A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic

Abstract

Recently we have seen a renewed interest in programming languages that tame the complexity of state and concurrency through refined type systems with more fine-grained control over effects. In addition to simplifying reasoning and eliminating whole classes of bugs, statically tracking effects opens the door to advanced compiler optimizations.

In this paper we present a relational model of a type-and-effect system for a higher-order, concurrent programming language. The model precisely captures the semantic invariants expressed by the effect annotations. We demonstrate that these invariants are strong enough to prove advanced program transformations, including automatic parallelization of expressions with suitably disjoint effects. The model also supports refinement proofs between abstract data type implementations with different internal data representations, including proofs that fine-grained concurrent algorithms refine their coarse-grained counterparts. This is the first model for such an expressive language that supports both effect-based optimizations and data abstraction.

The logical relation is defined in Iris, a state-of-the-art higher-order concurrent separation logic. This greatly simplifies proving well-definedness of the logical relation and provides us with a powerful logic for reasoning in the model.

1. Introduction

Programming with and reasoning about effects in higher-order programs is well-known to be very challenging. Over the years, there have therefore been many proposals of refined type systems for taming and simplifying reasoning about effectful programs. Examples include alias types [27], capability type systems [22], linear type systems [14, 17, 19] Hoare type theory [20], permissions-based type systems [23], type-and-effect systems [5, 6, 15, 18], etc. Lately, we have also witnessed some larger-scale implementation efforts on higher-order programming languages, e.g., the Mezzo programming language [23] and the Rust programming language [26], which employ refined type systems to control the use of state in the presence of concurrency.

In this paper, we provide a logical account of an expressive region-based type-and-effect system for a higher-order concurrent programming language $\lambda_{ref,conc}$ with general references (higher-order store). The type-and-effect system is taken from [11]; it is inspired by Lucassen and Gifford's seminal work [15, 18], but also features a notion of public and private regions, which can be used to limit interference from threads running in parallel. Hence it can be used to express effect-based optimizations, as emphasized for type-and-effect systems for sequential languages by Benton et al., see, e.g., [5, 6]. Effect-based optimizations are examples of so-called "free theorems", *i.e.,* they just depend on the types and effects of the involved expressions, not on the particular expressions involved. The most interesting effect-based optimization is a parallelization theorem expressing the equivalence of running expressions e_1 and e_2 in parallel and running them sequentially, assuming their effects are suitably disjoint. Note that this is a relational property, i.e., the intended invariants of the type-and-effect system are relational in nature. Our logical account of the type-and-effect system thus consists of a logical relations interpretation of the types in a program logic, and we prove that logical relatedness implies contextual equivalence. We show that our logical relations interpretation is strong enough to prove the soundness of effect-based optimizations, in particular the challenging parallelization theorem.

Since the programming language $\lambda_{ref,conc}$ includes higher-order store, it is non-trivial to define a logical relations interpretation of the types, as one is faced with the well-known type-world circularity [1] (see [10] for an overview). Here we factor out this challenge, by using a state-of-the-art program logic, Iris [16], as the logic in which we express the logical relations. Iris has direct support for impredicative invariants, as needed for defining logical relations for general references. Iris also supports reasoning about concurrency, in particular, it supports a form of rely-guarantee reasoning about shared state. We use this facility to capture invariants of private and public regions. Moreover, we show, using simple synthetic examples, how we can also use the logic to prove that syntactically ill-typed programs obey the semantic invariants enforced by the type system. This is important in practice: both Mezzo and Rust contain facilities for programming with statically ill-typed expressions (Mezzo uses dynamic type checks [24] and Rust has a facility for including unsafe code in statically typed programs [26]) so models of type-and-effect systems should preferably support reasoning about combinations of statically ill-typed and statically well-typed programs.

1.1 Overview of Challenges and Contributions

The typing judgments of our type-and-effect system take the form

$$\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$$

and express that the term e is of type τ and has effect ε in the typing context Γ mapping variables to types. The additional contexts Π and Λ consists of region variables ρ denoting, respectively, the public regions and the private regions that e may use. Intuitively, public regions are those that other threads may also use, whereas private regions are not subject to interference from other threads. Thus, from a thread-local perspective, the segregation describes an expression's expectations of interference from the environment. The effect ε is a finite set of read rd_ρ, write wr_ρ,

or allocation effects, al_ρ, the intuition being that if, *e.g.,* $rd_\rho \in \varepsilon$, then e may read a reference belonging to region ρ.

Effect-based optimizations. Using effect annotations we can express the idea of parallelization mentioned above formally as follows (where $\mathsf{rds}\ \varepsilon$ is the set of regions with read effects in ε and likewise for $\mathsf{wrs}\ \varepsilon$ and $\mathsf{als}\ \varepsilon$):

Theorem 1 (Parallelization)**.** *If* $\Lambda = \Lambda_1, \Lambda_2, \Lambda_3$ *and*

1. $\Lambda_3 \mid \Lambda_1 \mid \Gamma_1 \vdash e_1 : \tau_1, \varepsilon_1$ *and* $\Lambda_3 \mid \Lambda_2 \mid \Gamma_2 \vdash e_2 : \tau_2, \varepsilon_2$
2. $\mathsf{als}\ \varepsilon_1 \cup \mathsf{wrs}\ \varepsilon_1 \subseteq \Lambda_1$, $\mathsf{als}\ \varepsilon_2 \cup \mathsf{wrs}\ \varepsilon_2 \subseteq \Lambda_2$
3. $\mathsf{rds}\ \varepsilon_1 \subseteq \Lambda_1 \cup \Lambda_3$ *and* $\mathsf{rds}\ \varepsilon_2 \subseteq \Lambda_2 \cup \Lambda_3$

then $\cdot \mid \Lambda \mid \Gamma_1, \Gamma_2 \vdash e_1 \,||\, e_2 \cong_{ctx} (e_1, e_2) : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$.

Here Λ_1 are the private regions of e_1, and Λ_2 are the private regions of e_2, and Λ_3 are regions that can be used by both e_1 and e_2. The theorem then says, that if the expressions e_i only writes and allocates in their private regions (item 2) and only reads in private or shared regions (item 3), then, running e_1 in parallel with e_2 is contextually equivalent with running e_1 and e_2 sequentially, *if* the context is not allowed to access any locations used by the two expressions (expressed by the fact that $\Lambda_1, \Lambda_2, \Lambda_3$ are all private in the conclusion).

Intuitively, this theorem sounds very plausible, perhaps even quite obvious, but proving it formally was an open problem for more than 25 years [11] and it is still very difficult to prove for higher-order languages with general references, such as ours. Indeed, one of our key contributions is a novel proof technique for proving parallelization. To outline our approach, consider proving the left-to-right approximation of the parallelization theorem (Theorem 1). Then we, in particular, have to show that any reduction step taken by $e_1 \,||\, e_2$ can also be taken by (e_1, e_2). In the case where the expression $e_1 \,||\, e_2$ takes a step in the right branch, we cannot yet take the corresponding step in (e_1, e_2), unless e_1 has already reduced to a value. Previous methods for proving parallelization therefore relied on reordering steps taken in e_2 with steps from e_1, while preserving the semantic invariants - resulting in very difficult proofs [11] or trace-based arguments [9], which are not known to scale to a programming language with general references with dynamic allocation.

Our new technique is instead based on framing. We suspend the reduction on the right hand side temporarily, and first disentangle the reduction of $e_1 \,||\, e_2$ into two semi-independent ("semi" because they can read from shared regions) reductions for e_1 and e_2 respectively, which can then be reassembled into a reduction for (e_1, e_2) using framing. The disentanglement and the reassembling qua framing, of course, depends on the effect annotations, and our formal argument leverages Iris' facility for capturing sophisticated ownership disciplines. We present a more detailed description and the formal argument in Section 3.4.

Data abstraction and local state. The $\lambda_{ref,conc}$ language supports hiding of local state using closures. Hiding can be used to implement abstract data types (ADTs) that manipulate an internal data representation, which can only be accessed through the provided operations. Relating ADT implementations that use different internal data representations is well-studied in the setting of ML-like type systems (see, *e.g.,* [2, 31] and the references therein); effect tracking adds several interesting dimensions.

In the ML-setting the type system imposes no constraints on local state when relating ADT implementations. This is not the case in our setting. To illustrate, consider the following counter implemented using local state:

$$e_{\text{count}} \triangleq \mathsf{let}\ x = \mathsf{new}\ 0\ \mathsf{in}\ \mathsf{rec}\ inc().\mathsf{let}\ y = !x\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(x, y, y+1)\ \mathsf{then}\ y\ \mathsf{else}\ inc()$$

When e_{count} is run, it allocates a local reference x and then returns a function which increments x (using a compare-and-swap (**CAS**) operation in a loop to increment x atomically, to allow for concurrent use) and returns the old value.

The counter has the following type:

$$\rho \mid - \mid - \vdash e_{\text{count}} : 1 \to^{\rho,-}_{\{rd_\rho, wr_\rho\}} \mathsf{int}, \{al_\rho\}$$

The type is a function type, which is annotated with a latent effect, expressing that the returned function may read and write in the public region ρ. To prove soundness of effect-based transformations, it is, of course, crucial that the semantic model also enforces the semantic invariants expressed by the effect annotations on local state. Otherwise, if our semantic model would allow us to forget about the effects on the local reference x, then we would be able to show, using a semantic version of Theorem 1, that $\mathsf{let}\ g = e_{\text{count}}()\ \mathsf{in}\ g() \,||\, g()$ is contextually equivalent to $\mathsf{let}\ g = e_{\text{count}}()\ \mathsf{in}\ (g(), g())$, which is not the case (the first expression may evaluate to $(1, 0)$, while the second always evaluates to $(0, 1)$).

We can use the type-and-effect system to limit interference from the environment on the internal state of ADTs, when relating ADTs. For example, consider the two stack modules listed in *Figure* 1. The left stack-module, $\textsc{Stack}_1$, uses a single reference to a pure functional list whereas the right module, $\textsc{Stack}_2$, uses a linked list representation. Both stack implementations use a **CAS** operation to ensure that they function correctly in the presence of concurrent interference. The implementations (*i.e., stack*$_1$ and *stack*$_2$) can be given the following type $\tau_{\textsc{Stack}}$:

$$1 \to^{\rho,-}_{al_\rho} (\mathsf{int} \to^{\rho,-}_{\{wr_\rho, rd_\rho, al_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{\rho,-}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathsf{int})$$

This type expresses that each module will allocate in region ρ and return two functions *push* and *pop*. The type further expresses that *push* is allowed to have read, write and allocate effects on the local state described by ρ and that *pop* is allowed to read and write.

Intuitively, the two implementations are equivalent at this type, because their internal data representations are purely local and hidden from clients of the modules. Indeed, we can use our logical relation to prove:

Theorem 2. $\rho \mid - \mid - \vdash stack_1 \cong_{ctx} stack_2 : \tau_{\textsc{Stack}}, \{al_\rho\}$

Now, if we restrict possible interference from the environment by making the ρ region private, as expressed by the type $\tau'_{\textsc{Stack}}$ (ρ is now private on the latent effects, since it comes *after* the comma):

$$1 \to^{-,\rho}_{al_\rho} (\mathsf{int} \to^{-,\rho}_{\{wr_\rho, rd_\rho, al_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathsf{int})$$

then the two implementations are still contextually equivalent at this type.

Moreover, for this type, we can also prove that we can safely omit the **CAS** operation from the stack implementations (intuitively, because there is no possible concurrent interference). Thus, writing *stack_nc*$_i$ for the implementation of *stack*$_i$ without a **CAS** loop, we can use our logical relation to prove the following equivalences.

