A Higher-Order Logic for Concurrent Termination-Preserving Refinement

Joseph Tassarotti[1], Ralf Jung[2], and Robert Harper[1]

[1] Carnegie Mellon University, Pittsburgh, USA
[2] MPI-SWS, Saarland, Germany

Abstract. Compiler correctness proofs for higher-order concurrent languages are difficult: they involve establishing a termination-preserving refinement between a *concurrent* high-level source language and an implementation that uses low-level shared memory primitives. However, existing logics for proving concurrent refinement either neglect properties such as termination, or only handle first-order state. In this paper, we address these limitations by extending Iris, a recent higher-order concurrent separation logic, with support for reasoning about termination-preserving refinements. To demonstrate the power of these extensions, we prove the correctness of an efficient implementation of a higher-order, session-typed language. To our knowledge, this is the first program logic capable of giving a compiler correctness proof for such a language. The soundness of our extensions and our compiler correctness proof have been mechanized in Coq.

1   Introduction

Parallelism and concurrency impose great challenges on both programmers and compilers. In order to make compiled code more efficient and help programmers avoid errors, languages can provide type systems or other features to constrain the structure of programs and provide useful guarantees. The design of these kinds of concurrent languages is an active area of research. However, it is frequently difficult to prove that efficient compilers for these languages are correct, and that important properties of the source-level language are preserved under compilation.

For example, in work on session types [19, 45, 17, 10, 42], processes communicate by sending messages over channels. These channels are given a type which describes the kind of data sent over the channel, as well as the order in which each process sends and receives messages. Often, the type system in these languages ensures the absence of undesired behaviors like races and deadlocks; for instance, two threads cannot both be trying to send a message on the same channel simultaneously.

Besides preventing errors, the invariants enforced by session types also permit these language to be compiled efficiently to a shared-memory target language [43]. For example, because only one thread can be sending a message

on a given channel at a time, channels can be implemented without performing locking to send and receive messages. It is particularly important to prove that such an implementation does not *introduce* races or deadlocks, since this would destroy the very properties that make certain session-typed languages so interesting.

In this paper, we develop a higher-order program logic for proving the correctness of such concurrent language implementations, in a way that ensures that termination is preserved. We have used this program logic to give a machine-checked proof of correctness for a lock-free implementation of a *higher-order* session-typed language, *i.e.,* a language in which closures and channels can be sent over channels. To our knowledge, this is the *first such proof* of its kind.

As we describe below, previously developed program logics cannot be used to obtain these kinds of correctness results due to various limitations. In the remainder of the introduction, we will explain why it is so hard to prove refinements between higher-order, concurrent languages. To this end, we first have to provide some background.

Refinement for concurrent languages. To show that a compiler is correct, one typically proves that if a source expression E is well-typed, its translation $\widehat{E}$ *refines* E. In the sequential setting, this notion of refinement is easy to define[3]: (1) if the target program $\widehat{E}$ terminates in some value v, we expect E to also have an execution that terminates with value v, and (2) if $\widehat{E}$ diverges, then E should also have a diverging execution.

In the concurrent setting, however, we need to change this definition. In particular, the condition (2) concerning diverging executions is too weak. To see why, consider the following program, where x initially contains 0:

```
while (*x == 0)  {}     ||     *x = 1;
```

Here, || represents parallel composition of two threads. In every execution where the thread on the right eventually gets to run, this program will terminate. However, the program does have a diverging execution in which only the left thread runs: because x remains 0, the left thread continues to loop. Such executions are "unrealistic" in the sense that generally, we rely on schedulers to be *fair* and not let a thread starve. As a consequence, for purposes of compiler correctness, we do not want to consider these "unrealistic" executions which only diverge because the scheduler never lets a thread run.

Formally, an infinite execution is said to be *fair* [27] if every thread which does not terminate in a value takes infinitely many steps.[4] In the definition of refinement above, we change (2) to demand that if $\widehat{E}$ has a *fair* diverging execution, then E also has a *fair* diverging execution. We impose no such requirement

---

[3] Setting aside issues of IO behavior.

[4] This definition is simpler than the version found in Lehmann et al. [27], because there threads can be temporarily *disabled*, *i.e.,* blocked and unable to take a step. In the languages we consider, threads can always take a step unless they have finished executing or have "gone wrong".

about unfair diverging executions. This leads us to *fair termination-preserving refinement.*

Logics for proving refinement. To prove our compiler correct, we need to reason about the concurrent execution and (non)termination of the source and target programs. Rather than reason directly about all possible executions of these programs, we prefer to use a concurrent program logic in order to re-use ideas found in rely-guarantee reasoning [21] and concurrent separation logic [34]. However, although a number of concurrency logics have recently been developed for reasoning about termination and refinements, they cannot be used to prove our compiler correctness result because they either:

- are restricted to first-order state [18, 36, 29, 30, 28],
- only deal with termination, not refinement [18, 36], or
- handle a weaker form of refinement that is not fair termination-preserving [40, 29, 30].

Although the limitations are different in each of the above papers, let us focus on the approach by Turon et al. [40] since we will build on it. That paper establishes a termination-*insensitive* form of refinement, *i.e.,* a diverging program refines every program. Refinement is proven in a higher-order concurrent separation logic which, in addition to the usual points-to assertions $l \hookrightarrow v$, also provides assertions *about the source language's state.* For instance, the assertion[5] $\mathsf{source}(i, E)$ says thread i in the source language's execution is running expression E. A thread which "owns" this resource is allowed to modify the state of the source program by simulating steps of the execution of E. Then, we can prove that e refines E by showing:

$$\{\mathsf{source}(i, E)\} \; e \; \{v. \, \mathsf{source}(i, v)\}$$

As usual, the triple enforces that the post-condition holds on termination of e. Concretely for the triple above, the soundness theorem for the logic implies that if target expression e terminates with a value v, then there is an execution of source expression E that also terminates with value v. However, the Hoare triple above only expresses *partial correctness*. That means if e does not terminate, then the triple above is trivial, and so these triples can only be used to prove termination-insensitive refinements.

Ideally, one would like to overcome this limitation by adapting ideas from logics that deal with termination for first-order state. Notably, Liang *et al.* [28] have recently developed a logic for establishing *fair* refinements (as defined above).

However, there is a serious difficulty in trying to adapt these ideas. Semantic models of concurrency logics for higher-order state usually involve *step-indexing* [3, 7]. In step-indexed logics, the validity of Hoare triples is restricted to program executions of arbitrary *but finite* length. How can we use these to reason about fairness, a property which is inherently about *infinite* executions?

---

[5] The notation in Turon et al. [40] is different.

In this paper, we show how to overcome this difficulty: the key insight is that when the source language has only *bounded non-determinism*, step-indexed Hoare triples are actually sufficient to establish properties of infinite program executions. Using this observation, we extend Iris [23, 22], a recent higher-order concurrent separation logic, to support reasoning about fair termination-preserving refinement. The soundness of our extensions to Iris and our case studies have been verified in Coq.

Overview. We start by introducing the case study that we will focus on in this paper: a session-typed source language, a compiler into an ML-like language, and the compiler's correctness property – fair, termination-preserving refinement (§2). Then we present our higher-order concurrent separation logic for establishing said refinement (§3). We follow on by explaining the key changes to Iris that were necessary to perform this kind of reasoning (§4). We then use the extended logic to prove the correctness of the compiler for our session-typed language (§5). Finally, we conclude by describing connections to related work and limitations of our approach that we hope to address in future work (§6).

2   Session-Typed Language and Compiler

This section describes the case study that we chose to demonstrate our logic: a concurrent message-passing language and a type system establishing safety and race-freedom for this language. On top of that, we explain how to implement the message-passing primitives in terms of shared-memory concurrency, *i.e.,* we define a compiler translating the source language into an ML-like target language. Finally, we discuss the desired correctness statement for this compiler.

2.1   Source Language

The source language for our compiler is a simplified version of the language described in Gay and Vasconcelos [17]. The syntax and semantics are given in Figure 1. It is a functional language extended with primitives for message passing and a command $\mathsf{fork}\{E\}$ for creating threads. The semantics is defined by specifying a reduction relation for a single thread, which is then lifted to a concurrent semantics on thread-pools in which at each step a thread is selected non-deterministically to take the next step.

Threads can communicate asynchronously with each other by sending messages over *channels*. For example, consider the following program (which will be a running example of the paper):

$$\mathsf{let}\ (x, y) = \mathsf{newch}\ \mathsf{in}\ \big(\mathsf{fork}\{\mathsf{send}(x, 42)\}; \mathsf{let}\ (\_, v) = \mathsf{recv}(y)\ \mathsf{in}\ v\big) \qquad (1)$$

The command newch creates a new channel and returns two *end-points* (bound to x and y in the example). An end-point consists of a channel id c and a side s (either left or right), and is written as c_s. Each channel is a pair of

Syntax:

$Side$	$s ::= $ left $\mid$ right	
Val	$V ::= c_s \mid \lambda x.\,E_1 \mid (V_1, V_2) \mid () \mid n$	where $c \in \mathbb{N}$
$Expr$	$E ::= x \mid V \mid E_1\,E_2 \mid (E_1, E_2) \mid$ fork$\{E\} \mid$ newch $\mid$ recv(E)	
	$\mid$ send$(E_1, E_2) \mid$ let $(x, y) = E_1$ in $E_2 \mid \ldots$	
$Eval\ Ctx$	$K ::= [] \mid K\,E \mid V\,K \mid (K, E) \mid (V, K) \mid$ recv$(K) \mid$ send(K, E)	
	$\mid$ send$(V, K) \mid$ let $(x, y) = K$ in $E \mid \ldots$	
$State$	$\Sigma \in \mathbb{N} \to List\ Val \times List\ Val$	
$Config$	$\rho ::= [E_1, \ldots, E_n]; \Sigma$	
$Type$	$\tau ::=$ Int $\mid$ Unit $\mid \tau_1 \otimes \tau_2 \mid \tau_1 \multimap \tau_2 \mid S$	
$Session\ Type\ S$	$::= \,!\tau.\,S \mid \,?\tau.\,S \mid$ end	(co-inductive)
$Dual\ Type$	$\overline{?\tau.\,S} \triangleq\ !\tau.\,\overline{S} \qquad \overline{!\tau.\,S} \triangleq\ ?\tau.\,\overline{S} \qquad \overline{\text{end}} \triangleq$ end	

Per-Thread Reduction $E; \Sigma \to E'; \Sigma'$:   (Pure and symmetric rules ommitted.)

NEWCH
$$\frac{c = \min\{c' \mid c' \notin \mathrm{dom}(\Sigma)\}}{\text{newch}; \Sigma \to (c_{\mathsf{left}}, c_{\mathsf{right}}); [c \hookrightarrow ([], [])]\Sigma}$$

SENDLEFT
$$\frac{\Sigma(c) = (b_\to, b_\leftarrow)}{\text{send}(c_{\mathsf{left}}, V); \Sigma \to c_{\mathsf{left}}; [c \hookrightarrow (b_\to V, b_\leftarrow)]\Sigma}$$

RECVRIGHTIDLE
$$\frac{\Sigma(c) = ([], b_\leftarrow)}{\text{recv}(c_{\mathsf{right}}); \Sigma \to \text{recv}(c_{\mathsf{right}}); \Sigma}$$

RECVRIGHT
$$\frac{\Sigma(c) = (V\,b_\to, b_\leftarrow)}{\text{recv}(c_{\mathsf{right}}); \Sigma \to (c_{\mathsf{right}}, V); [c \hookrightarrow (b_\to, b_\leftarrow)]\Sigma}$$

Concurrent Semantics $\rho \to \rho'$:

$$\frac{E_i; \Sigma \to E_i'; \Sigma'}{[\ldots, K[E_i], \ldots]; \Sigma \to [\ldots, K[E_i'], \ldots]; \Sigma'} \qquad [\ldots, K[\text{fork}\{E_{\mathsf{f}}\}], \ldots]; \Sigma \to [\ldots, K[()], \ldots, E_{\mathsf{f}}]; \Sigma$$

Type system: (Standard rules for variables, integers and lambda omitted.)

FUN-ELIM
$$\frac{\Gamma \vdash E : \tau_1 \multimap \tau_2 \qquad \Gamma' \vdash E' : \tau_1}{\Gamma \uplus \Gamma' \vdash E\,E' : \tau_2}$$

PAIR-INTRO
$$\frac{\Gamma_1 \vdash E_1 : \tau_1 \qquad \Gamma_2 \vdash E_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash (E_1, E_2) : \tau_1 \otimes \tau_2}$$

PAIR-ELIM
$$\frac{\Gamma \vdash E : \tau_1 \otimes \tau_2 \qquad \Gamma', x : \tau_1, y : \tau_2 \vdash E' : \tau'}{\Gamma \uplus \Gamma' \vdash \text{let } (x, y) = E \text{ in } E' : \tau'}$$

FORK
$$\frac{\Gamma_1 \vdash E_{\mathsf{f}} : \tau' \qquad \Gamma_2 \vdash E : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{fork}\{E_{\mathsf{f}}\}; E : \tau}$$

NEWCHTYP
$$\Gamma \vdash \text{newch} : S \otimes \overline{S}$$

SEND
$$\frac{\Gamma_1 \vdash E_1 : !\tau.\,S \qquad \Gamma_2 \vdash E_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \text{send}(E_1, E_2) : S}$$

RECV
$$\frac{\Gamma \vdash E : ?\tau.\,S}{\Gamma \vdash \text{recv}(E) : S \otimes \tau}$$

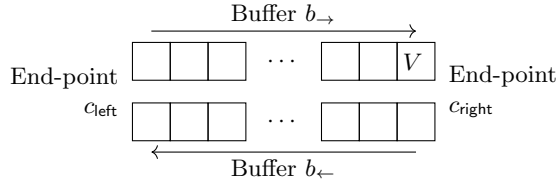**Buffer visualization:** Message V has been sent from the left end-point to the right.

Fig. 1. Syntax, semantics, and session type system of message-passing source language

buffers $(b_\rightarrow, b_\leftarrow)$, which are lists of messages. Buffer $b_\rightarrow$ stores messages traveling left-to-right (from x to y, in the example above), and $b_\leftarrow$ is for right-to-left messages, as shown in the visualization in Figure 1.

A thread can then use $\mathsf{send}(c_s, V)$ to send a value V along the channel c, with the side s specifying which buffer is used to store the message. For instance, when s is left, it inserts the value at the end of the first buffer (SENDLEFT). This value will then later be taken by a thread receiving on the *right* side (RECVRIGHT). Alternatively, if the buffer is empty when receiving, recv takes an "idle" step and tries again (RECVRIGHTIDLE). (The reason send and recv return the end-point again will become clear when we explain the type system.)

In the example above, after creating a new channel, the initial thread forks off a child which will send 42 from the left end-point, x. Meanwhile, the parent thread tries to receive from the right end-point y, and returns the message it gets. If the parent thread does this recv *before* the child has done its send, there will be no message and the parent thread will take an idle step. Otherwise, the receiver will see the message and the program will evaluate to 42.

2.2   Session Type System

A type system for this language is shown in Figure 1. This is a simplified version of the type system given in Gay and Vasconcelos [17].[6] In addition to base types Int and Unit, we have pair types $\tau_1 \otimes \tau_2$, function types $\tau_1 \multimap \tau_2$, and *session types* S. Session types are used to type the end-points of a channel. These types describe a kind of *protocol* specifying what types of data will flow over the channel, and in what order messages are sent. Notice that this type system is *higher-order* in the sense that both closures and channel end-points are first-class values and can, in particular, be sent over channels.

Session types. The possible session types are specified by the grammar in Figure 1. If an end-point has the session type $!\tau. S$, this means that the next use of this end-point must be to send a value of type τ (SEND). Afterward, the end-point that is returned by the send will have type S. Dually, $?\tau. S$ says that the end-point can be used in a receive (RECV), in which case the message read will have type τ, and the returned end-point will have type S. Notice that this is the same end-point that was passed to the command, but *at a different type.* The type of the end-point *evolves* as messages are sent and received, always representing the current state of the protocol. Finally, end is a session type for an end-point on which no further messages will be sent or received.

When calling newch to create a new channel, it is important that the types of the two end-points match: whenever one side sends a message of type τ, the other side should be expecting to receive a message of the same type. This relation is called *duality*. Given a session type S, its *dual* $\overline{S}$ is the result of swapping sends and receives in S. In our example (1), the end-point x is used to send a

---

[6] For the reader familiar with that work: we leave out subtyping and choice types. Also, we present an affine type system instead of a linear one.

single integer, so it can be given the type !Int. end. Conversely, y receives a single integer, so it has the dual type $\overline{\text{!Int. end}}$ = ?Int. end.

Affinity. The type system of the source language is *affine*, which means that a variable in the context can be used at most once. This can be seen, *e.g.,* in the rule FORK: the forked-off thread E_f and the local continuation E are typed using the two disjoint contexts Γ_1 and Γ_2, respectively.

One consequence of affinity is that after using an end-point to send or receive, the variable passed to send/recv has been "used up" and cannot be used anymore. Instead, the program has to use the channel returned from send/recv, which has the new "evolved" type for the end-point.

The type system given here ensures safety and race-freedom. However, it does not guarantee termination. We discuss alternative type systems guaranteeing different properties in the conclusion.

2.3   Compilation

We now describe a simple translation from this session-typed source language to a MiniML language with references and a forking primitive like the one in the source language. We omit the details of the MiniML syntax and semantics as they are standard.

Our translation needs to handle essentially one feature: the implementation of channel communication in terms of shared memory references.

The code for the implementation of the channel primitives is shown in Figure 2. We write $\widehat{E}$ for the translation in which we replace the primitives of the source language with the corresponding implementations. Concretely, applying the translation to our running example program we get:

$$
\begin{array}{lcl}
\text{let } (x, y) = \text{newch in} & & \text{let } (x, y) = \text{heapNewch in} \\
\text{fork\{send}(x, 42)\}; & \Rightarrow & \text{fork\{heapSend } x\ 42\}; \\
\text{let } (\_, v) = \text{recv}(y) \text{ in } v & & \text{let } (\_, v) = \text{heapRecv } y \text{ in } v
\end{array}
$$

Each channel is implemented as a linked list which represents *both* buffers. Nodes in this list are pairs (l, v), where l is a reference to the (optional) next node, and v is the message that was sent. Why is it safe to use just one list? Duality in the session types guarantees that if a thread is sending from one end-point, no thread can at the same time be sending a message on the other end-point. This ensures that at least one of the two buffers in a channel is always empty. Hence we just need one list to represent both buffers.

The implementation of newch, given by heapNewch, creates a new empty linked list by allocating a new reference l which initially contains none. The function heapSend implements send by appending a node to the end (l') of the list, and returning the new end. Meanwhile, for recv, heapRecv takes an end-point l and waits in a loop until it finds that the end-point contains a node.

$$\text{heapNewch} \triangleq$$
$$\quad \text{let } l = \text{ref none in } (l, l)$$

$$\text{heapSend } l\, v \triangleq$$
$$\quad \text{let } (l', v') = (l, v) \text{ in}$$
$$\quad \text{let } l_{new} = \text{ref none in}$$
$$\quad l' := \text{some } (l_{new}, v');$$
$$\quad l_{new}$$

$$\text{heapRecv} \triangleq \text{rec } f\, l.$$
$$\quad \text{match } !l \text{ with}$$
$$\quad\quad \mid \text{none} \Rightarrow f\, l$$
$$\quad\quad \mid \text{some } (l', v) \Rightarrow (l', v)$$
$$\quad \text{end}$$

Fig. 2. Implementation of message passing primitives.

2.4   Refinement

Having given the implementation, let us now clarify what it means for the compiler to be correct. Intuitively, we want to show that if we take a well-typed source expression E, all the *behaviors* of its translation $\widehat{E}$ are also *possible* behaviors of E. We say that $\widehat{E}$ *refines* E.

Before we come to the formal definition of refinement, we need to answer the question: which behaviors do we consider equivalent? In our case, the only observation that can be made about a whole program is its return value, so classifying "behaviors" amounts to relating return values. Formally speaking:

$$n \approx n \qquad () \approx () \qquad l \approx c_s \qquad \lambda x.e \approx \lambda x.E \qquad \frac{v_1 \approx V_1 \qquad v_2 \approx V_2}{(v_1, v_2) \approx (V_1, V_2)}$$

For integer and unit values, we expect them to be exactly equal; similarly, pairs are the same if their components are. Coming to locations/end-points and closures, we do not consider them to be interpretable by the user looking at the result of a closed program. So, we just consider all closures to be equivalent, and all heap locations to relate to all channel end-points. Of course, the *proof* of compiler correctness will use a more fine-grained logical relation between source and target values.

