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 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.

1 INTRODUCTION

Precise management of resources and the obligations they impose, such 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. In order to make reasoning about such resources possible, a plethora of variants of concurrent separation logic (CSL) have been proposed [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; Nanevski et al. 2014; O'Hearn 2007; Vafeiadis and Parkinson 2007], with increasingly sophisticated mechanisms for sharing resources. In particular, variants of *higher-order* concurrent separation logic [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a; Mansky et al. 2017; Svendsen and Birkedal 2014; Turon et al. 2013] have emerged which can be used to give general and modular specifications to libraries and programs.

A recent effort in this direction is Iris [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a]—a framework for concurrent separation logic with a minimal base logic, which is powerful enough to realize (in the logic) most of the reasoning principles found in other concurrent separation logics. Iris has been used to define logics for atomicity refinement of fine-grained concurrent data structures [Jung et al. 2015], Kripke logical-relations models for relational reasoning in ML-like languages [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017; Timany et al. 2018], program logics for relaxed memory models [Kaiser et al. 2017] and object capabilities [Swasey et al. 2017], and to give a safety proof for a realistic subset of the Rust programming language [Jung et al. 2018a].

Despite Iris's broad applicability for proving functional correctness, Iris has not been used to prove modularly that resources are *necessarily* used. Indeed, the prevalent understanding has been that one cannot use Iris to express, for example, that a program module is obligated to free all the memory it has allocated, or that it has released all the locks it has acquired. This understanding derives from the fact that Iris is an *affine* separation logic, which means that for every proposition P and Q, the weakening rule $P * Q \vdash P$ holds, and thus resources can be forgotten. As a result, one can silently throw away resources, such as the "points-to" connective $\ell \hookrightarrow v$, which expresses

Authors' addresses: Aleš Bizjak, Computer Science, Aarhus University, Aabogade 34, Aarhus N, 8200, Denmark, abizjak@cs.au.dk; Daniel Gratzer, Computer Science, Aarhus University, Aabogade 34, Aarhus N, 8200, Denmark, gratzer@cs.au.dk; Robbert Krebbers, Software Technology, Delft University of Technology, Mekelweg 4, Delft, 2628 CD, The Netherlands, mail@robbertkrebbers.nl; Lars Birkedal, Computer Science, Aarhus University, Aabogade 34, Aarhus N, 8200, Denmark, birkedal@cs.au.dk.

ownership of a location ℓ. There are several reasons why Iris is affine, but a key point is that Iris supports unstructured *fork*-style concurrency. Indeed, to that end, Iris includes invariants that can be freely shared among threads—this is in contrast to *scoped* CSL-style invariants [O'Hearn 2007], which suffice for structured concurrency, but not for unstructured *fork*-style concurrency. Iris's shareable invariants are not explicitly tracked in the logic (*i.e.,* they are treated intuitionistically) and thus resources can easily be leaked through invariants.

In this paper we present a new higher-order concurrent separation logic called **Iron**, which supports precise resource reasoning for *fork*-style concurrency. This is very important in practice, where an often used programming pattern is to transfer ownership of resources between dynamically allocated threads. For instance, Singularity OS [Fähndrich et al. 2006] used this pattern in its inter-thread communication mechanism. In Section 3 we show how to use Iron to specify and verify a non-trivial implementation of a channel module, inspired by Singularity OS. Moreover, we also show that a client can use the channel module to delegate memory reclamation to a forked-off thread and be provably certain that all memory is correctly disposed of. For concreteness, here is a much simpler example illustrating the transferal of resources from one thread to another, which then deallocates the transferred resources:

$$
\begin{array}{l}
\mathsf{let}\ \ell = \mathsf{ref}\,(\mathsf{None})\ \mathsf{in} \\
\mathsf{let\ rec}\ \mathsf{cleanup}() = \mathsf{match}\ !\,\ell\ \mathsf{with} \\
\qquad\qquad\qquad\qquad\quad \mathsf{None} \quad \Rightarrow \mathsf{cleanup}() \\
\qquad\qquad\qquad\qquad \mid \mathsf{Some}\,\ell' \Rightarrow \mathsf{free}(\ell');\mathsf{free}(\ell) \\
\qquad\qquad\qquad\qquad \mathsf{end\ in} \\
\mathsf{fork}\ \{\mathsf{cleanup}()\}\,; \\
\mathsf{let}\ \ell' = \mathsf{ref}\,(0)\ \mathsf{in}\ \ell \leftarrow \mathsf{Some}\,\ell'
\end{array}
$$

The idea 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 ℓ (the channel). After forking off the cleanup thread, the main thread allocates a resource, the location ℓ', which is sends to the forked-off thread (so that the forked-off thread may deallocate it). Using Iron we can prove that this program satisfies a Hoare triple, which provides an ironclad guarantee (via adequacy theorems for the Iron logic, see Sections 2.6) that when the two threads terminate, then all the allocated memory has indeed been disposed.

We model Iron on top of the *Iris base logic* [Krebbers et al. 2017a], which we do not change (in particular, the base logic is still affine). This simplifies the model construction significantly (*e.g.,* we do no have to solve recursive domain equations, that is done in the model of the Iris base logic [Jung et al. 2018b; Krebbers et al. 2017a]). It also allows us to reuse the Coq formalization of the Iris base logic [Krebbers et al. 2018, 2017b]. Thus we show that Iris's base logic can in fact be used to develop a program logic, which supports modular reasoning about obligations to use resources.