$stack_1() = \textsf{let } h = \textsf{new inj}_1\ () \textsf{ in } (push_1(h), pop_1(h))$
$push_1(h) = \textsf{rec } loop(n).\textsf{let } v = !h \textsf{ in}$
$\quad \textsf{if CAS}(h, v, \textsf{inj}_2\ (n, v)) \textsf{ then } () \textsf{ else } loop(n)$
$pop_1(h) = \textsf{rec } loop(\_).\textsf{let } v = !h \textsf{ in case}(v, \textsf{inj}_1\ () \Rightarrow \textsf{inj}_1\ (),$
$\quad \textsf{inj}_2\ (n, v') \Rightarrow \textsf{if CAS}(h, v, v') \textsf{ then inj}_2\ n \textsf{ else } loop())$

$stack_2() = \textsf{let } h = \textsf{new } (\textsf{new inj}_1\ ()) \textsf{ in } (push_2(h), pop_2(h))$
$push_2(h) = \textsf{rec } loop(n).\textsf{let } v = !h \textsf{ in}$
$\quad \textsf{if CAS}(h, v, \textsf{new inj}_2\ (n, v)) \textsf{ then } () \textsf{ else } loop(n)$
$pop_2(h) = \textsf{rec } loop(\_).\textsf{let } v = !h \textsf{ in case}(!v, \textsf{inj}_1\ () \Rightarrow \textsf{inj}_1\ (),$
$\quad \textsf{inj}_2\ (n, v'') \Rightarrow \textsf{if CAS}(h, v, v'') \textsf{ then inj}_2\ n \textsf{ else } loop())$

Figure 1. The left stack module, Stack_1, has a single reference to a pure list where Stack_2 uses a reference to a linked list.

Theorem 3.

$$- \mid \rho \mid - \vdash stack\_nc_1 \cong_{ctx} stack_1 : \tau'_{\text{STACK}}, \{al_\rho\}$$
$$\textit{and} \quad - \mid \rho \mid - \vdash stack\_nc_2 \cong_{ctx} stack_2 : \tau'_{\text{STACK}}, \{al_\rho\}$$

Our proofs of data abstraction, detailed in the Appendix, leverages Iris's facility for expressing invariants on local state. As pointed out in [11] the logical relation in *loc.cit.* could not be used to prove equivalences such as this one, since the logical relation there only allowed for much more restricted invariants.

Ill-typed terms. Here is a simple example of a statically ill-typed expression which nevertheless satisfies the semantic invariants enforced by the type system:

$$e \triangleq x := (); x := \textsf{true}$$

This expression first assigns the unit value to a boolean reference, and then assigns **true** to it.

This expression is not statically typable, due to the assignment of unit to a boolean reference. However, if the boolean reference is *private* then the rest of the program is not allowed to observe the ill-typed intermediate value and will thus never observe that the typing discipline has been broken. It is thus perfectly safe to use the untypable term e *as if* it had the following type:

$$- \mid \rho \mid x : \textsf{ref}_\rho\ \textsf{B} \vdash e : \textsf{1}, \{rd_\rho, wr_\rho\}$$

Our logical relations model allows us also to reason about such statically ill-typed terms and, *e.g.,* prove that e is equivalent a statically well-typed expression which only assigns **true** to x.

Summary of Contributions In summary, the contributions of this paper are:

- We show how to interpret types of a region-based type-and-effect system for a concurrent higher-order imperative programming language with higher-order store as logical relations in the state-of-the-art program logic Iris.
- We use the interpretation to prove soundness of effect-based optimizations. In particular, we prove the soundness of the parallelization theorem. Our parallelization theorem is a strengthening of the one in [11] and for our proof we use a novel proof technique, based on framing. The resulting proof is arguably a lot clearer and more abstract than the one in [11], thanks to the use of the logical features of Iris.
- We use the interpretation to prove contextual equivalence of fine-grained concurrent data structures that use local state to hide internal data representations. Our examples could not be proved with the logical relation [11].
- We show how the logic may be used to prove that syntactically ill-typed expressions obey the semantic properties enforced by the type system.
- We demonstrate that the logic allows us to give a modular definition of the logical relation and explain the relation by breaking it down into more manageable parts.

Outline We begin by formally defining the syntax and semantics of $\lambda_{ref,conc}$, the type-and-effect system, and contextual equivalence in Section 2. In Section 3 we turn our attention to logical relations for $\lambda_{ref,conc}$. We present our logical relation in four stages, starting from a unary relation that characterizes type inhabitance and ending with a binary relation for reasoning about contextual equivalence that supports advanced effect-based optimizations, each building on the previous relation. We conclude and discuss related and future work in Section 4.

2. $\lambda_{ref,conc}$ with Types, Regions and Effects

In this section we present the operational semantics and the type-and-effect system for $\lambda_{ref,conc}$, a call-by-value language with general references and concurrency primitives $||$ and **CAS** (compare-and-swap).

2.1 Syntax and Operational Semantics of $\lambda_{ref,conc}$

The syntax of $\lambda_{ref,conc}$ is shown in Figure 2 and the operational semantics is summarized in Figure 3. We assume given denumerably infinite sets of variables VAR, ranged over by x, y, f, and locations LOC, ranged over by l. We use v to range over the set of values, VAL, and e to range over the set of expressions, EXP. Note that expressions do not include types. We use **B**, **true**, **false** and **if** e **then** e_1 **else** e_2 as shorthands for booleans and branching encoded using sums.

$$v ::= () \mid n \mid (v, v) \mid \textsf{inj}_i\ v \mid \textsf{rec } f(x).e \mid x \mid l$$
$$e ::= v \mid e = e \mid e\ e \mid (e, e) \mid \textsf{prj}_i\ e \mid \textsf{inj}_i\ e \mid e + e \mid \textsf{new } e \mid !e$$
$$\mid e := e \mid \textsf{CAS}(e, e, e) \mid e \,||\, e \mid \textsf{case}(e, \textsf{inj}_1\ x \Rightarrow e, \textsf{inj}_2\ y \Rightarrow e)$$

Figure 2. Syntax of $\lambda_{ref,conc}$.

The operational semantics is defined by a small-step relation between configurations consisting of a heap and an expression. Heaps h are finite partial maps from locations to values. The semantics is defined in terms of evaluation contexts, $K \in \text{ECTX}$. We use $K[e]$ to denote the expression obtained by plugging e into the context K and $e[v/x]$ to denote capture-avoiding substitution of value v for variable x in expression e.

2.2 Types and Effects for $\lambda_{ref,conc}$

The set of types is defined by the following grammar:

$$\text{TYPE } \tau ::= \textsf{1} \mid \textsf{int} \mid \textsf{ref}_\rho\ \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau \to^{\Pi,\Lambda}_{\varepsilon} \tau$$

Π and Λ are finite sets of region variables, taken from a denumerably infinite set REGVAR ranged over by ρ. We use comma to denote disjoint union of sets of region variables.

$$K ::= [\,] \mid K = e \mid v = K \mid K\ e \mid v\ K \mid (K, e) \mid (v, K) \mid \mathbf{prj}_i\ K$$
$$\mid \mathbf{inj}_i\ K \mid \mathbf{case}(K, \mathbf{inj}_1\ x \Rightarrow e, \mathbf{inj}_2\ y \Rightarrow e) \mid \mathbf{new}\ K$$
$$\mid\ !K \mid K := e \mid v := K \mid K \,||\, e \mid e \,||\, K \mid \mathbf{CAS}(K, e, e)$$
$$\mid\ \mathbf{CAS}(v, K, e) \mid \mathbf{CAS}(v, v, K) \mid K + e \mid v + K$$

Pure reduction $\boxed{e \overset{\text{pure}}{\to} e'}$

$$v_1 \,||\, v_2 \overset{\text{pure}}{\to} (v_1, v_2)$$

Reduction $\boxed{h; e \to h'; e'}$

$$\begin{aligned}
h; e &\to h; e' && \text{if } e \overset{\text{pure}}{\to} e' \\
h; \mathbf{new}\ v &\to h \uplus [l \mapsto v]; l \\
h; !l &\to h; v && \text{if } h(l) = v \\
h[l \mapsto -]; l := v &\to h[l \mapsto v]; () \\
h; \mathbf{CAS}(l, v_o, v_n) &\to h; \mathbf{false} && \text{if } h(l) \neq v_o \\
h[l \mapsto v_o]; \mathbf{CAS}(l, v_o, v_n) &\to h[l \mapsto v_n]; \mathbf{true} \\
h; K[e] &\to h'; K[e'] && \text{if } h; e \to h'; e'
\end{aligned}$$

Figure 3. Operational semantics of $\lambda_{ref,conc}$. Remaining pure reductions are standard (see the Appendix).

An atomic effect on a region ρ is either a read effect, rd_ρ, a write effect, wr_ρ, or an allocation effect, al_ρ. An effect ε is a finite set of atomic effects.Typing judgments take the form $\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$. An excerpt of the typing rules are shown in Figure 4. All the typing rules can be found in the Appendix.

Regions can be introduced by the masking rule (TMask). The masking rule expresses when we can introduce a new private region ρ for the evaluation of an expression e and hide all of e's effects on region ρ. The condition $\rho \notin FRV(\Gamma, \tau)$ ensures that we do not leak any locations of ρ and hence, from the perspective of e, region ρ is private. The masking rule has been used to do memory-management [29] and to hide local effects to enable more program-transformations [4, 28].

Since the masking rule allows us to hide local state effects, a pure operation is not necessarily deterministic in our setting. For instance, the following code-snippet which non-deterministically returns **true** or **false** can be typed as a pure expression:

$$- \vdash \mathbf{let}\ x = \mathbf{new\ true\ in}\ (x := \mathbf{true} || x := \mathbf{false}); !x : \mathbf{B}, \emptyset$$

2.3 Contextual Equivalence for $\lambda_{ref,conc}$

We take contextual equivalence as our basic notion of equivalence. Contextual equivalence relates two expressions if no suitably typed context can distinguish them. For a concurrent language such as $\lambda_{ref,conc}$ we have to choose whether there simply has to exist an indistinguishable reduction (may-equivalence) or whether all possible reductions must be indistinguishable (must-equivalence). In this paper we study may-equivalence and may-approximation, as defined below.

Definition 1. *$\Pi \mid \Lambda \mid \Gamma \vdash e_1 \leq_{ctx} e_2 : \tau, \varepsilon$ iff for all contexts C, values v, and heaps h_1 such that $C : (\Pi \mid \Lambda \mid \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (- \mid - \mid - \vdash \boldsymbol{B}, \emptyset)$ and $[]; C[e_1] \to^* h_1; v$ there exists a heap h_2 such that $[]; C[e_2] \to^* h_2; v$.*

The $C : (\Pi \mid \Lambda \mid \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (\Pi' \mid \Lambda' \mid \Gamma' \vdash \tau', \varepsilon')$ relation expresses that the context C takes a term e of the former type to a term of the latter type; the definition is standard and relegated to the Appendix. Note that e_1 and e_2 are not required to be well-typed in the definition above. Contextual equivalence $\Pi \mid \Lambda \mid \Gamma \vdash e_1 \cong_{ctx} e_2 : \tau, \varepsilon$ is then defined as contextual approximation in both directions.