Based on this notion of equivalent observations, we define what it means for a MiniML program e to *refine* a source program E, written $e \sqsubseteq E$. When executing from an initial "empty" state $\emptyset$, the following conditions must hold:

1. If $([e], \emptyset) \to^* ([e_1, \dots, e_n], \sigma)$ then no e_i is stuck in state σ.
   In other words: the target program does not reach a stuck state.
2. If $([e], \emptyset) \to^* ([v_1, \dots, v_n], \sigma)$ then either:
   (a) $([E], \emptyset) \to^* ([V_1, \dots, V_m], \Sigma)$ and $v_1 \approx V_1$, or
   (b) there is an execution of $([E], \emptyset)$ in which some thread gets stuck.
   That is, if *all threads* of the target program terminate with a value, then either *all threads* of the source program terminate in some execution *and* the return values of the first (main) source thread and target thread are equivalent; or the source program can get stuck.

3. If $([e], \emptyset)$ has a fair diverging execution, then $([E], \emptyset)$ also has a fair diverging execution. Recall that an infinite execution is *fair* if every non-terminating thread takes infinitely many steps. This last condition makes the refinement a *fair, termination-preserving* refinement.

To understand why we have emphasized the importance of fair termination-preservation, suppose we had miscompiled our running example as:

$$\text{let } (x, y) = \text{heapNewch in let } (\_, v) = \text{heapRecv } y \text{ in } v$$

That is, we removed the sender thread. We consider this to be an incorrect compilation; *i.e.,* this program should *not* be considered a refinement of the source program. But imagine that we removed the word "fair" from condition (3) above: then this bad target program would be considered a refinement of the source. How is that? The program does not get stuck, so it satisfies condition (1). Condition (2) holds vacuously since the target program will never terminate; it will loop in heapRecv y, forever waiting for a message. Finally, to satisfy condition (3), we have to exhibit a diverging execution in the source program. Without the fairness constraint, we can pick the (unfair) execution in which the sender source thread never gets to run.

Notice that this unfair execution is very much like the example we gave in the introduction, where a thread waited forever for another one to perform a change in the shared state.

We consider such unfair executions to be unrealistic [27]; they should not give license to a compiler to entirely remove a thread from the compiled program. That's why our notion of refinement restricts condition (3) to *fair* executions, *i.e.,* executions in which all non-terminating threads take infinitely many steps.

Compiler correctness. We are now equipped to formally express the correctness statement of our compiler:

Theorem 1. *For every* well-typed *source program E, we have that:*

$$\widehat{E} \sqsubseteq E$$

We prove this theorem in §5. In the intervening sections, we first develop and explain a logic to help carry out this proof.

3   A Logic for Proving Refinement

Proving Theorem 1 is a challenging exercise. Both the source and the target program are written in a concurrent language with higher-order state, which is always a difficult combination to reason about. Moreover, the invariant relating the channels and buffers to their implementation as linked lists is non-trivial and relies on well-typedness of the source program.

The contribution of this paper is to provide a logic powerful enough to prove theorems like Theorem 1. In this section, we will give the reader an impression of both the logic and the proof by working through a proof of one concrete instance of our general result: we will prove that the translation of our running example is in fact a refinement of its source.

3.1   Refinement as a Hoare Logic

Our logic is an extension of *Iris* [23, 22], a concurrent higher-order separation logic. We use the ideas presented by Turon et al. [40] to extend this (unary) Hoare logic with reasoning principles for refinement. Finally, we add some further extensions which become necessary due to the *termination-preserving* nature of our refinement. We will highlight these extensions as we go.

The following grammar covers the assertions from our logic that we will need:[7]

$$P ::= \mathsf{False} \mid \mathsf{True} \mid P \vee P \mid P * P \mid \mathcal{A}(P) \mid \exists x.\, P \mid \forall x.\, P \mid l \hookrightarrow v \mid \mathsf{source}(i, E, d) \mid$$
$$\mathsf{Stopped} \mid c \hookrightarrow_{\mathsf{s}} (b_{\rightarrow}, b_{\leftarrow}) \mid \mathsf{StsSt}(s, T) \mid \{P\}\, e\, \{x.\, Q\} \mid P \Rrightarrow Q \mid P \Rrightarrow Q \mid ...$$

Many of these assertions are standard in separation logics, and our example proof will illustrate the non-standard ones.

Recalling the example and its translation, we want to prove:

let $(x, y) = \mathsf{heapNewch}$ in		let $(x, y) = \mathsf{newch}$ in
fork$\{\mathsf{heapSend}\, x\, 42\};$	$\sqsubseteq$	fork$\{\mathsf{send}(x, 42)\};$
let $(\_, v) = \mathsf{heapRecv}\, y$ in v		let $(\_, v) = \mathsf{recv}(y)$ in v

or, for short, $e_{\mathrm{ex}} \sqsubseteq E_{\mathrm{ex}}$. Following Ht-refine (Figure 3), it is enough to prove

$$\{\mathsf{source}(i, E_{\mathrm{ex}}, d)\}\, e_{\mathrm{ex}}\, \{v.\, \exists V.\, \mathsf{source}(i, V, 0) * v \approx V\} \tag{2}$$

In other words, we "just" prove a Hoare triple for e_{ex} (the MiniML program). In order to obtain a refinement from a Hoare proof, we equip our logic with assertions talking about the source program E. The assertion $\mathsf{source}(i, E, d)$ states that source-level thread i is about to execute E, and we have *delay d* left. (We will come back to delays shortly.) The assertion $c \hookrightarrow_{\mathsf{s}} (b_{\rightarrow}, b_{\leftarrow})$ says that source-level channel c currently has buffer contents $(b_{\rightarrow}, b_{\leftarrow})$. As usual in separation logic, both of these assertions furthermore assert *exclusive ownership* of their thread or channel. For example, in the case of $c \hookrightarrow_{\mathsf{s}} (b_{\rightarrow}, b_{\leftarrow})$, this means that no other thread can access the channel and we are free to mutate it (*i.e.,* send or receive messages) – we will see later how the logic allows threads to share these resources. Put together, these two assertions let us control the complete state of the source program's execution.

So far, we have not described anything new. However, to establish *termination-preserving* refinement, we have to add two features to this logic: *step shifts* and *linear assertions*.

---

[7] Note that many of these assertions are not primitive to the logic, but are themselves defined using more basic assertions provided by the logic. For instance, the Hoare triple is actually defined in terms of a *weakest precondition* assertion. See Jung et al. [23, 22] for further details.

Step Shift Rules: (all d and d' must be $\leq$ some fixed upper-bound D)

SRC-NEWCH
$$\mathsf{source}(i, K[\mathsf{newch}], d) \Rrightarrow \exists c.\, \mathsf{source}(i, K[(c_{\mathsf{left}}, c_{\mathsf{right}})], d') * c \hookrightarrow_{\mathsf{s}} ([], [])$$

SRC-RECV-RIGHT-MISS
$$\mathsf{source}(i, K[\mathsf{recv}(c_{\mathsf{right}})], d) * c \hookrightarrow_{\mathsf{s}} ([], b_{\leftarrow}) \Rrightarrow \mathsf{source}(i, K[\mathsf{recv}(c_{\mathsf{right}})], d') * c \hookrightarrow_{\mathsf{s}} ([], b_{\leftarrow})$$

SRC-RECV-RIGHT-HIT
$$\mathsf{source}(i, K[\mathsf{recv}(c_{\mathsf{right}})], d) * c \hookrightarrow_{\mathsf{s}} (v\, b_{\rightarrow}, b_{\leftarrow}) \Rrightarrow \mathsf{source}(i, K[(c_{\mathsf{right}}, v)], d') * c \hookrightarrow_{\mathsf{s}} (b_{\rightarrow}, b_{\leftarrow})$$

SRC-SEND-LEFT
$$\mathsf{source}(i, K[\mathsf{send}(c_{\mathsf{left}}, v)], d) * c \hookrightarrow_{\mathsf{s}} (b_{\rightarrow}, b_{\leftarrow}) \Rrightarrow \mathsf{source}(i, K[c_{\mathsf{left}}], d') * c \hookrightarrow_{\mathsf{s}} (b_{\rightarrow}\, v, b_{\leftarrow})$$

SRC-FORK
$$\mathsf{source}(i, K[\mathsf{fork}\{E\}], d) \Rrightarrow \exists j.\, \mathsf{source}(i, K[()], d') * \mathsf{source}(j, E, d_{\mathsf{f}})$$

SRC-DELAY
$$d' < d \vdash \mathsf{source}(i, K[E], d) \Rrightarrow \mathsf{source}(i, K[E], d')$$

SRC-PURE-STEP
$$\frac{e_1 \to e_2}{\mathsf{source}(i, e_1, d) \Rrightarrow \mathsf{source}(i, e_2, d')}$$

SRC-STOPPED
$$\mathsf{source}(i, V, 0) \vdash \mathsf{Stopped}$$

(Symmetric rules and side-condition on d' omitted.)

Basic Hoare Triples:

ML-ALLOC
$$\frac{\forall x.\, P \Rrightarrow Q}{\{P\}\ \mathsf{ref}\ v\ \{x.\, Q * x \hookrightarrow v\}}$$

ML-LOAD
$$\frac{P \Rrightarrow [v/y]Q}{\{P * x \hookrightarrow v\}\ !x\ \{y.\, Q * x \hookrightarrow v\}}$$

ML-STORE
$$\frac{P \Rrightarrow Q}{\{P * x \hookrightarrow v\}\ x := w\ \{Q * x \hookrightarrow w\}}$$

ML-FORK
$$\frac{P \Rrightarrow Q_0 * Q_1 \qquad \{Q_0\}\ e\ \{\mathsf{Stopped}\} \qquad \{Q_1\}\ e'\ \{R\}}{\{P\}\ \mathsf{fork}\{e\}; e'\ \{R\}}$$

ML-REC
$$\frac{P \Rrightarrow P' \qquad (\forall v.\, \{P\}\ (\mathsf{rec}\ f\, x.\, e)\ v\ \{w.\, Q\}) \Rightarrow \forall v.\, \{P'\}\ [\mathsf{rec}\ f\, x.\, e/f, v/x]e\ \{w.\, Q\}}{\forall v.\, \{P\}\ (\mathsf{rec}\ f\, x.\, e)\ v\ \{w.\, Q\}}$$

HT-FRAME
$$\frac{\{P\}\ e\ \{v.\, Q\}}{\{P * \mathcal{A}(R)\}\ e\ \{v.\, Q * \mathcal{A}(R)\}}$$

STEP-FRAME
$$\frac{P \Rrightarrow Q}{P * \mathcal{A}(R) \Rrightarrow Q * \mathcal{A}(R)}$$

HT-CSQ
$$\frac{P \Rrightarrow P' \qquad \{P'\}\ e\ \{v.\, Q'\} \qquad \forall v.\, Q' \Rrightarrow Q}{\{P\}\ e\ \{v.\, Q\}}$$

Refinement Rule:

HT-REFINE
$$\frac{\{\mathsf{source}(i, E, d)\}\ e\ \{v.\, \exists V.\, \mathsf{source}(i, V, 0) * v \approx V\}}{e \sqsubseteq E}$$

Fig. 3. Selection of rules for step shifts and Hoare triples

Step shifts. The rules given in Figure 3 let us manipulate the state of the source program's execution by *taking steps in the source program.* Such steps are expressed using *step shifts* $\Rrightarrow$. Every step shift corresponds to one rule in the operational semantics (Figure 1). For example, SRC-NEWCH expresses that if we have $\mathsf{source}(i, K[\mathsf{newch}], d)$ (which means that the source is about to create a new channel), we can "execute" that newch and obtain some fresh channel c and ownership of the channel $(c \hookrightarrow_{\mathsf{s}} ([], []))$. We also obtain $\mathsf{source}(i, K[c], d')$, so we can go on executing the source thread.

Crucially, having $P \Rrightarrow Q$ shows that in going from P to Q, the source *has* taken a step. We need to force the source to take steps because the refinement we show is *termination-preserving.* If a proof could just decide not to ever step the source program, we could end up with a MiniML program e diverging, while the corresponding source program E cannot actually diverge. That would make HT-REFINE unsound. So, to avoid this, all rules that take a step in the MiniML program (Figure 3) force us to also take a step shift.

A strict implementation of this idea requires a lock-step execution of source and target program. This is too restrictive. For that reason, the source assertion does not just record the state of the source thread, but also a *delay d*. Decrementing the delay counts as taking a step in the source (SRC-DELAY). When we take an actual source step, we get to reset the delay to some new d' – so long as d' is less than or equal to some fixed upper bound D that we use throughout the proof. There are also rules that allow executing *multiple* source steps when taking just a single step in the target program; we omit these rules for brevity. For the remainder of this proof, we will also gloss over the bookkeeping for the delay and just write $\mathsf{source}(i, e)$.

The assertion Stopped expresses that a source thread can no longer take steps. As expected, this happens when the source thread reaches a value (SRC-STOPPED).

Linearity. There is one last ingredient we have to explain before we start the actual verification: *linearity.* Assertions in our logic are generally *linear*, which means they cannot be "thrown away", *i.e.*, $P * Q \vdash P$ does not hold generically in P and Q. As a consequence, assertions represent not only the *right* to perform certain actions (like modifying memory), but also the *obligation* to keep performing steps in the source program. This ensures that we do not "lose track" of a source thread and stop performing step shifts justifying its continued execution.

The modality $\mathcal{A}(P)$ says that we have a proof of P, and that this is an *affine* proof – so there are no obligations encoded in this assertion, and we can throw it away. Some rules are restricted to affine assertions, *e.g.*, rules for framing around a Hoare triple or a step shift (Figure 3; the rule HT-CSQ will be explained later). Again, this affine requirement ensures that we do not "smuggle" a source thread around the obligation to perform steps in the source. All the base assertions, with the exception of $\mathsf{source}(i, e)$, are affine.

Coming back to the Hoare triple (2) above that we have to prove, the precondition $\mathsf{source}(i, E_{\mathrm{ex}})$ expresses that we start out with a source program executing E_{ex} (and not owning any channels), and we somehow have to take steps in the source program to end up with $\mathsf{source}(i, V)$ such that V is "equivalent" (in

the sense defined in §2.4) to the return value of the target program. Intuitively, because we can only manipulate source by taking steps in the source program, and because we end up stepping from $\mathsf{source}(i, E_{\mathrm{ex}})$ to "the same" return value as the one obtained from e, proving the Hoare triple actually establishes a refinement between the two programs. Furthermore, since source is linear and we perform a step shift at every step of the MiniML program, the refinement holds even for diverging executions.

3.2   Proof of the Example

The rest of this section will present in great detail the proof of our example (2). The rough structure of this proof goes as follows: after a small introduction covering the allocation of the channel, we will motivate the need for *state-transition systems* (STS), a structured way of controlling the interaction between cooperating threads. We will define the STS used for the example and decompose the remainder of the proof into two pieces: one covering the sending thread and one for the receiving thread.

Getting started. The first statement in both source and target program is the allocation of a channel. The following Hoare triple that's easily derived from ML-ALLOC summarizes the action of heapNewch: It allocates a channel in *both* programs.

$$\{\mathsf{source}(i, K[\mathsf{newch}])\} \; \mathsf{heapNewch}$$
$$\{x. \exists l, c.\, x = (l, l) * l \hookrightarrow \mathsf{none} * c \hookrightarrow_{\mathsf{s}} ([], []) * \mathsf{source}(i, K[(c_{\mathsf{left}}, c_{\mathsf{right}})])\} \tag{3}$$

Let us pause a moment to expand on that post-condition. On the source side, we have a channel c with both buffers being empty; on the target side we have a location l representing the empty buffer with none. The return value x is a pair with both components being l. Finally, the source thread changed from $K[\mathsf{newch}]$ in the pre-condition to $K[(c_{\mathsf{left}}, c_{\mathsf{right}})]$, meaning that the newch has been executed and the context can now go on with its evaluation based on the pair $(c_{\mathsf{left}}, c_{\mathsf{right}})$.

   We apply this triple for heapNewch with the appropriate evaluation context K for the source program, and the post-condition of (3) becomes our new context of current assertions. Next, we reduce the let on both sides, so we end up with

$$l \hookrightarrow \mathsf{none} * c \hookrightarrow_{\mathsf{s}} ([], []) * \mathsf{source}(i, e_{\mathrm{comm}}(c)) \tag{4}$$

where

$$e_{\mathrm{comm}}(c) \triangleq \mathsf{fork}\{\mathsf{send}(c_{\mathsf{left}}, 42)\}; \mathsf{let}\ (\_, v) = \mathsf{recv}(c_{\mathsf{right}})\ \mathsf{in}\ v$$

and the remaining MiniML code is

$$\mathsf{fork}\{\mathsf{heapSend}\ l\ 42\}; \mathsf{let}\ (\_, v) = \mathsf{heapRecv}\ l\ \mathsf{in}\ v$$

(In the following, we will perform these pure reduction steps and the substitutions implicitly.)

$$\mathcal{S} \triangleq \boxed{\text{INIT}} \rightarrow \boxed{\begin{array}{c}\text{SENT} \\ [\mathsf{S}]\end{array}} \rightarrow \boxed{\begin{array}{c}\text{RECEIVED} \\ [\mathsf{S}], [\mathsf{R}]\end{array}}$$

$$\varphi(\text{INIT}) \triangleq l \hookrightarrow \mathsf{none} * c \hookrightarrow_\mathsf{s} ([], [])$$
$$\varphi(\text{SENT}) \triangleq l \hookrightarrow \mathsf{some} \ (\_, 42) * c \hookrightarrow_\mathsf{s} ([42], [])$$
$$\varphi(\text{RECEIVED}) \triangleq \mathsf{True}$$
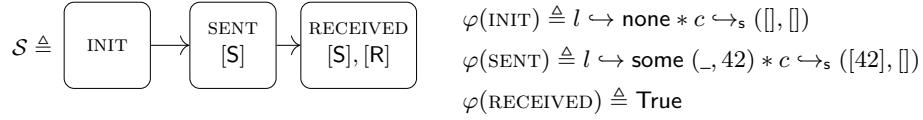
Fig. 4. STS for the example

As we can see, both programs are doing a fork to concurrently send and receive messages on the same channel. Usually, this would be ruled out by the exclusive nature of ownership in separation logic. To enable sharing, the logic provides a notion of *protocols* coordinating the interaction of multiple threads on the same shared state. The protocol governs ownership of both l (in the target) and c (in the source), and describes which thread can perform which actions on this shared state.

State-transition systems. A structured way to describe protocols is the use of state-transition systems (STS), following the ideas of Turon et al. [40]. An STS $\mathcal{S}$ consists of a directed graph with the nodes denoting *states* and the arrows denoting *transitions*.

The STS for our example is given in Figure 4. It describes the interaction of our two threads over the shared buffer happening in three phases. In the beginning, the buffer is empty (INIT). Then the message is sent by the forked-off sending thread (SENT). Finally, the message is received by the main thread (RECEIVED).

The STS also contains two *tokens*. Tokens are used to represent actions that only particular threads can perform. In our example, the state SENT requires the token [S]. The STS enforces that, in order to step from INIT to SENT, a thread must *provide* (and give up) ownership of [S]. This is called the *law of token preservation* [40]: Because SENT contains more tokens than INIT, the missing tokens have to be provided by the thread performing the transition. Similarly, [R] is needed to transition to the final state RECEIVED.

To tie the abstract state of the STS to the rest of the verification, every STS comes with an *interpretation* φ. For every state, it defines an affine assertion that has to hold at that state. In our case, we require the buffer to be initially empty, and to contain 42 in state SENT. Once we reach the final state, the programs no longer perform any action on their respective buffers, so we stop keeping track.

We need a way to track the state of the STS in our proof. To this end, the assertion $\mathsf{StsSt}(s, T)$ states that the STS is *at least* in state s, and that we own tokens T. We cannot know the *exact* current state of the STS because other threads may have performed further transitions in the mean time. The proof rules for STSs can be found in the appendix; in the following, we will keep the reasoning about the STS on an intuitive level to smooth the exposition.

Plan for finishing the proof. Let us now come back to our example program. We already described the STS we are going to use for the verification (Figure 4). The next step in the proof is thus to initialize said STS.

Remember our current context is (4). When allocating an STS, we get to pick its initial state – that would be INIT, of course. We have to provide $\varphi(\text{INIT})$ to initialize the STS, so we give up ownership of l and c. In exchange, we obtain StsSt and the tokens. Our context is now

$$\text{StsSt}(\text{INIT}, \{[\text{S}], [\text{R}]\}) * \text{source}(i, e_{\text{comm}}(c)) \tag{5}$$

The next command executed in both programs is fork. We are thus going to apply ML-FORK and prove the step shift using SRC-FORK. The two remaining premises of ML-FORK are the following two Hoare triples:

$$\{\text{StsSt}(\text{INIT}, [\text{S}]) * \text{source}(j, \text{send}(c_{\text{left}}, 42))\} \text{ heapSend } l \, 42 \, \{\text{Stopped}\} \tag{6}$$

$$\{\text{StsSt}(\text{INIT}, [\text{R}]) * \text{source}(j, \text{let } (\_, v) = \text{recv}(c_{\text{right}}) \text{ in } v)\}$$
$$\quad \text{let } (\_, v) = \text{heapRecv } l \text{ in } v \tag{7}$$
$$\{n. \, n = 42 * \text{source}(j, 42)\}$$

Showing these will complete the proof. The post-condition Stopped of (6) is mandated by ML-FORK; we will discuss it when verifying that Hoare triple. Note that we are splitting the StsSt to hand the two tokens that we own to two different threads.

