Iron: Managing Obligations in Higher-Order Concurrent Separation Logic

Daniel Gratzer[1], Aleš Bizjak[2], Robbert Krebbers[3], and Lars Birkedal[2]

[1] Carnegie Mellon University (jozefg@cmu.edu)
[2] Aarhus University ({abizjak,birkedal}@cs.au.dk)
[3] Delft University of Technology (mail@robbertkrebbers.nl)

Abstract. We present **Iron**, a higher-order concurrent separation logic with linear resources for managing *obligations* to use resources. We demonstrate how linear resources can be used to specify, *e.g.,* that clients of a lock module are obligated to release a lock after use. We also show how linear resources, in combination with a novel total correctness definition, can be used to reason about cost semantics. Iron is an extension of Iris, a state-of-the-art higher-order concurrent separation logic, and includes an "affine" modality, using which one can retain the full reasoning power of Iris. The soundness of our approach is ensured by a model, which links linear resources in the logic to the operational semantics, via a new interpretation of Hoare triples.

1 Introduction

Precise accounting of resources such as memory, locks, file handles, is hard—especially in 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 have been proposed [34,12,11,10,28,9,13,17], with increasingly sophisticated and powerful mechanism for managing resources. In particular, variants of *higher-order* concurrent separation logic [33,29,22,20,24,21] have emerged which can be used to give very general and modular specifications to libraries and programs.

A recent effort in this direction is Iris [22,20,24,21]—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 [22], Kripke logical-relations models for relational reasoning in ML-like languages [25,26,32], program logics for relaxed memory models [23] and object capability [30], and a safety proof for a realistic subset of the Rust programming language [19].

Despite Iris's broad applicability for proving functional correctness, it cannot be used to modularly prove that resources are *necessarily* used. For example, one cannot express in Iris 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. The reason why this kind of reasoning cannot be done in Iris is that Iris is an *affine* separation

logic, which means that for every proposition P and Q, the weakening rule $P * Q \vdash P$ holds. As a result, one can throw away resources, such as the "points-to" connective $\ell \hookrightarrow v$, arbitrarily. To the best of our knowledge, this shortcoming applies to all other existing logics for fine-grained concurrency [28,29,9] as well. In this we remedy this situation by presenting **Iron**, which extends Iris with a notion of *linear resources*—resources that cannot be thrown away.

In the remainder of the introduction, we show how Iris's restriction to affine resources allows one to give specifications to incorrect programs that exhibit a classic problem in concurrency—forgetting to release a lock. We then show how linear resources in Iron repair this problem.

Lock specification in higher-order concurrent separation logic. We first recall the specification of a lock module (which can be implemented in a fine-grained concurrent way using concurrency primitives as compare-and-swap):

$$\begin{aligned}
\{P\}\ &\mathsf{newLock}()\ \{v.\ \exists\gamma.\ \mathsf{islock}(v,P,\gamma)\}\\
\{\mathsf{islock}(v,P,\gamma)\}\ &\mathsf{acquire}\, v\ \{P * \mathsf{locked}(v,\gamma)\}\\
\{\mathsf{islock}(v,P,\gamma) * P * \mathsf{locked}(v,\gamma)\}\ &\mathsf{release}\, v\ \{\mathsf{True}\}\\
\mathsf{islock}(v,P,\gamma) \Rrightarrow\ &\Box\, \mathsf{islock}(v,P,\gamma)
\end{aligned}$$

The lock module has three methods, newLock, acquire and release for creation, acquisition, and release of the lock, respectively. Their specifications are parameterized by an arbitrary Iris proposition P, which describes the resources guarded by the lock. The specification uses two abstract predicates, $\mathsf{islock}(v,P,\gamma)$ and $\mathsf{locked}(v,\gamma)$. The former states that the value v is a lock, which protects the resources P. The latter is a predicate that we obtain when acquiring a lock and is necessary to release the lock. It prevents other threads from releasing the lock and then acquiring it themselves. The *ghost name* γ is used to associate a logical name to each lock, but is not important for the intuitive understanding of the specification.

The last rule uses the always modality $\Box$ to express that the abstract predicate $\mathsf{islock}(v,P,\gamma)$ is *persistent*, which means that it does not express any 'ownership' of resources but merely the 'knowledge' that the lock v guards the resources P. Persistency, in particular, implies that the predicate is duplicable, *i.e.,* $\mathsf{islock}(v,P,\gamma) \Leftrightarrow \mathsf{islock}(v,P,\gamma) * \mathsf{islock}(v,P,\gamma)$, which allows one to give islock predicates to any thread that wishes to use the lock.

Incorrect usage of a lock. We now demonstrate how the above lock specification does not prevent certain incorrect uses of the lock. Using a lock we implement a simple, coarse-grained bag with the following specification adapted from previous work [29,15]:

$$\begin{aligned}
\{\mathsf{True}\}\ &\mathsf{newBag}()\ \{b.\ \exists\gamma.\ \mathsf{isBag}(b,\Phi,\gamma)\}\\
\{\mathsf{isBag}(b,\Phi,\gamma) * \Phi(u)\}\ &\mathsf{insert}\, b\, u\ \{\mathsf{Emp}\}\\
\{\mathsf{isBag}(b,\Phi,\gamma)\}\ &\mathsf{remove}\, b\ \{v.\ v = \mathsf{None} \vee \exists x.\ v = \mathsf{Some}\, x \wedge \Phi(x)\}\\
\mathsf{isBag}(b,\Phi,\gamma) \Rrightarrow\ &\Box\, \mathsf{isBag}(b,\Phi,\gamma)
\end{aligned}$$

```
let newBag = λ_. (ref(None), newLock())
  let insert = λ(ℓ, lock) x. acquire lock; ℓ ← Some(x, !ℓ); release lock
let remove = λ(ℓ, lock). acquire lock;
                         let r = match !ℓ with
                                   None       ⇒ None
                                 | Some(x, ℓ') ⇒ ℓ ← ℓ'; Some x
                                 end in
                         release lock; r
```

Fig. 1.1. A coarse-grained bag implementation.

The bag is intended to be shared among threads, so we give it a specification which does not state exactly which elements are in the bag, but only that each element satisfies a given predicate Φ. The abstract predicate $\mathsf{isBag}(b, \Phi, \gamma)$ thus states that the value b is a bag, all of whose elements satisfy the predicate Φ.

A possible implementation of this specification is shown in Figure 1.1. A bag value is a pair of a container, its first component, and a lock guarding access to it, its second component. In Iris one can show that this implementation meets the above specification, and it does indeed work as intended. However, it is very easy to forget to release a lock in one of the methods and, for instance, write the following incorrect implementation of the insert method:

$$\mathsf{let}\ \mathsf{incorrect\_insert} = \lambda(\ell, \mathit{lock})\ x.\ \mathsf{acquire}\ \mathit{lock};\ \ell \leftarrow \mathsf{Some}(x, !\ell)$$

where we have forgotten to call release before returning. The problem is that this implementation can also be shown to satisfy the specification in Iris (and similar program logics) (!), even though that is obviously undesirable.

In Iris, it is impossible to disallow the faulty implementation, while allowing the correct one. In the Iron logic introduced in this paper, such faulty implementations can be disallowed and, indeed, we show that it is *impossible* to show that the incorrect_insert method satisfies the given specification.

How Iron avoids incorrect usage of locks. Let us see how this affects the lock specification of the running example:

$$\begin{aligned}
\{\blacksquare P\}\ \mathsf{newLock}()\ \{v.\ \exists\gamma.\ \mathsf{islock}(v, P, \gamma)\} \\
\{\mathsf{islock}(v, P, \gamma)\}\ \mathsf{acquire}\, v\ \{\blacksquare P * \mathsf{locked}(v, \gamma)\} \\
\{\mathsf{islock}(v, P, \gamma) * \blacksquare P * \mathsf{locked}(v, \gamma)\}\ \mathsf{release}\, v\ \{\mathsf{Emp}\} \\
\mathsf{islock}(v, P, \gamma) \Rightarrow \blacksquare\!\!\!\!\blacksquare \mathsf{islock}(v, P, \gamma)
\end{aligned}$$

There are a few differences from the previous specification. The first is that the post-condition of release is now Emp rather than True, which states that there are no linear resources left after calling release.[4] Since propositions are linear by

[4] The propositions Emp and True are different in Iron, whereas in Iris Emp is (necessarily) equivalent to True, due to the fact that Iris is an affine separation logic.

default in Iron, this means in particular that the proposition $\mathsf{locked}(v, \gamma)$ is linear, so when one does not release a lock in a method, it will be clearly visible in the post-condition of that method. In particular this means that we will *not* be able to verify that the faulty insert method meets the specification for insert: after we have acquired the lock in insert, we obtain the $\mathsf{locked}(v, \gamma)$ predicate and the only way we can meet the Emp post-condition of insert is by calling release.

A key feature of Iron is that it allows one to combine affine and linear resources seamlessly. This is achieved using the *affine modality* $\blacksquare$, which expresses that a proposition does not contain any linear resources, *i.e.,* it is an ordinary Iris proposition. The second difference to the lock specification is the use of the modality in the Hoare triples of the lock operations, where the modality makes sure that the resources P guarded by the lock are affine.

Since the lock is restricted to guard affine resources, it can be used for simple Iris-style thread-local reasoning when proving clients of the lock module—clients may give the islock predicate to arbitrarily many different threads and moreover do not need to keep track of who disposes the lock. This is formalized by the rule $\mathsf{islock}(v, P, \gamma) \Rightarrow \blacksquare\mathsf{islock}(v, P, \gamma)$, which uses Iron's new $\blacksquare$ modality, instead of Iris's $\square$ modality. The new $\blacksquare$ modality does not just express that a proposition is persistent, but also that it is *affine*, *i.e.,* that it can be thrown away.

Contributions. We present **Iron**, a higher-order concurrent separation logic with support for linear resources. Our contribution consists of the following parts:

- We describe a novel and generic way of instrumenting programs with annotations to describe obligations to use resources (Section 2).
- We incorporate linear resources at the level of the separation logic to reason about the correct usage of these annotations (Section 3) and prove a strong adequacy theorem (Theorem 1), which formalizes the connection between an instrumented operational semantics and the linear resources in the logic.
- We demonstrate that our approach is very flexible: it can be used to reason about a variety of resources, such as programmer defined locks, disposable references, and cost semantics including asymptotic analysis (Section 4).
- We show that Iron is sound by constructing a model (Section 5).
- The important parts of our development and most examples are formalized in Coq (Section 6 and the GitHub repository [14]).