3. A Logical Relation for $\lambda_{ref,conc}$

In this section we present a logical relation for $\lambda_{ref,conc}$. To aid exposition we present the logical relation in four steps. We start by defining a unary logical relation for a simplified type system without regions and effects. This allows us to focus on the use of Iris as a meta-language for logical relations and provide a gentle introduction to Iris. We then extend the unary relation to the full type system with regions and effects, focusing on how effects are translated into abstract descriptions of possible interference. These unary logical relations characterize type inhabitance, which suffices for establishing type soundness, but not for proving equivalences. As the third step we naively extend the unary relation for the full type system to a binary relation, focusing on how to express a binary relation in a unary program logic. This yields a logical relation that is sound with respect to contextual approximation and suffices for proving equivalences of many concrete examples, but not advanced effect-based equivalences such as parallelization. For the fourth and final relation, we refine the third relation further, to support reasoning using multiple simulations. This final relation validates parallelization and is also sound with respect to contextual approximation.

This staged presentation also highlights the modularity of using Iris as a meta-language for logical relations. In particular, each step builds naturally on the previous, only requiring small changes or additions between each relation.

3.1 Unary Relation for $\lambda_{ref,conc}$ Without Effects

We begin by defining a unary logical relation for $\lambda_{ref,conc}$ with a standard ML-like type system without regions and effects. The goal is to define a unary relation, LR_{ML} that characterizes type inhabitance semantically and is sound with respect to the syntactic typing rules. More precisely, we wish to define two unary relations, a value relation, $[\![\tau]\!]$, that characterizes values of type τ and an expression relation, $\mathcal{E}[\![\tau]\!]$, that characterizes expressions that either diverge or evaluate to values of type τ.

For ground types the definition of $[\![\tau]\!]$ is obvious: it is the values of the given type τ. The main difficulty arises when defining the interpretation of reference types. The idea is to take a location l to be an inhabitant of type $\mathbf{ref}\ \tau$ if location l contains a value of type τ in the current heap. $\lambda_{ref,conc}$ is a concurrent language and the context is free to update the heap as it sees fit. However, the context must preserve typing and we can thus think of $[\![\mathbf{ref}\ \tau]\!](l)$ as expressing an *invariant* that l must always contain a value v of the semantic type $[\![\tau]\!]$ in the heap. To formalize this we introduce our meta-language, Iris.

Iris and invariants. Iris is a generic framework for constructing higher-order separation logics. For the purposes of this paper we present one particular instance of this framework for the $\lambda_{ref,conc}$ language and we refer to this instance simply as Iris.

Figure 5 contains an excerpt of the Iris syntax and proof rules. Iris is a higher-order logic over a simply-typed term language. The set of Iris types, ranged over by κ, includes a type of $\lambda_{ref,conc}$ expressions **Exp** and values **Val**, a type of propositions, **Prop**, and is closed under products and function spaces. Iris includes the usual connectives ($\bot, \top, \wedge, \vee,$

$$\frac{(x:\tau)\in\Gamma}{\Pi \mid \Lambda \mid \Gamma \vdash x:\tau,\emptyset} \qquad \frac{}{\Xi \vdash ():\mathbf{1},\emptyset} \qquad \frac{\Xi \vdash e_1:\tau,\varepsilon_1 \qquad \Xi \vdash e_2:\tau,\varepsilon_2 \qquad eq_{type}(\tau)}{\Xi \vdash e_1 = e_2 : \mathbf{B}, \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Pi \mid \Lambda \mid \Gamma, f:\tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2, x:\tau_1 \vdash e:\tau_2,\varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{rec}\ f(x).e : \tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2, \emptyset}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e_1 : \tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_2 : \tau_1, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash e_1\ e_2 : \tau_2, \varepsilon \cup \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Xi \vdash e : \mathbf{ref}_\rho\ \tau, \varepsilon}{\Xi \vdash !e : \tau, \varepsilon \cup \{rd_\rho\}} \qquad \frac{\Pi \mid \Lambda \mid \Gamma \vdash e:\tau,\varepsilon \qquad \rho \in \Pi, \Lambda}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{new}\ e : \mathbf{ref}_\rho\ \tau, \varepsilon \cup \{al_\rho\}}$$

$$\frac{\Xi \vdash e_1 : \mathbf{ref}_\rho\ \tau, \varepsilon_1 \qquad \Xi \vdash e_2 : \tau, \varepsilon_2}{\Xi \vdash e_1\ :=\ e_2 : \mathbf{1}, \varepsilon_1 \cup \varepsilon_2 \cup \{wr_\rho\}} \qquad \frac{\Pi \mid \Lambda, \rho \mid \Gamma \vdash e:\tau,\varepsilon \qquad \rho \notin FRV(\Gamma,\tau)}{\Pi \mid \Lambda \mid \Gamma \vdash e:\tau,\varepsilon-\rho}\ \textsc{TMask} \qquad \frac{}{eq_{type}(\mathbf{1})}$$

$$\frac{\Pi, \Lambda_3 \mid \Lambda_1 \mid \Gamma_1 \vdash e_1 : \tau_1, \varepsilon_1 \qquad \Pi, \Lambda_3 \mid \Lambda_2 \mid \Gamma_2 \vdash e_2 : \tau_2, \varepsilon_2}{\Pi \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma_1, \Gamma_2 \vdash e_1 \,||\, e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Xi \vdash e_1 : \mathbf{ref}_\rho\ \tau, \varepsilon_1 \qquad \Xi \vdash e_2 : \tau, \varepsilon_2 \qquad \Xi \vdash e_3 : \tau, \varepsilon_3 \qquad eq_{type}(\tau)}{\Xi \vdash \mathbf{CAS}(e_1, e_2, e_3) : \mathbf{B}, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \cup \{wr_\rho, rd_\rho\}}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau_1, \varepsilon_1 \qquad \Pi, \Lambda \vdash \tau_1 \le \tau_2 \qquad \varepsilon_1 \subseteq \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau_2, \varepsilon_2} \qquad \frac{eq_{type}(\tau) \qquad eq_{type}(\sigma) \qquad op \in \{+, \times\}}{eq_{type}(\tau\ op\ \sigma)}$$

$$\frac{FRV(\tau) \in \Pi}{\Pi \vdash \tau \le \tau} \qquad \frac{\Pi \vdash \tau_1 \le \tau_1' \qquad \Pi \vdash \tau_2 \le \tau_2'}{\Pi \vdash \tau_1 \times \tau_2 \le \tau_1' \times \tau_2'} \qquad \frac{\Pi \vdash \tau_1' \le \tau_1 \qquad \Pi \vdash \tau_2 \le \tau_2' \qquad \varepsilon_1 \subseteq \varepsilon_2 \qquad \Pi_1 \subseteq \Pi_2 \qquad \Lambda_1 \subseteq \Lambda_2}{\Pi \vdash \tau_1 \to_{\varepsilon_1}^{\Pi_1,\Lambda_1} \tau_2 \le \tau_1' \to_{\varepsilon_2}^{\Pi_2,\Lambda_2} \tau_2'}$$

Figure 4. Excerpt of typing and sub-typing inference rules. We write $FV(e)$ and $FRV(e)$ for the sets of free program variables and region variables respectively. For all typing judgments $\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$ we implicitly assume that $FRV(\Gamma, \tau, \varepsilon) \in \Pi \cup \Lambda$. The equality type predicate, eq_{type}, defines the types we may test for equality. We use Ξ as shorthand for $\Pi \mid \Lambda \mid \Gamma$.

$$\begin{array}{rcl}
\kappa & ::= & 1 \mid \mathbf{Exp} \mid \mathbf{Val} \mid \mathbf{Name} \mid \mathbf{Prop} \mid \mathbf{Monoid} \mid \kappa \times \kappa \mid \cdots \\
t, \varphi, \iota, & ::= & x \mid \lambda x : \kappa.\, t \mid \varphi\ t \mid (t,t) \mid \pi_1(t) \mid \pi_2(t) \mid t =_\kappa t \\
P, \mathcal{M} & \mid & t \mapsto t \mid \bot \mid \top \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \\
& \mid & \forall x : \kappa.\, P \mid \exists x : \kappa.\, P \mid P * P \mid P \mathrel{-\!\!*} P \mid \Box P \\
& \mid & \triangleright P \mid \boxed{P}^\iota \mid \{P\}\ e\ \{v.\ Q\}_{\mathcal{M}} \mid P\ {}^{\mathcal{M}}\!\Rrightarrow^{\mathcal{M}} Q \mid \cdots
\end{array}$$

Figure 5. Excerpt of Iris syntax.

$\Rightarrow, \forall, \exists, *, \mathrel{-\!\!*}, =_\kappa$) and proof rules of higher-order separation logic. Iris extends this with a few new primitives, which we explain below.

Iris makes no distinction between assertions and specifications. Specifications are simply treated as special assertions that do not express ownership of any state. This is captured by the always modality, $\Box P$, which expresses that P holds and does not assert ownership of any state. Since $\Box P$ does not assert any ownership, it can be freely duplicated ($\Box P \iff \Box P * \Box P$). We therefore call assertions of the form $\Box P$ *pure*.

One of the main features of Iris is *invariants* for reasoning about shared state. The pure assertion $\boxed{P}^\iota$ asserts the existence of an invariant with the name ι that owns a resource satisfying the assertion P. Resources owned by invariants are shared by every thread and can be accessed freely by atomic operations, provided the invariant is preserved. For atomic operations we can thus *open* an invariant and take local ownership of the resource owned by the invariant for the duration of the operation, provided we transfer back a resource that satisfies the invariant assertion after the operation. In Iris this is captured formally by *view-shifts*. A view-shift, written $P \Rrightarrow Q$ expresses that it is possible to transform a resource satisfying P into a resource satisfying Q, without changing the underlying physical state. To reason about opening of invariants, view-shifts are further annotated with *invariant masks* indicating which invariants are required to hold before and after the view-shift. In the view-shift $P\ {}^{\mathcal{M}_1}\!\Rrightarrow^{\mathcal{M}_2} Q$, $\mathcal{M}_1$ and $\mathcal{M}_2$ are invariant masks (sets of invariant names) required to hold before and after the view-shift respectively. The invariant masks ensure that we do not open an invariant twice (which would not be sound in general). Opening and closing of invariants is captured by the two following view-shift axioms:

$$\frac{}{\boxed{P}^\iota\ {}^{\{\iota\}}\!\Rrightarrow^{\emptyset} \triangleright P}\ \textsc{InvOpen} \qquad \frac{}{\boxed{P}^\iota * \triangleright P\ {}^{\emptyset}\!\Rrightarrow^{\{\iota\}} \top}\ \textsc{InvClose}$$

The InvOpen rule allows us to take ownership of P upon opening the invariant ι, while the InvClose rule requires us to relinquish ownership of P to close the invariant ι. In both rules, the resource P is guarded by a modality, $\triangleright$, which we explain shortly.

To apply these view-shifts to open an invariant for the duration of an atomic operation, such as reading ($!e$), writing ($x := e$) or allocating ($\mathbf{new}\ e$), Iris features the following atomic rule-of-consequence.

$$\frac{e\ \text{atomic} \qquad P_1\ {}^{\mathcal{M}\uplus\mathcal{M}'}\!\Rrightarrow^{\mathcal{M}} P_2 \qquad \{P_2\}\ e\ \{Q_2\}_{\mathcal{M}} \qquad \forall v.\, Q_2(v)\ {}^{\mathcal{M}}\!\Rrightarrow^{\mathcal{M}\uplus\mathcal{M}'} Q_1(v)}{\{P_1\}\ e\ \{Q_1\}_{\mathcal{M}\uplus\mathcal{M}'}}\ \textsc{ACsq}$$

Iris triples, $\{P\}\ e\ \{Q\}_{\mathcal{M}}$, are also annotated with an invariant mask, $\mathcal{M}$, indicating which invariants are required to hold before, during and after the execution of e. The atomic rule-of-consequence allows us to change this mask to open invariants for the duration of an atomic expression e. View-shifts include implication ($\Box(p \Rightarrow q) \vdash p \Rrightarrow q$) and we can thus recover the usual rule-of-consequence from ACsq.