Verifying the sender. To prove the sending Hoare triple (6), the context we have available is $\text{StsSt}(\text{INIT}, [\text{S}]) * \text{source}(j, \text{send}(c_{\text{left}}, 42))$, and the code we wish to verify is (unfolding the definition of heapSend, and performing some pure reductions):

$$\text{let } l_{new} = \text{ref none in } l := \text{some } (l_{new}, 42); l_{new}$$

The allocation is easily handled with ML-ALLOC, and it turns out we don't even need to remember anything about the returned l_{new}.

The next step is the core of this proof: showing that we can change the value stored in l. Notice that we do not own $l \hookrightarrow \_$; the STS "owns" l as part of its interpretation. So we will *open* the STS to get access to l.

Looking at Figure 4, we can see that doing the transition from INIT to SENT requires the token $[\text{S}]$, *which we own* – as a consequence, nobody else could perform this transition. It follows that the STS is currently in state INIT. We obtain $\varphi(\text{INIT})$, so that we can apply ML-STORE with SRC-SEND-LEFT, yielding

$$l \hookrightarrow \text{some } (l', 42) * c \hookrightarrow_{\text{s}} ([], []) * \text{source}(j, c_{\text{left}}) \tag{8}$$

To finish up accessing the STS, we have to pick a new state and show that we actually possess the tokens to move to said state. In our case, we *cannot* pick RECEIVED, since we do not own the token $[\text{R}]$ necessary for that step. Instead, we

will pick SENT and give up our token. This means we have to establish $\varphi(\text{SENT})$. Doing so consumes most of our context (8), leaving only $\text{source}(j, c_{\text{left}})$. What remains to be done? We have to establish the post-condition of our triple (6), which is Stopped. By SRC-STOPPED, this immediately follows from the fact that we reduced the source thread to c_{left}, which is a value.

Notice that this last step was important: We showed that when the MiniML thread terminates, so does the source thread. The original fork rule for Iris allows picking *any* post-condition for the forked-off thread, because nothing happens any more with this thread once it terminates. However, we wish to establish that if all MiniML threads terminate, then so do all source threads – and for this reason, ML-FORK forces us to prove Stopped, which asserts that all the threads we keep track of have reduced to a value. This finishes the proof of the sender.

Verifying the receiver. The next (and last) step in establishing the refinement (2) is to prove the Hoare triple for the receiving thread (7). This is the target code to verify:

$$\text{let } (\_, v) = \text{heapRecv } l \text{ in } v$$

Since heapRecv is a recursive function, we use ML-REC, which says that we can assume that recursive occurrences of heapRecv have already been proven correct. It may be surprising to see this rule – after all, rules like ML-REC are usually justified by saying that all we do is partial correctness. Notice, however, that we are *not* showing that E_{ex} terminates. All we show is that, *if* E_{ex} diverges, then so does e_{ex}. That is, we are establishing termination-*preservation*, not termination.

In continuing the proof, we thus get to assume correctness of the recursive call. Our current context is

$$\text{StsSt}(\text{INIT}, [\text{R}]) * \text{source}(j, \text{let } (\_, v) = \text{recv}(c_{\text{right}}) \text{ in } v) \qquad (9)$$

and the code we are verifying is

$$\text{match } !l \text{ with none} \Rightarrow \text{heapRecv } l \mid \text{some } (l', v) \Rightarrow (l', v) \text{ end}$$

with post-condition $(\_, n). n = 42 * \text{source}(j, 42)$.

The first command of this program is $!l$. To access l, we have to again open the STS. Since we own $[\text{R}]$, we can rule out being in state RECEIVED. We perform a case distinction over the remaining two states.

– If we are in INIT, we get $l \hookrightarrow \text{none} * c \hookrightarrow_{\text{s}} ([], [])$ from the STS's $\varphi(\text{RECEIVED})$. We use ML-LOAD with SRC-RECV-RIGHT-MISS. Notice how we use $c \hookrightarrow_{\text{s}} ([], [])$ to justify performing an "idle" step in the source. This is crucial – after all, we are potentially looping indefinitely in the target, reading l over and over; we have to exhibit a corresponding diverging execution in the source.
  Since we did not change any state, we close the invariant again in the INIT state. Next, the program executes the none arm of the match: heapRecv l. Here, we use our assumption that the recursive call is correct to finish the proof.

– Otherwise, the current state is SENT, and we obtain $l \hookrightarrow \mathsf{some}\ (\_, 42) * c \hookrightarrow_{\mathsf{s}}$
  $([42], [])$. We use ML-LOAD with SRC-RECV-RIGHT-HIT; this time we know that
  the recv in the source will succeed. We also know that we are loading $(\_, 42)$
  from l. We pick RECEIVED as the next state (giving up our STS token), and
  trivially establish $\varphi(\text{RECEIVED})$. We can now throw away ownership of l and
  c as well as $\mathsf{StsSt}(\text{RECEIVED})$ since we no longer need them – we can do this
  because all these assertions are affine.
  All that remains is the source thread:

$$\mathsf{source}(j, \mathsf{let}\ (\_, v) = (c_{\mathsf{right}}, 42)\ \mathsf{in}\ v)$$

  Next, the target program will execute the some branch of the match. To
  finish, we need to justify the post-condition: $(\_, n).\ n = 42 * \mathsf{source}(j, 42)$. We
  already established that the second component of the value loaded from l is
  42, and the source thread is easily reduced to 42 as well.

This finishes the proof of (7) and therefore of (2): we proved that $e_{\mathrm{ex}} \sqsubseteq E_{\mathrm{ex}}$.

4   Soundness of the Logic

We have seen how to use our logic to establish a refinement for a particular
simple instance of our translation. We now need to show that this logic is sound.

As already mentioned, our logic is an extension of Iris, so we need to adapt
the soundness proof of Iris [22]. The two extensions that were described in §3.1
are:

1. We add a notion of a *step shift*, which is used to simulate source program
   threads.
2. We move from an affine logic to a linear logic. This is needed to capture the
   idea that some resources (like source) represent *obligations* that cannot be
   thrown away.

In this section we describe how we adapt the semantic model of Iris to handle
these changes. Although our extensions sound simple, the modification of the
model requires some care. Many of the features we used in §3, such as STSs [23]
and reasoning about the source language, are *derived* constructions that are not
"baked-in" to the logic. As we change the model, we need to ensure that all
of these features can still be encoded. We also strive to keep our extensions as
general as possible so as to not unnecessarily restrict the flexibility of Iris.

Brief review of the Iris model. We start by recalling some aspects of the Iris
model [22] that we modify in our extensions. A key concept is the notion of a
resource. Resources describe the physical state of the program as well as addi-
tional *ghost state* that is added for the purpose of verification and used, *e.g.,* to
interpret STSs or the assertions talking about source programs. Resources are
instances of a partial commutative monoid-like algebraic structure; in particular,
two resources a, b can be *composed* to $a \cdot b$. This operation is used to combine

resources held by different threads. When the composition $a \cdot b$ is defined, the elements a and b are said to be *compatible*. Iris always ensures that the resources held by different threads are compatible. This guarantees that, *e.g.,* different threads cannot own the same channel or the same STS token. The operation also gives rise to a pre-order on resources, defined as $a_1 \preccurlyeq a_2 \triangleq \exists a_3. \, a_1 \cdot a_3 = a_2$, *i.e.,* a_1 is included in a_2 if the former can be *extended* to the latter by adding some additional resource a_3.

Ideally, we would just interpret an assertion P as a set of resources. For technical reasons (that we will mostly gloss over), Iris needs an additional component: the *step-index n*. An assertion is thus interpreted as a set of pairs (n, a) of step-indices and resources. We write $n, a \models P$ to indicate that $(n, a) \in P$, and read this as saying that a satisfies P for n steps of the target program's execution.

Iris furthermore demands that assertions (interpreted as sets) satisfy two *closure properties*: They must be closed under larger resources and smaller step-indices. Formally:

1. If $n, a \models P$ and $a \preccurlyeq a'$, then $n, a' \models P$.
2. If $n, a \models P$ and $n' \leq n$, then $n', a \models P$.

The first point above makes Iris an *affine* as opposed to a linear logic: we can always "add-on" more resources and continue to satisfy an assertion. Put differently, there is no way to state an *upper bound* on our resources. The second point says that if P holds for n steps, then it also holds for fewer than n steps.

To give a model to assertions like $l \hookrightarrow v$, we need a function $\mathsf{HeapRes}(l, v)$ describing, as a resource, a heap which maps location l to v. We then define:

$$n, a \models l \hookrightarrow v \quad \text{iff} \quad \mathsf{HeapRes}(l, v) \preccurlyeq a$$

Notice the use of $\preccurlyeq$, ensuring that the closure property (1) holds.

Equipping Iris with linear assertions. In order to move to a linear setting with minimal disruption to the existing features of Iris, we replace the judgment $n, a \models P$ with $n, a, b \models P$. That is, assertions are now sets of triples: a step-index and *two* resources. The downward closure condition on n and the upward closure condition on a still apply, but we do not impose such a condition on b: this second resource will represent the "linear piece" of an assertion. Crucially, whereas affine assertions like $l \hookrightarrow v$ continue to "live" in the a piece, the linear source resides in b:

$$
\begin{aligned}
n, a, b &\models l \hookrightarrow v & &\text{iff} \quad \mathsf{HeapRes}(l, v) \preccurlyeq a \wedge b = \varepsilon \\
n, a, b &\models \mathsf{source}(i, E, d) & &\text{iff} \quad \mathsf{SourceRes}(i, E, d) = b
\end{aligned}
$$

where ε is the unit of the monoid. We assume $\mathsf{SourceRes}(i, E, d)$ to define, as a resource, a source thread i executing E with d delay steps left.

As we can see, source describes the *exact* linear resources b that we own, whereas $\hookrightarrow$ merely states a *lower bound* on the affine resources a (due to the

upwards closure on a). Notice that a and b are both elements of the same set of resources; it is just their treatment in the closure properties of assertions which makes one affine and the other linear. Because there is no upward closure condition on the second monoid element, the resulting logic is not affine: if $n, a, b \models P * Q$, then it is not necessarily the case that $n, a, b \models P$.

We define the affine modality by:

$$n, a, b \models \mathcal{A}(P) \quad \text{iff} \quad n, a, b \models P \wedge b = \varepsilon$$

This says that in addition to satisfying P, b should equal the unit of the monoid. That is, the linear part is "empty"; there are no obligations encoded in P. That makes it sound to throw away P or to frame it.

The advantage of this "two world" model is that it does not require us to change many of the encodings already present in Iris, like STSs.

Step Shifts. We are now ready to explain the ideas behind the *step shift*. Remember the goal here is to account for the steps taken in the source program, in a way that we can prove refinements by proving Hoare triples (HT-REFINE). This is subtle because by the definition of refinement (§2.4), we need to make statements even about infinite executions, *i.e.,* executions that never have to satisfy the post-condition.

The key idea is to equip the resources of Iris with a relation that represents a notion of *taking a (resource) step*. We write $a \curvearrowright b$, and say that a *steps to* b. We will then pick the resources in such a way as to represent the status of a source program,[8] and we define the resource step to be taking a step in the source program. All the other components of the resource, like STSs, will not be changed by resource steps.

Recall that the resources owned by different threads always need to be compatible. To ensure this, we define a relation that performs a step while maintaining compatibility with the resources owned by other threads. Formally, a *frame-preserving step-update* $a, b \rightsquigarrow a', b'$ holds if $b \curvearrowright b'$ and for all c such that $a \cdot b \cdot c$ is defined, so is $a' \cdot b' \cdot c$. The intuition is that, if a thread owns some resources a and b, that restricts the ownership of other threads to *frames c* that are compatible with a and b. Since a' and b' are also compatible with the frame, the step is guaranteed not to interfere with resources owned by other threads.

These frame-preserving step-updates are reflected into the logic through the *step shift* assertions: $P \Rrightarrow Q$ holds if, whenever some resources satisfy P, it is possible to perform a frame-preserving step-update to resources satisfying Q.

We then connect Hoare triples to these resource steps. To this end, we change the definition of Hoare triples so that whenever a target thread takes a step, we have to also take a step on our resources. This gives rise to the proof rules in Figure 3, which force the user of the logic to perform a step shift alongside every step of the MiniML program. We also enforce that forked-off threads must have a post-condition of Stopped, ensuring that target language threads cannot stop executing while source language threads are still running.

---

[8] Iris is designed to be parametric in the choice of resources, so we can pick a particular resource for this source language and still use most of the general Iris machinery.

Soundness of the refinement. Having extended the definition of Hoare triples in this way, we can prove our refinement theorem. Recall that the definition of refinement had three parts. For each of these parts, we proved an adequacy theorem for our extensions relating Hoare triples to properties of program executions. These theorems are parameterized by the kind of resource picked by the user, and in particular the kind of resource *step*. Below, we show these theorems specialized to the case where resource steps correspond to source language steps.

The first refinement condition, which says that the target program must not get stuck, follows from a "safety" theorem that was already present in the original Iris:

Lemma 2. *If* $\{source(i, E, d)\}\, e\, \{v.\, source(i, V, 0) * \mathcal{A}(v \approx V)\}$ *holds and we have* $([e], \emptyset) \to^* ([e_1, \ldots, e_n], \sigma)$, *then each* e_i *is either a value or it can take a step in state* σ.

The second refinement condition says that if the execution of e terminates, then there should be a related terminating execution in the source. Remember that the definition of the Hoare triple requires us to take a step in the source whenever the target steps (modulo a finite number of delays). Hence a proof of such a triple must have "built-up" the desired source execution:

Lemma 3. *If* $\{source(i, E, d)\}\, e\, \{v.\, source(i, V, 0) * \mathcal{A}(v \approx V)\}$ *holds and we have* $([e], \emptyset) \to^* ([v_1, \ldots, v_n], \sigma)$, *then there exists* $V_1, E_2, \ldots, E_m, \Sigma$ *s.t.* $([E], \emptyset) \to^* ([V_1, E_2, \ldots, E_m], \Sigma)$. *Moreover, each* E_i *is either stuck or a value, and* $v_1 \approx V_1$.

Here, we are already making crucial use of both linearity of source and the fact that forked-off threads must have post-condition Stopped: if it were not for these requirements, even when all target threads terminated with a value v_i, we could not rule out the existence of source threads that can go on executing.

Finally, we come to the third condition, which says fair diverging executions of the target should correspond to fair diverging executions of the source:

Lemma 4. *If* $\{source(i, E, d)\}\, e\, \{v.\, source(i, V, 0) * \mathcal{A}(v \approx V)\}$ *holds and* $([e], \emptyset)$ *has a diverging execution, then* $([E], \emptyset)$ *has a diverging execution as well. Moreover, if the diverging target execution is fair, then the source execution is too.*

This is the hardest part of the soundness proof. We would like to start by arguing that, just as for the finite case, if the target program took an infinite number of steps, then the proof of the refinement triple must give a corresponding infinite number of steps in the source program. Unfortunately, this argument is not so simple because of step-indexing.

In Iris, Hoare triples are themselves step-indexed sets. We write $n \models \{P\}\, e\, \{Q\}$ to say that the triple holds at step-index n. Then, when we say we have proved a Hoare triple, we mean the triple holds for all step-indices n and all resources satisfying the precondition. As is usual with step-indexing, when a triple $\{P\}\, e\, \{Q\}$ holds for step-index n, that means when the precondition is satisfied, execution of e is safe for up-to n steps, and if it terminates within those n steps, the post-condition holds. In our case, it also means that each step of the target program gives a step of the source program, for up to n target steps.

This restriction to only hold "up to n steps" arises due to the way Hoare triples are defined in the model: when proving the Hoare triple at step-index n, if e steps to e', we are only required to show $(n-1) \models \{P'\}\ e'\ \{Q\}$ for some P'.

The restriction to a finite number of steps did not bother us for Lemma 2 and Lemma 3. Since they only deal with finite executions, and the Hoare triple holds for *all* starting indices n, we can simply pick n to be greater than the finite execution we are considering. But we cannot do this when we want to prove something about a diverging execution of the target. Whatever n we start with, it is not big enough to get the infinite source execution we need.

Bounded non-determinism, infinite executions, and step-indexing. Our insight is that when the source language has only *bounded non-determinism*, we can set up a more careful inductive argument. By bounded non-determinism, we mean that each configuration $([E, ...], \Sigma)$ only has *finitely many* possible successor configurations. The key result is the following quantifier inversion lemma:

Lemma 5. *Let R be a step-indexed predicate on a finite set X. Then:*

$$(\forall n.\, \exists x.\, n \models R(x)) \Rightarrow (\exists x.\, \forall n.\, n \models R(x))$$

Proof. By assumption, for each n, there exists $x_n \in X$ such that $n \models R(x_n)$. Since X is finite, by the pigeon-hole principle, there must be some $x \in X$ such that $m \models R(x)$ for infinitely many values of m. Now, given arbitrary n, this means there exists $m > n$ such that $m \models R(x)$. Since step-indexed predicates are downward-closed, $n \models R(x)$. Hence $\forall n.\, n \models R(x)$.

Ignoring delay steps for the moment, we apply this lemma to our setting to get:

Lemma 6. *Suppose e steps to e' and $\forall n.\, \exists P_n.\, n \models \{\mathsf{source}(i, E) * P_n\}\ e\ \{Q\}$. Then, $\exists E'$ such that E steps to E' and $\forall n.\, \exists P_n'.\, n \models \{\mathsf{source}(i, E') * P_n'\}\ e'\ \{Q\}$.*

Proof. Let X by the set of E' that E can step to, which we know to be finite.[9] Consider the step-indexed predicate R on X defined by $n \models R(E') \triangleq (E \to E' \wedge \exists P_n'.\, n \models \{\mathsf{source}(i, E') * P_n'\}\ e'\ \{Q\})$. By assumption, for each $n > 0$, $n \models \{\mathsf{source}(i, E) * P_n\}\ e\ \{Q\}$ for some P_n. The definition of Hoare triples implies that there exists some E' such that $(n-1) \models R(E')$. Thus, $\forall n.\exists E'.\, n \models R(E')$, so we can apply Lemma 5 to get the desired result.

Notice that in the conclusion of Lemma 6, if e' takes another step, we can apply Lemma 6 again to the triples for e'. So, given some initial triple $\{\mathsf{source}(i, E)\}\ e\ \{Q\}$ and a diverging execution of e, by induction we can repeatedly apply Lemma 6 to construct an infinite execution of the source program. Finally, we prove that if the execution of e was fair, this source execution will be fair as well, giving us Lemma 4. Of course, for the full mechanized proof we have to take into account the delay steps and consider the case where the target thread multiple source threads. But all of these are *finite* additional possibilities, they do not fundamentally change the argument sketched above.

---

[9] To be precise we ought to mention the initial states σ and Σ that e and E run in and assume they satisfy the precondition of the triple.

5   Proof of Compiler Correctness

We now give a brief overview of our proof of Theorem 1. Recall that we want to show that if E is a well-typed source expression, then $\widehat{E} \sqsubseteq E$.

Our proof is a binary logical relations argument. We interpret each type τ as a relation on values from the target and source language, writing $v \simeq^{\mathcal{V}} V : \tau$ to say that v and V are related at type τ. However, following the example of [25, 24], these are relations *in our refinement logic*, which means we can use all of the constructs of the logic to describe the meaning of types. We then prove a fundamental lemma showing that well-typed expressions are logically related to their translation. Next, we show that our logical relation implies the triple used in Ht-refine. Theorem 1 is then a direct consequence of these two lemmas.

Details of these proofs can be found in the appendix; here we focus on the definition of the logical relation itself. For most types, the interpretation is straightforward and fairly standard. For instance, $v \simeq^{\mathcal{V}} V :$ Int holds exactly when $v = V = n$, for some integer n. The important exception, of course, is the interpretation of session types, in which we need to relate the encoding of channels as linked-lists to the source language's primitive buffers.

Sessions as an STS. To interpret session types, we generalize the state transition system from the example in §3 to handle the more complicated "protocols" that session types represent.

What should the states of this STS be? In the STS used in §3, we had three states: INIT, in which the message had not been sent; SENT, where a message had been sent from the left end-point, but not received; and RECEIVED, where the message had now been received at the right end-point. In the general case, we will have more than one message, so our states need to track how many messages have been sent/received on each end-point. We also need to know the "current" type of the end-points, but notice that if we know the starting type of an end-point, and how many messages have been sent/received on it, we can always recover these current types. We write S^n for the type after n messages have been sent/received starting from S.