2 Instrumentation of the programming language

Similar to Iris, the Iron logic is parameterized by the language of program expressions that one wishes to reason about. For the purpose of this paper, to make the discussion more concrete, we instantiate Iron with an ML-like language with higher-order store, explicit deallocation, fork, and compare-and-swap (cas), as given below (we omit the usual operations on pairs and sums):

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

The most interesting operations are emit and consume, which can be used to annotate programs with resource obligations. The semantics of these operations are given by means of multimaps from identifiers to values: the operation $\mathsf{emit}(\eta := e)$ adds an entry with key η and value e to the multimap of resource obligations, and conversely, the operation $\mathsf{consume}(\eta := e)$ cancels an obligation with key η and value e stored in the multimap. The keys η, called *tags*, are used to keep track of the program module to which a resource obligations belongs.

The choice of multimaps is a simplification made for this paper. In reality, Iron is designed so that any commutative monoid may be used to represent program resources. All that is needed is for the operational semantics of the language under consideration to be instrumented with the desired commutative monoid. All of the semantics (Section 5) and major results like Theorem 1 make no assumptions on the particular monoid being used. In order to emphasize this generality, we write $m \cdot n$ instead of $m \cup n$ and ε instead of $\emptyset$.

Example programs. In order to illustrate how the instrumentation using emit and consume is used we show some examples. In the first example we instrument the ordinary implementation of the spin lock as alluded to in the introduction:

$$\begin{aligned}
&\mathsf{let}\ \mathsf{newLock} = \lambda_.\ \mathsf{ref}(\mathsf{false}) \\
&\quad \mathsf{let}\ \mathsf{acquire} = \lambda\ell.\ \mathsf{if}\ \mathsf{cas}(\ell, \mathsf{false}, \mathsf{true})\ \mathsf{then}\ \mathsf{emit}(n_{\mathsf{lock}} := \ell)\ \mathsf{else}\ \mathsf{acquire}\ \ell \\
&\quad \mathsf{let}\ \mathsf{release} = \lambda\ell.\ \mathsf{consume}(n_{\mathsf{lock}} := \ell);\ \ell \leftarrow \mathsf{false}
\end{aligned}$$

Acquiring a lock emits a resource ℓ, which conversely, is consumed using by the release method. Using this instrumentation, we will show, in Section 4.1, how to use the Iron logic to verify that each acquire is indeed followed by a corresponding release, *i.e.,* we have not forgotten to release any locks.

A different way of using linear resources is to model a cost semantics, *i.e.,* to keep track of the number of steps a program takes to terminate. We demonstrate this by considering an algorithm for computing the n-th Fibonacci number:

$$\begin{aligned}
&\mathsf{let}\ \mathsf{worker} = \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ (1,0)\ \mathsf{else} \\
&\qquad\qquad \mathsf{let}\ (n_1, n_2) = (\mathsf{emit}(n_{\mathsf{cost}} := ()); \mathsf{worker}(n-1))\ \mathsf{in}\ (n_1 + n_2, n_1) \\
&\quad \mathsf{let}\ \mathsf{fib} = \lambda n.\ \pi_2\ (\mathsf{worker}(n))
\end{aligned}$$

In order to keep track of the number of steps the program takes, we have inserted an emit before each recursive call. Using this instrumentation, we can, as shown in Section 4.4, use the Iron logic to verify that the number of recursive calls, *i.e.,* the number of emits, is indeed linear with respect to the input.

Note that using the displayed instrumentation we count the recursive calls, and not the number of smaller operations, like addition. It is worth pointing out that the Iron setup is flexible: by providing a different instrumentation, we can count the number of smaller operations and prove properties about those.

Operational semantics. The operational semantics is defined by means of small-step operational semantics; we show selected rules in Figure 2.1. We first comment on the overall format of the operational semantics, which is fairly standard, and then comment on the novel instrumentation aspects.

Head reduction:

$$(\mathsf{ref}(v), \sigma) \xrightarrow{\varepsilon;\varepsilon}_{\mathbf{h}} (\ell, \sigma[\ell \mapsto v], \varepsilon) \qquad \text{if } \ell \notin \mathrm{dom}(\sigma)$$

$$(\mathsf{fork}\ \{e\}, \sigma) \xrightarrow{\varepsilon;\varepsilon}_{\mathbf{h}} ((), \sigma, e)$$

$$(\mathsf{emit}(\eta := v), \sigma) \xrightarrow{\varepsilon;\{(\eta,v)\}}_{\mathbf{h}} ((), \sigma, \varepsilon)$$

$$(\mathsf{consume}(\eta := v), \sigma) \xrightarrow{\{(\eta,v)\};\varepsilon}_{\mathbf{h}} ((), \sigma, \varepsilon)$$

Threadpool reduction:

$$\frac{(e, \sigma) \xrightarrow{m;n}_{\mathbf{h}} (e', \sigma', \vec{e_f})}{(E[\,e\,], \sigma) \xrightarrow{m;n}_{\mathbf{t}} (E[\,e'\,]; \vec{e_f}, \sigma')} \qquad \frac{(e, \sigma) \xrightarrow{m;n}_{\mathbf{t}} (e', \sigma', \vec{e_f})}{(T_1; e; T_2, \sigma) \xrightarrow{m;n}_{\mathbf{tp}} (T_1; e'; T_2; \vec{e_f}, \sigma')}$$

$$\frac{(T, \sigma) \xrightarrow{m_1;n_1}_{\mathbf{tp}} (T', \sigma') \qquad (T', \sigma') \xrightarrow{m_2;n_2}\!\!\!\!\!\rightarrow_{\mathbf{tp}} (T'', \sigma'')}{(T, \sigma) \xrightarrow{m_1 \cdot m_2;n_1 \cdot n_2}\!\!\!\!\!\rightarrow_{\mathbf{tp}} (T'', \sigma'')}$$

Fig. 2.1. Selected rules of the instrumented operational semantics.

Since our programming language includes the ability to fork off new threads, the main reduction relation involves threadpools, which are simply sequences of expressions. The instrumented operational semantics involves reductions of the form $(T, \sigma) \xrightarrow{m;n}_{\mathbf{tp}} (T', \sigma')$, for the single step reduction, and $(T, \sigma) \xrightarrow{m;n}\!\!\!\!\!\rightarrow_{\mathbf{tp}} (T', \sigma')$, for the multi-step reduction of the threadpool. These express that the configuration consisting of a threadpool T, which is a sequence of expressions, and a heap σ steps to a new configuration consisting of a threadpool T' and a heap σ' and, moreover, that m has been consumed and n has been emitted. Here m and n are elements of the multimap monoid used for linear resources.

Threadpools are reduced by non-deterministically choosing a thread to reduce and then making a thread-local reduction. In Figure 2.1 we present selected rules for thread-local reductions: A thread-local head reduction is of the form $(e, \sigma) \xrightarrow{m;n}_{\mathbf{h}} (e', \sigma', \vec{e}_f)$, where $\vec{e}_f$ is a sequence of forked off new threads, see the reduction rule for $\mathsf{fork}\ \{e\}$. As usual, head reductions are extended to thread-local reduction $(e, \sigma) \xrightarrow{m;n}_{\mathbf{t}} (e', \sigma', \vec{e}_f)$ by the use of evaluation contexts E.

Having explained the overall structure of the operational semantics, we can finally point out the novel instrumentation. Observe that emit and consume evaluate to the unit value in a single step, and that they also annotate the operational semantics with the emitted and consumed resources. Moreover, observe that in a sequence of reductions, the emitted and consumed resources accumulate on either side rather than forcing them to match directly. This prevents a configuration from becoming stuck because of a consume or an emit, an important property since semantically they ought to behave as annotations. This is reflected in the definition of the multi-step threadpool reduction $(T, \sigma) \xrightarrow{m;n}\!\!\!\!\!\rightarrow_{\mathbf{tp}} (T', \sigma')$ shown in Figure 2.1: it is essentially the transitive closure of a single step reduction, however it additionally accumulates all the consumed and emitted resources.

3 The Iron program logic

The setup of Iron is similar to that of Iris: at the core of Iron there is a small, resourceful *base logic*, which is a higher-order logic extended with the basic connectives of BI (separating conjunction and magic wand), predicates for resource ownership, and a handful of simple modalities. Similar to Iris, the Iron base logic does not bake in any propositions about programs as primitive, and it is only this base logic whose soundness must be proven directly against an underlying semantic model (see Section 5.1). On top of this base logic, we derive the fancier mechanisms of Iron, such as impredicative invariants (Section 5.2) and Hoare triples, both for partial program correctness (Section 5.3) and total program correctness (Section 5.4).

In this section, we focus on the important differences between the Iron and Iris base- and program logic. For clarity's sake, we make a distinction between primitive rules (rules whose soundness is proven in the model of the Iron base logic) and derived rules (rules that follow from the primitive rules of the Iron base logic), but we defer discussion about the Iron model and the encoding of the Iron program logic constructs in terms of the Iron base logic to Section 5.

The new rules of Iron are displayed in Figure 3.1. For reasons of space, we do not include a detailed description of existing Iris connectives (an extensive description can be found in [21]), but just focus on the new connectives:

- The Emp proposition, which asserts ownership of *no* resources. It is a unit for separating conjunction, as expressed by Emp-unit. Note that in Iron, Emp is distinct from True, and as such, True is not a unit for $*$.
- The connective $\mathsf{OwnL}(m)$ for ownership of linear resources allows one to keep track of the resource annotations in the programming language in the logic. Here, m ranges over the user-chosen commutative monoid for resource obligations. Similar to Iris's connective for ownership of affine resources, OwnL-op expresses that $*$ internalizes the monoid operation on linear resources.

In the subsequent paragraphs we describe how these connectives are used for reasoning in Iron.