The "later" modality, $\triangleright$, is used to express that a property is only required to hold after one step of execution. It is used in connection with invariants because an Iris invariant may contain *any* predicate P, including one referring to the invariant itself. To ensure this is well-defined, Iris uses a form of guarded recursion, an abstract version of step-indexing — where $\triangleright$ is used to *guard* the resource in the invariant. Since the later modality expresses that a predicate holds after one step of execution, we can remove a $\triangleright$ modality from a precondition whenever our program makes an operational step. This is captured by the frame rule for atomic expressions:

$$\frac{\{P\}\ e\ \{Q\} \qquad e\ \text{atomic}}{\{P * \triangleright R\}\ e\ \{v.\ Q(v) * R\}}\ \textsc{AFrame}$$

For many assertions it is also possible to remove laters without an operational step. We call these assertions *timeless* as they are independent of the number of steps left. For timeless assertions P we can view-shift away laters: $\triangleright P \Rrightarrow P$. Timeless assertions are closed under the connectives and quantifiers of first-order separation logic, but crucially does not include invariant assertions $\boxed{P}^{\iota}$, as the steps are precisely needed to model potentially self-referential invariants. With the exception of the reference invariants we use, all the invariants used throughout this article are timeless.

While the invariant names ι on invariant assertions, view shifts and Hoare triples are important for soundness, they are not important for understanding our encodings of logical relations in Iris. We have therefore chosen to elide all invariant names in the article, and refer interested readers to the Appendix, where everything is fully annotated.

Logical relation. We now have enough logical machinery to define the first unary logical relation in Iris. The full definition of LR_{ML} is given in Figure 6.

The value relation, $[\![\tau]\!]$, is defined by induction on τ and defines an Iris assertion of type $\mathbf{Val} \to \mathbf{Prop}$. The expression relation, $\mathcal{E}$, is defined independently of the value relation and takes an arbitrary value predicate and extends it to expressions. It has the following type in Iris:

$$\mathcal{E} : (\mathbf{Val} \to \mathbf{Prop}) \to (\mathbf{Exp} \to \mathbf{Prop})$$

As already mentioned, for ground types, $[\![\tau]\!]$ is simply the set of values of the given type τ. The definition for arrow-types follows the usual idea of related arguments to related values, with the added wrinkle that we only require the argument to be related later. This suffices since applying a function takes a step in the operational semantics. The always modality in the value relation for arrow types is there to ensure the value relation is pure, which allows us to duplicate the resource that witnesses that a value is well-typed. It is needed in the arrow-case as implication does not preserve purity in general. For space reasons we omit the cases for products, sum types, and **int** and refer the reader to the Appendix.

Finally, for reference types, **ref** τ, the value relation is the set of locations l such that there exists an invariant that owns the location l and contains a value v in $[\![\tau]\!]$. Resources owned by invariants are shared, which allows all concurrently executing threads to freely update references, provided they respect the typing of the reference. This type of invariant can be seen as a particularly simple instance of rely / guarantee reasoning, where the rely and the guarantee are the same: namely, to preserve the invariant. A large part of the challenge throughout the rest of this article boils down to refining this invariant to limit possible interference from the environment, based on the region and effect system.

The expression relation $\mathcal{E}(\phi)$ extends a value predicate ϕ to expressions e by requiring that, if e terminates, then it terminates with a value satisfying ϕ. Finally, $\Gamma \models_{\text{ML}} e : \tau$ extends this to open expressions, by closing under all substitutions. This semantic typing judgment is sound with respect to the usual typing rules, in the sense that for any well-typed term $\Gamma \vdash e : \tau$, the Iris assertion $\Gamma \models_{\text{ML}} e : \tau$ is provable in Iris.

Lemma 1 (Soundness)**.** *If* $\Gamma \vdash e : \tau$ *then* $\vdash_{\text{Iris}} \Gamma \models_{\text{ML}} e : \tau$.

We note in passing that this logical relation shows the power of using Iris as a meta-language for defining logical relations: Usually, to define logical relations for a language with general references, one would need to index semantic types by worlds containing semantic types for allocated locations and the worlds and semantic types would have to be recursively defined [1, 10]; here this is all taken care of by Iris' built-in general logical facility for defining and working with invariants.

$$[\![\mathbf{1}]\!] \triangleq \lambda x.\ x = () \qquad [\![\mathbf{int}]\!] \triangleq \lambda x.\ x \in \mathbb{N}$$

$$\textsc{Ref}(x, \phi) \triangleq \exists v.\ x \mapsto v * \phi(v) \qquad [\![\mathbf{ref}\ \tau]\!] \triangleq \lambda x.\ \boxed{\textsc{Ref}(x, [\![\tau]\!])}$$

$$[\![\tau_1 \to \tau_2]\!] \triangleq \lambda x.\ \Box \forall y.\ (\triangleright [\![\tau_1]\!](y)) \Rightarrow \mathcal{E}([\![\tau_2]\!])(x\ y)$$

$$\mathcal{E}(\phi)(e) \triangleq \{\top\}\ e\ \{v.\ \phi(v)\}_{\top}$$

Logical relatedness

$$\overline{x : \tau} \models_{\text{ML}} e : \tau \triangleq \ \vdash_{\text{Iris}} \forall \overline{x'}.\, [\![\tau]\!](\overline{x'}) \implies \mathcal{E}([\![\tau]\!])(e[\overline{x'}/\overline{x}])$$

Figure 6. LR_{ML}: Unary rel. for $\lambda_{ref,conc}$ sans effect-types.

3.2 Unary Relation for $\lambda_{ref,conc}$ with Effects

In this section we extend the unary relation from the previous section to the full type system with regions and effects. Note that simply extending the relation from the previous section to the full type system by ignoring all region and effect annotations already yields a relation that is sound with respect to the full type system. However, this is needlessly conservative and by interpreting region and effect annotations as restricting interference, we obtain a more precise semantic typing relation that is also sound.

The idea is to use the distinction between public and private regions to limit interference from the context, and the effect annotations to limit the effects of the given expression. We can encode this in Iris using tokens indexed by a region r corresponding to each type of effect: $[\textsc{Rd}]^{\pi}_r$, $[\textsc{Wr}]^{\pi}_r$, $[\textsc{Al}]^{\pi}_r$. Each token is intended to grant permission to perform the corresponding effect on region r and, depending on the fractional permission $\pi \in \{\pi \in \mathbb{Q} \mid 0 < \pi \leq 1\}$, prevent the context from performing the given effect. These tokens must satisfy the following properties (and likewise for Wr,Al):

$$[\textsc{Rd}]^{1}_r * [\textsc{Rd}]^{\pi}_r \Rightarrow \bot \qquad [\textsc{Rd}]^{\pi_1+\pi_2}_r \Leftrightarrow [\textsc{Rd}]^{\pi_1}_r * [\textsc{Rd}]^{\pi_2}_r \qquad (1)$$

expressing that the full permission ($\pi = 1$) really means exclusive ownership of the token and that these tokens can be split and recombined arbitrarily. In Iris we can define such tokens using *ghost state*. Below we give a brief introduction to ghost state in Iris; for a more thorough treatment, we refer the reader to [16].

Iris and ghost state. Ghost resources provides a modular way of reasoning about *knowledge* and *rights* to modify some shared state. Ghost state is modeled using partial commutative monoids in Iris. Formally, these partial commutative monoids are presented as total commutative monoids with a distinguished zero element, $\bot$. The assertion $\boxed{m : M}^{\gamma}$ asserts ownership of a ghost resource $m \in |M|$ of the monoid instance γ. Separating conjunction on ghost state is simply the lifting of the underlying monoid composition:

$$\boxed{m_1 : M}^{\gamma} * \boxed{m_2 : M}^{\gamma} \Leftrightarrow \boxed{m_1 \cdot_M m_2}^{\gamma} \qquad (2)$$

The zero-element represents an ill-defined resource and thus cannot be owned: $\boxed{\bot : M}^{\gamma} \Leftrightarrow \mathbf{false}$.

Tokens are a degenerate form of ghost state, consisting only of rights. The Frac monoid, defined below, allows us to define an effect token for a single region. The carrier is rationals between 0 and 1, with addition as composition and 0 as the unit (we typically omit the explicit zero element

from the definition of the monoid carrier and composition):

$$\textsc{Frac} = [0,1] \cap \mathbb{Q} \qquad q \cdot q' = q + q', \quad \text{if } q + q' \leq 1$$

The idea is that 0 represents no ownership, 1 exclusive ownership and anything rational in the interval $(0,1)$ non-exclusive ownership.

To define effect tokens for arbitrary regions, we also need the partial finite function monoid, $\textsc{FpFun}(X, M)$, whose carrier is functions f from a set X into the non-zero elements a monoid M, such that the set $\{x \in X \mid f(x) \neq \varepsilon\}$ is finite. Composition on $\textsc{FpFun}(X, M)$ is defined pointwise, but is only defined if all pointwise compositions are well-defined:

$$(f \cdot g)(x) \triangleq f(x) \cdot g(x) \quad \text{if } f(x) \cdot g(x) \neq \bot \text{ for all } x \in X$$

Effect tokens can now be defined as follows and proven to satisfy property (1). The proof is an easy consequence of (2) and the definition of the monoid.

$$[\mathrm{X}]^\pi_r \triangleq \boxed{[r \mapsto \pi] : \textsc{FpFun}(\mathcal{RN}, \textsc{Frac})}^X, \; X \in \{\textsc{Rd}, \textsc{Wr}, \textsc{Al}\}$$

Ghost state is a purely logical construct and is updated using view-shifts rather than assignments. To update a ghost resource we must ensure that our update is consistent what all ghost resources potentially owned by the environment. This is captured by the GhostUpd rule given below:

$$\textsc{GhostUpd} \quad \frac{\forall m_f.\, m \cdot m_f \neq \bot \Rightarrow \exists m' \in M'.\, m' \cdot m_f \neq \bot}{\boxed{m : M}^\gamma \Rrightarrow \exists m' \in M'.\, \boxed{m' : M}^\gamma}$$

To update our ghost resource m to some element $m' \in M'$, we have to show that doing so preserves all possible frames m_f composable with our resource m.

We can instantiate the finite partial functions monoid with locations and values to obtain the standard monoid of heaps used in separation logic. To define a monoid on values, we extend it with a unit element and a composition operator that is only defined if one of the two elements is unit.

$$\textsc{Heap} \triangleq \textsc{FpFun}(\textsc{Loc}, \textsc{Val} + \{\varepsilon\})$$

The ghost resource $\boxed{[l \mapsto v] : \textsc{Heap}}^\gamma$ asserts the exclusive right to modify location l in ghost heap γ and that location l currently contains the value v (here we use $[l \mapsto v]$ for the function that maps l to v and every other argument to ε). Using the GhostUpd rule, we can update ghost locations we own and allocate new ghost locations:

$$\boxed{[l \mapsto v] : \textsc{Heap}}^\gamma \Rrightarrow \boxed{[l \mapsto v'] : \textsc{Heap}}^\gamma \tag{3}$$

$$\boxed{[] : \textsc{Heap}}^\gamma \Rrightarrow \exists l.\, \boxed{[l \mapsto v] : \textsc{Heap}}^\gamma \tag{4}$$

Throughout the rest of the article we will need many ghost resources, including the Heap monoid. We will introduce them by explaining the properties we expect them to satisfy. Naturally, we must define monoids for all of these resources and prove that the desired properties hold. All of these definitions and proofs can be found in the Appendix.