Key Ideas. For reasoning about (non-disjoint) shared-memory concurrency, most logics employ some notion of invariant. As mentioned above, 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 where invariants are used in the logic. To reason about *fork*-style concurrency, Iris (like iCap [Svendsen and Birkedal 2014] and TaDa [da Rocha Pinto et al. 2014]) uses *shareable* invariants, which are not tracked in the logic. Technically, shareable means that Iris invariants are *persistent*, hence *duplicable*, and hence they behave as intuitionistic resources, which need not be explicitly tracked in pre- and postconditions of Hoare triples. Scoped invariants allow to control resources precisely, whereas with the more flexible

shareable Iris invariants we lose precise control over resources, since Iris invariants are treated intuitionistically in the logic. One of the key ideas of Iron is to use an intermediate form of invariants that we call *trackable invariants*, which are tracked in the logic (appear in pre- and postconditions), but are not scoped. Our trackable invariants do support a controlled form of sharing—technically, we have a so-called splitting rule, which allows for *controlled duplication* of trackable invariants. (Trackable invariants are not persistent and hence not freely duplicable.)

We define trackable invariants, tracked points-to connectives, and verify the soundness of the supporting proof rules that allow us to control the transferal of resources precisely, even when resources are transferred through (trackable) invariants. To support thread-local reasoning, we use *fractions* to control sharing; intuitively, a fraction π expresses how much we *know* about 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 other views of the heap).

Moreover, we introduce a new definition of weakest precondition and Hoare triples, and prove soundness of expected higher-order concurrent separation logic proof rules. Among other things, the new definition of weakest precondition keeps properly track of resources for forked-off threads (in Iris, forked-off threads always had True as postcondition).

In the first part of the paper, we use fractions explicitly to control sharing and accounting of resources, but we take care to do so in a principled and modular manner. As such, in all examples but one, fractions are treated parametrically. We capitalize on this insight by showing how to define a more abstract logic, called **Iron**$^{++}$, which hides the use of fractions and thus allows users of the logic to avoid having to track fractions explicitly. 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), in a manner which works in practice. Indeed, regarding practical use, we have formalized Iron and Iron^{++} in Coq so that one can also smoothly mechanize proofs in Iron^{++}. We have used this Coq formalization to verify the examples in the paper, including the challenging *channel module* inspired by Singularity OS. There is one example, which we cannot verify in the abstract Iron^{++} logic. It is the derivation of a proof rule for a parallel composition operator defined using a reference cell and fork. In this case, it is useful to be able to "drop down" to the basic Iron logic and prove that example 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, and then only later describe the more abstract Iron^{++} (rather than the other way round).

In most of the paper we focus on memory resources, but in Section 7 we discuss how the Iron approach also applies to precise reasoning about other resources.

Contributions. In summary, we make the following contributions:

- We present **Iron**, the first (to the best of our knowledge) higher-order concurrent separation that allows for precise reasoning about memory resources that are transferable among dynamically allocated threads (Section 2).
- We demonstrate how Iron can be used for verifying challenging examples (Section 3).
- 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 4).
- 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 formalizes that memory resources are accounted for precisely and not leaked (Section 5).
- We show that the Iron approach can also be used to reason precisely about other user-defined resources, such as locks (Section 7).

- We have formalized all of the developments in the Coq proof assistant on top of the recent MoSeL framework [Krebbers et al. 2018] (Section 7).

2 THE IRON LOGIC

Before describing the connectives and rules of Iron (Section 2.1–2.5), and its adequacy theorem (Section 2.6) we discuss the programming language that we will use throughout this paper.

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):

$$v \in \mathit{Val} ::= () \mid z \mid \mathsf{true} \mid \mathsf{false} \mid \ell \mid \lambda x.\, e \mid \ldots \qquad (z \in \mathbb{Z})$$
$$e \in \mathit{Exp} ::= v \mid x \mid e_1(e_2) \mid \mathsf{fork}\ \{e\} \mid \mathsf{ref}(e) \mid \mathsf{free}(e) \mid !\, e \mid e_1 \leftarrow e_2 \mid \mathsf{cas}(e, e_1, e_2) \mid \ldots$$

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.

2.1 The Iron Logic

In this section we describe the main ingredients of the Iron program logic. The Iron logic is built on top of the Iris base logic—a foundational logic of resources. For reasons of space, we do not include a detailed description of the Iris base logic; we refer to Jung et al. [2018b] for an extensive formal description, and Birkedal and Bizjak [2017] for a tutorial-style introduction. The grammar of Iron propositions is as follows (the novel assertions of Iron are highlighted in blue):

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

Figure 1 displays a selection of the rules for these connectives. (We use $P \dashv\vdash Q$ as notation for bidirectional entailment.) Throughout this section, we will explain Iron's rules for heap manipulation (Section 2.2), fork-based concurrency (Section 2.3), and trackable invariants (Section 2.4), as well as how Iris's machinery for ghost state is embedded into Iron (Section 2.5).

Note that many of the Iron rules involve the "later" modality ($\triangleright$). This is 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. 2018b].

2.2 Hoare Triples and Heap Manipulation

As in traditional separation logic, Iron features Hoare triples $\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}$, where the pre- and postcondition P and Q are *thread local*, *i.e.,* they provide a *local view* of the heap.[1] The primary separation logic connective for specifying the view of the heap is the *points-to* connective $\ell \hookrightarrow v$,

[1] Like in Iris, Hoare triples are annotated with a *mask* $\mathcal{E}$ to keep track of which invariants are currently in force. For the time being, we will omit these masks, but come back to them in Section 2.4, where we introduce invariants.

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 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
$$\ell \hookrightarrow_{\pi_1} v * \mathfrak{e}_{\pi_2} \dashv\vdash \ell \hookrightarrow_{\pi_1+\pi_2} v$$

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

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

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

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

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

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

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

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

Fig. 1. Selected rules of the Iron logic.

which expresses that the location ℓ points to a value v, *and* that the thread owns ℓ exclusively, *i.e.,* it makes sure that no other thread can read from or write to ℓ concurrently.