Affine propositions. The *affine modality* $\blacksquare P$, which is defined as:

$$\blacksquare P \triangleq P \wedge \mathsf{Emp}.$$

expresses ownership of P in the absence of any *linear* resources. We call a proposition *affine* whenever $P \vdash \blacksquare P$, that is, whenever P provably holds no linear resources. This modality makes it possible to retain the full reasoning power of Iris in Iron:

- As captured by the rule Affine-weakening, affine propositions enjoy weakening, *i.e.,* can be thrown, and thus behave like ordinary Iris propositions.
- The basic Iris connectives are affine. See for example the rules Affine-ghost and Affine-inv, which show that the Iris connectives for ghost ownership $\boxed{m}^{\gamma}$ and impredicative invariants $\boxed{P}$ are affine.

Primitive rules

EMP-UNIT
$P * \mathsf{Emp} \dashv\vdash P$

OWNL-OP
$\mathsf{OwnL}(m) * \mathsf{OwnL}(n) \dashv\vdash \mathsf{OwnL}(m \cdot n)$

OWNL-EMP
$\mathsf{OwnL}(\varepsilon) \dashv\vdash \mathsf{Emp}$

Derived rules

Rules of the affine modality:

AFFINE-ELIM
$\blacksquare P \vdash P$

AFFINE-IDEMP
$\blacksquare P \vdash \blacksquare\blacksquare P$

AFFINE-TRUE
$\blacksquare \mathsf{True} \dashv\vdash \mathsf{Emp}$

AFFINE-EMP
$\blacksquare P \vdash \mathsf{Emp}$

AFFINE-CONJ
$\blacksquare (P \wedge Q) \dashv\vdash \blacksquare P \wedge \blacksquare Q$

AFFINE-DISJ
$\blacksquare (P \vee Q) \dashv\vdash \blacksquare P \vee \blacksquare Q$

AFFINE-WEAKENING
$\blacksquare P * Q \vdash Q$

Rules of the combined affine/persistent modality:

COMB-ELIM
$\blacklozenge P \vdash P$

COMB-IDEMP
$\blacklozenge P \vdash \blacklozenge\blacklozenge P$

COMB-AFFINE
$\blacklozenge P \vdash \blacksquare \blacklozenge P$

COMB-DUPLICABLE
$\blacklozenge P \dashv\vdash \blacklozenge P * \blacklozenge P$

Iris constructs are affine:

AFFINE-INV
$\boxed{P} \vdash \blacksquare \boxed{P}$

AFFINE-GHOST
$\boxed{a}^{\gamma} \vdash \blacksquare \boxed{a}^{\gamma}$

Rules for invariants:

INV-ALLOC
$$\frac{\boxed{R} \vdash \{P\}\, e\, \{v.\, Q\}}{\mathsf{Emp} \vdash \{\blacksquare \triangleright R * P\}\, e\, \{v.\, Q\}}$$

INV-OPEN
$$\frac{\boxed{R} \vdash \{P * \blacksquare \triangleright R\}\, e\, \{v.\, Q * \blacksquare \triangleright R\} \qquad \mathsf{atomic(e)}}{\boxed{R} \vdash \{P\}\, e\, \{v.\, Q\}}$$

Selected rules for Hoare triples:

HT-EMIT
$\{\mathsf{Emp}\}\, \mathsf{emit}(\eta := w)\, \{v.\, \blacksquare (v = ()) * \boxed{\{w\}}^{\eta}\}$

HT-CONSUME
$\{\boxed{\{w\}}^{\eta}\}\, \mathsf{consume}(\eta := w)\, \{v.\, \blacksquare (v = ())\}$

HT-LOAD
$\{\ell \hookrightarrow v\}\ !\ell\, \{w.\, \blacksquare (w = v) * \ell \hookrightarrow v\}$

HT-STORE
$\{\ell \hookrightarrow \_\}\, \ell \leftarrow v\, \{w.\, \blacksquare (w = ()) * \ell \hookrightarrow u\}$

HT-ALLOC
$\{\mathsf{Emp}\}\, \mathsf{ref}(v)\, \{w.\, \exists \ell.\, \blacksquare (w = \ell) * \ell \hookrightarrow v\}$

HT-FREE
$\{\ell \hookrightarrow \_\}\, \mathsf{free}(\ell)\, \{v.\, \blacksquare (v = ())\}$

HT-FORK
$$\frac{\{\mathsf{Emp}\}\, e\, \{\mathsf{Emp}\}}{\{\mathsf{Emp}\}\, \mathsf{fork}\ \{e\}\ \{v.\, \blacksquare (v = ())\}}$$

Fig. 3.1. Selected rules of Iron.

– Affinity is preserved by the various connectives of the logic, as a consequence of the remaining structural rules.

We often wish to express that a proposition does not own *any* resources, *i.e.,* it owns neither linear nor affine resources. This is done using the "combined" *affine and always modality*, which is defined as:

$$\blacklozenge P \triangleq \blacksquare \square P$$

Here, $\square$ is a primitive modality of the Iron base logic, which is mostly a technical device used in the definition of $\blacklozenge$, and is not important for understanding the rest of the paper. For this reason we only discuss it in Section 5. The $\blacklozenge$ behaves very similarly to the always modality of Iris, in the sense that it makes predicates duplicable and affine, see the rules Comb-duplicable and Comb-affine. The rest of the rules are basic structural properties of the modality.

Linear ownership. Linear ownership is manifested by the *linear ownership assertion* $\mathsf{OwnL}(m)$. Here, m is a member of our chosen commutative monoid, which, for the language in Section 2, is a multimap of identifiers to values.

It is common to refer to linear ownership of a multiset of values all indexed by the same key η, this is denoted $\{v_1, \ldots, v_n\}^{\eta}$, instead of the lengthier $\mathsf{OwnL}(\{(\eta, v_1), \ldots, (\eta, v_n)\})$. This simulates a relationship much like the one between Iris's ownership $\mathsf{Own}(a)$ of the global ghost state and ownership $\lceil b \rceil^{\gamma}$ of a single ghost location γ.

A crucial different between the connective for affine ownership $\lceil b \rceil^{\gamma}$, and the one for linear ownership $\mathsf{OwnL}(m)$, is that the latter cannot be weakened, *i.e.,* it does *not* enjoy $P * \mathsf{OwnL}(m) \vdash P$ for arbitrary propositions P.

Impredicative invariants. Iron supports Iris-style *thread local* reasoning about higher-order programming languages with dynamically allocated threads. To this end, both logics include a powerful notion of *invariant* assertion[5] $\boxed{P}$. Invariants are used to share resources between threads (Inv-open) and they can be created at arbitrary points during proofs (Inv-alloc).

Invariants become more complicated in the presence of linear resources, as, in principle, arbitrary resources can be transferred into invariants. Since invariants can be duplicated (because the invariant assertion is persistent, *i.e.,* $\boxed{P} \vdash \square \boxed{P}$), ownership of the resources in an invariant becomes out of the control of a single thread. As such, it becomes impossible to guarantee that no resources were leaked via the use of invariants.

[5] Technically, every invariant has a name ι, which appears in the invariant assertion $\boxed{P}^{\iota}$. These names 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. In practise, this results in some additional bookkeeping when opening invariants. As this bookkeeping is orthogonal to our focus, we omit invariant names in this paper, and refer to [21] for the details.

To fix this issue, Iron forbids one to transfer linear resources, *e.g.,* resources containing $\mathsf{OwnL}(m)$ or Emp, into invariants. This is formally achieved by incorporating the $\blacksquare$ modality in the rules for invariants. The rule Inv-alloc for allocation of invariants is exactly as in Iris, except for the presence of the $\blacksquare$ modality, which makes sure only affine resources can be transferred into invariants. Similarly, the rule Inv-open is exactly as in Iris, except that the invariant R is obtained and has to be restored under the $\blacksquare$ modality.

Rules for Hoare triples. The rules for Hoare triples are mostly the same as in Iris, but there are some additional rules that we will describe. The rules Ht-emit and Ht-consume formalize the connection between the emit and consume operations and linear ownership. Furthermore, note the use of the affine modality in the post-conditions. These modalities are present because in Iron pure facts such as equality may hold arbitrarily many resources. Thus without the affine modality in the post-condition we could hide arbitrary linear resources into this assertion, which would weaken Theorem 1.

Adequacy. We now state the adequacy theorem, which expresses the connection between a specification of a program in Iron and its operational behavior.

Theorem 1 (Adequacy). *Let e be a closed program, ϕ a first-order assertion, σ a heap, and m a linear resource. Suppose $\mathsf{Emp} \vdash \{\mathsf{Emp}\}\, e\, \{v. \blacksquare \phi(v) * \mathsf{OwnL}(m)\}$ and suppose $(e, \sigma) \xrightarrow{m_1; m_2}{\!\!\!\!\twoheadrightarrow}_{\mathbf{tp}} ((e_1; e_2; \ldots; e_n), \sigma')$. Then:*

- *for all i, either e_i is a value or (e_i, σ') can step further, i.e., it is not stuck,*
- *if e_1 is a value v, then $\phi(v)$ holds,*
- *if* all of *e_i are values, then $m_2 = m_1 \cdot m$, i.e., m is the difference between emitted and consumed linear resources.*

Our Iron adequacy theorem is stronger than the corresponding theorem of the original Iris logic, because of the connection between the $\mathsf{OwnL}(m)$ predicate in the post-condition and its correspondence to the annotations in the operational semantics, see the last item in the list of consequences in the theorem. The proof of this theorem relies on a novel interpretation of Hoare triples, which we explain in Section 5. Finally note the use of the affine modality in the post-condition of $\{\mathsf{Emp}\}\, e\, \{v. \blacksquare \phi(v) * \mathsf{OwnL}(m)\}$. It is essential for the validity of the theorem, since we need to know that $\mathsf{OwnL}(m)$ are *all* the linear resources in the post-condition. The affine modality guarantees this since it states precisely that $\blacksquare \phi(v)$ contains no linear resources.

The proof of this theorem relies on the fact that emit and consume actually instrument the operational semantics. If we instead had used linear predicates to reason about Iris-like ghost state, then we would not have been able to draw any formal conclusions about the operational semantics.

In the following section we exemplify how this theorem can be used to conclude that resources are properly accounted for.

4 Examples

The main motivation behind Iron is to be able to prove more fine-grained, intensional properties about the handling of resources. In order to demonstrate that this is indeed possible, we present several examples of Iron being used for exactly this purpose.