Encoding effects using ghost state. Now that we have seen how to define and work with ghost state in Iris, we proceed with how to encode effects using ghost state.

A read-effect on a private region translates into exclusive ownership of the corresponding read token, while a read-effect on a public region translates into ownership of the corresponding read token with an arbitrary fractional permission π (and likewise for write and allocation effects).

The intended meaning of these tokens is enforced through the interplay between two invariants. A new *region* invariant, $\textsc{Reg}(r)$, linking references with their corresponding region, and an updated reference invariant, $\textsc{Ref}(r, \phi, x)$, indexed by a region identifier r and the reference's semantic type ϕ. Before we define these formally, we review some properties that should hold. If we own part of the read token for a region r then the context knows we might read references belonging to this region and must ensure that their values are well-typed. This is captured by the following property (where all free variables are universally quantified):

$$\boxed{\textsc{Reg}(r)}, \boxed{\textsc{Ref}(r, \phi, x)} \vdash \{[\textsc{Rd}]^\pi_r\}\, !x \{y.\, [\textsc{Rd}]^\pi_r * \phi(y)\} \tag{5}$$

Preservation of well-typedness is expressed by $\phi(y)$ in the post-condition. If we own part of the write token for a region r then we should be allowed to write any well-typed value to a reference belonging to region r:

$$\boxed{\textsc{Reg}(r)}, \boxed{\textsc{Ref}(r, \phi, x)}, \phi(v) \vdash \{[\textsc{Wr}]^\pi_r\} x := v \{y.\, [\textsc{Wr}]^\pi_r\} \tag{6}$$

Likewise, if we own any part of the allocation token for a region r we should be allowed to allocate a new reference and associate it with region r:

$$\boxed{\textsc{Reg}(r)}, \phi(v) \vdash \{[\textsc{Al}]^\pi_r\}\ \textbf{new}\ v\ \{y.\, [\textsc{Al}]^\pi_r * \boxed{\textsc{Ref}(r, \phi, y)}\}$$

Those three properties were fairly uneventful; the interesting properties deal with exclusive ownership of effect tokens.

Exclusive read effect. If we own the read token for region r exclusively, then the context cannot rely on references in region r containing well-typed values. If we additionally own a write token for region r, then we should be allowed to assign arbitrary values to references belonging to region r, provided we restore them with well-typed values before returning the exclusive read token. To capture this formally, we introduce two new tokens, $[\textsc{Rd}(x)]_r$ and $[\textsc{NoRd}(x)]_r$, which express that if location x belongs to region r then it contains a well-typed value and may contain a value that is not well-typed, respectively. If we own the read token on a region r exclusively, then the following property allows us to exchange it for tokens that force all locations belonging to region r to contain well-typed values.

$$\boxed{\textsc{Reg}(r)} \vdash [\textsc{Rd}]^1_r \Leftrightarrow \circledast_x [\textsc{Rd}(x)]_r \tag{7}$$

By giving up the token that expresses that a location contains a well-typed value, we can assign an arbitrary value to the location. If we later assign a well-typed value, we can recover the token witnessing the well-typedness of the location. This is captured by the following two properties.

$$\boxed{\textsc{Reg}(r)}, \boxed{\textsc{Ref}(r, \phi, x)} \tag{8}$$
$$\vdash \{[\textsc{Wr}]^\pi_r * [\textsc{Rd}(x)]_r\}\ x := v\ \{y.\, [\textsc{Wr}]^\pi_r * [\textsc{NoRd}(x)]_r\}$$

$$\boxed{\textsc{Reg}(r)}, \boxed{\textsc{Ref}(r, \phi, x)}, \phi(v) \tag{9}$$
$$\vdash \{[\textsc{Wr}]^\pi_r * [\textsc{NoRd}(x)]_r\}\ x := v\ \{y.\, [\textsc{Wr}]^\pi_r * [\textsc{Rd}(x)]_r\}$$

Exclusive write effect. If we own the full write token, $[\textsc{Wr}]^1_r$, then the context should not be allowed to modify references belonging to region r. Again, we capture this property by introducing new tokens $[\textsc{Wr}(x)]_r$ and $x \overset{\pi}{\hookrightarrow}_r v$. Both tokens express that if location x belongs to region r, then we own the exclusive right to update it; the latter token further asserts that the current value is v. As before, we can trade ownership of a per-region write token for region r for all per-location write tokens for region r:

$$\boxed{\textsc{Reg}(r)} \vdash [\textsc{Wr}]^1_r \Leftrightarrow \circledast_x [\textsc{Wr}(x)]_r \tag{10}$$

Given ownership of a per-location write token for a location x belonging to region r, we can trade the token for a *points-*

to proxy for x with fractional permission $\frac{1}{2}$:

$$\boxed{\textsc{Ref}(r,\phi,x)} \vdash [\textsc{Wr}(x)]_r \Longleftrightarrow \exists v.\, x \overset{\frac{1}{2}}{\hookrightarrow}_r v \tag{11}$$

This points-to proxy satisfies similar properties as the standard separation logic points-to: If we own the points-to proxy $x \overset{\pi}{\hookrightarrow}_r v$ and read location x, we will read the value v:

$$\boxed{\textsc{Reg}(r)} \vdash \{x \overset{\pi}{\hookrightarrow}_r v\}\ !x\ \{y.\, y = v * x \overset{\pi}{\hookrightarrow}_r v\} \tag{12}$$

If we own half of the points-to proxy for a location x we can also use it to assign a well-typed value to x:

$$\boxed{\textsc{Reg}(r)}, \boxed{\textsc{Ref}(r,\phi,x)}, \tag{13}$$
$$\phi(v_2) \vdash \{x \overset{\frac{1}{2}}{\hookrightarrow}_r v_1\}\ x := v_2\ \{y.\, x \overset{\frac{1}{2}}{\hookrightarrow}_r v_2\}$$

Exclusive ownership of the per-region write token thus allows us to reason about the exact value of all references belonging to the region.

Exclusive allocation effect. Exclusive ownership of a per-region allocation token allows us to lock the domain of the heap associated with the given region. By trading our exclusive per-region allocation token, we can take ownership of a new token, $[\textsc{Al}(h)]^{\pi}_r$, that witnesses the domain of the heap associated with the given region:

$$\boxed{\textsc{Reg}(r)} \vdash [\textsc{Al}]^r_1 \Longleftrightarrow \exists h.\, [\textsc{Al}(h)]^{\frac{1}{2}}_r \tag{14}$$

As usual, we use fractional permissions to share the $[\textsc{Al}(h)]^{\pi}_r$ token. Given fractional ownership of two parts of the $[\textsc{Al}(h)]^{\pi}_r$ token, the domains of the two heaps must agree:

$$[\textsc{Al}(h_1)]^{\pi_1}_r * [\textsc{Al}(h_2)]^{\pi_2}_r \Rightarrow [\textsc{Al}(h_1)]^{\pi_1}_r * [\textsc{Al}(h_2)]^{\pi_2}_r * \mathrm{dom}(h_1) = \mathrm{dom}(h_2)$$

Exclusive ownership ($\pi = 1$) is required to update the domain of the heap. It is possible to update the heap without exclusive access, as long as the domain is preserved:

$$[\textsc{Al}(h)]^1_r \Rrightarrow [\textsc{Al}(h')]^1_r$$
$$[\textsc{Al}(h)]^{\pi}_r * \mathrm{dom}(h) = \mathrm{dom}(h') \Rrightarrow [\textsc{Al}(h')]^{\pi}_r$$

Logical relation. The LR_{Eff} logical relation, including the new region invariant and updated reference invariant is defined in Figure 7. Changes to existing predicates are in green. The value relation, $[\![\tau]\!]^M$, is now indexed by an injective mapping M from region variables to Iris invariant names. This mapping allows us to model that the same region variable might refer to different regions in the case where a region ρ is created after hiding a region with the same name ρ. Likewise, the expression relation, $\mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}(\phi)$, is also indexed by the region mapping M, in addition to the region contexts Π, Λ and the effect typing ε.

To explain the relation, lets start with the reference invariant $\textsc{Ref}(r,\phi,x)$. Note first that the reference invariant no longer owns the underlying physical location (i.e., $x \mapsto v$). Instead it owns a proxy $x \overset{\frac{1}{2}}{\hookrightarrow}_r v$. The *effs* predicate encodes the meaning of the per-location read and write tokens. It allows us to exchange a write token $[\textsc{Wr}(x)]_r$ for a proxy that describes the current value of the location (property (11)) and track when the location contains a well-typed value (properties (8) and (9)).

The region invariant $\textsc{Reg}(r)$ consists of two resources, a token resource $toks(r)$ that ties all the per-region tokens together with the per-location tokens, and the $locs(r)$ resource that ties together the points-to proxies with the physical

New predicates

$$\begin{aligned}
effs(r,\phi,x,v) &\triangleq ([\textsc{Wr}(x)]_r \vee x \overset{\frac{1}{2}}{\hookrightarrow}_r v) * \\
&\quad ([\textsc{Rd}(x)]_r \vee (\phi(v) * [\textsc{NoRd}(x)]_r)) \\
\textsc{Reg}(r) &\triangleq locs(r) * toks(r) \\
locs(r) &\triangleq \exists h.\, rheap(h,r) * alloc(h,r) * \circledast_{(l,v)\in h} l \mapsto v \\
&\quad \circledast_{\{x \mid x \in (Loc \setminus \mathrm{dom}(h))\}} [\textsc{NoRd}(x)]_r \\
toks(r) &\triangleq ([\textsc{Rd}]^1_r \vee \circledast_{x\in Loc}[\textsc{Rd}(x)]_r) * \\
&\quad ([\textsc{Wr}]^1_r \vee \circledast_{x\in Loc}[\textsc{Wr}(x)]_r) \\
alloc(h,r) &\triangleq ([\textsc{Al}]^1_r * [\textsc{Al}(h)]^{\frac{1}{2}}_r) \vee [\textsc{Al}(h)]^1_r \\
P_{toks}(\rho,r,\pi,\varepsilon) &\triangleq (\rho \notin \mathsf{rds}\ \varepsilon \vee [\textsc{Rd}]^{\pi}_r) * (\rho \notin \mathsf{wrs}\ \varepsilon \vee [\textsc{Wr}]^{\pi}_r) * \\
&\quad (\rho \notin \mathsf{als}\ \varepsilon \vee [\textsc{Al}]^{\pi}_r) \\
P_{reg}(R,g,\varepsilon,M) &\triangleq \circledast_{\rho\in R} P_{toks}(\rho, M(\rho), g(\rho), \varepsilon) * \boxed{\textsc{Reg}(M(\rho))} \\
P^{\Pi,\Lambda}(g,\varepsilon,M) &\triangleq P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M) * P_{reg}(\Pi, g, \varepsilon, M)
\end{aligned}$$

Changes to previous definitions

$$\begin{aligned}
\textsc{Ref}(r,\phi,x) &\triangleq \exists v.\, x \overset{\frac{1}{2}}{\hookrightarrow}_r v * effs(r,\phi,x,v) \\
[\![\tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M &\triangleq \lambda x.\, \Box\forall y.\, (\triangleright y \in [\![\tau_1]\!]^M) \Rightarrow \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau_2]\!]^M)(x\ y) \\
[\![\mathsf{ref}_\rho\ \tau]\!]^M &\triangleq \lambda x.\, \boxed{\textsc{Ref}(M(\rho), [\![\tau]\!]^M, x)} \\
\mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}(\phi)(e) &\triangleq \forall g. \\
&\quad \{P^{\Pi,\Lambda}(g,\varepsilon,M)\}\ e\ \{v.\, \phi(v) * P^{\Pi,\Lambda}(g,\varepsilon,M)\}_{\top}
\end{aligned}$$

Logical relatedness

$$\Pi \mid \Lambda \mid \overline{x:\tau} \models_{\text{Eff}} e : \tau, \varepsilon \triangleq$$
$$\vdash_{\text{Iris}} \forall M. \forall \overline{x'}.\, [\![\tau]\!]^M(\overline{x'}) \implies \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau]\!]^M)(e[\overline{x'}/\overline{x}])$$

Figure 7. LR_{Eff}: Unary rel. for $\lambda_{ref,conc}$ with effect-types.

state. The $toks(r)$ resource allows us to exchange an exclusive per-region read or write token for all the corresponding per-location read or write tokens (properties (7) and (10)). It also enforces that if we only own a fraction of the per-region read or write token then the region invariant must own all per-location read and write tokens for the given region. This ensures that the location must contain a well-typed value and that we are allowed to update it, respectively (properties (5) and (6)).