Since pre- and postconditions are thread-local, each thread only has a partial local view of the global heap (other threads may own other parts of the heap). In order to precisely account for the use of resources (*e.g.,* to make sure that resources are not leaked), it is crucial to the exact resource footprint of a thread, *and* that there are no resources that are not tracked by any thread. In Iron, this is achieved by extending the points-to connective $\ell \hookrightarrow_\pi v$ with a fraction $\pi \in (0, 1]$, which represents the *view of the heap*. The semantics of this fraction is as follows: Whenever we own $\ell_1 \hookrightarrow_{\pi_1} v_1 * \ldots \ell_n \hookrightarrow_{\pi_n} v_1$ with $\sum \pi_i = 1$, we not only have unique ownership of these locations, but we also know that the global heap contains *exactly* the locations $\ell_1, \ldots, \ell_n$. If $\sum \pi_i < 1$, however, there may be other threads around that own different locations, *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 fractions of all threads sum up to 1 at all times.

In order express that the thread's view of the heap is empty, there is the connective $\mathfrak{e}_\pi$. Like the point-to connective, if we own $\mathfrak{e}_1$, we are sure that the global heap is empty. Indeed, given a Hoare triple $\{\mathfrak{e}_1\}\, e\, \{\mathfrak{e}_1\}$, adequacy (Theorem 2.2) guarantees that e does not leak memory, because the postcondition $\mathfrak{e}_1$ guarantees that the final heap (upon termination of all threads) will be empty.

There are a number of rules for manipulating $\mathfrak{e}_\pi$ and $\ell \hookrightarrow_\pi v$. First, since both $\mathfrak{e}_\pi$ and $\ell \hookrightarrow_\pi v$ represent a view of the heap, it is always possible to separate this view into an empty view $\mathfrak{e}_{\pi_1}$, 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 (HOARE-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 from our current view. Dually, the rule for deallocation (HOARE-FREE) gives back the permission $\mathfrak{e}_\pi$, which can subsequently be merged back into another $\mathfrak{e}_{\pi'}$ or $\ell \hookrightarrow_{\pi'} v$ using PT-SPLIT or EMP-SPLIT in left-to-right direction.

Unlike separation logic with fractional permissions [Boyland 2003], the rule for the store operation (HOARE-STORE) does not require the full fraction. Instead, owning $\ell \hookrightarrow_\pi v$ gives exclusive access to ℓ regardless of the fraction π, so owning both $\ell \hookrightarrow_\pi -$ and $\ell' \hookrightarrow_{\pi'} -$ implies $\ell \neq \ell'$ (rule PT-DISJ).

Let us see the rules we have seen so far in action on the following program:

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

In order to show that this program has no memory leaks, we shall prove $\{\mathfrak{e}_1\}\, e\, \{\mathfrak{e}_1\}$. First, we use EMP-SPLIT in right-to-left direction to turn the precondition $\mathfrak{e}_1$ into $\mathfrak{e}_{1/2} * \mathfrak{e}_{1/2}$. Then, we use HOARE-ALLOC twice, after which we need to prove $\{\ell_1 \hookrightarrow_{1/2} 0 * \ell_2 \hookrightarrow_{1/2} 0\}\, \mathsf{free}(\ell_1); \mathsf{free}(\ell_2)\, \{\mathfrak{e}_1\}$. This follows from two applications of HOARE-FREE, followed by EMP-SPLIT in left-to-right direction.

The accounting for resources by means of fractions is sound despite the fact that Iron is an *affine* separation logic, which means it enjoys the *weakening rule* $P * Q \vdash P$. Intuitively, if we would use weakening to drop a location $\ell \hookrightarrow_{\pi'} v$, we will not be able to reproduce $\mathfrak{e}_1$ in the postcondition.

As we already see in this simple example, keeping precise track of resources using fractions results in additional bookkeeping. However, as we illustrate through numerous examples in Section 3, this bookkeeping is principled, and can in fact be hidden in most cases as we will see in Section 4.

In the example, we fixed the total fraction to be 1. For more realistic proofs, however, it is desirable to write specifications that are *parametric* in the fraction π. This is crucial for composition: it is almost impossible to reuse a module which demands a total fraction of 1, because most of the time we will not be able to meet the preconditions. For the example, this means one should in fact prove $\{\mathfrak{e}_\pi\}\, e\, \{\mathfrak{e}_\pi\}$ *for any* π. Adapting the proof to this style is trivial, as all Iron's rules are parametric in the fraction. In the remainder of this paper we will thus consistently do so.

2.3 Fork-based Concurrency

In order to support concurrency, $\lambda_{\text{ref,conc}}$ has the expression 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 1.

The two rules deal with two different uses for fork $\{e\}$. The rule HOARE-FORK-TRUE 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 3.5, where we use this rule to prove the correctness of the parallel composition operator, implemented using a synchronization mechanism). Note that HOARE-FORK-TRUE is like the fork rule of Iris.

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. As such, 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 $\mathfrak{e}_\pi$ as well. Before taking a look at the actual example in Section 3.3, 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 2.2, 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 $\{\ell_1 \hookrightarrow_{\pi/2} 0 * \ell_2 \hookrightarrow_{\pi/2} 0\}\, \mathsf{fork}\, \{\mathsf{free}(\ell_1)\}\,; \mathsf{free}(\ell_2)\, \{\mathfrak{e}_\pi\}$. To do so, we use HOARE-FORK-EMP, after which it suffices to prove the following Hoare triples:

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

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:

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

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.