4.1 A linear specification for locks

We first outline how a spin-lock implementation shown in Section 2 can be shown to satisfy the lock specification from the introduction. Recall that our Iron specification for locks is intended to force the lock client to call release after each call to acquire. Further remember that the specification says that there exist predicates islock and locked such that the following holds (where free variables are implicitly universally quantified):

$$\begin{aligned}
\{\blacksquare P\}\ \mathsf{newLock}()\ \{v.\ \exists\gamma.\ \mathsf{islock}(v,P,\gamma)\}\\
\{\mathsf{islock}(v,P,\gamma)\}\ \mathsf{acquire}\, v\ \{\blacksquare P * \mathsf{locked}(v,\gamma)\}\\
\{\mathsf{islock}(v,P,\gamma) * \blacksquare P * \mathsf{locked}(v,\gamma)\}\ \mathsf{release}\, v\ \{\mathsf{Emp}\}\\
\mathsf{islock}(v,P,\gamma) \Rightarrow \mathrel{\rotatebox{180}{$\blacktriangleright$}}\mathsf{islock}(v,P,\gamma)
\end{aligned}$$

The key point here is that the $\mathsf{locked}(v,\gamma)$ predicate is linear.

We define the locked and islock predicates as follows:

$$\mathsf{locked}(v,\gamma) \triangleq \boxed{K}^{\gamma} * \boxed{\{v\}}^{n_{\mathsf{lock}}}$$

$$\mathsf{islock}(v,P,\gamma) \triangleq \boxed{\exists \ell, b.\ \blacksquare (v = \ell) * \ell \hookrightarrow b * \left(\left(b = \mathsf{false} * \blacksquare P * \boxed{K}^{\gamma}\right) \vee b = \mathsf{true}\right)}$$

The predicate $\mathsf{locked}(v,\gamma)$ includes (affine) ghost state $\boxed{K}^{\gamma}$. The idea is that K is a token, which can be thought of as a "key" to the lock; it is used to ensure that only the thread that has acquired the lock can release it. The K token is defined using existing Iris monoid constructions so that it satisfies:

$$\mathsf{Emp} \vdash \exists\gamma.\boxed{K}^{\gamma} \qquad\qquad \boxed{K}^{\gamma} * \boxed{K}^{\gamma} \vdash \mathsf{False}$$

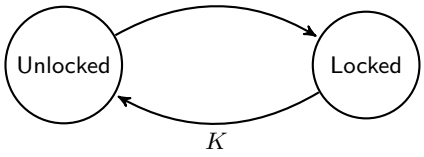
The first property is used to "allocate" a key for each lock. The second ensures that ownership of the key is unique, as such only the client which has acquired the lock can release it. The $\mathsf{locked}(v,\gamma)$ predicate also includes a linear predicate $\boxed{\{v\}}^{n_{\mathsf{lock}}}$ to track the linear resource emitted in acquire.

There is some redundancy between $\boxed{\{v\}}^{n_{\mathsf{lock}}}$, which a thread possesses in the critical section, and $\boxed{K}^{\gamma}$, which a thread possesses when it has the right to release a lock. On first glance, they serve much the same purpose, both are produced by calling acquire and are needed in order to call release. They conceptually, however, serve different roles. The $\boxed{\{v\}}^{n_{\mathsf{lock}}}$ exists to force a thread to call release. It does not interact with the invariant of the lock or any part of the lock structure; it

merely represents the obligation to dispose of an acquired lock. On the other hand, $\lceil \bar{K} \rceil^{\gamma}$ is necessary to ensure that release can be called correctly. It forces the invariant to be the correct shape and is needed to reestablish the invariant.

It is natural to wonder whether these two distinct roles might be unified to decrease the complexity of the specification of locks. This is currently impossible since, as was explained earlier, only affine propositions may be stored in invariants. This means that we cannot store $\{v\}^{n_{\mathsf{lock}}}$ in an invariant, and, since the operational semantics can only be instrumented to produce linear resources we cannot replace $\{v\}^{n_{\mathsf{lock}}}$ with $\lceil \bar{K} \rceil^{\gamma}$.

The islock(v, P, γ) predicate is defined as an invariant; it represents knowledge of the lock that can be shared between threads. The invariant expresses that the lock can be in one of two states, as shown in the following state transition system:



The Locked state corresponds to the second disjunct in the invariant, where the reference used for the lock holds the value true. The Unlocked state corresponds to the first disjunct in the invariant, where the reference used for the lock holds the value false, and the lock module owns the key, and the resource invariant P. Note that we make use of the affine modality on P; that is necessary since, as explained earlier, we can only use affine predicates in invariants.

Theorem 2. *The spin-lock satisfies the lock specification.*

The proof of the Hoare triples is quite similar to the proof in Iris of a corresponding affine lock specification; the only difference is that $\{v\}^{n_{\mathsf{lock}}}$ is propagated as a post-condition for acquire and as a precondition for release. In particular, the linear nature of Iron is not an impediment to using a proof originally constructed in Iris because most of the propositions under consideration are affine. This means that normal reasoning steps, including dropping unused assumptions, may be used. But, of course, one has to be careful—that is why we have formalized the proof in Coq, see Section 6.

Lastly, using the rules of Iron we can show that islock(v, P, γ) is affine and persistent as required for the lock specification.

4.2 Mixing disposable and non-disposable resources

In the ML-like language used in this paper, allocated memory can be disposed of with free. There is a natural question when free is in the language however: is every allocated resource freed? Using linearity, this question becomes tractable. We simply ensure that the proposition representing a disposable location is linear and that the only way to use such a proposition is the rule for free. In this case, the logic itself will ensure that all allocated resources are accounted for. It

would be desirable, though, if not all resources needed to be subjected to this bookkeeping. Programmers might want to exercise fine control over large chunks of allocated memory but small pieces are not worth the trouble.

In Iron we can accommodate both of these goals. The idea is to instrument the operational semantics so that allocation produces a linear proposition which represents the obligation to dispose of it. In order to ensure that ref and free both make suitable use of linear resources, we define a simple wrapper:

$$\begin{aligned}\mathsf{let}\,\mathsf{lref} &= \lambda x.\,\mathsf{let}\,\ell = \mathsf{ref}(x)\,\mathsf{in}\,\mathsf{emit}(n_{\mathsf{alloc}} := \ell);\, l\\ \mathsf{let}\,\mathsf{lfree} &= \lambda \ell.\,\mathsf{consume}(n_{\mathsf{alloc}} := \ell);\,\mathsf{free}(\ell)\end{aligned}$$

To distinguish these emits and consumes from the ones used to instrument locks we use a different tag: n_{alloc}.

By instrumenting these operations with emit and consume we make it so that every allocated resource poses an obligation to dispose of it. Furthermore, if we avoid placing a $\mathsf{consume}(n_{\mathsf{alloc}} := -)$ elsewhere in the program, the only way that this can be done is to call lfree. Since the normal ref and free are available, this does not prevent the use of normal, affine locations. In fact, it allows the fluid mixing of both linear and affine locations together since the linear resources are being used to represent the obligation to dispose of a location rather than baking it into the $\ell \hookrightarrow v$ connective itself.

These derived forms can be specified in much the same way that ref and free were originally specified, the crucial difference is that they now *also* contain a linear resource, apart from just the (affine) point-to predicate:

$$\textsc{Ht-lalloc}\quad \{\mathsf{Emp}\}\ \mathsf{lref}(v)\left\{w.\,\exists \ell.\,\blacksquare\,(w = \ell) * \ell \hookrightarrow v * \boxed{\{\ell\}}^{n_{\mathsf{alloc}}}\right\}$$

$$\textsc{Ht-lfree}\quad \left\{\ell \hookrightarrow v * \boxed{\{\ell\}}^{n_{\mathsf{alloc}}}\right\}\ \mathsf{lfree}(\ell)\ \{w.\,\blacksquare\,(w = ())\}$$

Notably, the rules for accessing and setting references do not change to include linear resources in any way. This means that disposability is an entirely orthogonal issue to how a reference might be used; whether a resource is allocated using lref or just the unwrapped ref makes no difference.

Finally, it is provable that the linearity of propositions does in fact imply the linearity of usage in the language.

Theorem 3. *If* $\{\mathit{Emp}\}\,e\,\{\mathit{Emp}\}$ *holds and the only occurrences of* $\mathit{emit}(n_{\mathsf{alloc}} := -)$ *and* $\mathit{consume}(n_{\mathsf{alloc}} := -)$ *are in* lref *and* lfree *respectively, then every application of* lref *can be uniquely paired with an application of* lfree *in every execution of* e *to a value.*

4.3 A disposable bag

We now use the lock and the disposable references to define a complex concurrent data structure. Recall the concurrent bag defined in the introduction. We will

adapt it to make use of disposable references (Section 4.2) and linear locks (Section 4.1). By making use of linear locks, we rule out the incorrect implementation as shown in the example incorrect_insert illustrated in the introduction.

The implementation of the disposable bag is almost identical in implementation of the code presented in Figure 1.1. The only difference is that ref in newBag is replaced by lref so that we may take advantage of linear locations.

A tricky aspects of disposable bags is the specifying and proving the correctness of the dispose operations. Whereas the bag can be shared by arbitrarily many threads, it may only be disposed when there are no threads operating on the bag anymore. In order to account for the fact that no threads operate on the bag anymore, we use fractional permissions.

Fractional permissions $\lceil\overline{p}\rfloor^{\gamma}$, where p is a rational number $0 < p \leq 1$, can be modeled using (affine) ghost state in Iris, and enjoy the following rules:

$$\textsc{Frac-Split}\quad \lceil\overline{p+q}\rfloor^{\gamma} \vdash \lceil\overline{p}\rfloor^{\gamma} * \lceil\overline{q}\rfloor^{\gamma} \qquad \textsc{Frac-Cap}\quad \lceil\overline{1}\rfloor^{\gamma} * \lceil\overline{p}\rfloor^{\gamma} \vdash \mathsf{False}$$

We will use a fractional permission $\lceil\overline{p}\rfloor^{\gamma}$ to represent ownership of a fraction p of the bag. When allocating a bag, one obtains the full fractional permission $\lceil\overline{1}\rfloor^{\gamma}$. In order to insert or remove elements of the bag, one only needs a fragment of the fractional permission, *i.e.,* $\lceil\overline{p}\rfloor^{\gamma}$ for some $0 < p \leq 1$. Using the rule Frac-Split, one can obtain as many fragments as the number of threads that need to operate on the bag. Disposal of the bag is only possible if no threads are inserting or removing elements anymore. As such, it will require surrendering ownership of $\lceil\overline{1}\rfloor^{\gamma}$, *i.e.,* ownership of the full fractional permission.