The local points-to proxies for a region r are tied to the physical state using the global counter-part $rheap(h,r)$ resource in $locs(r)$. The local points-to proxy always agrees with the global heap proxy:

$$rheap(h,r) * x \overset{\pi}{\hookrightarrow}_r v \Rrightarrow rheap(h,r) * x \overset{\pi}{\hookrightarrow}_r v * h(x) = v$$

To update a points-to proxy thus requires ownership of both the corresponding global heap proxy and exclusive ownership of the local points-to proxy:

$$rheap(h,r) * x \overset{1}{\hookrightarrow}_r v \Rrightarrow rheap(h[x \mapsto v'], r) * x \overset{1}{\hookrightarrow}_r v'$$

The $locs(r)$ resource asserts ownership of physical points-to resources for each location and value in the global points-to proxy for region r. Since these are tied together with the local points-to proxies, the local points-to proxies must agree with the underlying physical state (properties (12) and

(13)). $locs(r)$ also asserts ownership of all the $[\text{NoRd}(x)]_r$ resources for locations x not belonging to the region.

The reason for introducing the indirection of proxies, is to allow reasoning about the set of locations belonging to a region, to interpret allocation effects. This is captured by the $alloc(h, r)$ resource, which allows clients to trade the exclusive allocation token for a lock on the set of locations belonging to the region (property (14)).

Finally, P_{reg} specifies how the effect annotation translates into ownership of the corresponding effect tokens. Effects in private regions yield exclusive ownership, while effects in public regions yield non-exclusive ownership.

Example: type violating update. To illustrate how we can use this stronger semantic typing judgment to semantically type-check code that is not syntactically well-typed, recall the previous mentioned type-violating example.

$$- \mid \rho \mid x : \mathsf{ref}_\rho\ \mathsf{B} \models_{\text{Eff}} x := (); x := \mathsf{true} : \mathbf{1}, \{rd_\rho, wr_\rho\}$$

The read and write effect on the private region ρ translates into exclusive ownership of the read and write token. Using properties (7), (8) and (9) we can thus easily verify that the example is semantically well-typed.

Context: $\boxed{\text{Reg}(r)}, \boxed{\text{Ref}(r, [\![\mathbf{B}]\!]^M, x)}$
$\left\{[\text{Wr}]^1_r * [\text{Rd}]^1_r\right\} \Rrightarrow \left\{[\text{Wr}]^1_r * \circledast_{y \in Loc} [\text{Rd}(y)]_r\right\}$
$x := ()$
$\left\{[\text{Wr}]^1_r * [\text{NoRd}(x)]_r * \circledast_{y \in Loc \setminus \{x\}} [\text{Rd}(y)]_r\right\}$
$x := \mathsf{true}$
$\left\{[\text{Wr}]^1_r * \circledast_{y \in Loc} [\text{Rd}(y)]_r\right\} \Rrightarrow \left\{[\text{Wr}]^1_r * [\text{Rd}]^1_r\right\}$

3.3 Binary Relation for $\lambda_{ref,conc}$ with Effects

Previously we looked at unary relations for semantically characterizing type inhabitance. Now, we switch to binary relations intended to imply contextual approximation.

We define two families of binary relations, $[\![\tau]\!]^M$ and $\mathcal{E}([\![\tau]\!]^M)$, that characterize contextual approximation on values and expressions of type τ, respectively. Generalizing the value relation to contextual approximation is fairly straightforward: on ground types it is simply the identity relation on the values of the given type; for arrow types it relates functions that map related arguments to related expressions, and for reference types it relates two locations if they contain related values.

The expression relation is more interesting: intuitively it should express that e_I approximates e_S, if any step that e_I can make can be simulated by zero or more steps of e_S. We think of e_I as an "implementation" and of e_S as a "specification". We follow the approach of Turon et. al. [30] and capture this relational property as a unary Hoare triple on e_I by requiring the triple to update ghost resources that force the execution of e_S. The idea is to introduce a ghost resource $j \Rightarrow_S e$ that expresses that the expression e is in an evaluation context on the "specification" side and the exclusive right to reduce this expression. With this ghost resource we can express a simulation between an implementation e_I and a specification e_S as follows:

$$e_I \le e_S \approx \{j \Rightarrow_S e_S\}\ e_I\ \{v_I. \exists v_S.\, j \Rightarrow_S v_S * \phi(v_I, v_S)\}$$

By requiring e_I to update the ghost resource from e_S to a value v_S, we are forced to show that we can reduce e_S, which appears in an evaluation context on the specification side, to the value v_S. We refer to $j \Rightarrow_S e$ as a local expression resource, as it allows us to reason locally about reductions on sub-expressions of the full specification program.

New predicates

$$\text{Spec}(h_0, e_0) \triangleq \exists h, e.\, heap_S(h) * mctx(e) * (h_0; e_0 \to^* h; e)$$

Changes to previous definitions

$$\text{Ref}(r, \phi, x) \triangleq \exists v.\ x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v_S * \mathit{effs}(r, \phi, x, v)$$

$$\mathit{effs}(r, \phi, x, v) \triangleq ([\text{Wr}(x)]_r \vee (x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v_S)) * ([\text{Rd}(x)]_r \vee (\phi(v_I, v_S) * [\text{NoRd}(x)]_r))$$

$$locs(r) \triangleq \exists h.\, rheap_I(h_I, r) * rheap_S(h_S, r) * alloc(h, r) \circledast_{(l,v) \in h_I} l \mapsto_I v * \circledast_{(l,v) \in h_S} l \mapsto_S v \circledast_{x \in Loc \setminus \text{dom}(h_I) \times (Loc \setminus \text{dom}(h_S))} [\text{NoRd}(x)]_r$$

$$tokens(r) \triangleq ([\text{Wr}]^1_r \vee \circledast_{x \in Loc^2} [\text{Wr}(x)]_r) * ([\text{Rd}]^1_r \vee \circledast_{x \in Loc^2} [\text{Rd}(x)]_r)$$

$$alloc(h, r) \triangleq ([\text{Al}]^1_r * [\text{Al}(h_I, h_S)]^{\frac{1}{2}}_r) \vee [\text{Al}(h_I, h_S)]^1_r$$

$$[\![\mathbf{1}]\!]^M \triangleq \lambda x.\ x_I = x_S = ()$$

$$[\![\mathsf{int}]\!]^M \triangleq \lambda x.\ x_I, x_S \in \mathbb{N} \wedge x_I = x_S$$

$$[\![\tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M \triangleq \lambda x.\ \Box \forall y.\ (\triangleright [\![\tau_1]\!]^M)(y_I, y_S) \Rrightarrow \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau_2]\!]^M)(x_I\ y_I, x_S\ y_S)$$

$$[\![\mathsf{ref}_\rho\ \tau]\!]^M \triangleq \lambda x.\ \boxed{\text{Ref}(M(\rho), [\![\tau]\!]^M, x)}$$

$$\mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}(\phi)(e_I, e_S) \triangleq \forall g, j, h_0, e_0.\ \boxed{\text{Spec}(h_0, e_0)} \vdash \{j \Rightarrow_S e_S * P^{\Pi,\Lambda}(g, \varepsilon, M)\}\ e_I \{v_I.\ \exists v_S.\ j \Rightarrow_S v_S * \phi(v_I, v_S) * P^{\Pi,\Lambda}(g, \varepsilon, M)\}_\top$$

Logical relatedness

$$\Pi \mid \Lambda \mid \overline{x : \tau} \models_{\text{Bin}} e_1 \le_{log} e_2 : \tau, \varepsilon \triangleq \vdash_{\text{Iris}} \forall M. \forall \overline{x}. [\![\tau]\!]^M(\overline{x}) \Rightarrow \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau]\!]^M)(e_1[\overline{x_I}/\overline{x}], e_2[\overline{x_S}/\overline{x}])$$

Figure 8. LR_{Bin}: Binary rel. for $\lambda_{ref,conc}$ with effect-types.

The generalization to expressions in evaluation contexts is necessary to prove that the relation is a congruence. In particular, to prove the following congruence property:

$$e_{1I} \le e_{1S} \wedge e_{2I} \le e_{2S} \Rightarrow e_{1I} || e_{2I} \le e_{1S} || e_{2S}$$

We need to be able to split the local expression resource $j \Rightarrow_S e_{1S} || e_{2S}$ into two separate resources, one for e_{1S} and another for e_{2S}. Then we can pass one to e_{1I} and the other to e_{2I} and they can each reduce their corresponding expression on the right, independently of the other. This is possible because e_{1S} and e_{2S} both occur in evaluation contexts. This is also the reason why local expression resources $j \Rightarrow_S e$ are indexed. The j serves as a logical "thread identifier", allowing us to distinguish different local expression resources.

Specification resources. To formalize this idea, we need a number of ghost resources. In addition to the local expression resource, $j \Rightarrow_S e$ we also need a global expression resource, $mctx(e)$, for reasoning about the full specification program. Naturally, the global expression resource and all local expression resources must agree on the specification program, so splitting a local expression resource requires ownership of both. The following lemma allows us to introduce and eliminate a local expression resource for an expression that occurs in an evaluation context inside another

local expression resources:

$$j \Rightarrow_S \kappa[e_1] * mctx(e) \Longleftrightarrow \exists i.\, j \Rightarrow_S \kappa[i] * i \Rightarrow_S e_1 * mctx(e) \tag{15}$$

This is achieved by introducing a new logical thread identifier i for the new local expression resource for e_1 and replacing e_1 with i in the original local expression resource. Here κ is an evaluation context extended to expressions that may contain logical thread identifiers. By applying the above property twice, we can split a local expression resource for a parallel composition into two:

$$j \Rightarrow_S e_1||e_2 * mctx(e) \Longleftrightarrow \exists i_1, i_2.\\ j \Rightarrow_S i_1||i_2 * i_1 \Rightarrow_S e_1 * i_2 \Rightarrow_S e_2 * mctx(e)$$

Since all local specification expressions are in an evaluation context of the global specification expression, any reduction of a local specification expression can be extended to the global specification expression:

$$j \Rightarrow_S e_1 * mctx(e) * (h; e_1 \to h'; e_1') \Rrightarrow \\ \exists e'.\, j \Rightarrow_S e_1' * mctx(e') * (h; e \to h'; e') \tag{16}$$

In the case where there exists just one local expression resource that contains no free logical thread identifiers, then the local expression should agree with the global expression. To formalize this, we treat the thread identifier 0 as the "root" local expression:

$$0 \Rightarrow_S e_1 * mctx(e_2) * FA(e_1) = \emptyset \;\Rightarrow\\ 0 \Rightarrow_S e_1 * mctx(e_2) * e_1 = e_2 \tag{17}$$

where $FA(e)$ is the set of free logical thread identifiers in e. These local and global expression resources are definable in Iris and we refer the reader to the Appendix for detailed definitions.