We also need to know which heap locations l_l and l_r currently represent the end-points of the channel. All together then, the states will be tuples (n_l, n_r, l_l, l_r) describing how many messages have been sent/received on each end-point, and the corresponding heap locations.

Remember that we also need to define the tokens and transitions associated with each state of our STS. The transitions are simple: we can either advance the left end-point, incrementing n_l and updating l_l, and similarly for the right end-point. For the tokens, recall that in our example proof, we had [S] and [R] tokens used by each thread to advance the state when they had interacted with their respective end-points. In general, the threads will now use the end-points multiple times, so we need a token for each of these uses on both sides. Concretely, we will have two kinds of tokens, [Left n] and [Right n], which are used when advancing the left and right end-point counter to n, respectively.

To complete the description of the STS, we have to talk about the interpretation of the states. This interpretation has to relate the messages in the source channel's current buffers to the nodes in the linked list on the target heap. The individual messages should, of course, be related by our logical relation ($\simeq^\mathcal{V}$). We lift this relation to lists of messages ($\simeq^\mathcal{L}$) as follows:

$$[] \simeq^\mathcal{L} [] : S \qquad \frac{\text{L-cons}}{vL_\mathsf{h} \simeq^\mathcal{L} VL_\mathsf{c} : ?\tau.\,S}$$

For now, ignore the $\triangleright$ symbol. The left rule says that two empty lists are equivalent at any session type. The right rule says two lists are related *at a receive type* $?\tau.\,S$, if their heads are related under τ, and the remainders of each list are related at S. It is important that this is a receive type: if the current type of the end-point is a send type, then there should not be any messages in its receive buffer, so the rule for empty lists is the only one that applies.

We can now give our state interpretation, φ, which is parameterized by (a) the starting type S of the left end-point (the right end-point's starting type is by necessity dual so there is no need to track it), and (b) the name c of the channel:

$$\varphi_{S,c}(n_\mathsf{l}, n_\mathsf{r}, l_\mathsf{l}, l_\mathsf{r}) \triangleq \exists L_\mathsf{c}, L_\mathsf{h}.\quad \Big(c \hookrightarrow_\mathsf{s} (L_\mathsf{c}, []) * \mathsf{linklist}(L_\mathsf{h}, l_\mathsf{l}, l_\mathsf{r}) * \tag{10}$$

$$(L_\mathsf{h} \simeq^\mathcal{L} L_\mathsf{c} : S^{n_\mathsf{l}}) * n_\mathsf{l} + |L_\mathsf{c}| = n_\mathsf{r} \Big) \vee \dots \tag{11}$$

Let us explain this piece by piece. To start, we have that there exists a list of *source* values L_c and a list of *target* values L_h, representing the messages that are stored in the buffer right now. We then distinguish between two cases: either the first buffer is empty or the second buffer is empty. We omit the second case (corresponding to the second disjunct) because it is symmetric. In the first case, the channel's first buffer contains L_c and the second buffer is empty (10, left). On the target side, the buffer is represented as a linked list from l_l to l_r containing the values L_h (10, right). Of course, the lists of values need to be related according to the end-point's current type S^{n_l} (11, left). Finally, the number of messages sent/received through the left end-point, plus the number of messages still in the buffer, should equal the total number of messages sent/received through the right end-point (11, right). Therefore, when these remaining messages are received by the left end-point, the two types will again be dual.

Informally then, the value relation at session types $l \simeq^\mathcal{V} c_s : S$ says that there exists an appropriate STS and tokens for the session S which relates l and c_s. We can then prove Hoare triples for the message-passing primitives that manipulate this STS. For instance, for $\mathsf{heapRecv}$ we have (omitting delay steps):

$$\{\mathsf{source}(i, K[\mathsf{recv}(c_s)]) * l \simeq^\mathcal{V} c_s : ?\tau.\,S\} \qquad \mathsf{heapRecv}\ l$$

$$\{(l', v).\, \exists V.\, \mathsf{source}(i, K[(c_s, V)]) * (v \simeq^\mathcal{V} V : \tau) * l' \simeq^\mathcal{V} c_s : S\}$$

This triple closely corresponds to the typing rule Recv (Figure 1): typing judgments in the premise become value relations in the pre-condition, and the

conclusion is analogously transformed into the postconditon. Indeed, the proof of the fundamental lemma for the logical relation essentially just appeals to these triples.

There is something we have glossed over: when we defined the logical relation, we used the STS, but the STS interpretation used the logical relation! This circularity is the reason for the $\triangleright$ symbol guarding the recursive occurrence of $(\simeq^{\mathcal{V}})$ in L-cons. The details are spelled out in the appendix.

6   Conclusion and Related Work

We have presented a logic for establishing *fair, termination-preserving* refinement of higher-order, concurrent languages. To our knowledge, this is the first logic combining higher-order reasoning (and in particular, step-indexing) with reasoning for termination-sensitive concurrent refinement. Moreover, we applied this logic to verify the correctness of a compiler that translates a session-typed source language with channels into an ML-like language with a shared heap.

All of these results have been fully mechanized in Coq. Our mechanization builds on the Coq development described in Jung et al. [22] and the proof-mode from Krebbers et al. [24]. The proofs use the axioms of excluded middle and indefinite description. The proof scripts can be found online [1].

Second Case Study. Our logic is not tied to this source language and translation: we have used it to mechanize a proof that the Craig-Landin-Hagersten queue lock [11, 31] refines a ticket lock. Further details can be found in the appendix.

Linearity. Linearity has been used in separation logics to verify the absence of memory leaks: if heap assertions like $l \hookrightarrow v$ are linear, and the only way to "dispose" of them is by freeing the location l, then post conditions must mention all memory that persists after a command completes [20]. Our treatment of linearity has limitations that make it unsuitable for tracking resources like the heap. First, in our logic, only affine assertions can be framed (see Ht-frame), because framing could hide the obligation to perform steps on source threads. Of course, for resources like the heap this would be irrelevant, and this rule could be generalized. Second, linear resources cannot be put in STS interpretations, so they cannot be shared between threads. Since STSs are implemented in terms of a more primitive feature in Iris called *invariants*, which are affine, allowing linear resources to be put inside would circumvent the precise accounting that motivates linearity in the first place. Thus, we would need to extend Iris with a useful form of "linear" shared invariants, which we leave to future work.

Session Types. Starting from the seminal work of Honda [19], a number of session-type systems have been presented with different features [45, 17, 10, 39, 42] (among many others). The language presented here is a simplified version of the one in Gay and Vasconcelos [17]. Wadler [42] has shown that a restricted subset of the language in [17] does enjoy a deadlock freedom property. This property

holds only when the type system is *linear*, like the original in [17]. Pérez et al. [35] and Caires et al. [9] give logical relations for session-typed languages, which they use to prove strong normalization and contextual equivalence results. Their logical relation is defined "directly", instead of translating into an intermediary logic. Early versions of another session-typed system [43] used a ring-buffer to represent channels instead of linked lists, which would be interesting to verify.

Logics for Concurrency, Termination, and Refinement. There is a vast literature on program logics for concurrency [34, 8, 40, 23, 22, 14, 37, 12, 36, 33, 15, 41, 16, 29, 30, 28, 18]. Indeed, the reason for constructing a logical relation on top of a program logic, as in Krogh-Jespersen et al. [25], is so that we can take advantage of the many ideas that have proliferated in this community.

Focusing on logics for refinement and termination properties: Benton [4] pioneered the use of a *relational* Hoare logic for showing the correctness of compiler transformations in the sequential setting. Yang [44] generalized this to relational separation logic. We have already described [40], which developed a higher-order concurrent separation logic for termination-insensitive refinement. Liang et al. [29] also allow non-terminating programs to refine terminating ones. This was extended in [30] for a termination-preserving refinement, but this deals with termination-preservation *without* fairness. Most recently Liang and Feng [28] addressed fair termination-preserving refinement. In their logic, threads can explicitly reason about how their actions may or may not further delay other threads, which is more general than our approach and may be needed for verifying some of the examples they consider. It would be interesting to adapt this more explicit fairness reasoning to the higher-order setting.

Hoffmann et al. [18] features a concurrent separation logic for total correctness. Threads own resources called "tokens", which must be "used up" every time a thread repeats a while loop. This "using up" of tokens inspired our step shifts. Later, da Rocha Pinto et al. [36] generalized this by using ordinals instead of tokens: threads decrease the ordinal they own as they repeat a loop. This is useful for languages with unbounded non-determinism. Our technique for coping with step-indexing in §4 relied on bounded non-determinism. It may be possible to remove this limitation by using *transfinite* step-indexing [5, 38] instead.

References

1. Website with Coq development. http://www.cs.cmu.edu/~jtassaro/papers/iris-refinement (2016)
2. America, P., Rutten, J.: Solving reflexive domain equations in a category of complete metric spaces. JCSS 39(3), 343–375 (1989)
3. Appel, A., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. TOPLAS 23(5), 657–683 (2001)
4. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL (2004)
5. Birkedal, L., Bizjak, A., Schwinghammer, J.: Step-indexed relational reasoning for countable nondeterminism. Logical Methods in Computer Science 9(4) (2013)
6. Birkedal, L., Bizjak, A.: A taste of categorical logic — tutorial notes (Oct 2014), available at http://users-cs.au.dk/birke/modures/tutorial/categorical-logic-tutorial-notes.pdf
7. Birkedal, L., Støvring, K., Thamsborg, J.: The category-theoretic solution of recursive metric-space equations. Theoretical Computer Science 411(47), 4102–4122 (2010)
8. Brookes, S.D.: Variables as resource for shared-memory programs: Semantics and soundness. Electr. Notes Theor. Comput. Sci. 158, 123–150 (2006)
9. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: ESOP. pp. 330–349 (2013)
10. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR. pp. 222–236 (2010)
11. Craig, T.S.: Building fifo and priority-queueing spin locks from atomic swap. Tech. Rep. 93-02-02, Computer Science Department, University of Washington (1993)
12. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: ECOOP. pp. 207–231 (2014)
13. David, T., Guerraoui, R., Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In: SOSP (2013)
14. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. pp. 504–528 (2010)
15. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: Compositional reasoning for concurrent programs. In: POPL (2013)
16. Feng, X.: Local rely-guarantee reasoning. In: POPL. pp. 315–327 (2009)
17. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. 20(1), 19–50 (2010)
18. Hoffmann, J., Marmar, M., Shao, Z.: Quantitative reasoning for proving lock-freedom. In: LICS. pp. 124–133 (2013)
19. Honda, K.: Types for dyadic interaction. In: CONCUR. pp. 509–523 (1993)
20. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL. pp. 14–26 (2001)
21. Jones, C.B.: Tentative steps toward a development method for interfering programs. TOPLAS 5(4), 596–619 (1983)
22. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order ghost state. In: ICFP (2016), (to appear)
23. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL. pp. 637–650 (2015)

24. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: POPL (2017), to appear.
25. Krogh-Jespersen, M., Svendsen, K., Birkedal, L.: A relational model of types-and-effects in higher-order concurrent separation logic. In: POPL (2017), to appear.
26. Lea, D.: The java.util.concurrent synchronizer framework. Sci. Comput. Program. 58(3), 293–309 (2005)
27. Lehmann, D.J., Pnueli, A., Stavi, J.: Impartiality, justice and fairness: The ethics of concurrent termination. In: Automata, Languages and Programming. pp. 264–277 (1981)
28. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: POPL. pp. 385–399 (2016)
29. Liang, H., Feng, X., Fu, M.: Rely-guarantee-based simulation for compositional verification of concurrent program transformations. ACM Trans. Program. Lang. Syst. 36(1), 3 (2014)
30. Liang, H., Feng, X., Shao, Z.: Compositional verification of termination-preserving refinement of concurrent programs. In: CSL-LICS. pp. 65:1–65:10 (2014)
31. Magnusson, P.S., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: International Symposium on Parallel Processing. pp. 165–171 (1994)
32. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9(1), 21–65 (1991)
33. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: ESOP. pp. 290–310 (2014)
34. O'Hearn, P.: Resources, concurrency, and local reasoning. TCS 375(1), 271–307 (2007)
35. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations for session-based concurrency. In: ESOP. pp. 539–558 (2012)
36. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular termination verification for non-blocking concurrency. In: ESOP. pp. 176–201 (2016)
37. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: ESOP. pp. 149–168 (2014)
38. Svendsen, K., Sieczkowski, F., Birkedal, L.: Transfinite step-indexing: Decoupling concrete and logical steps. In: ESOP. pp. 727–751 (2016)
39. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: ESOP. pp. 350–369 (2013)
40. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In: ICFP. pp. 377–390 (2013)
41. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: CONCUR. pp. 256–271 (2007)
42. Wadler, P.: Propositions as sessions. J. Funct. Program. 24(2-3), 384–418 (2014)
43. Willsey, M., Prabhu, R., Pfenning, F.: Design and implementation of concurrent C0. In: Linearity (2016)
44. Yang, H.: Relational separation logic. TCS 375(1–3), 308–334 (2007)
45. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electr. Notes Theor. Comput. Sci. 171(4), 73–93 (2007)

Contents of Appendices

A   Extensions to Iris 2.0

This section is a reproduction of the manual for the Iris 2.0 logic (taken from a revised version of the technical appendix of Jung et al. [22]), with the modifications needed for our extension. This reproduction is done with permission of

the authors of Jung et al. [22]. Our modifications are highlighted in blue, like so. Hence, this section is best viewed in color. If a section or paragraph heading is highlighted in blue, everything in that section is new.

Disclaimer: The Coq development of the logic is taken to be authoritative. Any discrepancies between this document and the Coq code should therefore be regarded as errors. Some important rules developed in the Coq proof may have been left out. Moreover, any mistakes in this document may have been introduced by the present authors: readers interested in the original Iris should consult its documentation. Also, the reader should be aware that the latest version of Iris has departed significantly from the version with which we began our extensions.

A.1 Algebraic Structures

COFE The model of Iris lives in the category of *Complete Ordered Families of Equivalences* (COFEs). This definition varies slightly from the original one in [6].

Definition 7 (Chain). *Given some set T and an indexed family $(\stackrel{n}{=} \subseteq T \times T)_{n \in \mathbb{N}}$ of equivalence relations, a* chain *is a function $c : \mathbb{N} \to T$ such that $\forall n, m. \, n \leq m \Rightarrow c(m) \stackrel{n}{=} c(n)$.*

Definition 8. *A* complete ordered family of equivalences *(COFE) is a tuple $(T, (\stackrel{n}{=} \subseteq T \times T)_{n \in \mathbb{N}}, \lim : \mathrm{chain}(T) \to T)$ satisfying*

$$\forall n. \, (\stackrel{n}{=}) \text{ is an equivalence relation} \qquad (\textsc{cofe-equiv})$$

$$\forall n, m. \, n \geq m \Rightarrow (\stackrel{n}{=}) \subseteq (\stackrel{m}{=}) \qquad (\textsc{cofe-mono})$$

$$\forall x, y. \, x = y \Leftrightarrow (\forall n. \, x \stackrel{n}{=} y) \qquad (\textsc{cofe-limit})$$

$$\forall n, c. \, \lim(c) \stackrel{n}{=} c(n) \qquad (\textsc{cofe-compl})$$

The key intuition behind COFEs is that elements x and y are n-equivalent, notation $x \stackrel{n}{=} y$, if they are *equivalent for n steps of computation, i.e.,* if they cannot be distinguished by a program running for no more than n steps. In other words, as n increases, $\stackrel{n}{=}$ becomes more and more refined ($\textsc{cofe-mono}$)— and in the limit, it agrees with plain equality ($\textsc{cofe-limit}$). In order to solve the recursive domain equation in §A.6 it is also essential that COFEs are *complete, i.e.,* that any chain has a limit ($\textsc{cofe-compl}$).

Definition 9. *An element $x \in T$ of a COFE is called* discrete *if*

$$\forall y \in T. \, x \stackrel{0}{=} y \Rightarrow x = y$$

A COFE A is called discrete *if all its elements are discrete. For a set X, we write ΔX for the discrete COFE with $x \stackrel{n}{=} x' \triangleq x = x'$*

Definition 10. *A function* $f : T \to U$ *between two COFEs is* non-expansive *(written* $f : T \xrightarrow{ne} U$ *) if*

$$\forall n, x \in T, y \in T. \, x \stackrel{n}{=} y \Rightarrow f(x) \stackrel{n}{=} f(y)$$

It is contractive *if*

$$\forall n, x \in T, y \in T. \, (\forall m < n. \, x \stackrel{m}{=} y) \Rightarrow f(x) \stackrel{n}{=} f(y)$$

Intuitively, applying a non-expansive function to some data will not suddenly introduce differences between seemingly equal data. Elements that cannot be distinguished by programs within n steps remain indistinguishable after applying f. The reason that contractive functions are interesting is that for every contractive $f : T \to T$ with T inhabited, there exists a *unique* fixed-point $fix(f)$ such that $fix(f) = f(fix(f))$.

Definition 11. *The category* $\mathcal{COFE}$ *consists of COFEs as objects, and non-expansive functions as arrows.*

Note that $\mathcal{COFE}$ is cartesian closed. In particular:

Definition 12. *Given two COFEs* T *and* U, *the set of non-expansive functions* $\left\{ f : T \xrightarrow{ne} U \right\}$ *is itself a COFE with*

$$f \stackrel{n}{=} g \triangleq \forall x \in T. \, f(x) \stackrel{n}{=} g(x)$$

Definition 13. *A (bi)functor* $F : \mathcal{COFE} \to \mathcal{COFE}$ *is called* locally non-expansive *if its action* F_1 *on arrows is itself a non-expansive map. Similarly,* F *is called* locally contractive *if* F_1 *is a contractive map.*

The function space $(-) \xrightarrow{ne} (-)$ is a locally non-expansive bifunctor. Note that the composition of non-expansive (bi)functors is non-expansive, and the composition of a non-expansive and a contractive (bi)functor is contractive. The reason contractive (bi)functors are interesting is that by America and Rutten's theorem [2, 7], they have a unique[10] fixed-point.

RA

---

[10] Uniqueness is not proven in Coq.

Definition 14. *A* resource algebra *(RA) is a tuple*
$(M, \mathcal{V} \subseteq M, |-| : M \to M^?, (\cdot) : M \times M \to M, (\curvearrowright) \subseteq M \times M)$ *satisfying:*

$$\forall a, b, c.\, (a \cdot b) \cdot c = a \cdot (b \cdot c) \tag{RA-ASSOC}$$

$$\forall a, b.\, a \cdot b = b \cdot a \tag{RA-COMM}$$

$$\forall a.\, |a| \in M \Rightarrow |a| \cdot a = a \tag{RA-CORE-ID}$$

$$\forall a.\, |a| \in M \Rightarrow ||a|| = |a| \tag{RA-CORE-IDEM}$$

$$\forall a, b.\, |a| \in M \wedge a \preccurlyeq b \Rightarrow |b| \in M \wedge |a| \preccurlyeq |b| \tag{RA-CORE-MONO}$$

$$\forall a, b.\, (a \cdot b) \in \mathcal{V} \wedge |a| \in M \wedge |b| \in M \Rightarrow$$
$$||a| \cdot |b|| = |a| \cdot |b| \tag{RA-CORE-DISTRIB}$$

$$\forall a, b.\, (a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V} \tag{RA-VALID-OP}$$

where $\qquad M^? \triangleq M \uplus \{\top\} \qquad\qquad a^? \cdot \top \triangleq \top \cdot a^? \triangleq a^?$

$$a \preccurlyeq b \triangleq \exists c \in M.\, b = a \cdot c \tag{RA-INCL}$$

RAs are closely related to *Partial Commutative Monoids* (PCMs), with two key differences:

1. The composition operation on RAs is total (as opposed to the partial composition operation of a PCM), but there is a specific subset $\mathcal{V}$ of *valid* elements that is compatible with the composition operation (RA-VALID-OP).
   This take on partiality is necessary when defining the structure of *higher-order* ghost state, CMRAs, in the next subsection.
2. Instead of a single unit that is an identity to every element, we allow for an arbitrary number of units, via a function $|-|$ assigning to an element a its *(duplicable) core* $|a|$, as demanded by RA-CORE-ID. We further demand that $|-|$ is idempotent (RA-CORE-IDEM) and monotone (RA-CORE-MONO) with respect to the *extension order*, defined similarly to that for PCMs (RA-INCL). Notice that the domain of the core is $M^?$, a set that adds a dummy element $\top$ to M. Thus, the core can be *partial*: not all elements need to have a unit. We use the metavariable $a^?$ to indicate elements of $M^?$. We also lift the composition ($\cdot$) to $M^?$. Partial cores help us to build interesting composite RAs from smaller primitives.
   Notice also that the core of an RA is a strict generalization of the unit that any PCM must provide, since $|-|$ can always be picked as a constant function.
3. We add an aditional relation $\curvearrowright$ that captures "taking a step" on a resource. In most cases this will be the full relation, as we have no useful notion of "stepping" such resources.But, for resources used for establishing refinements, this will correspond to taking some kind of step in the source program.