This approach originates in Boyland [4] where permissions are used to distinguish when a reference may be read from or written to. Full permission is needed to write. Reading, on the other hand, requires only ownership of a fraction of the permission because two threads reading a reference concurrently causes no issues. Our approach is identical except instead of preventing conflicting writes we prevent the bag from being disposed while it is still in use.

Recall from the introduction that the bag specification makes use of some proposition $\mathsf{isBag}(b, \Phi, \gamma)$. This proposition indicates that b is a bag named by γ which is comprised of elements that satisfy Φ. Additionally, we define a proposition, $\mathsf{bagOb}(b)$ which represents the obligation to dispose of a bag. The new specification is then:

$$\begin{aligned}
&\{\mathsf{Emp}\}\ \mathsf{newBag}()\ \{b.\, \exists\gamma.\ \mathsf{bagOb}(b) * \mathsf{isBag}(b, \Phi, \gamma) * \lceil\overline{1}\rfloor^{\gamma}\}\\
&\{\lceil\overline{p}\rfloor^{\gamma} * \mathsf{isBag}(b, \Phi, \gamma) * \blacksquare\, \Phi(u)\}\ \mathsf{insert}\, b\, u\ \{\lceil\overline{p}\rfloor^{\gamma}\}\\
&\{\mathsf{isBag}(b, \Phi, \gamma) * \lceil\overline{p}\rfloor^{\gamma}\}\ \mathsf{remove}\, b\ \{v.\, (v = \mathsf{None} \vee \exists w.\, v = \mathsf{Some}\, w * \blacksquare\, \Phi(w)) * \lceil\overline{p}\rfloor^{\gamma}\}\\
&\{\mathsf{bagOb}(b) * \mathsf{isBag}(b, \Phi, \gamma) * \lceil\overline{1}\rfloor^{\gamma}\}\ \mathsf{freeBag}\, b\ \{\mathsf{Emp}\}\\
&\mathsf{isBag}(b, \Phi, \gamma) \Rightarrow \mathsf{isBag}(b, \Phi, \gamma)
\end{aligned}$$

The central difference is that isBag now produces an obligation to dispose of the bag, $\mathsf{bagOb}(b)$. This is a result of switching from ref to lref in the implementation. These new specifications also thread a fractional permission to use the bag

throughout the triples. It is important that the fractional permission is threaded through the specification rather than merely demanded as a precondition. After all, eventually a client will wish to combine all of these fractional permissions into the full fractional permission, $\boxed{1}^{\gamma}$, so that the bag can be disposed of using the freeBag operation.

As discussed previously, disposing of the bag requires full ownership of the bag. The specification also requires the linear resource representing the obligation to dispose of the bag, $\mathsf{bagOb}(b)$. It produces nothing, unlike the other operations we do not thread the partial ownership of the bag through the specification because this ownership represents the right to dispose the bag. Once the bag is disposed once no thread may claim any right to dispose it again. This operation can be implemented as:

$$\mathsf{let}\ \mathsf{freeBag} = \lambda(\ell, \mathit{lock}).\ \mathsf{acquire}\ \mathit{lock};\ \ \mathsf{lfree}(\ell);\ \ \mathsf{release}\ \mathit{lock}$$

A client of this bag would allocate the bag in a main thread. This thread would maintain ownership of the obligation to dispose of the bag, $\mathsf{bagOb}(b)$. As different threads would seek to make use of this bag, the main thread would separate the ownership of $\boxed{1}^{\gamma}$ into many fractional permissions. One would go to each worker thread and the worker thread would return this permission upon completion of its job. Finally, the main thread would collect up all of these fractured permissions and recreate $\boxed{1}^{\gamma}$ and, using $\mathsf{bagOb}(b)$, dispose of the bag.

It remains to define isBag and $\mathsf{bagOb}(b)$. We start with $\mathsf{bagOb}(b)$, this should be the linear obligation to dispose of b. In particular, this will be $\{\ell\}^{n_{\mathsf{alloc}}}$ where ℓ is the reference to the container because that is what is emitted by $\mathsf{lref}()$ in newBag. Since this reference is stored in the first component of the tuple we know b must be $\mathsf{bagOb}(b)$ can be defined as:

$$\mathsf{bagOb}(b) \triangleq \exists \ell, v.\ \blacksquare\,(b = (\ell, v)) * \{\ell\}^{n_{\mathsf{alloc}}}$$

Next we define $\mathsf{isBag}(b, \Phi, \gamma)$. If isBag did not need to account for the possibility of disposal, we might define it as follows:

$$\mathsf{isBag}(b, \Phi, \gamma) \triangleq \exists \ell, v, \gamma'.\ \blacksquare\,(b = (\ell, v)) * \mathsf{islock}(v, \exists w.\ \ell \hookrightarrow w * \mathsf{bagList}(\Phi, w), \gamma')$$

Here, $\mathsf{bagList}(\Phi, b)$ is used to ensure that b is a list of values each of which satisfy Φ. It can be specified internally using the feature of Iris to form guarded fixed points, for instance:

$$\mathsf{bagList}(\Phi, v) \triangleq (v = \mathsf{inj}_1\ ()) \vee \exists h, t.\ \blacksquare\,(v = \mathsf{inj}_2\ (h, t)) * \Phi(h) * \triangleright\, \mathsf{bagList}(\Phi, t)$$

Using this as our resource guarded by the lock would be sufficient to prove the specifications for most of the bag operations, but it will not allow us to specify freeBag because eventually it will be required to surrender $\ell \hookrightarrow w$ to lfree. In order to resolve this, a case is added to isBag for the case where the bag has been disposed of:

$$\begin{aligned}\mathsf{isBag}(b, \Phi, \gamma) \triangleq \exists \gamma', \ell, v.\ \blacksquare\,(b = (\ell, v)) *\\ \mathsf{islock}\left(v, \boxed{1}^{\gamma} \vee (\exists w.\ \ell \hookrightarrow w * \mathsf{bagList}(\Phi, w)), \gamma'\right)\end{aligned}$$

Importantly, because isBag is still just an equality together with islock, it is still possible to demonstrate that it is both affine and persistent:

$$\mathsf{isBag}(b, \Phi, \gamma) \vdash \blacksquare\, \mathsf{isBag}(b, \Phi, \gamma)$$

This is required for our specification since $\mathsf{isBag}(b, \Phi, \gamma)$ will be shared between all the thread making use of the bag.

By guarding this proposition in the lock, every time we acquire a lock we will receive either $\boxed{1}^{\gamma}$ or the actual information stored by the bag. However, each of the specifications for operations on the bag require some ownership of the bag, $\boxed{p}^{\gamma}$. By rule Frac-Cap this means that the left branch of this disjunction implies a contradiction. Therefore, each of this operations may work as though the resource guarded by the lock is still just $\exists w.\, \ell \hookrightarrow w * \mathsf{bagList}(\Phi, w)$. This means that the proofs for each operation may proceed essentially as if the bag will never be disposed.

The new operation, freeBag, will surrender its ownership of $\boxed{1}^{\gamma}$ to prove the left branch of the disjunction. In exchange for doing this it is able to keep $\ell \hookrightarrow w$ outside of the critical region and is thus able to surrender it to lfree using Ht-lfree and free the bag.

Theorem 4. *The bag implementation satisfies its specification.*

Being able to specify correct programs is only part of the goal of more precise program specifications. Equally important is the ability to demonstrate that incorrect implementations are ruled out by these finer specifications. It is provable in the logic that the specifications do *not* hold for certain incorrect ones. In particular, a version of $\mathsf{insert}(v)$, which does not call release cannot be assigned the desired specification.

Theorem 5. *The following proposition is not derivable:*

$$\left\{\boxed{p}^{\gamma} * \mathsf{isBag}(b, \Phi, \gamma) * \Phi(u)\right\}\ \mathsf{incorrect\_insert}\, b\, u\, \left\{\boxed{p}^{\gamma}\right\}$$

4.4 Cost semantics

It is well known that separation logic can be used to reason about the complexity of programs [1,2,5,6,7,8]. Traditionally, in these separation logics, cost is encoded in the logic by verifying triples of the form:

$$\{\mathsf{ticket}(m)\}\, e\, \{v.\, Q\}$$

Here, $\mathsf{ticket}(m)$ is a logical connective, which represents the right to take m steps. During the course of the proof of the triple it would be necessary to expend some of the tickets and at the end we will have upper-bounded the cost.

As an alternative, though, since Iron is a linear logic it is possible to accumulate ticket propositions that would be produced by the verification of the triple. In Iron it is natural to represent $\mathsf{ticket}(m)$ as $\{(), \ldots, ()\}^{n_{\mathsf{cost}}}$, a multiset

tagged with n_{cost} consisting of m copies of $()$. For ease of notation, we use the convention of denoting such a multiset by m.

All that is needed to ensure that we track cost correctly is to instrument our programs with emits so that each step that should be counted as expensive is annotated with a single $\mathsf{emit}(n_{\mathsf{cost}} := 1)$ operation. In our example in the introduction with worker and fib, cost at the granularity of "number of function calls" was considered. That is why every recursive call to worker was paired with an $\mathsf{emit}(n_{\mathsf{cost}} := 1)$. Thus, a program which costs n would have the trace:

$$(e_1, \sigma) \xrightarrow{\varepsilon; m}\!\!\!\!\rightarrow_{\mathbf{tp}} ((v_1, \ldots, v_n), \sigma')$$

Where m is a multimap containing exactly n occurrences of $(n_{\mathsf{cost}}, ())$.

Our adequacy statement then gives us a correspondence between such traces and specifications of the form $\{\mathsf{Emp}\}\, e \left\{ m \overset{n_{\mathsf{cost}}}{} \right\}$, *i.e.,* we can reason about the cost m in the logic.