2.4 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 turn any proposition I into an invariant $\boxed{I}^{\mathcal{N}}$.[2] This assertion is *duplicable*[3] (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 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 $\ell \hookrightarrow_\pi v$ 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 $\ell \hookrightarrow_\pi v$ 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 in Iris, but this 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 4.2, 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 4. 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 thee three resources:

- The *trackable invariant assertion* $\boxed{\pi.I}^{\mathcal{N},\gamma}$, which expresses the knowledge that the invariant exists.[4] 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

[2] Every invariant has a namespace $\mathcal{N}$, 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 that are opened. When the mask $\mathcal{E}$ is omitted, is is assumed to be $\top$, the set of all masks. In practice, the use of masks results in some additional bookkeeping. As this bookkeeping is orthogonal to our focus, we mostly ignore namespaces in this paper, and refer to Jung et al. [2018b] for further details.

[3] 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. [2018b] for further details.

[4] 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.

express the view of the heap. The fraction p is a fractional permission [Boyland 2003] 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 $\mathsf{OPerm}_\gamma\,(1)$), it can be deallocated using the rule TINV-DEALLOC. Before coming back to 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 split among, and used by connectives $\mathfrak{e}_\pi$ and $\ell \hookrightarrow_\pi v$ appearing in I. A way to think about the use of $\mathfrak{e}_\pi$ 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 3.2, 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 in the fraction π, and thus cannot be put into the invariant.

Thread-local head reduction:

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

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

Threadpool reduction:

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

Fig. 2. Selected rules of the operational semantics of $\lambda_{\mathrm{ref,conc}}$.

2.5 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 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:

$$\frac{P \Rrightarrow_{\mathcal{E}} P' \qquad \{P'\}\, e\, \{w.\, Q\}_{\mathcal{E}} \qquad \forall w.\, Q \Rrightarrow_{\mathcal{E}} Q'}{\{P\}\, e\, \{w.\, Q\}_{\mathcal{E}}}\ \textsc{hoare-cons}$$

We will see an example of ghost state in Section 3.1.

2.6 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 $\lambda_{\mathrm{ref,conc}}$.

Operational semantics. The operational semantics of $\lambda_{\mathrm{ref,conc}}$ is given by means of small-step operational semantics; we show selected rules in Figure 2. 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* $(T, \sigma) \to_{\mathbf{tp}} (T', \sigma')$, and its reflexive-transitive closure $(T, \sigma) \to\!\!\!\to_{\mathbf{tp}} (T', \sigma')$.

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* $(e_1, \sigma_1) \to_{\mathbf{t}} (e_2, \sigma_2, \vec{e}_f)$. Following the conventions in Iris [Jung et al. 2018b], 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 $(e_1, \sigma_1) \to_{\mathbf{h}} (e_2, \sigma_2, \vec{e}_f)$, which is lifted by means of standard call-by-value evaluation contexts K to thread-local reductions $(K[\, e_1\,], \sigma_1) \to_{\mathbf{t}} (K[\, e_2\,], \sigma_2, \vec{e}_f)$.

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

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

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.*

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 2.2 (Adequacy for correct usage of resources). *Suppose the Hoare triple $\{\mathfrak{e}_1\}\ e\ \{\mathfrak{e}_1\}$ is derivable in Iron. Now, if we have:*

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

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

Note that the adequacy theorem for correct disposal 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 5.

3 EXAMPLES

In this section, we will specify and verify a channel module inspired by the one in Singularity OS (Section 3.3), as well as a client of that channel module (Section 3.4). Before doing that, we verify a simpler example—first in the setting of scoped concurrency, using the parallel composition operator (Section 3.1), and then in the setting of unscoped concurrency, using fork (Section 3.2).

Parallel composition. The parallel composition operator is not primitive in our language, but it is definable via fork in the usual way (Section 3.5). Parallel composition can be given the following specification (where P_i and Q_i are arbitrary Iron propositions):

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

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 4.3 we show how to hide this fraction.) We will prove this rule in Section 3.5, since its proof illustrates an important feature of Iron. For now, we will assume this rule to hold.

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

Fig. 3. The rules for the transition system used in Section 3.1, 3.2 and 3.5.

3.1 Resource Transfer Using Parallel Composition

Consider the following example program:

$$\begin{array}{l} e_{\mathsf{par}} \triangleq \mathsf{let}\ \ell = \mathsf{ref}\,(\mathsf{None})\ \mathsf{in} \\ \quad \begin{array}{l} \mathsf{let}\ \ell' = \mathsf{ref}\,(0)\ \mathsf{in} \\ \ell \leftarrow \mathsf{Some}\,\ell' \end{array} \ \Big\|\ \begin{array}{l} (\mathsf{rec}\,\mathsf{cleanup}() = \mathsf{match}\,!\,\ell\,\mathsf{with} \\ \qquad \mathsf{None} \Rightarrow \mathsf{cleanup}() \\ \quad \mid \mathsf{Some}\,\ell' \Rightarrow \mathsf{free}(\ell') \\ \quad \mathsf{end})\,() \end{array} \\ \quad \mathsf{free}(\ell) \end{array}$$

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 of 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:

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

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 3.