Definition 15. *It is possible to do a* frame-preserving update *from $a \in M$ to $B \subseteq M$, written $a \rightsquigarrow B$, if*

$$\forall a_{\mathrm{f}}^? \in M^?.\, a \cdot a_{\mathrm{f}}^? \in \mathcal{V} \Rightarrow \exists b \in B.\, b \cdot a_{\mathrm{f}}^? \in \mathcal{V}$$

We further define $a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$.

The assertion $a \rightsquigarrow B$ says that every element $a_{\mathrm{f}}^?$ compatible with a (we also call such elements *frames*), must also be compatible with some $b \in B$. Notice that $a_{\mathrm{f}}^?$ could be $\top$, so the frame-preserving update can also be applied to elements that have *no* frame. Intuitively, this means that whatever assumptions the rest of the program is making about the state of γ, if these assumptions are compatible with a, then updating to b will not invalidate any of these assumptions. Since Iris ensures that the global ghost state is valid, this means that we can soundly update the ghost state from a to a non-deterministically picked $b \in B$.

Definition 16. *It is possible to do a* frame-preserving step update *from $b \in M$ with $a \in M$ to $B \subseteq M \times M$, written $a, b \rightsquigarrow\!\!* B$, if*

$$\forall a_{\mathrm{f}}^? \in M^?. \, a \cdot b \cdot a_{\mathrm{f}}^? \in \mathcal{V} \Rightarrow \exists (a', b') \in B. \, a' \cdot b' \cdot a_{\mathrm{f}}^? \in \mathcal{V} \wedge b \curvearrowright b'$$

We further define $a, b \rightsquigarrow\!\!* a', b' \triangleq a, b \rightsquigarrow\!\!* \{(a', b')\}$.

We can regard this as a transformation of two compatible resources a and b in which we do a frame-preserving update on the first component a and a step on the second component b.

CMRA

Definition 17. *A CMRA is a tuple $(M : \mathcal{COFE}, (\mathcal{V}_n \subseteq M)_{n \in \mathbb{N}},$*
$|-| : M \xrightarrow{ne} M^?, (\cdot) : M \times M \xrightarrow{ne} M, ((\curvearrowright^n) \subseteq M \times M)_{n \in \mathbb{N}})$ *satisfying:*

$$\forall n, a, b. \, a \stackrel{n}{=} b \wedge a \in \mathcal{V}_n \Rightarrow b \in \mathcal{V}_n \qquad\qquad (\text{CMRA-VALID-NE})$$

$$\forall n, m. \, n \geq m \Rightarrow \mathcal{V}_n \subseteq \mathcal{V}_m \qquad\qquad (\text{CMRA-VALID-MONO})$$

$$\forall n, m. \, n \geq m \Rightarrow \curvearrowright^n \subseteq \curvearrowright^m \qquad\qquad (\text{CMRA-STEP-MONO})$$

$$\forall a, b, c. \, (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad\qquad (\text{CMRA-ASSOC})$$

$$\forall a, b. \, a \cdot b = b \cdot a \qquad\qquad (\text{CMRA-COMM})$$

$$\forall a. \, |a| \in M \Rightarrow |a| \cdot a = a \qquad\qquad (\text{CMRA-CORE-ID})$$

$$\forall a. \, |a| \in M \Rightarrow ||a|| = |a| \qquad\qquad (\text{CMRA-CORE-IDEM})$$

$$\forall a, b. \, |a| \in M \wedge a \preccurlyeq b \Rightarrow |b| \in M \wedge |a| \preccurlyeq |b| \qquad (\text{CMRA-CORE-MONO})$$

$$\forall n, a, b. \, (a \cdot b) \in \mathcal{V}_n \wedge |a| \in M \wedge |b| \in M \Rightarrow$$
$$||a| \cdot |b|| \stackrel{n}{=} |a| \cdot |b| \qquad\qquad (\text{CMRA-CORE-DISTRIB})$$

$$\forall n, a, b. \, (a \cdot b) \in \mathcal{V}_n \Rightarrow a \in \mathcal{V}_n \qquad\qquad (\text{CMRA-VALID-OP})$$

$$\forall n, a, b_1, b_2. \, a \in \mathcal{V}_n \wedge a \stackrel{n}{=} b_1 \cdot b_2 \Rightarrow$$
$$\exists c_1, c_2. \, a = c_1 \cdot c_2 \wedge c_1 \stackrel{n}{=} b_1 \wedge c_2 \stackrel{n}{=} b_2 \qquad (\text{CMRA-EXTEND})$$

where

$$a \preccurlyeq b \triangleq \exists c. \, b = a \cdot c \qquad\qquad (\text{CMRA-INCL})$$

$$a \stackrel{n}{\preccurlyeq} b \triangleq \exists c. \, b \stackrel{n}{=} a \cdot c \qquad\qquad (\text{CMRA-INCLN})$$

This is a natural generalization of RAs over COFEs. All operations have to be non-expansive, and the validity predicate $\mathcal{V}$ can now also depend on the step-index. We define the plain $\mathcal{V}$ as the "limit" of the $\mathcal{V}_n$:

$$\mathcal{V} \triangleq \bigcap_{n \in \mathbb{N}} \mathcal{V}_n$$

*The extension axiom (*CMRA-EXTEND*).* Notice that the existential quantification in this axiom is *constructive, i.e.,* it is a sigma type in Coq. The purpose of this axiom is to compute a_1, a_2 completing the following square:

$$
\begin{array}{ccc}
a & \stackrel{n}{=} & b \\
\| & & \| \\
a_1 \cdot a_2 & \stackrel{n}{=} & b_1 \cdot b_2
\end{array}
$$

where the n-equivalence at the bottom is meant to apply to the pairs of elements, *i.e.,* we demand $a_1 \stackrel{n}{=} b_1$ and $a_2 \stackrel{n}{=} b_2$. In other words, extension carries the decomposition of b into b_1 and b_2 over the n-equivalence of a and b, and yields a corresponding decomposition of a into a_1 and a_2. This operation is needed to prove that $\triangleright$ commutes with separating conjunction:

$$\triangleright(P * Q) \Leftrightarrow \triangleright P * \triangleright Q$$

Definition 18. *An element ε of a CMRA M is called the* unit *of M if it satisfies the following conditions:*

1. *ε is valid:*
   $\forall n.\, \varepsilon \in \mathcal{V}_n$
2. *ε is a left-identity of the operation:*
   $\forall a \in M.\, \varepsilon \cdot a = a$
3. *ε is a discrete COFE element*
4. *ε is its own core:*
   $|\varepsilon| = \varepsilon$

Lemma 19. *If M has a unit ε, then the core $|-|$ is total,* i.e., *$\forall a.\, |a| \in M$.*

Definition 20. *It is possible to do a* frame-preserving update *from $a \in M$ to $B \subseteq M$, written $a \rightsquigarrow B$, if*

$$\forall n, a_{\mathrm{f}}^?.\, a \cdot a_{\mathrm{f}}^? \in \mathcal{V}_n \Rightarrow \exists b \in B.\, b \cdot a_{\mathrm{f}}^? \in \mathcal{V}_n$$

We further define $a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$.

Definition 21. *It is possible to do a* frame-preserving step update *from $b \in M$ with $a \in M$ to $B \subseteq M \times M$, written $a \rightsquigarrow\!\!\!* B$, if*

$$\forall n, a_{\mathrm{f}}^?.\, a \cdot b \cdot a_{\mathrm{f}}^? \in \mathcal{V}_n \Rightarrow \exists (a', b') \in B.\, a' \cdot b' \cdot a_{\mathrm{f}}^? \in \mathcal{V}_n \wedge b \curvearrowright^n b'$$

We further define $a, b \rightsquigarrow\!\!\!* a', b' \triangleq a \rightsquigarrow\!\!\!* \{(a', b')\}$.

Note that for RAs, this and the RA-based definition of a frame-preserving update and frame-preserving step update coincide.

Definition 22. *A CMRA M is* discrete *if it satisfies the following conditions:*

1. M *is a discrete COFE*
2. $\mathcal{V}$ *ignores the step-index:*
   $\forall a \in M.\, a \in \mathcal{V}_0 \Rightarrow \forall n, a \in \mathcal{V}_n$

Note that every RA is a discrete CMRA, by picking the discrete COFE for the equivalence relation. Furthermore, discrete CMRAs can be turned into RAs by ignoring their COFE structure, as well as the step-index of $\mathcal{V}$.

Definition 23. *A function $f : M_1 \to M_2$ between two CMRAs is* monotone *(written $f : M_1 \xrightarrow{mon} M_2$) if it satisfies the following conditions:*

1. f *is non-expansive*
2. f *preserves validity:*
   $\forall n, a \in M_1.\, a \in \mathcal{V}_n \Rightarrow f(a) \in \mathcal{V}_n$
3. f *preserves CMRA inclusion:*
   $\forall a \in M_1, b \in M_1.\, a \preccurlyeq b \Rightarrow f(a) \preccurlyeq f(b)$

Definition 24. *The category $\mathcal{CMRA}$ consists of CMRAs as objects, and monotone functions as arrows.*

Note that every object/arrow in $\mathcal{CMRA}$ is also an object/arrow of $\mathcal{COFE}$. The notion of a locally non-expansive (or contractive) bifunctor naturally generalizes to bifunctors between these categories.

A.2 COFE constructions

Next (type-level later) Given a COFE T, we define $\blacktriangleright T$ as follows (using a datatype-like notation to define the type):

$$\blacktriangleright T \triangleq \mathsf{next}(x : T)$$

$$\mathsf{next}(x) \overset{n}{=} \mathsf{next}(y) \triangleq n = 0 \vee x \overset{n-1}{=} y$$

Note that in the definition of the carrier $\blacktriangleright T$, next is a constructor (like the constructors in Coq), *i.e.,* this is short for $\{\mathsf{next}(x) \,|\, x \in T\}$.

$\blacktriangleright(-)$ is a locally *contractive* functor from $\mathcal{COFE}$ to $\mathcal{COFE}$.

Uniform Predicates Given a CMRA M, we define the COFE $UPred(M)$ of *uniform predicates* over M as follows:

$$UPred(M) \triangleq \left\{ \varphi : \mathbb{N} \times M \times M \to Prop \,\middle|\, \begin{array}{l} (\forall n, x, y, x', y'.\, \varphi(n, x, y) \wedge x \preccurlyeq x' \wedge y \overset{n}{=} y' \Rightarrow \varphi(n, x', y')) \wedge \\ (\forall n, m, x, y.\, \varphi(n, x, y) \wedge x \in \mathcal{V}_m \wedge y \in \mathcal{V}_m \Rightarrow \varphi(m, x, y)) \end{array} \right\}$$

One way to understand this definition is to re-write it a little. We start by defining the COFE of *step-indexed propositions*: For every step-index, the proposition either holds or does not hold.

$$SProp \triangleq \wp^{\downarrow}(\mathbb{N})$$
$$\triangleq \{X \in \wp(\mathbb{N}) \mid \forall n, m. \, n \geq m \Rightarrow n \in X \Rightarrow m \in X\}$$
$$X \stackrel{n}{=} Y \triangleq \forall m \leq n. \, m \in X \Leftrightarrow m \in Y$$

Notice that this notion of *SProp* is already hidden in the validity predicate $\mathcal{V}_n$ of a CMRA: We could equivalently require every CMRA to define $\mathcal{V}_-(-) : M \xrightarrow{\text{ne}} SProp$, replacing CMRA-VALID-NE and CMRA-VALID-MONO.

Now we can rewrite $UPred(M)$ as step-indexed predicates over pairs of M eleements, which is "monotone" in a certain sense with respect to the first element:

$$UPred(M) \cong M \xrightarrow{\text{mon}} M \to SProp$$
$$\triangleq \left\{ \varphi : M \xrightarrow{\text{ne}} M \xrightarrow{\text{ne}} SProp \;\middle|\; \begin{array}{l} \forall n, m, x, y, x'. \, n \in \varphi(x) \wedge x \preccurlyeq x' \wedge m \leq n \wedge x' \in \mathcal{V}_m \\ \qquad\qquad \Rightarrow m \in \varphi(x', y) \end{array} \right\}$$

A.3 RA and CMRA constructions

When describing a CMRA construction, unless specified otherwise, the step relation is taken to be the full relation.

Product Given a family $(M_i)_{i \in I}$ of CMRAs (I finite), we construct a CMRA for the product $\prod_{i \in I} M_i$ by lifting everything pointwise.

Frame-preserving updates on the M_i lift to the product:

PROD-UPDATE
$$\frac{a \rightsquigarrow_{M_i} B}{f[i \hookrightarrow a] \rightsquigarrow \{f[i \hookrightarrow b] \mid b \in B\}}$$

Finite partial function Given some infinite countable K and some CMRA M, the set of finite partial functions $K \xrightarrow{\text{fin}} M$ is equipped with a COFE and CMRA structure by lifting everything pointwise.

We obtain the following frame-preserving updates:

FPFN-ALLOC-STRONG
$$\frac{G \text{ infinite} \qquad a \in \mathcal{V}}{\emptyset \rightsquigarrow \{[\gamma \hookrightarrow a] \mid \gamma \in G\}}$$

FPFN-ALLOC
$$\frac{a \in \mathcal{V}}{\emptyset \rightsquigarrow \{[\gamma \hookrightarrow a] \mid \gamma \in K\}}$$

FPFN-UPDATE
$$\frac{a \rightsquigarrow_M B}{f[i \hookrightarrow a] \rightsquigarrow \{f[i \hookrightarrow b] \mid b \in B\}}$$

Above, $\mathcal{V}$ refers to the validity of M.

$K \xrightarrow{\text{fin}} (-)$ is a locally non-expansive functor from $\mathcal{CMRA}$ to $\mathcal{CMRA}$.

Agreement Given some COFE T, we define $\text{AG}(T)$ as follows:

$$\text{AG}(T) \triangleq \{(c, V) \in (\mathbb{N} \to T) \times SProp\} \,/\sim$$

$$\text{where } a \sim b \triangleq a.V = b.V \wedge \forall n.\, n \in a.V \Rightarrow a.c(n) \overset{n}{=} b.c(n)$$

$$a \overset{n}{=} b \triangleq (\forall m \le n.\, m \in a.V \Leftrightarrow m \in b.V) \wedge (\forall m \le n.\, m \in a.V \Rightarrow a.c(m) \overset{m}{=} b.c(m))$$

$$\mathcal{V}_n \triangleq \left\{ a \in \text{AG}(T) \,\middle|\, n \in a.V \wedge \forall m \le n.\, a.c(n) \overset{m}{=} a.c(m) \right\}$$

$$|a| \triangleq a$$

$$a \cdot b \triangleq \left(a.c, \left\{ n \,\middle|\, n \in a.V \wedge n \in b.V \wedge a \overset{n}{=} b \right\} \right)$$

$\text{AG}(-)$ is a locally non-expansive functor from $\mathcal{COFE}$ to $\mathcal{CMRA}$.

You can think of the c as a *chain* of elements of T that has to converge only for $n \in V$ steps. The reason we store a chain, rather than a single element, is that $\text{AG}(T)$ needs to be a COFE itself, so we need to be able to give a limit for every chain of $\text{AG}(T)$. However, given such a chain, we cannot constructively define its limit: Clearly, the V of the limit is the limit of the V of the chain. But what to pick for the actual data, for the element of T? Only if $V = \mathbb{N}$ we have a chain of T that we can take a limit of; if the V is smaller, the chain "cancels", *i.e.*, stops converging as we reach indices $n \notin V$. To mitigate this, we apply the usual construction to close a set; we go from elements of T to chains of T.

We define an injection ag into $\text{AG}(T)$ as follows:

$$\text{ag}(x) \triangleq \left\{ c \triangleq \lambda\_.\, x, V \triangleq \mathbb{N} \right\}$$

There are no interesting frame-preserving updates for $\text{AG}(T)$, but we can show the following:

AG-VAL
$$\text{ag}(x) \in \mathcal{V}_n$$

AG-DUP
$$\text{ag}(x) = \text{ag}(x) \cdot \text{ag}(x)$$

AG-AGREE
$$\text{ag}(x) \cdot \text{ag}(y) \in \mathcal{V}_n \Rightarrow x \overset{n}{=} y$$

Exclusive CMRA Given a COFE T equipped with a step-indexed relation $(\curvearrowright^n)$, we define a CMRA $\text{Ex}(T)$ such that at most one $x \in T$ can be owned:

$$\text{Ex}(T) \triangleq \text{ex}(T) + \bot$$

$$\mathcal{V}_n \triangleq \{a \in \text{Ex}(T) \mid a \ne \bot\}$$

All cases of composition go to $\bot$.

$$|\text{ex}(x)| \triangleq \top \qquad\qquad |\bot| \triangleq \bot$$

Remember that $\top$ is the "dummy" element in $M^?$ indicating (in this case) that $\text{ex}(x)$ has no core.

The step-indexed equivalence is inductively defined as follows:

$$\frac{x \overset{n}{=} y}{\text{ex}(x) \overset{n}{=} \text{ex}(y)} \qquad\qquad \bot \overset{n}{=} \bot$$

$\text{Ex}(-)$ is a locally non-expansive functor from $\mathcal{COFE}$ to $\mathcal{CMRA}$.

We obtain the following frame-preserving update:

$$\text{EX-UPDATE}$$
$$\text{ex}(x) \rightsquigarrow \text{ex}(y)$$

We lift the stepping relation to:

$$\frac{x \curvearrowright^n y}{\text{ex}(x) \curvearrowright^n \text{ex}(y)}$$

STS with tokens Given a state-transition system (STS, *i.e.,* a directed graph) $(\mathcal{S}, \rightarrow \subseteq \mathcal{S} \times \mathcal{S})$, a set of tokens $\mathcal{T}$, and a labeling $\mathcal{L} : \mathcal{S} \rightarrow \wp(\mathcal{T})$ of *protocol-owned* tokens for each state, we construct an RA modeling an authoritative current state and permitting transitions given a *bound* on the current state and a set of *locally-owned* tokens.

The construction follows the idea of STSs as described in CaReSL [40]. We first lift the transition relation to $\mathcal{S} \times \wp(\mathcal{T})$ (implementing a *law of token conservation*) and define a stepping relation for the *frame* of a given token set:

$$(s, T) \rightarrow (s', T') \triangleq s \rightarrow s' \wedge \mathcal{L}(s) \uplus T = \mathcal{L}(s') \uplus T'$$

$$s \xrightarrow{\overline{T}} s' \triangleq \exists T_1, T_2.\, T_1 \,\#\, \mathcal{L}(s) \cup T \wedge (s, T_1) \rightarrow (s', T_2)$$

We further define *closed* sets of states (given a particular set of tokens) as well as the *closure* of a set:

$$\text{closed}(S, T) \triangleq \forall s \in S.\, \mathcal{L}(s) \,\#\, T \wedge \left(\forall s'.\, s \xrightarrow{\overline{T}} s' \Rightarrow s' \in S \right)$$

$$\uparrow(S, T) \triangleq \left\{ s' \in \mathcal{S} \,\middle|\, \exists s \in S.\, s \xrightarrow{\overline{T}}^* s' \right\}$$

The STS RA is defined as follows

$$M \triangleq \{\text{auth}((s, T) \in \mathcal{S} \times \wp(\mathcal{T})) \,|\, \mathcal{L}(s) \,\#\, T\} +$$
$$\{\text{frag}((S, T) \in \wp(\mathcal{S}) \times \wp(\mathcal{T})) \,|\, \text{closed}(S, T) \wedge S \neq \emptyset\} + \bot$$

$$\text{frag}(S_1, T_1) \cdot \text{frag}(S_2, T_2) \triangleq \text{frag}(S_1 \cap S_2, T_1 \cup T_2) \qquad \text{if } T_1 \,\#\, T_2 \text{ and } S_1 \cap S_2 \neq \emptyset$$

$$\text{frag}(S, T) \cdot \text{auth}(s, T') \triangleq \text{auth}(s, T') \cdot \text{frag}(S, T) \triangleq \text{auth}(s, T \cup T') \qquad \text{if } T \,\#\, T' \text{ and } s \in S$$

$$|\text{frag}(S, T)| \triangleq \text{frag}(\uparrow(S, \emptyset), \emptyset)$$

$$|\text{auth}(s, T)| \triangleq \text{frag}(\uparrow(\{s\}, \emptyset), \emptyset)$$

$$a \curvearrowright a' \triangleq \exists s, T, b, s', T', b'.\, a = \text{auth}(s, T) \cdot b \wedge a' = \text{auth}(s', T') \cdot b' \wedge$$
$$(s, T) \rightarrow^* (s', T')$$

The remaining cases are all $\bot$.