This shows that Iron is capable of encoding cost semantics using emit and consume operations: cost is simply a mode of use of our instrumented semantics. Note that many other natural notions of cost, such as memory usage, may be recorded in a similar style.

When using Hoare triples for partial program correctness, the Iron adequacy statement gives a slightly odd specification to triples like $\{\mathsf{Emp}\}\, e \left\{ m \overset{n_{\mathsf{cost}}}{} \right\}$. It states that *if* a program terminates then it does so in a particular number of steps. If the program diverges, then there are no bounds placed on its time, memory usage, or whatever choice of instrumentation has used.

Fortunately, the Iron setup, which is adapted from Iris 3.0, is general enough to remedy this issue: Hoare triples for partial program correctness are a defined notion, and as such, it is possible to define Hoare triples for total correctness in a similar way as a defined notion. We will show the definition in Section 5.

For now, let us write the new total correctness version of Hoare triples as $[P]\, e\, [v.\, Q]$. As expected they enjoy a stronger adequacy statement than partial correctness triples.

Theorem 6. *For any assertions P and Q and program e we have the entailment $[P]\, e\, [v.\, Q] \vdash \{P\}\, e\, \{v.\, Q\}$. Moreover, if $\mathsf{Emp} \vdash [\mathsf{Emp}]\, e\, [v.\, \mathsf{True}]$ then for any physical state σ, all reductions paths from (e, σ) are terminating, i.e., there is no infinite execution path.*

The first part of the theorem implies that given a first-order assertion ϕ for which holds $\mathsf{Emp} \vdash [\mathsf{Emp}]\, e\, [v.\, \mathsf{OwnL}(m) * \blacksquare \phi(v)]$, then we can use the adequacy for the ordinary Hoare triples (Theorem 1) to conclude that e does not get stuck and in all its executions to a value the difference between emitted and consumed resources is m. Additionally, Theorem 6 tells us that our cost semantics correspond directly to the number of steps taken.

Occasionally one wishes to only account for less precise bounds on cost, upper-bounds only for instance. This is the only method of specification available in an affine setting where $\mathsf{ticket}(n)$ may always be dropped. It is recoverable in

Iron with a predicate like:

$$\mathsf{upto}(n) \triangleq \exists c.\, \blacksquare\, (c \leq n) * c^{\,n_{\mathsf{cost}}}$$

In a similar vein, asymptotic analysis can be encoded using this notion of cost. To express that a function f takes a linear number of steps in its input a possible specification is:

$$\exists c.\, \forall n.\, [\mathsf{Emp}]\; f(n)\; [v.\; \Phi(v) * \mathsf{upto}(c \cdot n)]$$

All of these are results of the fact that by starting out with a more precise accounting of cost, it is easy to relax restrictions. By comparison, if one starts with only asymptotic bounds then it is difficult to recover more fine-grained bounds on cost.

There is a natural limitation to our approach, which is consistent with prior work on cost semantics in separation logic [1,2,7,8]: specifying cost is limited to sequential programs. This is because the forked off threads have the post-condition Emp (see HT-Fork) This means that no forked off threads may produce linear resources. In this scheme, that means that the cost of forked of threads must be zero. As a result programs which can be specified are essentially the sequential ones.

Recall the example of the fib function from Section 2. It is clear that each recursive call corresponds exactly to one $\mathsf{emit}(n_{\mathsf{cost}} := 1)$, and thus we can state and prove in the logic the time complexity of this program, as well as its functional correctness. The specification we wish to show is that:

$$\forall n.\, \{\mathsf{Emp}\}\; \mathsf{fib}(n)\; \left\{v.\, v \text{ is the } n\text{th Fibonacci number} * n^{\,n_{\mathsf{cost}}}\right\}$$

In order to prove this, the following specification for worker is needed:

$$\forall n.\, \{\mathsf{Emp}\}\; \mathsf{worker}(n)\; \left\{v.\, v \text{ is the } n+1\text{th and } n\text{th Fibonacci numbers} * n^{\,n_{\mathsf{cost}}}\right\}$$

The essence of these specifications is that the post-condition captures the number of resources generated, which is directly tied to the number of recursive calls made. These specifications are straightforward to show.

Finally as a corollary of the adequacy theorem we have:

Theorem 7. *Any execution of* $\mathsf{fib}(n)$ *makes at most* n *calls to* worker.

5 Semantics of Iron

In this section we show that the Iron logic is sound by constructing a model. We follow the recent approach of Iris 3.0 by having a minimalistic base logic with only a few connectives and rules—the base logic can then be used to define the high-level features needed for program proving. As explained in Section 3, the base logic of Iron is the base logic of Iris, extended with the linear ownership assertion $\mathsf{OwnL}(m)$ satisfying rules in Figure 3.1. For reasons of space we cannot explain the model of Iris in detail. We briefly recall it, but for understanding it

in detail the reader should consult previous work [24,21]. In this paper we focus on pointing out and explaining the changes needed to model Iron.

All results in this section, apart from the total weakest precondition, have been formalized in Coq; see Section 6.

5.1 The semantics of the base logic

The model of the Iron base logic is based closely on the model of Iris 3.0 base logic [24], which we now briefly recall.

The Iris model is based on a notion of a *camera*, which is a partial commutative monoid like structure, enriched with step-indexing. Elements of a camera are called resources, and in particular they can be composed using an associative and commutative operation, which we write as $x \cdot y$. There is a decreasing family $\mathcal{V}$ of subsets of resources $\mathcal{V}_0 \supseteq \mathcal{V}_1 \supseteq \cdots$, which are thought of as resources valid for n "steps". The steps are related to the operational steps of the program in the definition of the weakest precondition. An Iris assertion P is modeled as a decreasing family of subsets of resources $[\![P]\!]_0 \supseteq [\![P]\!]_1 \supseteq \cdots$ satisfying a couple of closure conditions, the most important of which is that, for any n, if $a \in [\![P]\!]_n$ and $a \cdot b \in \mathcal{V}_n$ then $a \cdot b \in [\![P]\!]_n$. That is, $[\![P]\!]_n$ is upwards closed with respect to the extension order of the resources. This exact closure property is the reason why Iris validates the rule $P * Q \vdash P$, for any assertions P and Q.

The camera used to define the model of Iris is typically a product of many different cameras, each component corresponding to different ghost state used in different verifications. It is defined as a solution to a recursive domain equation, to model impredicative invariants [24]. In the following, $\mathcal{M}$ will denote the camera used to define the model.

The semantics of Iron keeps this structure, but is additionally based on a commutative monoid $\mathcal{N}$. We call elements of $\mathcal{N}$ *linear resources*. In the following, if we need to avoid confusion between elements of $\mathcal{N}$ and $\mathcal{M}$ we shall call elements of $\mathcal{M}$ *affine resources*. An Iron assertion is modeled as a decreasing family of *pairs* of a resource and a linear resource $[\![P]\!]_0 \supseteq [\![P]\!]_1 \supseteq \cdots$, satisfying some closure properties. The most important one for this paper is that, for any n, if $(a, m) \in [\![P]\!]_n$ and $a \cdot b \in \mathcal{V}_n$ then $(a \cdot b, m) \in [\![P]\!]_n$. That is, $[\![P]\!]_n$ is upwards closed with respect to the affine resources, but we do not require upwards closure with respect to the linear resources—assertions about linear resources are exact, not lower bounds.

Semantics of selected basic assertions are shown in Figure 5.2. The semantics of the remaining assertions are straightforward modifications of the semantics of Iris assertions, *e.g.*,

$$[\![P * Q]\!]_n \triangleq \left\{ (b, m) \;\middle|\; \exists b_1, b_2, m_1, m_2.\; \begin{array}{c} b = b_1 \cdot b_2 \wedge m = m_1 \cdot m_2 \wedge \\ (b_1, m_1) \in [\![P]\!]_n \wedge (b_2, m_2) \in [\![Q]\!]_n \end{array} \right\}$$

The semantics of Emp states that no linear resources are owned, but arbitrary affine resources are owned. The semantics of $\mathsf{Own}(x)$ is similar to that of Iris, except that it contains no linear resources. The semantics of $\mathsf{OwnL}(m)$ is dual, it contains exactly the linear resource m, and arbitrary affine resources.

$$\textsc{Always-mono}\ \frac{P \vdash Q}{\Box P \vdash \Box Q} \qquad \textsc{Always-idemp}\ \Box P \vdash \Box\Box P \qquad \textsc{Always-emp}\ \mathsf{True} \vdash \Box\,\mathsf{Emp} \qquad \textsc{Always-and}\ \Box P \land \Box Q \vdash \Box(P \land Q)$$

$$\textsc{Always-or}\ \Box(P \lor Q) \vdash \Box P \lor \Box Q \qquad \textsc{Always-sep}\ \Box P * Q \vdash \Box P \qquad \textsc{Always-and-emp}\ \Box P \land Q \vdash (\mathsf{Emp} \land P) * Q$$

Fig. 5.1. Rules of the always modality.

$$\begin{aligned}
[\![\mathsf{Emp}]\!]_n &\triangleq \{(b,\varepsilon) \mid b \in \mathcal{M}\} \\
[\![\mathsf{Own}(a)]\!]_n &\triangleq \{(b,\varepsilon) \mid \exists c, a \cdot c = b\} \\
[\![\mathsf{OwnL}(m)]\!]_n &\triangleq \{(b,m) \mid b \in \mathcal{M}\} \\
[\![\Box P]\!]_n &\triangleq \left\{(b,m) \;\middle|\; (|b|,\varepsilon) \in [\![P]\!]_n\right\} \\
[\![\Rightarrow\!\!\!\Rightarrow P]\!]_n &\triangleq \left\{(b,m) \;\middle|\; \forall k \le n, \forall b_f, b \cdot b_f \in \mathcal{V}_k \Rightarrow \exists c, c \cdot b_f \in \mathcal{V}_k \land (c,m) \in [\![P]\!]_k\right\} \\
[\![\ulcorner \phi \urcorner]\!]_n &\triangleq \{(b,m) \mid \phi\}
\end{aligned}$$

Fig. 5.2. Semantics of selected basic assertions. We use ε both for the unit of the camera of resources as well as for the monoid of linear resources. $|\_|$ is the *core* operation of the camera $\mathcal{M}$.

