of Iron (TINV-OPEN), the opening rule of Iron^{++} (LTINV-OPEN), requires I to be *uniform*. For Iron^{++} uniformity is defined in a similar way:[7]

$$\mathsf{uniform}(I : \mathsf{Prop}^{++}) \triangleq \forall \pi_1, \pi_2 > 0.\ I(\pi_1 + \pi_2) \dashv\vdash I(\pi_1) * \mathfrak{e}_{\pi_2}.$$

The class of uniform propositions enjoys good closure properties: it is closed under disjunction, existentials, separating conjunction, and all affine propositions are uniform. Recall that the invariant deallocation token is not affine, nor is it uniform, and thus cannot be put into the invariant.

5.3 Examples

We can verify all of the examples we have presented in this paper in Iron^{++} (including the channel module from Section 4.3), with the exception of the parallel composition operator (Section 4.5). The proofs follow exactly the same structure as in the plain Iron logic, except that there is no manual fraction accounting—all that is handled abstractly by Iron^{++}.

For example, we can derive the following specifications for the channel module (provided we have $\{\Phi(w)\}\ d(w)\ \{\mathsf{Emp}\}$ for any value w):

$$\begin{aligned}
\{\mathsf{Emp}\}&\ \mathsf{newchannel}(d)\ \{(\mathsf{ep}_1, \mathsf{ep}_2).\ \mathsf{Endpoint}_1(\mathsf{ep}_1, \gamma) * \mathsf{Endpoint}_2(\mathsf{ep}_2, \gamma)\} \\
\{\mathsf{Endpoint}_i(\mathsf{ep}, \gamma) * \Phi(u)\}&\ \mathsf{send}(\mathsf{ep}, u)\ \{\mathsf{Endpoint}_i(\mathsf{ep}, \gamma)\} \\
\{\mathsf{Endpoint}_i(\mathsf{ep}, \gamma)\}&\ \mathsf{receive}(\mathsf{ep})\ \{w.\ \mathsf{Endpoint}_i(\mathsf{ep}, \gamma) * \Phi(w)\} \\
\{\mathsf{Endpoint}_i(\mathsf{ep}, \gamma)\}&\ \mathsf{close}(\mathsf{ep})\ \{\mathsf{Emp}\}
\end{aligned}$$

Note that we have not just hidden the fractions at the end (by lifting the specification from Iron to Iron^{++}). Instead, the *entire proof* of this specification can be carried out in Iron^{++}. In fact, we have done so in Coq all the way—from the (precise HOCAP-style [Svendsen et al. 2013]) specifications and proofs of the messages queues, to the final channel specifications, to the client of the channel module.

This is in contrast to the parallel composition operator, whose Iron^{++} specification is as follows:

$$\frac{\overset{\textsc{lhoare-par}}{}\quad i \in \{1, 2\} \qquad \{P_i\}\ e_i\ \{w_i.\ Q_i\}}{\{P_1 * P_2\}\ e_1 \mid\mid e_2\ \{(w_1, w_2).\ Q_1 * Q_2\}}$$

While we can recover this specification from the one in plain Iron, we cannot prove this specification entirely in Iron^{++}. If we attempt that, the best we can prove entirely in Iron^{++} is a version where Q_2 should be uniform as we are limited to trackable invariants. This restriction is undesirable in general, and prevents *e.g.*, allocating an invariant in e_2 (since $\widehat{\mathsf{DPerm}_\gamma}$ is not uniform).

[7] Note that even though I ranges over non-negative fractions, we require uniformity only for strictly positive fractions. If we would require the property to hold for all fractions, the condition would force I to be essentially $\mathfrak{e}$.

6 SEMANTICS OF IRON

The semantics of Iron builds upon the semantics of Iris [Jung et al. 2016, 2018; Krebbers et al. 2017a], which is staged in two parts:

- The **Iris base logic**, which comprises only the assertion layer of vanilla separation logic, plus a handful of simple modalities for dealing with (higher-order) ghost state and step-indexing. The semantics of the Iris base logic is given by solving a recursive domain equation.
- The **Iris program logic**, which provides machinery for invariants and program specifications on top of the Iris base logic. The semantics of the Iris program logic is given by defining its connectives in terms of the Iris base logic.