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 $\ell \hookrightarrow_{\pi/4} \mathsf{None}$ (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 3, which sets us up with all the resources needed to establish the initial state of the invariant I. So, using TINV-ALLOC, we obtain $\pi.I(\pi)^{\mathcal{N},\gamma}$, thereby giving up $s_1(\gamma_{\mathsf{sts}})$ and $\ell \hookrightarrow_{\pi/4} \mathsf{None}$ (*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:

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

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:

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

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 $\ell' \hookrightarrow_{\pi/8} 0$ (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 $\ell \hookrightarrow_{\pi/8} \mathsf{None}$, and by using the assignment rule HOARE-STORE, we then obtain $\ell \hookrightarrow_{\pi/8} \mathsf{Some}\,\ell'$. Combining this with $\ell' \hookrightarrow_{\pi/8} 0$, 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 3). 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 $\ell \hookrightarrow_{\pi_{41}} \mathsf{Some}\,\ell'$, and $\ell' \hookrightarrow_{\pi_{42}} -$, and $\pi/4 = \pi_{41} + \pi_{42}$. We then update the transition system to $t_3(\gamma_{\mathsf{sts}})$, and close the invariant using $\ell \hookrightarrow_{\pi_{41}} -$, 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:

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

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 $\ell \hookrightarrow_{\pi/4} -$, 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 4), we can hide the fractions completely.

3.2 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:

$$\begin{aligned}
e_{\mathsf{fork}} \triangleq\ & \mathsf{let}\ \ell = \mathsf{ref}\,(\mathsf{None})\ \mathsf{in}\\
& \mathsf{let\ rec}\ \mathsf{cleanup}() = \mathsf{match}\ !\,\ell\ \mathsf{with}\\
& \qquad\qquad\qquad\qquad \mathsf{None} \Rightarrow \mathsf{cleanup}()\\
& \qquad\qquad\qquad\qquad \mid \mathsf{Some}\,\ell' \Rightarrow \mathsf{free}(\ell');\mathsf{free}(\ell)\\
& \qquad\qquad\qquad\qquad \mathsf{end}\ \mathsf{in}\\
& \mathsf{fork}\ \{\mathsf{cleanup}()\}\,;\\
& \mathsf{let}\ \ell' = \mathsf{ref}\,(0)\ \mathsf{in}\ \ell \leftarrow \mathsf{Some}\,\ell'
\end{aligned}$$

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):

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

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.

3.3 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:

$$
\begin{aligned}
\mathsf{newchannel}(d) \triangleq\ & \mathsf{let}\ q_x = \mathsf{qNew}\,()\ \mathsf{in}\ \mathsf{let}\ q_y = \mathsf{qNew}\,()\ \mathsf{in} \\
& \mathsf{let}\ x_a = \mathsf{ref}\,(\mathsf{true})\ \mathsf{in}\ \mathsf{let}\ y_a = \mathsf{ref}\,(\mathsf{true})\ \mathsf{in} \\
& \mathsf{let\ rec}\ \mathsf{cleanup}() = \mathsf{if}\,!\,x_a\ \mathsf{then}\ \mathsf{cleanup}() \\
& \qquad\qquad\qquad\qquad \mathsf{else\ if}\,!\,y_a\ \mathsf{then}\ \mathsf{cleanup}() \\
& \qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{qDealloc}(d, q_x); \mathsf{qDealloc}(d, q_y); \mathsf{free}(x_a); \mathsf{free}(y_a) \\
& \mathsf{fork}\ \{\mathsf{cleanup}()\}\,; \\
& ((q_x, q_y, x_a), (q_y, q_x, y_a))
\end{aligned}
$$

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:

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

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 send over the channel should satisfy. For instance, $\Phi(w, \pi)$ could be $\exists n.\ w \hookrightarrow_\pi n * \mathsf{even}(n)$, 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):

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

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

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

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

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):[5]

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

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

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

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

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:

$$\mathsf{Endpoint}_i((q_1,q_2,x),\gamma,\pi)\triangleq\exists\pi_1,\pi_2,\pi_3.\ \mathsf{rHandle}(q_1,\gamma,\pi_1)*\mathsf{iHandle}(q_2,\gamma,\pi_2)*x\hookrightarrow_{\pi_3}\mathsf{true}*\mathfrak{C}\mathfrak{I}$$

[5] 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.

Each endpoint has two handles, one for sending, and one for receiving messages, and we know that the endpoint is alive, hence we own the flag $x \hookrightarrow_{\pi_3} \mathsf{true}$ saying so. 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 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 $x \hookrightarrow_{\pi_3} \mathsf{true}$ 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.

3.4 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:

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

The method mkList(n) creates a linked list of integers $n, n-1, \ldots, 1$, and 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 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 2.2), we can thus conclude that all the allocated memory has been disposed of when the program terminates.

3.5 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:

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

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

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

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 as follows:

$$\mathsf{joinHandle}(c, \Phi, \pi) \triangleq \exists \gamma_{\mathsf{sts}}.\, t_2(\gamma_{\mathsf{sts}}) * \boxed{\begin{array}{l}(s_1(\gamma_{\mathsf{sts}}) * c \hookrightarrow_\pi \mathsf{None}) \vee\\ (s_2(\gamma_{\mathsf{sts}}) * (\exists w.\, c \hookrightarrow_\pi \mathsf{Some}\, w * \Phi(w))) \vee\\ s_3(\gamma_{\mathsf{sts}})\end{array}}^{\mathcal{N}}$$

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 3. 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.

4 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 entailment relation ($\vdash$) is defined as:

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

(Recall that Prop is the type of Iris propositions.) 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 ($\overset{\frown}{\mapsto}$). Below we give the definitions of some of these connectives:

$$\begin{aligned}
(P \wedge Q)(\pi) &\triangleq P(\pi) \wedge Q(\pi) & \mathsf{Emp}(\pi) &\triangleq \pi = 0\\
(P \Rightarrow Q)(\pi) &\triangleq P(\pi) \Rightarrow Q(\pi) & (P -\!\!* Q)(\pi) &\triangleq \forall \pi'.\, P(\pi') -\!\!* Q(\pi + \pi')\\
(t = s)(\pi) &\triangleq (t = s) & (\ell \overset{\frown}{\mapsto} v)(\pi) &\triangleq \pi > 0 \wedge \ell \hookrightarrow_\pi v
\end{aligned}$$

An important feature of Iron^{++} is that it, unlike Iris and Iron, *does not* satisfy the weakening rule $P * Q \vdash P$. In particular, $\ell \overset{\frown}{\mapsto} v * \ell' \overset{\frown}{\mapsto} v' \vdash \ell \overset{\frown}{\mapsto} v$ 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.