We will need the following frame-preserving update:

$$\text{STS-STEP} \quad \frac{(s, T) \to^* (s', T')}{\mathsf{auth}(s, T) \rightsquigarrow \mathsf{auth}(s', T')}$$

$$\text{STS-WEAKEN} \quad \frac{\mathsf{closed}(S_2, T_2) \quad S_1 \subseteq S_2 \quad T_2 \subseteq T_1}{\mathsf{frag}(S_1, T_1) \rightsquigarrow \mathsf{frag}(S_2, T_2)}$$

At the moment we do not make use of the non-trivial step structure on STS's – all instances of STS's that are used in our present proofs are wrapped in another construction that makes the step structure trivial.

The core is not a homomorphism. The core of the STS construction is only satisfying the RA axioms because we are *not* demanding the core to be a homomorphism—all we demand is for the core to be monotone with respect to the RA-INCL and have RA-CORE-DISTRIB property. This last rule is kind of like homomorphism *for elements that are already cores*, which is weaker than normal homomorphism.

In other words, the following does *not* hold for the STS core as defined above:

$$|a| \cdot |b| = |a \cdot b|$$

To see why, consider the following STS:

Now consider the following two elements of the STS RA:

$$a \triangleq \mathsf{frag}(\{s_1, s_2\}, \{T_1\}) \qquad b \triangleq \mathsf{frag}(\{s_1, s_3\}, \{T_2\})$$

We have:

$$a \cdot b = \mathsf{frag}(\{s_1\}, \{T_1, T_2\}) \qquad |a| = \mathsf{frag}(\{s_1, s_2, s_4\}, \emptyset)$$

$$|b| = \mathsf{frag}(\{s_1, s_3, s_4\}, \emptyset) \qquad |a| \cdot |b| = \mathsf{frag}(\{s_1, s_4\}, \emptyset) \neq |a \cdot b| = \mathsf{frag}(\{s_1\}, \emptyset)$$

A.4   Language

A *language* Λ consists of a set *Expr* of *expressions* (metavariable e), a set *Val* of *values* (metavariable v), and a set *State* of *states* (metvariable σ) such that

- There exist functions val2expr : *Val* $\to$ *Expr* and expr2val : *Expr* $\rightharpoonup$ *val* (notice the latter is partial), such that

  $$\forall e, v. \, \mathrm{expr2val}(e) = v \Rightarrow \mathrm{val2expr}(v) = e \qquad \forall v. \, \mathrm{expr2val}(\mathrm{val2expr}(v)) = v$$

- There exists a *primitive reduction relation*

  $$(-, - \to -, -, -) \subseteq \textit{Expr} \times \textit{State} \times \textit{Expr} \times \textit{State} \times (\textit{Expr} \uplus \{\bot\})$$

  We will write $e_1, \sigma_1 \to e_2, \sigma_2$ for $e_1, \sigma_1 \to e_2, \sigma_2, \bot$.
  A reduction $e_1, \sigma_1 \to e_2, \sigma_2, e_\mathsf{f}$ indicates that, when e_1 reduces to e_2, a *new thread* e_f is forked off.

– All values are stuck:

$$e, \_ \to \_, \_, \_ \Rightarrow \text{expr2val}(e) = \bot$$

Definition 25. *An expression e and state σ are* reducible *(written $\text{red}(e, \sigma)$) if*

$$\exists e_2, \sigma_2, e_\text{f}. e, \sigma \to e_2, \sigma_2, e_\text{f}$$

Definition 26. *An expression e is said to be* atomic *if it reduces in one step to a value:*

$$\forall \sigma_1, e_2, \sigma_2, e_\text{f}. e, \sigma_1 \to e_2, \sigma_2, e_\text{f} \Rightarrow \exists v_2. \text{expr2val}(e_2) = v_2$$

Definition 27 (Context). *A function $K : Expr \to Expr$ is a* context *if the following conditions are satisfied:*

1. *K does not turn non-values into values:*
   $\forall e. \text{expr2val}(e) = \bot \Rightarrow \text{expr2val}(K(e)) = \bot$
2. *One can perform reductions below K:*
   $\forall e_1, \sigma_1, e_2, \sigma_2, e_\text{f}. e_1, \sigma_1 \to e_2, \sigma_2, e_\text{f} \Rightarrow K(e_1), \sigma_1 \to K(e_2), \sigma_2, e_\text{f}$
3. *Reductions stay below K until there is a value in the hole:*
   $\forall e_1', \sigma_1, e_2, \sigma_2, e_\text{f}. \text{expr2val}(e_1') = \bot \wedge K(e_1'), \sigma_1 \to e_2, \sigma_2, e_\text{f} \Rightarrow \exists e_2'. e_2 = K(e_2') \wedge e_1', \sigma_1 \to e_2', \sigma_2, e_\text{f}$

Concurrent language For any language Λ, we define the corresponding thread-pool semantics. The step relation for thread pool configurations is indexed by the number of the thread that performed a step.

Machine syntax

$$T \in ThreadPool \triangleq \bigcup_n Expr^n$$

$$\rho \in Config \triangleq ThreadPool \times State$$

Machine reduction
$$\boxed{[T]; \sigma \xrightarrow{i} [T']; \sigma'}$$

$$\frac{e_1, \sigma_1 \to e_2, \sigma_2, e_\text{f} \qquad e_\text{f} \neq \bot \qquad |T| = i}{[T \mathbin{+\!\!+} [e_1] \mathbin{+\!\!+} T']; \sigma_1 \xrightarrow{i} [T \mathbin{+\!\!+} [e_2] \mathbin{+\!\!+} T' \mathbin{+\!\!+} [e_\text{f}]]; \sigma_2}$$

$$\frac{e_1, \sigma_1 \to e_2, \sigma_2 \qquad |T| = i}{[T \mathbin{+\!\!+} [e_1] \mathbin{+\!\!+} T']; \sigma_1 \xrightarrow{i} [T \mathbin{+\!\!+} [e_2] \mathbin{+\!\!+} T']; \sigma_2}$$

Definition 28. *We say thread index i is* enabled *in $[T]; \sigma$ if there exists T' and σ' such that $[T]; \sigma \xrightarrow{i} [T']; \sigma'$.*

Definition 29. *A diverging execution[11] of $[T]; \sigma$ is a function $F : \mathbb{N} \to Config \times \mathbb{N}$ such that:*

1. *$F(0) = ([T]; \sigma, i)$ for some i.*
2. *For all n, if $F(n) = ([T_n]; \sigma_n, j)$ and $F(n + 1) = ([T_{n+1}]; \sigma_{n+1}, j')$ then $[T_n]; \sigma_n \xrightarrow{j} [T_{n+1}]; \sigma_{n+1}$.*

Definition 30. *We say that thread index i is* eventually always enabled *in a diverging execution F if there exists N such that $\forall n \geq N$, i is enabled in $\pi_1(F(n))$.*

Definition 31. *We say that thread index i* always eventually steps *in a diverging execution F if for all n, there exists $n' \geq n$ such that $\pi_2(F(n')) = i$.*

Definition 32. *A diverging execution is* (weakly) fair *if for all i, if i is eventually always enabled in F, then i always eventually steps in F.*

A.5   Logic

To instantiate Iris, you need to define the following parameters:

- A language Λ, and
- a locally contractive bifunctor $\Sigma : \mathcal{COFE} \to \mathcal{CMRA}$ defining the ghost state, such that for all COFEs A, the CMRA $\Sigma(A)$ has a unit. (By Lemma 19, this means that the core of $\Sigma(A)$ is a total function.)

As usual for higher-order logics, you can furthermore pick a *signature* $\mathcal{S} = (\mathcal{T}, \mathcal{F}, \mathcal{A})$ to add more types, symbols and axioms to the language. You have to make sure that $\mathcal{T}$ includes the base types:

$$\mathcal{T} \supseteq \{\mathsf{Val}, \mathsf{Expr}, \mathsf{State}, \mathsf{M}, \mathsf{InvName}, \mathsf{InvMask}, \mathsf{Prop}\}$$

Elements of $\mathcal{T}$ are ranged over by T.

Each function symbol in $\mathcal{F}$ has an associated *arity* comprising a natural number n and an ordered list of $n + 1$ types τ (the grammar of τ is defined below, and depends only on $\mathcal{T}$). We write

$$F : \tau_1, ..., \tau_n \to \tau_{n+1} \in \mathcal{F}$$

to express that F is a function symbol with the indicated arity.

Furthermore, $\mathcal{A}$ is a set of *axioms*, that is, terms t of type Prop. Again, the grammar of terms and their typing rules are defined below, and depends only on $\mathcal{T}$ and $\mathcal{F}$, not on $\mathcal{A}$. Elements of $\mathcal{A}$ are ranged over by A.

---

[11] This may also be defined co-inductively; in the Coq formalization we use a co-inductive definition and give the definition here as a derived one.

Grammar

Syntax. Iris syntax is built up from a signature $\mathcal{S}$ and a countably infinite set *Var* of variables (ranged over by metavariables x, y, z):

$$\tau ::= T \mid 1 \mid \tau \times \tau \mid \tau \to \tau$$

$$
\begin{aligned}
t, P, \varphi ::= {} & x \mid F(t_1, \ldots, t_n) \mid () \mid (t, t) \mid \pi_i\, t \mid \lambda x : \tau.\, t \mid t(t) \mid \varepsilon \mid |t| \mid t \cdot t \mid \\
& \mathsf{False} \mid \mathsf{True} \mid \mathsf{Emp} \mid t =_\tau t \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid P * P \mid P \mathrel{-\!\!*} P \mid \\
& \mu x : \tau.\, t \mid \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \\
& \boxed{P}^{\,t} \mid \lceil\mathrel{\underline{t}}\rceil \mid \lceil\mathrel{\underline{t}}\rceil^{\mathsf{L}} \mid \mathcal{V}(t) \mid \mathsf{Stopped} \mid \mathcal{V}(t) \mid \mathsf{Phy}(t) \mid \Box P \mid \mathcal{A}(P) \mid \triangleright P \mid {}^t\!\!\Rrightarrow^t P \mid {}^t\!\!\Rrightarrow^t P \mid \mathsf{wp}_t\, t\, \{x.\, t\}
\end{aligned}
$$

Recursive predicates must be *guarded*: in $\mu x.\, t$, the variable x can only appear under the later $\triangleright$ modality.

Note that $\Box$, $\triangleright$ bind more tightly than $*$, $\mathrel{-\!\!*}$, $\wedge$, $\vee$, and $\Rightarrow$. We will write $\Rrightarrow_t P$ for ${}^t\!\!\Rrightarrow^t P$, and similarly for $\Rrightarrow_t P$. If we omit the mask, then it is $\top$ for weakest precondition $\mathsf{wp}\, e\, \{x.\, P\}$ and $\emptyset$ for primitive view shifts $\Rrightarrow P$ and primitive step shifts $\Rrightarrow P$.

Some propositions are *timeless*, which intuitively means that step-indexing does not affect them. This is a *meta-level* assertion about propositions, defined as follows:

$$\Gamma \vdash \mathsf{timeless}(P) \triangleq \Gamma \mid \triangleright P \vdash P \vee \triangleright \mathsf{False}$$

Similarly, some propositions are *affine timeless*, which means that step-indexing does not affect them *when under an affine modality*:

$$\Gamma \vdash \mathsf{atimeless}(P) \triangleq \Gamma \mid \mathcal{A}(\triangleright P) \vdash \mathcal{A}(P) \vee \triangleright \mathsf{False}$$

Metavariable conventions. We introduce additional metavariables ranging over terms and generally let the choice of metavariable indicate the term's type:

metavariable	type
t, u	arbitrary
v, w	Val
e	Expr
σ	State

metavariable	type
ι	InvName
$\mathcal{E}$	InvMask
a, b	M
P, Q, R	Prop
φ, ψ, ζ	$\tau \to \mathsf{Prop}$ (when τ is clear from context)

Variable conventions. We assume that, if a term occurs multiple times in a rule, its free variables are exactly those binders which are available at every occurrence.

Types Iris terms are simply-typed. The judgment $\Gamma \vdash t : \tau$ expresses that, in variable context Γ, the term t has type τ.

A variable context, $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, declares a list of variables and their types. In writing $\Gamma, x : \tau$, we presuppose that x is not already declared in Γ.

Well-typed terms $\boxed{\Gamma \vdash_{\mathcal{S}} t : \tau}$

$$x : \tau \vdash x : \tau \qquad \frac{\Gamma \vdash t : \tau}{\Gamma, x : \tau' \vdash t : \tau} \qquad \frac{\Gamma, x : \tau', y : \tau' \vdash t : \tau}{\Gamma, x : \tau' \vdash t[x/y] : \tau}$$

$$\frac{\Gamma_1, x : \tau', y : \tau'', \Gamma_2 \vdash t : \tau}{\Gamma_1, x : \tau'', y : \tau', \Gamma_2 \vdash t[y/x, x/y] : \tau}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash t_n : \tau_n \qquad F : \tau_1, \ldots, \tau_n \to \tau_{n+1} \in \mathcal{F}}{\Gamma \vdash F(t_1, \ldots, t_n) : \tau_{n+1}} \qquad \Gamma \vdash () : 1$$

$$\frac{\Gamma \vdash t : \tau_1 \qquad \Gamma \vdash u : \tau_2}{\Gamma \vdash (t, u) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i\, t : \tau_i} \qquad \frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x.t : \tau \to \tau'}$$

$$\frac{\Gamma \vdash t : \tau \to \tau' \qquad u : \tau}{\Gamma \vdash t(u) : \tau'} \qquad \Gamma \vdash \varepsilon : \mathsf{M} \qquad \frac{\Gamma \vdash a : \mathsf{M}}{\Gamma \vdash |a| : \mathsf{M}} \qquad \frac{\Gamma \vdash a : \mathsf{M} \qquad \Gamma \vdash b : \mathsf{M}}{\Gamma \vdash a \cdot b : \mathsf{M}}$$

$$\Gamma \vdash \mathsf{False} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{True} : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{Emp} : \mathsf{Prop}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash u : \tau}{\Gamma \vdash t =_\tau u : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \Rightarrow Q : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \wedge Q : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \vee Q : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P * Q : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \twoheadrightarrow Q : \mathsf{Prop}}$$

$$\frac{\Gamma, x : \tau \vdash t : \tau \qquad x \text{ is guarded in } t}{\Gamma \vdash \mu x : \tau.\, t : \tau} \qquad \frac{\Gamma, x : \tau \vdash P : \mathsf{Prop}}{\Gamma \vdash \exists x : \tau.\, P : \mathsf{Prop}}$$

$$\frac{\Gamma, x : \tau \vdash P : \mathsf{Prop}}{\Gamma \vdash \forall x : \tau.\, P : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash \iota : \mathsf{InvName}}{\Gamma \vdash \boxed{P}^{\iota} : \mathsf{Prop}} \qquad \frac{\Gamma \vdash a : \mathsf{M}}{\Gamma \vdash \lceil a \rceil : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash a : \tau \qquad \tau \text{ is a CMRA}}{\Gamma \vdash \mathcal{V}(a) : \mathsf{Prop}} \qquad \frac{\Gamma \vdash \sigma : \mathsf{State}}{\Gamma \vdash \mathsf{Phy}(\sigma) : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash \square P : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash \mathcal{A}(P) : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash \triangleright P : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash \mathcal{E} : \mathsf{InvMask} \qquad \Gamma \vdash \mathcal{E}' : \mathsf{InvMask}}{\Gamma \vdash {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}'} P : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash \mathcal{E} : \mathsf{InvMask} \qquad \Gamma \vdash \mathcal{E}' : \mathsf{InvMask}}{\Gamma \vdash {}^{\mathcal{E}}\!\!\Rrightarrow^{\mathcal{E}'} P : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash e : \mathsf{Expr} \qquad \Gamma, x : \mathsf{Val} \vdash t : \mathsf{Prop} \qquad \Gamma \vdash \mathcal{E} : \mathsf{InvMask}}{\Gamma \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, t\} : \mathsf{Prop}}$$

Proof rules The judgment $\Gamma \mid \Theta \vdash P$ says that with free variables Γ, proposition P holds whenever all assumptions Θ hold. We implicitly assume that an arbitrary variable context, Γ, is added to every constituent of the rules. Furthermore, an arbitrary *boxed* assertion context $\Box\Theta$ may be added to every constituent. Axioms $\Gamma \mid P \dashv\vdash Q$ indicate that both $\Gamma \mid P \vdash Q$ and $\Gamma \mid Q \vdash P$ can be derived.

$$\boxed{\Gamma \mid \Theta \vdash P}$$

Laws of intuitionistic higher-order logic with equality. This is entirely standard.

$$\frac{\mathrm{ASM}}{P \in \Theta} \qquad \frac{\mathrm{EQ}}{\Theta \vdash P \quad \Theta \vdash t =_\tau t'} \qquad \frac{\mathrm{REFL}}{\Theta \vdash t =_\tau t} \qquad \frac{\perp\mathrm{E}}{\Theta \vdash \mathsf{False}} \qquad \frac{\top\mathrm{I}}{\Theta \vdash \mathsf{True}}$$

$$\frac{\Theta \vdash P[t'/t]}{} \qquad \frac{\Theta \vdash P}{}$$

$$\frac{P \in \Theta}{\Theta \vdash P}\ \mathrm{ASM} \qquad \frac{\Theta \vdash P \qquad \Theta \vdash t =_\tau t'}{\Theta \vdash P[t'/t]}\ \mathrm{EQ} \qquad \frac{}{\Theta \vdash t =_\tau t}\ \mathrm{REFL} \qquad \frac{\Theta \vdash \mathsf{False}}{\Theta \vdash P}\ \perp\mathrm{E} \qquad \frac{}{\Theta \vdash \mathsf{True}}\ \top\mathrm{I}$$

$$\frac{\Theta \vdash P \qquad \Theta \vdash Q}{\Theta \vdash P \wedge Q}\ \wedge\mathrm{I} \qquad \frac{\Theta \vdash P \wedge Q}{\Theta \vdash P}\ \wedge\mathrm{EL} \qquad \frac{\Theta \vdash P \wedge Q}{\Theta \vdash Q}\ \wedge\mathrm{ER} \qquad \frac{\Theta \vdash P}{\Theta \vdash P \vee Q}\ \vee\mathrm{IL}$$

$$\frac{\Theta \vdash Q}{\Theta \vdash P \vee Q}\ \vee\mathrm{IR} \qquad \frac{\Theta \vdash P \vee Q \qquad \Theta, P \vdash R \qquad \Theta, Q \vdash R}{\Theta \vdash R}\ \vee\mathrm{E} \qquad \frac{\Theta, P \vdash Q}{\Theta \vdash P \Rightarrow Q}\ \Rightarrow\mathrm{I}$$

$$\frac{\Theta \vdash P \Rightarrow Q \qquad \Theta \vdash P}{\Theta \vdash Q}\ \Rightarrow\mathrm{E} \qquad \frac{\Gamma, x : \tau \mid \Theta \vdash P}{\Gamma \mid \Theta \vdash \forall x : \tau.\, P}\ \forall\mathrm{I} \qquad \frac{\Gamma \mid \Theta \vdash \forall x : \tau.\, P \qquad \Gamma \vdash t : \tau}{\Gamma \mid \Theta \vdash P[t/x]}\ \forall\mathrm{E}$$

$$\frac{\Gamma \mid \Theta \vdash P[t/x] \qquad \Gamma \vdash t : \tau}{\Gamma \mid \Theta \vdash \exists x : \tau.P}\ \exists\mathrm{I} \qquad \frac{\Gamma \mid \Theta \vdash \exists x : \tau.\, P \qquad \Gamma, x : \tau \mid \Theta, P \vdash Q}{\Gamma \mid \Theta \vdash Q}\ \exists\mathrm{E}$$

Furthermore, we have the usual η and β laws for projections, λ and μ.

Laws of bunched implications.

$$\mathsf{Emp} * P \dashv\vdash P$$
$$P * Q \dashv\vdash Q * P$$
$$(P * Q) * R \dashv\vdash P * (Q * R)$$

∗-MONO
$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}$$

─∗ I-E
$$\frac{P * Q \vdash R}{P \vdash Q \mathbin{-\!\!*} R}$$

Laws for ghosts and physical resources.

$$\boxed{a} * \boxed{b} \dashv\vdash \boxed{a \cdot b}$$
$$\boxed{a} \vdash \mathcal{V}(a) \qquad\qquad \mathsf{Phy}(\sigma) * \mathsf{Phy}(\sigma') \vdash \mathsf{False}$$
$$\mathsf{True} \vdash \boxed{\varepsilon}$$

$$\frac{\forall n. \not\exists a'. a \curvearrowright^n a'}{\boxed{a}^{\mathsf{L}} \vdash \mathsf{Stopped}}$$

Similar rules hold for $\boxed{a}^{\mathsf{L}}$.