For the semantics of Iron, we use the Iris base logic in its original form, and only modify the program logic in the following ways:

(1) We change the *state interpretation* of Hoare triples (which are defined in terms of weakest preconditions) to incorporate trackable points-to connectives $\ell \mapsto_\pi v$.
(2) We change the definition of Hoare triples to account for the resources of forked-off threads.
(3) We define trackable invariants as a layer on top of ordinary Iris invariants.

In this section we will focus on the first two points. For reasons of space, we cannot recall the Iris semantics here, so we presume that the reader is familiar with the semantics of Iris.

Definition of weakest precondition. In Iron we use the following definition of weakest preconditions, with changes from the Iris definition highlighted in blue. We explain the changes below.

$$\begin{aligned}
\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq{}& (e \in \mathit{Val} \land {\models\!\!\!\Rrightarrow}_{\mathcal{E}} \Phi(e)) \\
&\lor\Big(e \notin \mathit{Val} \land \forall \sigma_1, \pi_1.\, S(\sigma_1, \pi_1) \mathrel{-\!\!*} {}^{\mathcal{E}}{\models\!\!\!\Rrightarrow}^{\emptyset}\ (\mathrm{red}(e, \sigma_1) \\
&\quad\land \triangleright \forall e_2, \sigma_2, \vec{e}_f.\, ((e, \sigma_1) \to_{\mathsf{t}} (e_2, \sigma_2, \vec{e}_f)) \mathrel{-\!\!*} {}^{\emptyset}{\models\!\!\!\Rrightarrow}^{\mathcal{E}} \\
&\qquad \exists \vec{\pi}_f.\, \big(S(\sigma_2, \pi_1 + \textstyle\sum \vec{\pi}_f) * \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\} * \mathop{\ast\!\!\!\ast}_{(e',\pi') \in \overrightarrow{e_f, \pi_f}} \mathsf{wp}_{\top}\, e'\, \{\_.\mathfrak{e}_{\pi'}\}\big)\big)\Big)
\end{aligned}$$

Here, γ_h is a fixed ghost name, σ's range over the physical state, and S is the state interpretation. The state interpretation $S(\sigma, \pi)$ in Iron has an additional argument π which is the amount of knowledge the forked-off threads have about the heap. Concretely, we use the resource algebra $\textsc{Auth}\big((0, \infty) \times (\mathit{Loc} \xrightarrow{\text{fin}} \mathrm{Ex}(\mathit{Val})) + \{\varepsilon\}\big)$ for the state interpretation, and define:

$$S(\sigma, \pi) \triangleq \boxed{\bullet\, (1 + \pi, \sigma)}^{\gamma_h} \qquad \ell \mapsto_\pi v \triangleq \boxed{\circ\, (\pi, \{\ell \mapsto v\})}^{\gamma_h} \qquad \mathfrak{e}_\pi \triangleq \boxed{\circ\, (\pi, \emptyset)}^{\gamma_h}$$

This resource algebra is very close to the partial commutative monoid we used in Section 2.4, except that we allow arbitrary positive fractions π (instead of $\pi \le 1$). The reason for this is that we need to accommodate the postconditions of the forked-off threads. The invariant maintained by the definition of weakest precondition is that the total fraction is always $1 + \sum_i \pi_i$ where i ranges over the forked-off threads, and $\mathfrak{e}_{\pi_i}$ is the postcondition[8] of the i-th thread.[9] Using this invariant we can conclude, once all the threads have terminated, and we have $\mathfrak{e}_1$ in the postcondition of the main thread, that the heap is empty. This is because $S(\sigma, 1 + \pi) * \mathfrak{e}_{1+\pi}$ implies $\sigma = \emptyset$. A precise adequacy statement is the following theorem.

[8] Here we allow $\pi = 0$ and define $\mathfrak{e}_0$ to be True in order to be able to have True as a postcondition of the new thread.