We need another two ghost resources, $heap_S(h)$ and $l \mapsto_S v$, for reasoning about specification heaps. This is in fact the HEAP monoid we have seen before, with some additional structure. The $heap_S(h)$ resource asserts global ownership of the full specification heap h, while $l \mapsto_S v$ asserts local ownership of a single location l, respectively. We require that the global heap agrees with the local heap resources:

$$heap_S(h) * l \mapsto_S v \Rrightarrow heap_S(h) * l \mapsto_S v * h(l) = v \tag{18}$$

Updating a location l requires both local ownership of l and the global heap resource and allocating require ownership of the global heap resource, both lifted from updating and allocating ghost locations seen before in (3) and (4).

$$heap_S(h) * l \mapsto_S v \Rrightarrow heap_S(h[l \mapsto v']) * l \mapsto_S v' \tag{19}$$

$$heap_S(h) * l \notin \mathrm{dom}(h) \Rrightarrow heap_S(h[l \mapsto v]) * l \mapsto_S v \tag{20}$$

With these resources in hand, we can now formally define a simulation as a Hoare triple. We define a specification invariant that asserts ownership of the global specification heap and expression. Additionally, it also requires that there exists a reduction from some initial configuration $h_0; e_0$ to the current global specification heap and expression:

$$\textsc{Spec}(h_0, e_0) \triangleq \exists h, e.\, heap_S(h) * mctx(e) * (h_0; e_0 \to^* h; e)$$

By requiring the Hoare triple to update the local expression e_S of a to a value v_S, we thus force it to show the existence of a reduction:

$$\boxed{\textsc{Spec}(h_0, e_0)} \vdash \{j \Rightarrow_S e_S\}\ e_I\ \{v_I.\, \exists v_S.\, j \Rightarrow_S v_S\}$$

The only way to update the local expression resource $j \Rightarrow_S e$ is through property (16) which also requires opening and reestablishing the specification invariant to gain access to the global expression resource.

The logical relation. Now that we have seen how we can express relational properties as unary Hoare triples, we just need to integrate this idea with the techniques from the previous section for translating region and effect annotations into specifications of abstract interference.

Consider the reference invariant, $\textsc{Ref}(r, \phi, x)$. In the unary setting it asserts ownership of a proxy for the underlying heap location that, depending on ownership of the per-location read and write tokens, contains a well-typed value and may be updated. In the binary setting, x is now a pair of locations (x_I, x_S) and the invariant asserts ownership of proxies for both the implementation and specification side heaps, but otherwise the structure of the definition remains the same. The binary reference invariant is defined in Figure 8. We use x_I and x_S as shorthand for the first and second projection of a pair x. Note that, in the binary setting, per-location read, write and no-read tokens are now indexed by a pair of locations, rather than just a single location. The $[\textsc{Rd}(x_I, x_S)]_r$ token now expresses that if locations x_I and x_S are related and belong to region r, then they contain related values, and likewise for the other tokens.

The $\mathrm{LR}_{\textsc{Bin}}$ logical relation satisfies the fundamental theorem of logical relations (Theorem 4), which expresses that all well-typed terms are related to themselves. It is also sound with respect to contextual approximation (Theorem 5).

Theorem 4 (Fundamental Theorem)**.** *If* $\Pi \mid \Delta \mid \Gamma \vdash e : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \models_{\textsc{Bin}} e \leq_{log} e : \tau, \varepsilon$

Theorem 5 (Soundness)**.** *If* $\Pi \mid \Delta \mid \Gamma \models_{\textsc{Bin}} e_I \leq_{log} e_S : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \vdash e_I \leq_{ctx} e_S : \tau, \varepsilon$.

3.4 Binary Relation for $\lambda_{ref,conc}$ with Effects Using Multiple Simulations

The $\mathrm{LR}_{\textsc{Bin}}$ relation supports proofs of contextual approximations by showing that each step on the left can be simulated on the right. However, it requires that each thread on the left owns the local expression resource of the thread on the right that simulates the thread on the left. This is too restrictive in cases where multiple threads on the left are simulated by a single thread on the right, such as the case of parallelization. In this section we introduce our final logical relation, $\mathrm{LR}_{\textsc{Par}}$, that removes this restriction.

The idea is simple: The $\mathrm{LR}_{\textsc{Bin}}$ relation allowed us to reason about a single simulation; now, we generalize the relation to allow reasoning about multiple simulations, such that multiple threads on the left can be given ownership of an expression resource for the same thread on the right, in different simulations.

Multiple simulations. To make this precise, we generalize the existing specification ghost resources, so that we can have multiple independent copies, by indexing the global and local expression resources ($mctx(e, \zeta)$ and $j \overset{\zeta}{\Rightarrow}_S e$) and heap resources ($heap_S(h, \zeta)$ and $l \overset{\zeta}{\mapsto}_S v$) with a simulation identifier ζ. For each simulation identifier ζ, the resources $mctx(e, \zeta)$ and $j \overset{\zeta}{\Rightarrow}_S e$ satisfy the same properties as before (properties (15) to (17)) and likewise for the heap resources (properties (18) and (19)). We can allocate new expression and heap resources initialized with an arbitrary expression e and an empty heap:

$$\top \Rrightarrow \exists \zeta.\, mctx(e, \zeta) * heap_S([], \zeta) * 0 \overset{\zeta}{\Rightarrow}_S e \tag{21}$$

The idea is to relate e_I and e_S if e_S can simulate any step performed by e_I in an arbitrary simulation ζ in which e_S appears in an evaluation context:

$$\forall \zeta, i, h_0, e_0. \boxed{\exists h, e.\, heap_S(h, \zeta) * mctx(e, \zeta) * (h_0; e_0 \to^* h; e)} \\ \vdash \{i \overset{\zeta}{\Rightarrow}_S e_S\}\, e_I\, \{v_I.\, \exists v_S.\, i \overset{\zeta}{\Rightarrow}_S v_S\}$$

This allows the caller of e_I to choose in which simulation ζ the specification e_S must simulate e_I. It also allows e_I to simulate sub-expressions of e_I in different simulations than ζ, provided it can still prove a simulation in ζ at the end.

We can allocate new simulations with in arbitrary initial configuration $h; e$ and take ownership of the local heap and expression resources for this simulation, using (21) and (20).

This ability to simulate sub-expressions in different simulations is exactly what we need to disentangle an execution of $e_1 \,||\, e_2$ into two independent executions of e_1 and e_2, when proving parallelization. To show that $e_1 \,||\, e_2$ is related to (e_1, e_2) in the expression relation, we (roughly) prove the following triple:

$$\boxed{\exists h, e.\, heap_S(h, \zeta) * mctx(e, \zeta) * (h_0; e_0 \to^* h; e)} \vdash \\ \{i \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * \cdots\}\ e_1 \,||\, e_2\ \{v_I.\, \exists v_S.\, i \overset{\zeta}{\Rightarrow}_S v_S * \cdots\}$$

Recall from the Introduction that the idea is to use the effect annotations to prove that an execution of $e_1 \,||\, e_2$ can be disentangled into semi-independent executions of e_1 and e_2.

Since e_1 and e_2 are well-typed, it follows by the fundamental theorem of logical relations that they are related to themselves. To use these assumptions we must pass ownership of a local expression resource to each of e_1 and e_2 with e_1 and e_2 in an evaluation context, respectively. We could use the ζ simulation with e_1 since e_1 is already in an evaluation context in the ζ simulation. However, this leaves us without an expression resource for e_2.

Instead, the idea is to suspend the ζ simulation and create two new simulations ζ_1 and ζ_2 with e_1 as the full specification of the ζ_1 simulation and e_2 as the full specification of the ζ_2 simulation. Then we can appeal to relatedness of e_1 and e_2 to themselves with ζ_1 and ζ_2 as the respective simulations, which will show the existence of the two independent executions of e_1 and e_2. Once $e_1 \,||\, e_2$ has terminated on the left, we can resume the ζ simulation and use the two independent executions of e_1 and e_2 to take the appropriate steps in the ζ simulation.

The reason this works, is the effect annotations, which ensure that e_1 and e_2 are semi-independent. In particular, all locations accessed by both e_1 and e_2 are read-only and can therefore soundly be shared between the ζ_1 and ζ_2 simulations.

Relating heaps in multiple simulations. In previous relations, the region invariant ensured that all specification heap proxies ($l \overset{\pi}{\hookrightarrow}_{S,r} v$) matched the contents of the actual specification heap. With multiple simulations, we have multiple specification heaps. The idea is to allow proxies to be tied to multiple specification heaps, provided we can guarantee that the given references are immutable. In cases where we cannot guarantee immutability, we still only allow proxies to be tied to a single simulation, to ensure we can reason locally about reductions in simulations.

To capture this formally, we introduce a new ghost resource, to specify whether a region is immutable or not, and which simulations the regions proxies are tied to. The $[\textsc{Im}(r, S, h)]^\pi$ resource asserts that region r is immutable and the current specification heap of the region is h, while $[\textsc{Mu}(r, S)]^\pi$ asserts that it is mutable. In both cases the set S specifies which simulations the proxies of region r are tied to. In the mutable state, we require that the set S is a singleton. We call this ghost resource the *specification link* resource.

The fractional permission is used to track whether we are allowed to change the state of a region. If we own a specification link resource exclusively (i.e., $\pi = 1$), then we can change its state between mutable and immutable and which simulations the proxies of the region are tied to.

$$[\textsc{Mu}(r, S)]^1 \Leftrightarrow [\textsc{Im}(r, S', h)]^1 \tag{22}$$

Both tokens can be split arbitrarily using the fraction. Any two fractional immutable tokens must agree on the current heap and which simulations the region is tied to:

$$[\textsc{Im}(r, S_1, h_1)]^{\pi_1} * [\textsc{Im}(r, S_2, h_2)]^{\pi_2} \implies \\ [\textsc{Im}(r, S_1, h_1)]^{\pi_1} * [\textsc{Im}(r, S_2, h_2)]^{\pi_2} * (h_1 = h_2) * (S_1 = S_2) \tag{23}$$

Disjointness of allocations. We need two final bits of ghost state before we can define the full logical relation. Namely, we need a way to control which locations simulations use when allocating new locations.

To facilitate this level of control over locations, we introduce a ghost resource, $[X]$, for asserting ownership of a set of locations X. These can be split and recombined and ensure that disjoint resources refer to disjoint sets of locations:

$$[X_1 \uplus X_2] \Leftrightarrow [X_1] * [X_2] \tag{24}$$

$$[X_1] * [X_2] \implies [X_1] * [X_2] * X_1 \cap X_2 = \emptyset \tag{25}$$

The idea is to give each specification invariant ownership of a countably infinite set of locations that only that simulation may use for future allocations.

To allow simulations to replay reductions from other simulations, we also need a way of deactivating a simulation, such that we can take back ownership of that simulation's locations. To achieve this we introduce a ghost resource $[\textsc{Sr}]^\pi_\zeta$, which we refer to as a *specification runner resource*, to track whether a simulation is active. Ownership of any fraction of this token witnesses that the simulation is active.