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 $\ell \overset{\frown}{\mapsto} v$ 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 4.1). We then show how trackable invariants are integrated into Iron^{++} (Section 4.2), and finally conclude with some examples (Section 4.3). A selection of Iron^{++} rules is shown in Figure 4.

4.1 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:[6]

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

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 inherent 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 2.1 and 2.2) 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 $\emptyset$).

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

Ordinary separation logic:

$$\textsc{lhoare-frame}\ \frac{\{P\}\, e\, \{w.\, Q\}_{\mathcal{E}}}{\{P * R\}\, e\, \{w.\, Q * R\}_{\mathcal{E}}} \qquad \textsc{lhoare-val}\ \{\mathsf{Emp}\}\, v\, \{w.\, w = v\}_{\mathcal{E}} \qquad \textsc{lhoare-}\lambda\ \frac{\{P\}\, e[v/x]\, \{w.\, Q\}_{\mathcal{E}}}{\{\triangleright P\}\, (\lambda x.\, e)\, v\, \{w.\, Q\}_{\mathcal{E}}}$$

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

Heap manipulation:

$$\textsc{lpt-disj}\ \ell_1 \mathrel{\widehat{\hookrightarrow}} - * \ell_2 \mathrel{\widehat{\hookrightarrow}} - \vdash \ell_1 \neq \ell_2 \qquad \textsc{lhoare-alloc}\ \{\mathsf{Emp}\}\, \mathsf{ref}(v)\, \{\ell.\, \ell \mathrel{\widehat{\hookrightarrow}} v\}_{\mathcal{E}} \qquad \textsc{lhoare-free}\ \{\triangleright\, \ell \mathrel{\widehat{\hookrightarrow}} -\}\, \mathsf{free}(\ell)\, \{\mathsf{Emp}\}_{\mathcal{E}}$$

$$\textsc{lhoare-load}\ \{\triangleright\, \ell \mathrel{\widehat{\hookrightarrow}} v\}\, !\ell\, \{w.\, w = v * \ell \mathrel{\widehat{\hookrightarrow}} v\}_{\mathcal{E}} \qquad \textsc{lhoare-store}\ \{\triangleright\, \ell \mathrel{\widehat{\hookrightarrow}} -\}\, \ell \leftarrow w\, \{\ell \mathrel{\widehat{\hookrightarrow}} w\}_{\mathcal{E}}$$

Fork-based concurrency:

$$\textsc{lhoare-fork}\ \frac{\{P\}\, e\, \{\mathsf{Emp}\}}{\{\triangleright P\}\, \mathsf{fork}\ \{e\}\ \{w.\, w = () \wedge \mathsf{Emp}\}_{\mathcal{E}}}$$

Trackable (Iron) invariants:

$$\textsc{ltinv-split}\ \widehat{\mathsf{OPerm}}_\gamma\,(p_1 + p_2) \dashv\vdash \widehat{\mathsf{OPerm}}_\gamma\,(p_1) * \widehat{\mathsf{OPerm}}_\gamma\,(p_2) \qquad \textsc{ltinv-dup}\ \boxed{I}^{\mathcal{N},\gamma} * \boxed{I}^{\mathcal{N},\gamma} \dashv\vdash \boxed{I}^{\mathcal{N},\gamma}$$

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

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

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

Fig. 4. Selected rules of the Iron^{++} logic.

Theorem 4.1 (Lifted basic adequacy). *Given a first-order predicate over values ϕ, and suppose the Hoare triple $\left\{\ast_{(\ell,v)\in\sigma}\, \ell \mathrel{\widehat{\hookrightarrow}} v\right\}\, e\, \{w.\, \phi(w)\}$ is derivable in Iron^{++}. Now, if we have:*

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

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.*

Theorem 4.2 (Lifted adequacy for correct usage of resources). *Suppose the Hoare triple* $\{*_{(\ell,v)\in\sigma} \ell \hookrightarrow v\}\, e\, \{w.\ *_{(\ell,v)\in\sigma'} \ell \hookrightarrow v\}$ *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* $\sigma'' = \sigma'$.

4.2 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.

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 $p \in (0,1]$):

$$\boxed{I}^{\mathcal{N},\gamma}(\pi) \triangleq \pi = 0 \land \boxed{\pi.\exists \pi_1 \geq 0, \pi_2 > 0.\ (\pi = \pi_1 + \pi_2) * I(\pi_1) * \mathfrak{e}_{\pi_2}}^{\mathcal{N},\gamma}$$

$$\widehat{\mathsf{OPerm}}_\gamma(p)(\pi) \triangleq \pi = 0 \land \mathsf{OPerm}_\gamma(p)$$

$$\widehat{\mathsf{DPerm}}_\gamma(\pi) \triangleq \pi > 0 \land \mathsf{DPerm}_\gamma(\pi)$$

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 affine propositions. Recall that the invariant deallocation token is not affine, and thus cannot be put into the invariant.