Next, Iron contains two modalities, the always modality $\Box$ and the update modality. The rules for the update modality are the same as in Iris. The rules for the always modality $\Box$ are slightly different, and are listed in Figure 5.1. The $\Box$ modality does not have conceptually as good properties as $\blacklozenge$ or $\blacksquare$. However it is technically better behaved. In particular $\Box$ modality is better behaved with respect to other connectives, *e.g.,* it commutes with the later modality, which is needed to prove some of the derived rules of Iris/Iron.

The semantics of the basic update modality $\Rightarrow\!\!\!\Rightarrow$ is similar to its semantics in Iris, in the sense that it only affects the affine resources and only threads through the linear resources, as can be clearly seen from the definition. This point is particularly important in the proof of the adequacy theorem.

Finally we have the embedding of pure propositions $\ulcorner \phi \urcorner$, such as equality. The important point to notice here is that these assertions contain arbitrary resources, or none. In particular, they are *not* affine, which is the reason why we need to occasionally use the affine modality (*e.g.,* in the Hoare rules in Figure 3.1)

In the following subsections we use the Iron base logic to define invariants, weakest preconditions, and related constructs entirely in the base logic, without reference to the model. We assume the base logic has been instantiated with a sufficiently rich camera $\mathcal{M}$ that supports all the monoid constructions we need for encoding invariants. For reasons of space we cannot detail how such a camera is constructed, please see previous work [20,21].

5.2 Encoding invariants in Iron

The encoding of invariants is very similar to the encoding of invariants that appeared in Iris 3.0. The main difference is that we enforce that invariants only contain *affine* propositions. As discussed in the preceding sections this is necessary because Iron supports invariants, which can be shared arbitrarily among multiple threads. Since we do not keep precise track of the number of threads making use of the invariant, it should not be possible to transfer non-affine resources into invariants, and then forgotten about them.

Since the encoding is so close to the encoding of invariants in Iris 3, we omit it here due to space reasons. For the following sections we only remark that invariants are manipulated using the so-called fancy update modality. Invariants in the Iron base logic are manipulated via the fancy update modality ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$, which is defined exactly the same as in Iris. The fancy update modality enjoys exactly the same rules as in Iris, hence we omit listing them here.

5.3 Encoding Hoare triples in Iron

Hoare triples are not a primitive concept, but are defined in the Iron base logic in terms of the weakest precondition assertion, which in turn is defined in terms of the concepts we have introduced above. In Iron Hoare triples are defined as:

$$\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \triangleq \blacklozenge \left(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \left\{v.\, Q\right\}\right)$$

where $\mathsf{wp}_{\mathcal{E}}\, e\, \left\{v.\, Q\right\}$ is the weakest precondition assertion defined below. Here $\mathcal{E}$ is the set of names of invariants, which can be used when proving the specification. Note that this definition is as in Iris, apart from the use of $\blacklozenge$ modality in place of Iris's always modality $\Box$. This modality ensures that all exclusive resources needed by e are contained in the precondition P.

We now define the Iron weakest preconditions. We proceed by showing how it is related to the definition in Iris; the change in definition is quite minimal and principled, but allows us to prove a stronger adequacy theorem.

Recall that in Iris 3.0, the weakest precondition is defined as the unique assertion satisfying the following equality:

$$\begin{aligned}\mathsf{wp}_{\mathcal{E};I}\, e\, \left\{w.\, Q\right\} \triangleq\ & (e \in \mathit{Val} \wedge Q[e/w]) \vee \\ & (e \notin \mathit{Val} \wedge \forall \sigma. I(\sigma) \mathrel{-\!\!*} \\ & \quad {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \mathrm{red}((e,\sigma)) * \triangleright \forall e', \sigma', \vec{e}_f.\ ((e,\sigma) \mapsto (e', \sigma', \vec{e}_f)) \mathrel{-\!\!*} \\ & \quad\quad {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} (I(\sigma') * \mathsf{wp}_{\mathcal{E};I}\, e'\, \left\{w.\, Q\right\} * \mathop{\Large *}_{e_f \in \vec{e}_f} \mathsf{wp}_{\top;I}\, e_f\, \left\{\mathsf{True}\right\}))\end{aligned}$$

where:

- The *state interpretation* I maps physical states, *e.g.,* heaps, to Iris assertions. The state interpretation is used to connect the physical state to ghost state, in particular the points-to predicate $\ell \hookrightarrow v$, in the desired way, and depends on the language that Iris has been instantiated with.

- The mask $\mathcal{E}$ is a set of invariant names that are allowed to be used.
- The proposition $\mathrm{red}(e, \sigma)$ states that the configuration (e, σ) is not stuck.

Omitting some details, what the definition states is that $\mathsf{wp}_{\mathcal{E};I}\, e \left\{w.\, Q\right\}$ holds if e is a value, in which case $Q[e/w]$ holds, or, if e is not a value, then, for any state σ in the state interpretation I:

- The configuration (e, σ) must be able to take a step, *i.e.,* it is not stuck.
- For any configuration $(e', \sigma', \vec{e}_f)$ the configuration (e, σ) can step to:
  - the state σ' must be in the state interpretation
  - $\mathsf{wp}_{\mathcal{E};I}\, e' \left\{w.\, Q\right\}$ must hold
  - the new threads $e_f \in \vec{e}_f$ must be safe, *i.e.,* $\mathsf{wp}_{\top;I}\, e_f \left\{\mathsf{True}\right\}$ holds.

Additionally, the fancy update modalities are used so that for the duration of the execution step all invariants in $\mathcal{E}$ can be disabled (${}^{\mathcal{E}}\!\Rrightarrow^{\emptyset}$), but they must be re-enabled (${}^{\emptyset}\!\Rrightarrow^{\mathcal{E}}$) afterwards.

Keeping track of annotations. To incorporate the additional information gained by annotating the operational semantics with emit and consume statements we change the weakest precondition definition only slightly. For every step $(e, \sigma) \xrightarrow{m;n}_{\mathbf{t}} (e', \sigma', \vec{e}_f)$ we must establish ownership of linear resources $\mathsf{OwnL}(m)$, and in turn we get ownership of $\mathsf{OwnL}(n)$ to continue with the execution of e'.

The final change is that instead of requiring that the newly created threads are safe ($\mathsf{wp}_{\top;I}\, e_f \left\{\mathsf{True}\right\}$ in the Iris definition), we require that they are safe *and* that they do not leak any resources ($\mathsf{wp}_{\top;I}\, e_f \left\{\mathsf{Emp}\right\}$). Thus the weakest precondition in Iron is defined as the *unique* assertion satisfying the equality:

$$
\begin{aligned}
\mathsf{wp}_{\mathcal{E};I}\, e \left\{w.\, Q\right\} \triangleq\ & (e \in \mathit{Val} \wedge Q[e/w]) \vee \\
& \big(e \notin \mathit{Val} \wedge \forall \sigma. I(\sigma) \mathrel{-\!\!*} \\
& \quad {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \mathrm{red}((e, \sigma)) * \triangleright \forall e', \sigma', \vec{e}_f, m, n.\ \Big((e, \sigma) \xrightarrow{m;n}_{\mathbf{t}} (e', \sigma', \vec{e}_f)\Big) \mathrel{-\!\!*} \\
& \quad\ \mathsf{OwnL}(m) * \big(\mathsf{OwnL}(n) \mathrel{-\!\!*} \\
& \quad\quad {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} (I(\sigma') * \mathsf{wp}_{\mathcal{E};I}\, e' \left\{w.\, Q\right\} * \mathop{\Large *}_{e_f \in \vec{e}_f} \mathsf{wp}_{\top;I}\, e_f \left\{\mathsf{Emp}\right\})\big)\Big)
\end{aligned}
$$

5.4 Total weakest preconditions

Recall the statement of adequacy for the weakest precondition asserting we have defined in the preceding section. This weakest precondition guarantees that a program does not get stuck, and if it terminates the post-condition holds. This latter property cannot be strengthened to showing that a program does terminate. The limitation stems from the fact that $\mathsf{wp}_{\mathcal{E};I}\, e \left\{w.\, Q\right\}$ is defined by *guarded recursion*, using the later modality to guard recursive occurrences, which gives the definition an essentially coinductive nature. In particular using Löb induction, a proof technique similar to coinduction, a looping program can be proved

correct with any post-condition. The use of the later modality is however important in connection with impredicative, and higher order, invariants.

However since the definition of weakest precondition is on such a high-level it is easy to modify it. In particular the recursive occurrence of $\mathsf{wp}\, e \left\{ w.\, Q \right\}$ on the right is *positive*. Hence we can use the Knaster-Tarski fixed point theorem to define an analogue of $\mathsf{wp}\, e \left\{ w.\, Q \right\}$ *inductively*. Since Iron base logic is a higher-order logic Knaster-Tarski theorem is a theorem of the logic.

Thus we define $\mathsf{wp}\, e \left[w.\, Q \right]$ as the least predicate satisfying:

$$\begin{aligned}
\mathsf{wp}_{\mathcal{E};I}\, e \left[w.\, Q \right] \triangleq\ & (e \in \mathit{Val} \wedge Q[e/w]) \vee \\
& \Big(e \notin \mathit{Val} \wedge \forall \sigma. I(\sigma) \mathrel{-\!\!*} \\
& \quad {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \mathrm{red}((e,\sigma)) * \forall e', \sigma', \vec{e}_f, m, n.\ \Big((e,\sigma) \xrightarrow{m;n}_{\mathbf{t}} (e', \sigma', \vec{e}_f)\Big) \mathrel{-\!\!*} \\
& \quad\quad \mathsf{OwnL}(m) * \big(\mathsf{OwnL}(n) \mathrel{-\!\!*} \\
& \quad\quad\quad {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} (I(\sigma') * \mathsf{wp}_{\mathcal{E};I}\, e' \left[w.\, Q \right] * \mathop{\Large *}_{e_f \in \vec{e}_f} \mathsf{wp}_{\top;I}\, e_f \left[\mathsf{Emp} \right]))\Big)
\end{aligned}$$

Note that there is no later modality on the right hand-side. This is because, as mentioned above, if the recursive occurrences are guarded by the later modality, there is *exactly* one predicate satisfying the equation, whereas here we are aiming to have a stronger definition.