[9] In the Coq formalization, the treatment of postconditions of forked-off threads is more general. Instead of parameterizing the state interpretation S by a fraction, we parameterize it by the number of forked-off threads, and have a fixed postcondition for all forked-off threads. The version in the paper can be derived via a step of indirection using Iris's ghost state mechanism. We ignore this extra generality in the paper as it is unnecessary for our purposes.

Theorem 6.1. *Let ϕ be a first-order predicate over the initial state, final state, and value, and suppose the following Hoare triple is derivable:*

$$\{\mathfrak{e}_\pi\}\ e\ \{v.\ Q\} \qquad\qquad \forall v, \sigma', \vec{\pi}_f.\ Q * S(\sigma', \textstyle\sum \vec{\pi}_f) * \left(\mathop{\Huge *}_{\pi' \in \vec{\pi}_f} \mathfrak{e}_{\pi'}\right) \vdash \phi(\sigma, \sigma', v)$$

Now, if we have $(e, \sigma) \twoheadrightarrow_{\mathbf{tp}} ((e_1, e_2, \ldots, e_n), \sigma')$, and all expressions e_i are values, then $\phi(\sigma, \sigma', e_1)$ holds at the meta-level.

Note that like Iron's adequacy statement (Theorem 3.2) we require *all* the threads to terminate before we can apply this theorem. This condition is crucial—intuitively, the main thread may have disposed of all of its memory, so that $\mathfrak{e}_1$ holds, but the postconditions of the forked-off threads are needed to ensure that no memory has leaked anywhere else, and indeed the forked-off thread might not yet have disposed of the memory when the main thread terminates.

In order to show that the actual adequacy result (Theorem 3.2) follows from Theorem 6.1, we let $\pi \triangleq 1$ and $Q \triangleq \mathfrak{e}_1$ and $\phi(v, \sigma, \sigma') \triangleq (\sigma = \sigma')$.

7 RELATED WORK

We already discussed some related work in the introduction (Section 1); in this section we consider some further related work.

Already in early work on separation logic for sequential languages [Ishtiaq and O'Hearn 2001; Reynolds 2000, 2002], different models were considered; see [Cao et al. 2017; Krebbers et al. 2018] for a recent account. In particular, models aimed at reasoning about explicit memory deallocation, often referred to as "classical" separation logic, and models aimed at reasoning about garbage collected languages without explicit memory deallocation, often referred to as "intuitionistic" separation logic. The classical models were defined using the full powerset of the set of heaps, whereas the intuitionistic models used upwards closed subsets of heaps (with respect to the extension order of heaps). In our terminology, the "intuitionistic" logic is affine and the "classical" is linear.

Ishtiaq and O'Hearn [2001] pointed out that one could recover the "intuitionistic" (affine) logic from the "classical" (linear). Here instead, as mentioned in Section 5, we recover a linear logic from an affine one; for a much richer higher-order logic, for a concurrent language, with invariants *etc.*

There has previously been a variant of Iris with a notion of linearity [Tassarotti et al. 2017]. It was used to reason about concurrent termination-preserving refinement. However, the treatment of linearity by Tassarotti et al. [2017] is limited: in contrast to Iron only affine propositions can be framed and it is not possible to share linear propositions through invariants, which means that the logic, in the words of the authors, is "unsuitable for tracking resources like the heap". As part of future work, we would like to investigate whether the approach we have used in Iron can also be used for proving termination-preserving refinements.

Krebbers et al. [2018] described a variant of the Iris model that can handle a mixture of linear and affine resources, by equipping Iris's algebraic structure (a *resource algebra*) for modeling resources with a pre-order. While this model encompasses the model of Tassarotti et al. [2017], it is not clear how its generality can be exploited to prove program specifications. The authors only defined the basic BI connectives, but did not fully generalize the other Iris connectives (like the update modalities, invariant machinery, and weakest preconditions).

Gotsman et al. [2007] described a variant of concurrent separation logic where resources can be transferred between dynamically allocated threads. To prevent leaking of resources their resource invariants need to satisfy a *global* admissibility condition. This condition needs to be checked for a set of resource invariants simultaneously. In contrast our reasoning is local, and the fraction accounting in Iron ensures that any resource leaks are visible in the specifications.