Laws for the later modality.

▷-MONO
$$\frac{\Theta \vdash P}{\Theta \vdash \triangleright P}$$

LÖB
$$(\triangleright P \Rightarrow P) \vdash P$$

U-LÖB
$$\mathcal{A}(\Box(\mathcal{A}(\Box\triangleright P) \mathbin{-\!\!*} \mathcal{A}(\Box P))) \vdash \Box P$$

▷-∃
$$\frac{\tau \text{ is inhabited}}{\triangleright \exists x : \tau.\, P \vdash \exists x : \tau. \triangleright P}$$

$$\triangleright(P \wedge Q) \dashv\vdash \triangleright P \wedge \triangleright Q$$
$$\triangleright(P \vee Q) \dashv\vdash \triangleright P \vee \triangleright Q$$

$$\triangleright \forall x.\, P \dashv\vdash \forall x. \triangleright P$$
$$\exists x. \triangleright P \vdash \triangleright \exists x.\, P$$
$$\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$$

A type τ being *inhabited* means that $\vdash t : \tau$ is derivable for some t.

$$\frac{t \text{ or } t' \text{ is a discrete COFE element}}{\mathsf{timeless}(t =_\tau t')}$$

$$\frac{a \text{ is a discrete COFE element}}{\mathsf{timeless}(\boxed{a})}$$

$$\frac{a \text{ is an element of a discrete CMRA}}{\mathsf{timeless}(\mathcal{V}(a))}$$

$$\mathsf{timeless}(\mathsf{Phy}(\sigma))$$

$$\frac{\Gamma \vdash \mathsf{timeless}(Q)}{\Gamma \vdash \mathsf{timeless}(P \Rightarrow Q)}$$

$$\frac{\Gamma \vdash \mathsf{timeless}(Q)}{\Gamma \vdash \mathsf{timeless}(P \mathbin{-\!\!*} Q)}$$

$$\frac{\Gamma, x : \tau \vdash \mathsf{timeless}(P)}{\Gamma \vdash \mathsf{timeless}(\forall x : \tau.\, P)}$$

$$\frac{\Gamma, x : \tau \vdash \mathsf{timeless}(P)}{\Gamma \vdash \mathsf{timeless}(\exists x : \tau.\, P)}$$

$$\frac{t \text{ or } t' \text{ is a discrete COFE element}}{\mathsf{atimeless}(t =_\tau t')}$$

$$\frac{a \text{ is a discrete COFE element}}{\mathsf{atimeless}(\lfloor a \rfloor)}$$

$$\frac{a \text{ is an element of a discrete CMRA}}{\mathsf{atimeless}(\mathcal{V}(a))}$$

$$\mathsf{atimeless}(\mathsf{Phy}(\sigma))$$

$$\frac{\Gamma \vdash \mathsf{atimeless}(Q)}{\Gamma \vdash \mathsf{atimeless}(P \Rightarrow Q)}$$

$$\frac{\Gamma \vdash \mathsf{atimeless}(Q)}{\Gamma \vdash \mathsf{atimeless}(P \twoheadrightarrow Q)}$$

$$\frac{\Gamma \vdash \mathsf{atimeless}(P) \quad \Gamma \vdash \mathsf{atimeless}(Q)}{\Gamma \vdash \mathsf{atimeless}(P \wedge Q)}$$

$$\frac{\Gamma \vdash \mathsf{atimeless}(\mathcal{A}(P)) * \Gamma \vdash \mathsf{atimeless}(\mathcal{A}(Q))}{\Gamma \vdash \mathsf{atimeless}(\mathcal{A}(P) * \mathcal{A}(Q))}$$

$$\frac{\Gamma \vdash \mathsf{atimeless}(P) \quad \Gamma \vdash \mathsf{atimeless}(Q)}{\Gamma \vdash \mathsf{atimeless}(P \vee Q)}$$

$$\frac{\Gamma, x : \tau \vdash \mathsf{atimeless}(P)}{\Gamma \vdash \mathsf{atimeless}(\forall x : \tau. P)}$$

$$\frac{\Gamma, x : \tau \vdash \mathsf{atimeless}(P)}{\Gamma \vdash \mathsf{atimeless}(\exists x : \tau. P)}$$

Laws for the always/relevant modality.

$$\Box\mathrm{I} \quad \frac{\Box\Theta \vdash P}{\Box\Theta \vdash \Box P}$$

$$\Box\mathrm{E} \quad \Box P \vdash P$$

$$\Box(P \wedge Q) \vdash \Box(P * Q)$$
$$\Box P \wedge Q \vdash \Box P * Q$$
$$\Box \triangleright P \dashv\vdash \Box \triangleright \Box P$$
$$\Box \triangleright P \vdash \triangleright \Box P$$

$$\Box P * \Box Q \vdash \Box P * Q$$
$$\Box P \vdash \Box P * \Box P$$
$$\Box(P \wedge Q) \dashv\vdash \Box P \wedge \Box Q$$
$$\Box(P \vee Q) \dashv\vdash \Box P \vee \Box Q$$
$$\Box \exists x. P \dashv\vdash \exists x. \Box P$$

$$t =_\tau t' \vdash \Box t =_\tau t' \qquad \boxed{P}^\iota \vdash \Box \boxed{P}^\iota \qquad \lfloor a \rfloor \vdash \Box \lfloor a \rfloor \qquad \mathcal{V}(a) \vdash \Box \mathcal{V}(a)$$

$$\frac{\tau \text{ is inhabited}}{\Box \forall x : \tau. P \dashv\vdash \forall x. \Box P}$$

Laws for the affine modality.

$$\mathcal{A}(P) \;\vdash\; P$$
$$\mathcal{A}(\mathsf{True}) \;\dashv\vdash\; \mathsf{Emp}$$
$$\mathcal{A}(\mathcal{A}(P)) \;\dashv\vdash\; \mathcal{A}(P)$$
$$P * \mathcal{A}(Q) \;\vdash\; P$$
$$\mathcal{A}(\mathcal{A}(P) * \mathcal{A}(Q)) \;\dashv\vdash\; \mathcal{A}(P) * \mathcal{A}(Q)$$
$$\mathcal{A}(P) \wedge Q \;\dashv\vdash\; \mathcal{A}(P \wedge Q)$$
$$\mathcal{A}(\triangleright P) \;\dashv\vdash\; \mathcal{A}(\triangleright \mathcal{A}(P))$$
$$\mathcal{A}(\triangleright P) \;\vdash\; \triangleright \mathcal{A}(P)$$
$$\mathcal{A}(\triangleright(\mathcal{A}(P) * \mathcal{A}(Q))) \;\vdash\; \mathcal{A}(\triangleright P) * \mathcal{A}(\triangleright P)$$
$$\mathcal{A}(\square P) \;\dashv\vdash\; \square \mathcal{A}(P)$$
$$t =_\tau t' \;\vdash\; \mathcal{A}(t =_\tau t')$$
$$\boxed{P}^{\iota} \;\vdash\; \mathcal{A}(\boxed{P}^{\iota})$$
$$\lfloor \overline{a} \rfloor \;\vdash\; \mathcal{A}(\lfloor \overline{a} \rfloor)$$
$$\mathcal{V}(a) \;\vdash\; \mathcal{A}(\mathcal{V}(a))$$

$$\frac{\tau \text{ is inhabited}}{\mathcal{A}(\forall x : \tau.\, P) \dashv\vdash \forall x.\, \mathcal{A}(P)} \qquad \frac{\tau \text{ is inhabited}}{\mathcal{A}(\exists x : \tau.\, P) \dashv\vdash \exists x.\, \mathcal{A}(P)}$$

Laws of primitive view shifts.

PVS-INTRO
$$P \vdash \Rrightarrow_{\mathcal{E}} P$$

PVS-MONO
$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q}$$

PVS-TIMELESS
$$\frac{\mathsf{timeless}(P)}{\triangleright P \vdash \Rrightarrow_{\mathcal{E}} P}$$

PVS-ATIMELESS
$$\frac{\mathsf{atimeless}(P)}{\mathcal{A}(\triangleright P) \vdash \Rrightarrow_{\mathcal{E}} \mathcal{A}(P)}$$

PVS-TRANS
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1 \cup \mathcal{E}_3}{{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\!\!\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_3} P}$$

PVS-MASK-FRAME
$${}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_{\mathsf{f}}}\!\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_{\mathsf{f}}} P$$

PVS-FRAME
$$Q * {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q * P$$

PVS-ALLOCI
$$\frac{\mathcal{E} \text{ is infinite}}{\mathcal{A}(\triangleright P) \vdash \Rrightarrow_{\mathcal{E}} \exists \iota \in \mathcal{E}.\, \boxed{P}^{\iota}}$$

PVS-OPENI
$$\boxed{P}^{\iota} \vdash {}^{\{\iota\}}\!\!\Rrightarrow^{\emptyset} \mathcal{A}(\triangleright P)$$

PVS-CLOSEI
$$\boxed{P}^{\iota} * \mathcal{A}(\triangleright P) \vdash {}^{\emptyset}\!\!\Rrightarrow^{\{\iota\}} \mathsf{Emp}$$

PVS-UPDATE
$$\frac{a \rightsquigarrow B}{\lfloor \overline{a} \rfloor \vdash \Rrightarrow_{\mathcal{E}} \exists b \in B.\, \lfloor \overline{b} \rfloor}$$

PVS-AFFINE
$$\mathcal{A}({}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P) \dashv\vdash {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} \mathcal{A}(P)$$

Laws of primitive step shifts.

PSVS-INTRO
$$P \vdash \Rrightarrow_{\mathcal{E}} P$$

PSVS-MONO
$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q}$$

PVS-PSVS
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1 \cup \mathcal{E}_3}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} P}$$

PSVS-PVS
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1 \cup \mathcal{E}_3}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} P}$$

PSVS-MASK-FRAME
$${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_f} P$$

PSVS-FRAME
$$\mathcal{A}(Q) * {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} (\mathcal{A}(Q) * P)$$

PSVS-STEP
$$\frac{a, b \rightsquigarrow B}{\lceil a \rceil * \lceil b \rceil^{\mathsf{L}} \vdash \Rrightarrow_{\mathcal{E}} \exists a', b' \in B. \lceil a' \rceil * \lceil b' \rceil^{\mathsf{L}}}$$

Laws of weakest preconditions.

WP-VALUE
$$P[v/x] \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{x.\, P\}$$

WP-MONO
$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad x : \mathsf{val} \mid P \vdash Q}{\mathsf{wp}_{\mathcal{E}_1}\, e\, \{x.\, P\} \vdash \mathsf{wp}_{\mathcal{E}_2}\, e\, \{x.\, Q\}}$$

PVS-WP
$$\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\}$$

WP-PVS
$$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \Rrightarrow_{\mathcal{E}} P\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\}$$

WP-ATOMIC
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1 \qquad \mathsf{atomic}(e)}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e\, \{x.\, {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} P\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{x.\, P\}}$$

WP-FRAME
$$\mathcal{A}(Q) * \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \mathcal{A}(Q) * P\}$$

WP-FRAME-STEP
$$\frac{\mathrm{expr2val}(e) = \bot \qquad \mathcal{E}_2 \subseteq \mathcal{E}_1}{\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} * \mathcal{A}({}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \triangleright {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} Q) \vdash \mathsf{wp}_{\mathcal{E} \uplus \mathcal{E}_1}\, e\, \{x.\, \mathcal{A}(Q) * P\}}$$

WP-BIND
$$\frac{K \text{ is a context}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \mathsf{wp}_{\mathcal{E}}\, K(\mathrm{val2expr}(x))\, \{y.\, P\}\} \vdash \mathsf{wp}_{\mathcal{E}}\, K(e)\, \{y.\, P\}}$$

Lifting of operational semantics.

WP-LIFT-STEP

$$\mathcal{E}_2 \subseteq \mathcal{E}_1 \qquad \text{expr2val}(e_1) = \bot$$

$$\frac{}{{}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} \exists \sigma_1.\, \mathcal{A}(\text{red}(e_1, \sigma_1) \wedge \rhd \mathsf{Phy}(\sigma_1))\, * }$$

$$\rhd \forall e_2, \sigma_2, e_{\mathsf{f}}.\, ((e_1, \sigma_1 \to e_2, \sigma_2, e_{\mathsf{f}}) \wedge \mathsf{Phy}(\sigma_2)) \mathrel{-\!\!*} {}^{\mathcal{E}_2}\!\!\Rrightarrow^{\mathcal{E}_1} \mathsf{wp}_{\mathcal{E}_1} e_2 \{x.\, P\} * \mathsf{wp}_\top e_{\mathsf{f}} \{\_.\, \mathsf{Stopped}\}$$

$$\vdash \mathsf{wp}_{\mathcal{E}_1} e_1 \{x.\, P\}$$

WP-LIFT-PURE-STEP

$$\frac{\text{expr2val}(e_1) = \bot \qquad \forall \sigma_1.\, \text{red}(e_1, \sigma_1) \qquad \forall \sigma_1, e_2, \sigma_2, e_{\mathsf{f}}.\, e_1, \sigma_1 \to e_2, \sigma_2, e_{\mathsf{f}} \Rightarrow \sigma_1 = \sigma_2}{\rhd \forall \sigma, e_2, e_{\mathsf{f}}.\, (e_1, \sigma \to e_2, \sigma, e_{\mathsf{f}}) \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}_1} \mathsf{wp}_{\mathcal{E}_1} e_2 \{x.\, P\} * \mathsf{wp}_\top e_{\mathsf{f}} \{\_.\, \mathsf{Stopped}\} \vdash \mathsf{wp}_{\mathcal{E}_1} e_1 \{x.\, P\}}$$

Notice that primitive view shifts cover everything to their right, *i.e.,* $\Rrightarrow P * Q \triangleq \Rrightarrow (P * Q)$, and similarly for primitive step shifts.

Here we define $\mathsf{wp}_{\mathcal{E}} e_{\mathsf{f}} \{x.\, P\} \triangleq \mathsf{Emp}$ if $e_{\mathsf{f}} = \bot$ (remember that our stepping relation can, but does not have to, define a forked-off expression).

Adequacy

Finite Executions and Safety. The adequacy statement concerning functional correctness reads as follows:

$$\forall \mathcal{E}, e, v, \varphi, \sigma, a, b, \sigma', T'.$$
$$(\forall n.\, a \cdot b \in \mathcal{V}_n) \Rightarrow$$
$$(\mathsf{Phy}(\sigma) * \lceil a \rceil * \lceil b \rceil^{\mathsf{L}} \vdash \mathsf{wp}_{\mathcal{E}} e \{x.\, \varphi(x)\}) \Rightarrow$$
$$[\sigma]; [e] \to^* [\sigma']; [v] \mathbin{+\!\!+} T' \Rightarrow$$
$$\varphi(v)$$

where φ is a *meta-level* predicate over values, *i.e.,* it can mention neither resources nor invariants.

Furthermore, the following adequacy statement shows that our weakest preconditions imply that the execution never gets *stuck*: Every expression in the thread pool either is a value, or can reduce further.

$$\forall \mathcal{E}, e, \sigma, a, b, \sigma', T'.$$
$$(\forall n.\, a * b \in \mathcal{V}_n) \Rightarrow$$
$$(\mathsf{Phy}(\sigma) * \lceil a \rceil * \lceil b \rceil^{\mathsf{L}} \vdash \mathsf{wp}_{\mathcal{E}} e \{x.\, \varphi(x)\}) \Rightarrow$$
$$[\sigma]; [e] \to^* [\sigma']; T' \Rightarrow$$
$$\forall e' \in T'.\, \text{expr2val}(e') \neq \bot \vee \text{red}(e', \sigma')$$

Notice that this is stronger than saying that the thread pool can reduce; we actually assert that *every* non-finished thread can take a step.

Diverging Executions. Remember that our goal is to show that if we have proved $\mathsf{wp}\, e\, \{x.\, Q\}$ and e has a fair diverging execution, then there is a fair diverging execution of some corresponding source program. Of course, there is no such notion of a "source program" baked into the logic at this point yet.

All we have is the step relation $\curvearrowright^n$ on CMRA elements. The rules above for $\mathsf{wp}\, e\, \{x.\, Q\}$ suggest that[12] for each step e takes, we are required to perform a step-shift, thereby performing reduction steps on CMRA resources.

We can lift the $\curvearrowright^n$ relation from CMRA elements to lists of CMRA elements, much as we lift the per-thread step relation to an indexed relation on threadpools:

$$\frac{a_i \curvearrowright^n (a'_i \cdot a_{\mathsf{f}})}{[a_0, \dots, a_i, \dots, a_k] \curvearrowright^n_i [a_0, \dots, a'_i, \dots, a_k, a_{\mathsf{f}}]}$$

$$\frac{a_i \curvearrowright^n a'_i}{[a_0, \dots, a_i, \dots, a_k] \curvearrowright^n_i [a_0, \dots, a'_i, \dots, a_k]}$$

In general we shall use A and B as metavariables for such lists of CMRA elements, and write $\text{\Large$\ast$}\, A$ to represent the product of the elements of A.

Definition 33. *We say index i is n-enabled in A if there exists B such that $A \curvearrowright^n_i B$.*

Somehow, we want to connect up these CMRA steps to steps in some source language, for a suitably chosen CMRA. Still working a bit more abstractly than that for the moment, let U be some COFE equipped with a family of relations $(\xrightarrow{i} : U^? \times U^?)_{i \in \mathbb{N}}$. We can adapt all of our definitions about fairness to the setting of this reduction on $U^?$, e.g.:

Definition 34. *We say index i is enabled in $a \in U^?$ if there exists b such that $a \xrightarrow{i} b$.*

Definition 35. *A diverging execution of a is a function $F : \mathbb{N} \to U^? \times \mathbb{N}$ such that:*

1. *$F(0) = (a, i)$ for some i.*
2. *For all n, if $F(n) = (b, j)$ and $F(n+1) = (b', j')$ then $b \xrightarrow{j} b'$.*

and so on.

Definition 36. *We say $H : \mathbb{N} \to List\, M \to U^?$ is a step-preserving map if the following conditions hold:*

1. *If i is enabled in $H(n, A)$ then i is n-enabled in A.*
2. *If $\text{\Large$\ast$}\, A \in \mathcal{V}_n$, $\text{\Large$\ast$}\, B \in \mathcal{V}_n$, $A \curvearrowright^n_i B$ and $H(n, A) \in U$, then $H(n, B) \in U$ and $H(n, A) \xrightarrow{i} H(n, B)$*

---

[12] We shall see that this is indeed the case when we examine the definition of weakest precondition in the model.

3. If $\mathbin{\Large\ast} A \in \mathcal{V}_n$ then $\forall n' \leq n,\ H(n, A) = H(n', A)$

4. If $H(n, A) \xrightarrow{i} H(n, B)$ then $\forall n' \leq n,\ H(n', A) \xrightarrow{i} H(n', B)$

Definition 37. *U has* bounded non-determinism *if $\forall a \in U$, the set $\{b \mid \exists i.\, a \xrightarrow{i} b\}$ is finite.*

We are now able to state the infinite adequacy theorem:

Theorem 38. *Assume U has bounded non-determinism under a step relation $(\xrightarrow{i}: U^? \times U^?)_{i \in \mathbb{N}}$. Let H be a step-preserving map to U. If $[e]; \sigma$ has a diverging execution, and all of the following hold:*

1. $n > 2$,
2. $\forall n'.\, a \cdot b \in \mathcal{V}_{n'}$,
3. $\mathsf{Phy}(\sigma) * \lceil \underline{a} \rceil * \lceil \underline{b} \rceil^{\mathsf{L}} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \varphi(x)\})$,
4. $\forall x.\, \varphi(x) \vdash \mathsf{Stopped}$,
5. $H(n, [b]) \in U$

then there exists a diverging execution of $H(n, [b])$. Moreover, if the execution of $[e]; \sigma$ was fair, so too is the execution of $H(n, [b])$.

A.6   Model and semantics

The semantics closely follows the ideas laid out in [6].

Generic model of base logic The base logic including equality, later, always, and a notion of ownership is defined on $UPred(M)$ for any CMRA M.

Interpretation of base assertions
$$\boxed{[\![\boldsymbol{\Gamma} \vdash \boldsymbol{t} : \mathsf{Prop}]\!] : [\![\boldsymbol{\Gamma}]\!] \xrightarrow{\mathrm{ne}} UPred(\boldsymbol{M})}$$

The type $UPred(M)$ is isomorphic to $M \xrightarrow{\mathrm{mon}} M \rightarrow SProp$. We are thus going to define the assertions as mapping pairs of CMRA elements to sets of step-indices.