4.3 Examples

We can verify all of the examples we have presented in this paper in Iron^{++} (including the channel module from Section 3.3), with the exception of the parallel composition operator (Section 3.5). 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):

$$\{\mathsf{Emp}\}\ \mathsf{newchannel}(d)\ \{(\mathsf{ep}_1,\mathsf{ep}_2).\ \mathsf{Endpoint}_1(\mathsf{ep}_1,\gamma) * \mathsf{Endpoint}_2(\mathsf{ep}_2,\gamma)\}$$

$$\{\mathsf{Endpoint}_i(\mathsf{ep},\gamma) * \Phi(u)\}\ \mathsf{send}(\mathsf{ep},u)\ \{\mathsf{Endpoint}_i(\mathsf{ep},\gamma)\}$$

$$\{\mathsf{Endpoint}_i(\mathsf{ep},\gamma)\}\ \mathsf{receive}(\mathsf{ep})\ \{w.\ \mathsf{Endpoint}_i(\mathsf{ep},\gamma) * \Phi(w)\}$$

$$\{\mathsf{Endpoint}_i(\mathsf{ep},\gamma)\}\ \mathsf{close}(\mathsf{ep})\ \{\mathsf{Emp}\}$$

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

[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 trivialize I.

have done so in Coq all the way—from the (precise HOCAP-style) 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:

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

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).

5 SEMANTICS OF IRON

The semantics of Iron builds upon the semantics of Iris [Jung et al. 2016, 2018b; 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:

- We change the *state interpretation* of Hoare triples (which are defined in terms of weakest preconditions) to incorporate tracked points-to connectives $\ell \hookrightarrow_\pi v$.
- We change the definition of Hoare triples to account for the resources of forked-off threads.
- 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.

Weakest preconditions in Iris. Weakest preconditions in Iris are defined as follows:

$$\begin{aligned}
\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq\ & (e \in \mathit{Val} \wedge \Rightarrow_{\mathcal{E}} \Phi(e)) && \text{(return value)}\\
& \vee \Big(e \notin \mathit{Val} \wedge \forall \sigma_1.\, S(\sigma_1) \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rightarrow^{\emptyset} \big(\exists e_2, \sigma_2, \vec{e}_f.\, (e, \sigma_1) \rightarrow_{\mathbf{h}} (e_2, \sigma_2, \vec{e}_f)\big) && \text{(safety)}\\
& \quad \wedge \triangleright \forall e_2, \sigma_2, \vec{e}_f.\, \big((e, \sigma_1) \rightarrow_{\mathbf{h}} (e_2, \sigma_2, \vec{e}_f)\big) \mathrel{-\!\!*} {}^{\emptyset}\!\Rightarrow^{\mathcal{E}} && \text{(preservation)}\\
& \qquad \big(S(\sigma_2) * \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\} * \textstyle\ast_{e' \in \vec{e}_f} \mathsf{wp}_{\top}\, e'\, \{v.\mathsf{True}\}\big)\Big)
\end{aligned}$$

In this definition, the *state interpretation* $S : \mathit{State} \to \mathsf{Prop}$ links the physical state σ to a proposition in the logic. For an ML-like language with states being just heaps (*i.e.,* maps from locations to values, $\mathit{State} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}$), as used in most Iris papers, one would use:

$$S(\sigma) \triangleq \boxed{\bullet\, \sigma}^{\gamma_h} \qquad\qquad \ell \hookrightarrow v \triangleq \boxed{\circ\, \{\ell \mapsto v\}}^{\gamma_h}$$

Here, γ_h is a fixed ghost name, and the *resource algebra* that is being used is $\textsc{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} \mathrm{Ex}(\mathit{Val}))$. In this, owning a location $\ell \hookrightarrow v$ ensures that the heap σ contains *at least* the location ℓ with value w, but there may be more locations. However, there is no way for a client to express that the heap consists *exactly* of certain locations (*e.g.,* to express that the heap is empty).

Tracked points-to connectives. The first change we make towards modeling the Iron logic is to change the state interpretation so that we can express that the heap *surely* contains certain locations. This is needed to incorporate the tracked points-to connective $\ell \hookrightarrow_\pi v$ and the $\mathfrak{e}_\pi$ connective. To do so, we use the resource algebra $\textsc{Auth}\big((0,1] \times (Loc \xrightarrow{\text{fin}} \text{Ex}(Val)) + \{\varepsilon\}\big)$, and define:

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

The crucial benefit of the new resource algebra and state interpretation is that $\mathfrak{e}_1 * S(\sigma)$ implies that σ is empty. This is exactly the property that we need to prove Iron's adequacy statement (Theorem 2.2). Similarly, $\ell \hookrightarrow_1 v * S(\sigma)$ implies not only that ℓ is in σ, but that σ is exactly the singleton heap. This is in contrast to $\ell \hookrightarrow_\pi v * S(\sigma)$ for $\pi < 1$, which only implies that the domain of σ contains at least ℓ. (This is sufficient for proving soundness of the rules for heap manipulation, for which, contrary to adequacy, the fraction π could be anything).

Accounting for resources of forked-off threads. We can get very far by instantiating the vanilla Iris weakest preconditions with a custom state interpretation, but it has one important limitation: at each step the newly created threads ($\vec{e}_f$) are only required to be safe, *i.e.,* have postcondition True. Thus with the vanilla definition, it is impossible to transfer resources to a forked-off thread and know that they have been correctly disposed of. Hence, it is then impossible to derive the strong fork rule HOARE-FORK-EMP. What is needed is the ability for the parent thread to specify what the postcondition of the forked-off thread should be, instead of it being fixed to being True.

We achieve this in two steps: (1) we extend the definition of weakest preconditions to take the postconditions of forked threads into account and change the type of the state interpretation S accordingly, and, (2) we define a corresponding state interpretation S for $\lambda_{\text{ref,conc}}$.