8 DISCUSSION AND FUTURE WORK

Trackable invariants. Recall that Iron's trackable invariants require that the proposition that is put into the invariant is uniform with respect to fractions. As we have demonstrated by the examples in the paper, the uniformity requirement does not seem like a very restrictive condition, as we have been able to reason about many challenging examples involving transfer of resources between dynamically allocated threads.

The uniformity requirement does, however, mean that there are some examples which are out of reach of Iron^{++}. We already mentioned that the parallel composition operator cannot be verified completely in Iron^{++}, but requires one to "drop down" to the lower-level Iron logic. For this particular example, we believe that one could define a bespoke sharing mechanism in Iron and then use that carry out the main proof in Iron^{++}. Other examples which are out of reach of the Iron^{++} are those where it is not statically known who disposes of the shared resources. An example of such a program is the channel module implementation we have considered, but instead of having a background thread in charge of cleaning up the resources, the resources would have to be disposed by the last user to close the endpoint. We have a generalization of trackable invariants which would be sufficient to verify such an implementation of the channel module also in Iron^{++}, however this generalization is tailored rather specifically to the kind of invariant we need in this example. Designing a more principled sharing mechanism more general than the trackable invariants we have considered in this paper is left as future work. For now such examples can still be done by dropping down to Iron, which also shows the power of having this logic available, even if most examples can already be done entirely in Iron^{++}.

Managing other resources. We believe that the methodology of trackable resources is also applicable to precise reasoning about other kinds of resources. The key idea is simple enough—to combine the real resource we care about, *e.g.,* the heap, together with a fraction and make it so that splitting of the resource is tied to the splitting of the fraction, such as in the rule PT-SPLIT. The fraction can be used to avoid silently leaking resources in the proof through, for instance, the weakening rule. If we do, we *lose* a degree of knowledge about the resource, and with this property we *gain* the ability to say exactly what the resources are, which is a property typically only expressible in a linear separation logic. In retrospect, this is not surprising since the use of fractions in Iron is intimately related to linearity, as seen in the definition of the logic Iron^{++}.

For example, if we extended the language with file handles, or input/output channels we could follow an approach analogous to how we manage the heap. Each different kind of resource would be paired with a fraction, as is the points-to connective.

Interestingly, we can also use the same approach for keeping track of resources which are not primitive to the language. As an example let us look at how we can ensure that locks (which are not primitive in $\lambda_{\text{ref,conc}}$) are *used* correctly. That is, if a program acquires a lock it should release it before it terminates. The simplest lock implementation is the spin lock, which consists of the following four methods:

$$\begin{aligned}
\mathsf{newLock}() &\triangleq \mathsf{ref}(\mathsf{false}) \\
\mathsf{dispose}(\ell) &\triangleq \mathsf{free}(\ell) \\
\mathsf{acquire}(\ell) &\triangleq \mathsf{if}\ \mathsf{cas}(\ell, \mathsf{false}, \mathsf{true})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{acquire}(\ell) \\
\mathsf{release}(\ell) &\triangleq \ell \leftarrow \mathsf{false}
\end{aligned}$$

The lock itself is a Boolean flag, which is false when the lock is released, and true when it is acquired. Since multiple threads can race to acquire the lock, the acquire method is implemented via a cas loop so that checking the flag and (potentially) setting it to true is done in a single atomic step.

The release method is only called by the method which has acquired the lock, and thus can simply set the flag without any additional synchronization mechanisms. The dispose method should only be called when there is a single thread using the lock and the lock is released, and thus it simply disposes of the flag, reclaiming the memory.