Logical relation. The $\text{LR}_{\textsc{Par}}$ logical relation is defined in Figure 9. The most important difference compared to the $\text{LR}_{\textsc{Bin}}$ relation, is in the $locs(r)$ predicate contained in the region invariant Reg. Reg now asserts fractional ownership of a specification link resource for the given region through the *slink* predicate. In case the region is immutable, the pair of heaps given by the specification link resource must match the actual implementation and specification heap for the references belonging to the given region. The region invariant further asserts ownership of the local specification heap resource $l \mapsto^\zeta_S v$ for every simulation $\zeta \in S$ tied to the given region through the specification link resource.

The specification invariant, Spec, has been extended to support global freshness when allocating, as explained above. Either the specification invariant owns half of the specification runner resource, in which case it also asserts ownership of countably infinite sets of fresh locations through the *disj* predicate. Otherwise, the specification invariant is inactive and asserts exclusive ownership of the specification runner resource.

Finally, the expression relation now asserts fractional ownership of the specification runner resource and fractional ownership of specification link resources, for all regions in the context. The specification runner resource ensures that the ζ simulation is active. In case a region is private or the effect mask contains a write or allocation effect for the given

New predicates

$$P_{par}(R,g,\varepsilon,M,\zeta) \triangleq \circledast_{\rho\in mutable(R,g,\varepsilon)}[\textsc{Mu}(M(\rho),\{\zeta\})]^{g(\rho)} * \circledast_{\rho\in R\setminus mutable(R,g,\varepsilon)}\exists h,S.\mathit{slink}(M(\rho),\{\zeta\}\uplus S,h,g(\rho),g(\rho))$$

$$\mathit{slink}(r,S,h) \triangleq [\textsc{Mu}(r,S)]^{\frac{1}{2}} \vee [\textsc{Im}(r,S,h)]^{\frac{1}{4}}$$

$$\mathit{disj}(X_0,X) \triangleq \exists Y.\ [Y] \wedge \mathrm{dom}(X_0)\cap Y = \emptyset \wedge (\mathrm{dom}(X)\setminus\mathrm{dom}(X_0)) \subset Y$$

$$\mathit{mutable}(R,g,\varepsilon) \triangleq \mathsf{wrs}\ \varepsilon \cup \mathsf{als}\ \varepsilon \cup \left\{\rho \mid \rho\in R \wedge g(\rho) = \tfrac{1}{2}\right\}$$

Changes to previous definitions

$$\mathit{locs}(r) \triangleq \exists h,S.\ \mathit{slink}(r,S,h_S) * \mathit{rheap}_I(h_I,r) * \mathit{rheap}_S(h_S,r) * \mathit{alloc}(h,r) * \circledast_{(l,v)\in h_I} l\mapsto_I v * \circledast_{\zeta\in S}\circledast_{(l,v)\in h_S} l\mapsto^{\zeta}_S v$$

$$\textsc{Spec}(h_0,e_0,\zeta) \triangleq \exists h,e,\pi.\ \mathit{heap}_S(h,\zeta) * \mathit{mctx}(e,\zeta) * (h_0,e_0)\to^*(h,e) * ([\textsc{Sr}]^{1}_{\zeta} \vee ([\textsc{Sr}]^{\frac{1}{2}}_{\zeta} * \mathit{disj}(h_0,h_S)))$$

$$P_{reg}(\cdots,\zeta) \triangleq \cdots * P_{par}(R,\tfrac{1}{2}\circ g,\varepsilon,M,\zeta)$$

$$\mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}(\phi)(e_I,e_S) \triangleq \forall g,j,h_0,e_0,\pi,\zeta.\ \boxed{\textsc{Spec}(h_0,e_0,\zeta)} \vdash \left\{j \overset{\zeta}{\Rightarrow}_S e_S * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda,\mathbf{1},\varepsilon,M,\zeta) * P_{reg}(\Pi,g,\varepsilon,M,\zeta)\right\}\ e_I\ \left\{\begin{array}{l} v_I.\ \exists v_S.\ j\overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi}_{\zeta} * \phi(v_I,v_S) * \\ P_{reg}(\Lambda,\mathbf{1},\varepsilon,M,\zeta) * P_{reg}(\Pi,g,\varepsilon,M,\zeta)\end{array}\right\}_{\top}$$

Figure 9. LR_{PAR}: Binary rel. for $\lambda_{ref,conc}$ with effect-types and effect-based simulations.

region, then the region must be in the mutable state and tied only to the simulation ζ. Otherwise, the region may be in either the mutable or the immutable state, as long as it is tied to the ζ simulation.

The LR_{PAR} relation is sound with respect to contextual approximation and supports parallelization.

Theorem 6 (Soundness)**.** *If* $\Pi \mid \Delta \mid \Gamma \models_{\text{PAR}} e_I \leq_{log} e_S : \tau,\varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \vdash e_I \leq_{ctx} e_S : \tau,\varepsilon$.

Theorem 7 (Parallelization (semantically))**.** *If*

1. $\mathcal{E}^{\Lambda_3,\Lambda_1}_{\varepsilon_1,M}([\![\tau_1]\!]^M)(e_{1I},e_{1S})$ *and* $\mathcal{E}^{\Lambda_3,\Lambda_2}_{\varepsilon_2,M}([\![\tau_2]\!]^M)(e_{2I},e_{2S})$
2. $\mathsf{als}\ \varepsilon_1 \cup \mathsf{wrs}\ \varepsilon_1 \subseteq \Lambda_1$, $\mathsf{als}\ \varepsilon_2 \cup \mathsf{wrs}\ \varepsilon_2 \subseteq \Lambda_2$
3. $\mathsf{rds}\ \varepsilon_1 \subseteq \Lambda_1 \cup \Lambda_3$ *and* $\mathsf{rds}\ \varepsilon_2 \subseteq \Lambda_2 \cup \Lambda_3$

then $\mathcal{E}^{-,(\Lambda_1,\Lambda_2,\Lambda_3)}_{\varepsilon_1\cup\varepsilon_2,M}([\![\tau_1\times\tau_2]\!]^M)(e_{1I}\,||\,e_{2I},(e_{1S},e_{2S}))$.

Using the LR_{PAR} relation we can prove the contextual refinements from in the Introduction (Theorems 2 and 3). Proofs can be found in the Appendix.

4. Discussion

We have already mentioned some related work along the way; here we discuss some other related work.

Benton et. al. initiated a line of work on relational models of type-and-effect systems to formally justify effect-based program transformations for increasingly sophisticated sequential programming languages and increasingly expressive effect systems [4–8, 28]. Birkedal et. al. showed how to extend this approach to a concurrent language [11]. The effect system we use here is from *loc. cit.* Birkedal et. al.'s relational interpretation defined by a concrete step-indexed Kripke logical relation. They used the model to prove a parallelization theorem similar to ours, but the proof was very technical and consisted of manual disentangling and re-ordering of computation steps. Part of the reason for this was that support for parallelization was not built into their logical relation and had to be proven separately. In contrast, we build in support for parallelization in the LR_{PAR} relation through its support for multiple simulations. This allows us to reduce the proof of the parallelization theorem to the essence of why the it holds: framing. Moreover, as mentioned in the Introduction, it makes it possible to use the program logic to show that an expression satisfies the semantic invariants imposed by the type system even if the expression is not statically well-typed and to reason about refinements.

In recent work, Benton et. al. [9] have also considered a concurrent language, which in contrast to the language considered here only includes first-order store. Technically, this makes the construction of a logical relations model simpler, since one avoids having to deal with the type-world circularity mentioned in the Introduction. Their type-and-effect system does not support dynamic allocation of abstract locations (which correspond to regions in our setup), requiring all abstract locations to be given up front. Our type-and-effect system supports dynamic allocation and hiding of regions, through the masking rule. On the other hand, their effect system supports a notion of abstract effects, which means, e.g., that an operation in a data structure module can be considered pure even if it uses effects internally, as long as those effects are not observable outside the module boundary. Benton et. al. use this facility for treating refinement of fine-grained concurrent data structures, illustrated using an idealized Michael-Scott queue. Our semantics also supports refinements between fine-grained concurrent data structures, using Iris' support for general invariants. In this paper we have focused on an example of a refinement that only holds by restricting interference through the type-and-effect system. Our method also scales to fine-grained concurrent data structures that use helping, thanks to Iris [16].

Raza et. al. [25] and Botincan et. al. [12], both explore automatic parallelization of sequential programs verified in separation logic. Raza *et al.* rely on specifications inferred from a shape-analysis. Botincan *et al.* explore the idea of using the proof to insert synchronization that ensures the dependencies of the original program are preserved. These analyses focus on first-order programs, whereas our type-and-effect system applies to higher-order programs.

The idea of defining logical relations in a program logic goes back at least to Plotkin and Abadi, who used a second-order logic to define logical relations for a second-order lambda calculus [21]. Dreyer et. al. used a second-order logic with a Löb modality, inspired by [3], to give a logical relations interpretation of a programming language with recursive types [13]. The logic used by Dreyer et. al. did not support invariants and hence it did not support the interpretation of reference types. Turon et. al. showed how to use a variant of second-order concurrent separation logic with invariants for giving a logical relations interpretation of an ML-like type system for a language similar to the one considered in this paper [30]. To define logical relations in the unary separation logic, their logic had a *built-in* notion of specification resources and a single specification invariant. In contrast, here we use a higher-order concurrent separation logic, Iris, which is flexible enough that one can define specification resources and invariants in it. We rely crucially on this flexibility for the LR_{PAR} relation to support multiple simulations, as discussed in Section 3.4.

References

[1] A. Ahmed, A. Appel, and R. Virga. A stratified semantics of general references A stratified semantics of general references. In *LICS*, 2002.

[2] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[3] A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.

[4] N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI*, 2007.

[5] N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations. In *PLAS*. Springer, 2006.

[6] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*, 2007.

[7] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*, 2009.

[8] N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. In *POPL*, 2014.

[9] N. Benton, M. Hofmann, and V. Nigam. Effect-dependent transformations for concurrent programs. In *PPDP*, 2016.

[10] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed kripke models over recursive worlds. In *POPL*, 2011.

[11] L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In *CSL*, 2012.

[12] M. Botincan, M. Dodds, and S. Jagannathan. Proof-Directed Parallelization Synthesis by Separation Logic. *TOPLAS*, 35 (2), 2013.

[13] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

[14] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, 2002.

[15] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LISP*, 1986.

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

[17] N. Krishnaswami, P. Pradic, and N. Benton. Integrating linear and dependent types. In *POPL*, 2015.

[18] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.

[19] G. Morrisett, A. Ahmed, and M. Fluet. L^3: A linear language with locations. In *TLCA*, 2005.

[20] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *ICFP*, 2006.

[21] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA*, 1993.

[22] F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, 2008.

[23] F. Pottier and J. Protzenko. Programming with permissions in Mezzo. In *ICFP*, 2013.

[24] F. Pottier and J. Protzenko. A few lessons from the mezzo project. In *SNAPL*, 2015.

[25] M. Raza, C. Calcagno, and P. Gardner. Automatic Parallelization with Separation Logic. In *ESOP*, 2009.

[26] Rust Language. `https://doc.rust-lang.org`, 2016.

[27] F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, 2000.

[28] J. Thamsborg and L. Birkedal. A kripke logical relation for effect-based program transformations. In *ICFP*, 2011.

[29] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *POPL*, 1994.

[30] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.

[31] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.