For step (1), we change the type of the state interpretation to be a function from states and non-negative rationals to propositions, *i.e.,* $S : State \times [0,\infty) \to \text{Prop}$, and correspondingly alter the definition of weakest precondition definition (changes are highlighted in blue):[8]

$$\begin{aligned}\text{wp}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq\ & (e \in Val \wedge \Rrightarrow_{\mathcal{E}} \Phi(e)) \\ & \vee \Big(e \notin Val \wedge \forall \sigma_1, \pi_1.\, S(\sigma_1, \pi_1) \mathrel{-\!\!*}\ {}^{\mathcal{E}}\Rrightarrow^{\emptyset}\ \big(\text{red}(e,\sigma_1) \\ & \quad \wedge \triangleright \forall e_2, \sigma_2, \vec{e}_f.\, \big((e,\sigma_1) \to_{\mathrm{h}} (e_2, \sigma_2, \vec{e}_f)\big) \mathrel{-\!\!*}\ {}^{\emptyset}\Rrightarrow^{\mathcal{E}} \\ & \qquad \exists \vec{\pi}_f.\, \big(S(\sigma_2, \pi_1 + \textstyle\sum \vec{\pi}_f) * \text{wp}_{\mathcal{E}}\, e_2\, \{\Phi\} * \mathbin{\text{✻}}_{(e',\pi') \in \overrightarrow{e_f, \pi_f}} \text{wp}_{\top}\, e'\, \{\_.\mathfrak{e}_{\pi'}\}\big)\big)\Big)\end{aligned}$$

In order to account for both the postconditions True (for HOARE-FORK-TRUE) and $\mathfrak{e}_\pi$ (for HOARE-FORK-EMP), we extend the domain of $\mathfrak{e}_\pi$ from $\pi \in (0,1]$ to $\pi \in [0,1]$ by letting $\mathfrak{e}_0 \triangleq$ True.

For step (2) we make a crucial alteration to the ghost state used to model the heap. Instead of the standard fractions $(0,1]$ in the resource algebra construction, we use arbitrary positive rational numbers. Formally, we use $\textsc{Auth}\big((0,\infty) \times (Loc \xrightarrow{\text{fin}} \text{Ex}(Val)) + \{\varepsilon\}\big)$, and define:

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

The new resource algebra construction provides the increased flexibility in the sense that it allows one to allocate a fresh permission $\mathfrak{e}_\pi$ for each forked-off thread. As such, the maximal fraction that is allowed in the state interpretation is thus no longer 1, but 1 plus the sum of the fractions handed out to all forked-off threads.

[8] In the Coq formalization, the postconditions of forked-off threads can be arbitrary propositions, and do not need to be of the shape $\mathfrak{e}_\pi$. Correspondingly, the state interpretation takes a list of propositions rather than a single number. We ignore this extra generality in the paper as it is unnecessary for our purposes.

This new version of weakest preconditions comes with a stronger adequacy statement that connects up with the postconditions of the forked-off threads. When all threads have terminated, this adequacy statement allows one to use the postconditions of all forked-off threads to establish the final result. A corollary of this stronger adequacy theorem (that is sufficient to prove the adequacy property in Theorem 2.2) is as follows.

Theorem 5.1. *Given a first-order predicate over an initial states, final states, and values ϕ, and suppose the following Hoare triple is derivable:*

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

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 2.2) 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 (we prove partial correctness, so forked-off threads may just loop and never dispose their resources).

In order to show that the actual adequacy result (Theorem 2.2) follows from Theorem 5.1, we let $\pi \triangleq 1$ and $Q \triangleq \mathfrak{e}_1$ and $\phi(v, \sigma, \sigma') \triangleq (\sigma = \sigma')$. Consequently, it suffices to show:

$$\mathfrak{e}_1 * S(\sigma, \sum \vec{\pi}_f) * \left(\divideontimes_{\pi' \in \vec{\pi}_f} \mathfrak{e}_{\pi'}\right) \vdash \emptyset = \sigma$$

This statement follows from some reasoning about the used resource algebra. The key point is that we have obtained a total sum of $\mathfrak{e}$ connectives whose summed fraction is $1 + \sum \vec{\pi}_f$, *i.e.,* the fraction in the state interpretation.

6 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 4, 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).

7 DISCUSSION AND FUTURE WORK

Trackable invariants. Recall that Iron's trackable invariants require that the proposition that it 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^{++}. However, it is unclear how to generalize such a bespoke sharing mechanism to the case where it is not known statically which thread is going to dispose of resources. An example of where it is not known statically is a variant of the message passing example, where the cleanup thread would be replaced by two threads, one for each endpoint. The last thread to run would then also cleanup the two flags, and reclaim all the resources. We stress that this only means that this variant could not be verified in Iron^{++} directly, it would still be possible to verify it by dropping down to Iron (like we have done to verify the parallel composition operator).

Managing other resources. We believe that the methodology we have described 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:

$$\mathsf{newLock}() \triangleq \mathsf{ref}(\mathsf{false}) \qquad \mathsf{dispose}(\ell) \triangleq \mathsf{free}(\ell)$$

$$\mathsf{acquire}(\ell) \triangleq \mathsf{if}\,\mathsf{cas}(\ell, \mathsf{false}, \mathsf{true})\,\mathsf{then}\,()\,\mathsf{else}\,\mathsf{acquire}(\ell) \qquad \mathsf{release}(\ell) \triangleq \ell \leftarrow \mathsf{false}$$

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:

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

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 Iron. 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 5) 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.

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*.

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, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. Rustbelt: Securing the foundations of the rust programming language. *PACMPL* 2, POPL (2018).

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. 2018b. 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*.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*.

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*.

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects 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.

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. In *OOPSLA*.

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

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of `runST`. In *POPL*.

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*.