In Iron^{++}, we can define two abstract predicates, $\mathsf{isLock}(w, \gamma, p)$ and $\mathsf{locked}(w, \gamma, p)$, parameterized over a proposition I, and give the following specifications to the methods of the lock:

$$\begin{aligned}
\{I\}\ &\mathsf{newLock}()\ \{w.\ \exists\gamma.\ \mathsf{isLock}(w, \gamma, 1)\}\\
\{\mathsf{isLock}(w, \gamma, p)\}\ &\mathsf{acquire}(w)\ \{I * \mathsf{locked}(w, \gamma, p)\}\\
\{I * \mathsf{locked}(w, \gamma, p)\}\ &\mathsf{release}(w)\ \{\mathsf{isLock}(w, \gamma, p)\}\\
\{\mathsf{isLock}(w, \gamma, 1)\}\ &\mathsf{dispose}(w)\ \{I\}
\end{aligned}$$

Here, $p \in (0, 1]$ is a fraction. The analogy we wish to make is that $\mathsf{isLock}(w, \gamma, p)$ is akin to the $\mathfrak{e}_\pi$ proposition of Iron. We can split it, *i.e.,* $\mathsf{isLock}(w, \gamma, p_1 + p_2) \dashv\vdash \mathsf{isLock}(w, \gamma, p_1) * \mathsf{isLock}(v, \gamma, p_2)$, and the operations acquire and release behave analogously to ref and free.

The fact that we can define the above specification for locks is not specific to Iron; it can also be done in vanilla Iris. What is novel however, is that with these abstract predicates we can prove the following (simplified here, for clarity) analogue of *adequacy for usage of locks.* If the following specification of an arbitrary expression e is provable:

$$\{\mathsf{isLock}(w, \gamma, 1)\}\ e\ \{\mathsf{isLock}(w, \gamma, 1)\},$$

then if the program e terminates, every acquire has an accompanying release. A formal statement and proof of this requires a slight extension of the adequacy theorem to take into account the fact that invariants hold at the end of the execution of the program. Further details are beyond the scope of this paper but are worked out in the accompanying Coq formalization.

The key idea in related examples is to tie a concrete operation on a shared data structure (necessarily guarded by an invariant) with transferring the permission to use the invariant into the invariant itself, connected to some particular physical state. This way the specification of a method using the data structure will expose whether certain methods are called or not.

Coq formalization. We have formalized Iron and Iron^{++} using the recent MoSeL framework [Krebbers et al. 2018]—which provides an extensive set of tactics for making separation logics proofs look like Coq proofs. To do so, we have instantiated MoSeL's modal BI (MoBI) interface with both the propositions of Iron (which comes in the form of a fork of Iris, with the changes described in Section 6) and those of Iron^{++} (whose formalization in Coq is based on fractional predicates over arbitrary BI logics). Having instantiated the MoSeL interfaces, and having taught MoSeL about some of the Iron specific features by instantiating corresponding type classes, we were then able to reason both in Iron^{++}, and to "drop down" to Iron when needed. In fact, all proofs in Coq were directly carried out in Iron^{++}, with the exception the parallel composition operator, for whose proof we dropped down to Iron.

ACKNOWLEDGMENTS

We thank Thomas Dinsdale-Young and Morten Krogh-Jespersen for invaluable discussions, and the reviewers for suggestions on how to make the paper more accessible.

This research was supported by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

REFERENCES

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf.

John Boyland. 2003. Checking interference with fractional permissions. In *SAS*.

Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. 2017. Bringing order to the separation logic jungle. In *APLAS*.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *ECOOP*.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP*.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. 2006. Language support for fast and reliable message-based communication in singularity os. In *EuroSys*.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*.

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*.

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local reasoning about storable locks and threads. In *APLAS*.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle semantics for concurrent separation logic. In *ESOP*.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an assertion language for mutable data structures. *ACM SIGPLAN Notices* (2001).

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* (2018). To Appear.

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*.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. ICFP (2018).

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *ESOP*.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*.

William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A verified messaging system. In *OOPSLA*.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*.

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *TCS* (2007).

John C. Reynolds. 2000. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*. 303–321.

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP*.

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular reasoning about separation of concurrent data structures. In *ESOP*. 169–188.

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A higher-order logic for concurrent termination-preserving refinement. In *ESOP*.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*.

Viktor Vafeiadis and Matthew Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *CONCUR*.