By induction (which follows from Knaster-Tarski's fixed point theorem) we can show the following theorem.

Theorem 8. *For any closed expression e, predicate Q, state interpretation I and set of invariant names $\mathcal{E}$ we have:*

$$\mathit{wp}_{\mathcal{E};I}\, e \left[w.\, Q \right] \vdash \mathit{wp}_{\mathcal{E};I}\, e \left\{ w.\, Q \right\}$$

which immediately shows that $\mathsf{wp}\, e \left[w.\, Q \right]$ guarantees safety.

Moreover, since total weakest precondition is defined inductively we can additionally show termination in the following sense.

Theorem 9. *For any closed expression e, predicate Q, state interpretation I and set of invariant names $\mathcal{E}$, if $\mathit{wp}_{\mathcal{E};I}\, e \left[w.\, Q \right]$ holds then for any σ such that $I(\sigma)$ the reduction from (e, σ) is well-founded, i.e., e is strongly normalizing.*

The downsides of this definition are that it is no longer possible to use proof by Löb induction to prove correct looping programs without a clear termination criterion, such as the spin lock. Moreover, it is not possible to use impredicative and higher-order invariants. The reason for this is that opening an invariant gives access to the resources only under the later modality. This is necessary to keep the logic consistent in presence of higher-order and impredicative invariants [24]. However using first-order invariants is still possible, since opening these invariants does not introduce a later modality.

The small nature of the change from the usual definition of weakest preconditions however emphasizes the modularity of Iris and Iron; by encoding so much of the program logic on top of an expressive base logic, it is possible to change the program logic in a relatively straightforward manner.

6 Coq formalization

The majority of the development of Iron has been formalized in the Coq proof assistant [27], and is available through GitHub [14].

In the Coq formalization of Iron we have constructed the model of the logic and proved both the soundness of all primitive and derived rules listed in this paper. We have also formalized the program logic's adequacy (see Theorem 1) for the partial correctness Hoare triples, and all the examples except for those in Section 4.4. We have not yet formalized the total correctness Hoare triples, but expect that to be done before the final version of this paper.

The formalization is based on the formalization of Iris 3.0 [24,21]: the primitive rules of the Iron are proved directly in a model of the logic in Coq, and the interactive proof mode [25] is used to prove properties of derived notions, *e.g.,* invariants, and weakest precondition assertions.

7 Discussion and related work

We have introduced the Iron higher-order concurrent separation logic as an extension of Iris, aimed at supporting more fine-grained reasoning about resources, in particular about obligations to use resources. We have achieved this by (1) making propositions in Iron linear by default and equipping Iron with an affine modality for recovering the usual affine reasoning of Iris, and (2) by introducing the emit and consume constructs for instrumentation of programs.

With regard to linearity (1), one may wonder why Iris was not just linear by default and/or why we could not achieve our goals simply by changing the standard interpretation of Iris so that the logic would be linear by default. The reason is that Iris supports a powerful notion of impredicative invariants, which are a valuable tool for specification and which are inherently affine. The impredicativity allows an invariant to contain an arbitrary propositions, including potentially even itself. This allows for more modular specifications. For instance, the examples in Sections 4.1 and 4.3 depend crucially on impredicative invariants to allow locks to guard arbitrary resources in a modular way. The impredicativity also means that it is possible to "leak" resources through invariants; and hence, impredicative invariants are somewhat antithetical to linearity in concurrent separation logic. This is why Iron does not allow for arbitrary linear resources in invariants. Iron now supports a mixture of linear and affine reasoning and we hope that linearity makes Iron sufficiently expressive for reasoning about obligations to use resources, when necessary, while still keeping the simplicity afforded by affine reasoning, when that suffices.

This approach to linearity was also taken in the recent work by Tassarotti *et al.*, who developed a linear version of Iris for reasoning about termination-preserving refinement [31]. The model of *loc. cit.* is quite similar to the model of Iron; a key technical difference is our introduction of emit and consume, and our new definition of weakest precondition, which links the linear reasoning in the logic to the operational behavior of emit and consume, and which is the

basis for our important adequacy theorem. Another difference between our logic and the one of Tassarotti *et al.* is that the latter is developed as a variant on Iris 2.0. Therefore, weakest preconditions, invariants, and a number of other connectives are introduced primitively in *loc. cit.* A contribution of our work is to demonstrate that the Iris 3.0 definitions of these connectives also function in a linear setting. Whether one needs to accommodate some notion of linear invariants in the future is an open question. We feel that Iron's approach of containing only a small base logic and creating most constructs as derived forms makes it an ideal setting in which to explore this question.

One possible application for being able to store linear resources in invariants is demonstrated in Hoffmann *et al.* [18]. In this work they use a novel scheme for proving lock-freedom in which affine tokens are used to pay for the delays caused by contention on a shared data structure. In Iron it is not currently possible to encode this because it would require the passing of linear propositions between threads with invariants, as linear resources are what is tied to the operational semantics of our language. As future work for Iron we wish to pursue the implementation of these techniques in Iron.

With regard to instrumentation (2), we note that it can be thought of as a kind of ghost state—so one may wonder why we did not simply use the ghost state facilities already supported by Iris. The reason is that the instrumentation allows us to prove a stronger adequacy theorem, which enables us to show that certain faulty implementations do *not* satisfy a specification (see Theorem 5). Moreover, the instrumentation also means that we can reason about more intensional properties of programs, *e.g.,* cost semantics.

Instrumentation has been used before to reason about more intensional properties in program logics. Aspinall *et al.* [1] created a variety of program logics for reasoning about resource usage and cost. Atkey used instrumentation for reasoning about amortized complexity using separation logic [2] as did Charguéraud and Pottier use instrumentation for reasoning about amortized complexity of the union-find algorithm in a separation logic for a sequential language [7,8], and Birkedal *et al.* used a construct like our emit in connection with a separation logic for a sequential language for reasoning about intensional trace properties, *e.g.,* API call ordering [3]. The latter made use of new basic predicates for reasoning about traces in the logic. In contrast to *loc. cit.* Iron is a *concurrent* separation logic and it supports any construct which manipulates linear resources, of which emit is just a particular instance. We can recover the ability to reason about API call orderings by ensuring that calls to emit match calls to consume. Our adequacy theorem is enough to give us that various API call requirements hold without baking it directly into the program logic.

Future work for Iron includes lifting the restriction to sequential programs in the encoding of cost semantics. Being able to encode the cost semantics of a concurrent language could allow for the verification of parallel speedups and the formalization of Brent-type theorem [16]. Additional future work includes the exploration of the logic for constructions other than program logics as has been done in Iris [25,26,32,23,30,19].

References

1. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.W., Momigliano, A.: A program logic for resources. Theoretical Computer Science pp. 411–445 (2007)
2. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science pp. 85–103 (2011)
3. Birkedal, L., Dinsdale-Young, T., Jaber, G., Svendsen, K., Tzevelekos, N.: Trace properties from separation logic specifications (2017), http://arxiv.org/abs/1702.02972
4. Boyland, J.: Checking interference with fractional permissions. In: SAS. pp. 55–72 (2003)
5. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z.: End-to-end verification of stack-space bounds for c programs. In: PLDI. pp. 270–281 (2014)
6. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: PLDI. pp. 467–478 (2015)
7. Charguéraud, A., Pottier, F.: Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: ITP. pp. 137–153 (2015)
8. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning (2017)
9. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: ECOOP. pp. 207–231 (2014)
10. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. pp. 504–528 (2010)
11. Feng, X.: Local rely-guarantee reasoning. In: POPL. pp. 315–327 (2009)
12. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: ESOP. pp. 173–188 (2007)
13. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about optimistic concurrency using a program logic for history. In: CONCUR. pp. 388–402 (2010)
14. Gratzer, D., Bizjak, A., Krebbers, R., Birkdal, L.: The Iron Coq Formalization. https://github.com/jozefg/iron (2017)
15. Gratzer, D., Høier, M., Bizjak, A., Birkedal, L.: Formalizing concurrent stacks with helping: A case study in Iris (2017), available at `http://iris-project.org`
16. Gustafson, J.L.: Brent's theorem. In: Encyclopedia of Parallel Computing. pp. 182–185 (2011)
17. Hobor, A., Appel, A.W., Zappa Nardelli, F.: Oracle semantics for concurrent separation logic. In: ESOP. pp. 353–367 (2008)
18. Hoffmann, J., Marmar, M., Shao, Z.: Quantitative reasoning for proving lock-freedom. In: LICS. pp. 124–133 (2013)
19. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. In: POPL (2018), conditionally accepted to POPL 2018
20. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order ghost state. In: ICFP. pp. 256–269 (2016)
21. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic (2017), conditionally accepted to JFP
22. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL. pp. 637–650 (2015)

23. Kaiser, J.O., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In: ECOOP (2017)
24. Krebbers, R., Jung, R., Bizjak, A., Jourdan, J., Dreyer, D., Birkedal, L.: The essence of higher-order concurrent separation logic. In: ESOP. pp. 696–723 (2017)
25. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: POPL. pp. 205–217 (2017)
26. Krogh-Jespersen, M., Svendsen, K., Birkedal, L.: A relational model of types-and-effects in higher-order concurrent separation logic. In: POPL. pp. 218–231 (2017)
27. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), http://coq.inria.fr
28. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: ESOP. pp. 290–310 (2014)
29. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: ESOP. pp. 149–168 (2014)
30. Swasey, D., Garg, D., Dreyer, D.: Robust and compositional verification of object capability patterns. In: OOPSLA. pp. 89:1–89:26 (2017)
31. Tassarotti, J., Jung, R., Harper, R.: A higher-order logic for concurrent termination-preserving refinement. In: ESOP. pp. 909–936 (2017), http://dx.doi.org/10.1007/978-3-662-54434-1_34
32. Timany, A., Stefanesco, L., Krogh-Jespersen, M., Birkedal, L.: A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of `runST`. In: POPL (2018), conditionally accepted to POPL 2018
33. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In: ICFP. pp. 377–390 (2013)
34. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. pp. 256–271 (2007)