We introduce an additional logical connective $\mathsf{Own}(a)$ and $\mathsf{OwnL}(a)$, which will later be used to encode all of $\overline{[P]}^{\iota}$, $\lceil \underline{a} \rceil$, $\lceil \underline{a} \rceil^{\mathsf{L}}$ and $\mathsf{Phy}(\sigma)$.

$$\llbracket \Gamma \vdash t =_\tau u : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda \_, b. \left\{ n \,\middle|\, \llbracket \Gamma \vdash t : \tau \rrbracket_\gamma \overset{n}{=} \llbracket \Gamma \vdash u : \tau \rrbracket_\gamma \wedge b \overset{n}{=} \varepsilon \right\}$$

$$\llbracket \Gamma \vdash \mathsf{False} : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda \_, \_. \emptyset$$

$$\llbracket \Gamma \vdash \mathsf{True} : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda \_, \_. \mathbb{N}$$

$$\llbracket \Gamma \vdash \mathsf{Emp} : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda \_, b. \left\{ n \,\middle|\, b \overset{n}{=} \varepsilon \right\}$$

$$\llbracket \Gamma \vdash P \wedge Q : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(a,b) \cap \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\gamma(a,b)$$

$$\llbracket \Gamma \vdash P \vee Q : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(a,b) \cup \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\gamma(a,b)$$

$$\llbracket \Gamma \vdash P \Rightarrow Q : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \left\{ n \,\middle|\, \begin{array}{c} \forall m, a'. \, m \le n \wedge a \preccurlyeq a' \wedge a' \in \mathcal{V}_m \wedge b \in \mathcal{V}_m \Rightarrow \\ m \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(a',b) \Rightarrow \\ m \in \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\gamma(a',b) \end{array} \right\}$$

$$\llbracket \Gamma \vdash \forall x : \tau. \, P : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \left\{ n \,\middle|\, \forall v \in \llbracket \tau \rrbracket. \, n \in \llbracket \Gamma, x : \tau \vdash P : \mathsf{Prop} \rrbracket_{\gamma[x \hookrightarrow v]}(a,b) \right\}$$

$$\llbracket \Gamma \vdash \exists x : \tau. \, P : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \left\{ n \,\middle|\, \exists v \in \llbracket \tau \rrbracket. \, n \in \llbracket \Gamma, x : \tau \vdash P : \mathsf{Prop} \rrbracket_{\gamma[x \hookrightarrow v]}(a,b) \right\}$$

$$\llbracket \Gamma \vdash \Box P : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(|a|,|b|) \cap \left\{ n \,\middle|\, b \overset{n}{=} |b| \right\}$$

$$\llbracket \Gamma \vdash \mathcal{A}(P) : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(a,b) \cap \left\{ n \,\middle|\, b \overset{n}{=} \varepsilon \right\}$$

$$\llbracket \Gamma \vdash \triangleright P : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \{ n \mid n = 0 \vee n - 1 \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(a,b) \}$$

$$\llbracket \Gamma \vdash P * Q : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \left\{ n \,\middle|\, \begin{array}{c} \exists a_1, a_2, b_1, b_2. \, a \overset{n}{=} a_1 \cdot a_2 \wedge b \overset{n}{=} b_1 \cdot b_2 \wedge \\ n \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(a_1,b_1) \wedge n \in \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\gamma(a_2,b_2) \end{array} \right\}$$

$$\llbracket \Gamma \vdash P \mathbin{-\!*} Q : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b. \left\{ n \,\middle|\, \begin{array}{c} \forall m, a', b'. \, m \le n \wedge a \cdot a' \in \mathcal{V}_m \wedge b \cdot b' \in \mathcal{V}_m \Rightarrow \\ m \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(a',b') \Rightarrow \\ m \in \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\gamma(a \cdot a', b \cdot b') \end{array} \right\}$$

$$\llbracket \Gamma \vdash \mathsf{Own}(a) : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a', b. \left\{ n \,\middle|\, \llbracket \Gamma \vdash a' : \mathsf{M} \rrbracket \overset{n}{\preccurlyeq} a \wedge b \overset{n}{=} \varepsilon \right\}$$

$$\llbracket \Gamma \vdash \mathsf{OwnL}(b) : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda a, b'. \left\{ n \,\middle|\, b \overset{n}{=} b' \right\}$$

$$\llbracket \Gamma \vdash \mathcal{V}(a) : \mathsf{Prop} \rrbracket_\gamma \triangleq \lambda \_, b. \left\{ n \,\middle|\, \llbracket \Gamma \vdash a : \tau \rrbracket \in \mathcal{V}_n \wedge b \overset{n}{=} \varepsilon \right\}$$

Note: There are slight differences between the definition here and the version in the Coq development. For instance, $\mathsf{Own}(a)$ in the Coq development does not stipulate that the second component is in fact equivalent to ε, which makes it non-affine; but in practice we almost always use $\mathcal{A}(\mathsf{Own}(a))$, so here we just present a version equivalent to that.

For every definition, we have to show all the side-conditions: The maps have to be non-expansive and monotone.

Iris model

Semantic domain of assertions. The first complicated task in building a model of full Iris is defining the semantic model of Prop. We start by defining the functor that assembles the CMRAs we need to the global resource CMRA:

$$ResF(T^{\mathrm{op}}, T) \triangleq \left\{ w : \mathbb{N} \xrightarrow{\mathrm{fin}} \mathrm{AG}(\blacktriangleright T), \pi : \mathrm{Ex}(State)^?, g : \Sigma(T^{\mathrm{op}}, T) \right\}$$

Above, $M^?$ is the monoid obtained by adding a unit to M. (It's not a coincidence that we used the same notation for the range of the core; it's the same type either way: $M + 1$.) Remember that Σ is the user-chosen bifunctor from $\mathcal{COFE}$ to $\mathcal{CMRA}$ (see §A.5). $ResF(T^{\mathrm{op}}, T)$ is a CMRA by lifting the individual CMRAs pointwise. Furthermore, since Σ is locally contractive, so is $ResF$.

Now we can write down the recursive domain equation:

$$iPreProp \cong UPred(ResF(iPreProp, iPreProp))$$

iPreProp is a COFE defined as the fixed-point of a locally contractive bifunctor. This fixed-point exists and is unique by America and Rutten's theorem [2, 7]. We do not need to consider how the object is constructed. We only need the isomorphism, given by

$$Res \triangleq ResF(iPreProp, iPreProp)$$
$$Prop \triangleq UPred(Res)$$
$$\xi : Prop \xrightarrow{\mathrm{ne}} iPreProp$$
$$\xi^{-1} : iPreProp \xrightarrow{\mathrm{ne}} Prop$$

We then pick *Prop* as the interpretation of Prop:

$$\llbracket \mathsf{Prop} \rrbracket \triangleq Prop$$

Interpretation of assertions. *Prop* is a *UPred*, and hence the definitions from §A.6 apply. We only have to define the interpretation of the missing connectives, the most interesting bits being primitive view shifts and weakest preconditions.

World satisfaction $\boxed{- \models_- -- : \boldsymbol{\Delta}State \times \boldsymbol{\Delta}\wp(\mathbb{N}) \times Res \times Res \xrightarrow{\mathrm{ne}} SProp}$

$$pre\text{-}wsat(n, \mathcal{E}, \sigma, R, r, s) \triangleq (r \cdot s) \in \mathcal{V}_{n+1} \wedge r.\pi = \mathsf{ex}(\sigma) \wedge \mathrm{dom}(R) \subseteq \mathcal{E} \cap \mathrm{dom}(r.w) \wedge$$
$$\forall \iota \in \mathcal{E}, P \in Prop. (r.w)(\iota) \overset{n+1}{=} \mathsf{ag}(\mathsf{next}(\xi(P))) \Rightarrow n \in P(R(\iota), \varepsilon)$$

$$\sigma \models_{\mathcal{E}} r, s \triangleq \{0\} \cup \left\{ n+1 \, \middle| \, \exists R : \mathbb{N} \xrightarrow{\mathrm{fin}} Res. \, pre\text{-}wsat(n, \mathcal{E}, \sigma, R, r \cdot \prod_{\iota} R(\iota), s) \right\}$$

Notice that the assertion P corresponding to the world ι must be *affine*, in the sense that $n \in P(R(\iota), \varepsilon)$. Without this stipulation, threads would be able to

put linear resources inside invariants and STS interpretations. We prevent this in the current formulation of the logic because otherwise the rules for dealing with invariants would have to be stricter if we still wanted to establish fair refinements.

Primitive view-shift $\boxed{\boldsymbol{pvs}^-_-(-) : \Delta(\wp(\mathbb{N})) \times \Delta(\wp(\mathbb{N})) \times Prop \xrightarrow{\text{ne}} Prop}$

$$pvs^{\mathcal{E}_2}_{\mathcal{E}_1}(P) = \lambda r, s. \left\{ n \;\middle|\; \begin{array}{c} \forall r_{\mathsf{f}}, s_{\mathsf{f}}, k, \mathcal{E}_{\mathsf{f}}, \sigma.\, 0 < k \leq n \wedge (\mathcal{E}_1 \cup \mathcal{E}_2) \,\#\, \mathcal{E}_{\mathsf{f}} \wedge k \in \sigma \models_{\mathcal{E}_1 \cup \mathcal{E}_{\mathsf{f}}} r \cdot r_{\mathsf{f}}, s \cdot s_{\mathsf{f}} \Rightarrow \\ \exists r'.\, k \in P(r', s) \wedge k \in \sigma \models_{\mathcal{E}_2 \cup \mathcal{E}_{\mathsf{f}}} r' \cdot r_{\mathsf{f}}, s \cdot s_{\mathsf{f}} \end{array} \right\}$$

Primitive step-shift $\boxed{\boldsymbol{psvs}^-_-(-) : \Delta(\wp(\mathbb{N})) \times \Delta(\wp(\mathbb{N})) \times Prop \xrightarrow{\text{ne}} Prop}$

$$psvs^{\mathcal{E}_2}_{\mathcal{E}_1}(P) = \lambda r, s. \left\{ n \;\middle|\; \begin{array}{c} \forall r_{\mathsf{f}}, s_{\mathsf{f}}, k, \mathcal{E}_{\mathsf{f}}, \sigma.\, 0 < k \leq n \wedge (\mathcal{E}_1 \cup \mathcal{E}_2) \,\#\, \mathcal{E}_{\mathsf{f}} \wedge k \in \sigma \models_{\mathcal{E}_1 \cup \mathcal{E}_{\mathsf{f}}} r \cdot r_{\mathsf{f}}, s \cdot s_{\mathsf{f}} \Rightarrow \\ \exists r', s'.\, k \in P(r', s') \wedge k \in \sigma \models_{\mathcal{E}_2 \cup \mathcal{E}_{\mathsf{f}}} r' \cdot r_{\mathsf{f}}, s' \cdot s_{\mathsf{f}} \wedge s \curvearrowright^k s' \end{array} \right\}$$

Weakest precondition $\boxed{\boldsymbol{wp}_-(-,-) : \Delta(\wp(\mathbb{N})) \times \Delta(Exp) \times (\Delta(Val) \xrightarrow{\text{ne}} Prop) \xrightarrow{\text{ne}} Prop}$
wp is defined as the fixed-point of a contractive function.

$$pre\text{-}wp(wp)(\mathcal{E}, e, \varphi) \triangleq \lambda r, s. \left\{ n \;\middle|\; \begin{array}{l} \forall r_{\mathsf{f}}, s_{\mathsf{f}}, m, \mathcal{E}_{\mathsf{f}}, \sigma.\, 0 \leq m < n \wedge \mathcal{E} \,\#\, \mathcal{E}_{\mathsf{f}} \wedge m + 1 \in \sigma \models_{\mathcal{E} \cup \mathcal{E}_{\mathsf{f}}} r \cdot r_{\mathsf{f}}, s \cdot s_{\mathsf{f}} \Rightarrow \\ (\forall v.\, \mathsf{expr2val}(e) = v \Rightarrow \exists r'.\, m + 1 \in \varphi(v)(r', s) \wedge m + 1 \in \sigma \models_{\mathcal{E} \cup \mathcal{E}_{\mathsf{f}}} r' \cdot r_{\mathsf{f}}, s \cdot s_{\mathsf{f}}) \wedge \\ (\mathsf{expr2val}(e) = \bot \wedge 0 < m \Rightarrow \mathsf{red}(e, \sigma) \wedge \forall e_2, \sigma_2, e_{\mathsf{f}}.\, e, \sigma \to e_2, \sigma_2, e_{\mathsf{f}} \Rightarrow \\ \quad \exists r_1, r_2, s_1, s_2.\, m \in \sigma \models_{\mathcal{E} \cup \mathcal{E}_{\mathsf{f}}} r_1 \cdot r_2 \cdot r_{\mathsf{f}} \wedge m \in wp(\mathcal{E}, e_2, \varphi)(r_1, s_1) \wedge \\ \quad ((e_{\mathsf{f}} = \bot \wedge s_2 \overset{m}{=} \varepsilon) \vee m \in wp(\top, e_{\mathsf{f}}, \lambda\_.\, \lambda\_.\, \mathbb{N})(r_2, s_2)) \wedge \\ \quad s \curvearrowright^m s_1 \cdot s_2 \end{array} \right\}$$

$$wp_{\mathcal{E}}(e, \varphi) \triangleq \mathit{fix}(pre\text{-}wp)(\mathcal{E}, e, \varphi)$$

Interpretation of program logic assertions $\boxed{\llbracket \Gamma \vdash t : \mathsf{Prop} \rrbracket : \llbracket \Gamma \rrbracket \xrightarrow{\text{ne}} Prop}$ $\boxed{\overline{P}^{\iota}}$,
$\boxed{\overline{a}}$ and $\mathsf{Phy}(\sigma)$ are just syntactic sugar for forms of $\mathsf{Own}(-)$.

$$\overline{P}^{\iota} \triangleq \mathsf{Own}([\iota \hookrightarrow \mathsf{ag}(\mathsf{next}(\xi(P)))], \varepsilon, \varepsilon)$$
$$\overline{a} \triangleq \mathsf{Own}(\varepsilon, \varepsilon, a)$$
$$\overline{a}^{\mathsf{L}} \triangleq \mathsf{OwnL}(\varepsilon, \varepsilon, a)$$
$$\mathsf{Phy}(\sigma) \triangleq \mathsf{Own}(\varepsilon, \mathsf{ex}(\sigma), \varepsilon)$$

$$\llbracket \Gamma \vdash {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P : \mathsf{Prop} \rrbracket_{\gamma} \triangleq pvs^{\llbracket \Gamma \vdash \mathcal{E}_2 : \mathsf{InvMask} \rrbracket_{\gamma}}_{\llbracket \Gamma \vdash \mathcal{E}_1 : \mathsf{InvMask} \rrbracket_{\gamma}}(\llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_{\gamma})$$

$$\llbracket \Gamma \vdash {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} P : \mathsf{Prop} \rrbracket_{\gamma} \triangleq psvs^{\llbracket \Gamma \vdash \mathcal{E}_2 : \mathsf{InvMask} \rrbracket_{\gamma}}_{\llbracket \Gamma \vdash \mathcal{E}_1 : \mathsf{InvMask} \rrbracket_{\gamma}}(\llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_{\gamma})$$

$$\llbracket \Gamma \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} : \mathsf{Prop} \rrbracket_{\gamma} \triangleq wp_{\llbracket \Gamma \vdash \mathcal{E} : \mathsf{InvMask} \rrbracket_{\gamma}}(\llbracket \Gamma \vdash e : \mathsf{Expr} \rrbracket_{\gamma}, \lambda v.\, \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_{\gamma[x \hookrightarrow v]})$$

Remaining semantic domains, and interpretation of non-assertion terms. The remaining domains are interpreted as follows:

$$\llbracket \mathsf{InvName} \rrbracket \triangleq \Delta\mathbb{N} \qquad \llbracket \mathsf{Val} \rrbracket \triangleq \Delta Val \qquad \llbracket 1 \rrbracket \triangleq \Delta\{()\}$$

$$\llbracket \mathsf{InvMask} \rrbracket \triangleq \Delta_{\wp}(\mathbb{N}) \qquad \llbracket \mathsf{Expr} \rrbracket \triangleq \Delta Expr \qquad \llbracket \tau \times \tau' \rrbracket \triangleq \llbracket \tau \rrbracket \times \llbracket \tau \rrbracket$$

$$\llbracket \mathsf{M} \rrbracket \triangleq F(Prop) \qquad \llbracket \mathsf{State} \rrbracket \triangleq \Delta State \qquad \llbracket \tau \to \tau' \rrbracket \triangleq \llbracket \tau \rrbracket \xrightarrow{\mathrm{ne}} \llbracket \tau \rrbracket$$

For the remaining base types τ defined by the signature $\mathcal{S}$, we pick an object X_τ in $\mathcal{COFE}$ and define

$$\llbracket \tau \rrbracket \triangleq X_\tau$$

For each function symbol $F : \tau_1, \dots, \tau_n \to \tau_{n+1} \in \mathcal{F}$, we pick a function $\llbracket F \rrbracket :$ $\llbracket \tau_1 \rrbracket \times \cdots \times \llbracket \tau_n \rrbracket \xrightarrow{\mathrm{ne}} \llbracket \tau_{n+1} \rrbracket$.

Interpretation of non-propositional terms $\qquad\qquad \boxed{\llbracket \boldsymbol{\Gamma \vdash t : \tau} \rrbracket : \llbracket \boldsymbol{\Gamma} \rrbracket \xrightarrow{\mathrm{ne}} \llbracket \boldsymbol{\tau} \rrbracket}$

$$\llbracket \Gamma \vdash x : \tau \rrbracket_\gamma \triangleq \gamma(x)$$

$$\llbracket \Gamma \vdash F(t_1, \dots, t_n) : \tau_{n+1} \rrbracket_\gamma \triangleq \llbracket F \rrbracket(\llbracket \Gamma \vdash t_1 : \tau_1 \rrbracket_\gamma, \dots, \llbracket \Gamma \vdash t_n : \tau_n \rrbracket_\gamma)$$

$$\llbracket \Gamma \vdash \lambda x : \tau.\, t : \tau \to \tau' \rrbracket_\gamma \triangleq \lambda u : \llbracket \tau \rrbracket.\, \llbracket \Gamma, x : \tau \vdash t : \tau \rrbracket_{\gamma[x \hookrightarrow u]}$$

$$\llbracket \Gamma \vdash t(u) : \tau' \rrbracket_\gamma \triangleq \llbracket \Gamma \vdash t : \tau \to \tau' \rrbracket_\gamma(\llbracket \Gamma \vdash u : \tau \rrbracket_\gamma)$$

$$\llbracket \Gamma \vdash \mu x : \tau.\, t : \tau \rrbracket_\gamma \triangleq \mathit{fix}(\lambda u : \llbracket \tau \rrbracket.\, \llbracket \Gamma, x : \tau \vdash t : \tau \rrbracket_{\gamma[x \hookrightarrow u]})$$

$$\llbracket \Gamma \vdash () : 1 \rrbracket_\gamma \triangleq ()$$

$$\llbracket \Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2 \rrbracket_\gamma \triangleq (\llbracket \Gamma \vdash t_1 : \tau_1 \rrbracket_\gamma, \llbracket \Gamma \vdash t_2 : \tau_2 \rrbracket_\gamma)$$

$$\llbracket \Gamma \vdash \pi_i(t) : \tau_i \rrbracket_\gamma \triangleq \pi_i(\llbracket \Gamma \vdash t : \tau_1 \times \tau_2 \rrbracket_\gamma)$$

$$\llbracket \Gamma \vdash \varepsilon : \mathsf{M} \rrbracket_\gamma \triangleq \varepsilon$$

$$\llbracket \Gamma \vdash |a| : \mathsf{M} \rrbracket_\gamma \triangleq |\llbracket \Gamma \vdash a : \mathsf{M} \rrbracket_\gamma|$$

$$\llbracket \Gamma \vdash a \cdot b : \mathsf{M} \rrbracket_\gamma \triangleq \llbracket \Gamma \vdash a : \mathsf{M} \rrbracket_\gamma \cdot \llbracket \Gamma \vdash b : \mathsf{M} \rrbracket_\gamma$$

An environment Γ is interpreted as the set of finite partial functions ρ, with $\mathrm{dom}(\rho) = \mathrm{dom}(\Gamma)$ and $\rho(x) \in \llbracket \Gamma(x) \rrbracket$.

Logical entailment. We can now define *semantic* logical entailment.

Interpretation of entailment $\qquad\qquad\qquad\qquad \boxed{\llbracket \boldsymbol{\Gamma \mid \Theta \vdash P} \rrbracket : Prop}$

$$\llbracket \Gamma \mid \Theta \vdash P \rrbracket \triangleq \forall n \in \mathbb{N}.\ \forall r, s \in \mathit{Res}.\ \forall \gamma \in \llbracket \Gamma \rrbracket,$$

$$\big(\forall Q \in \Theta.\, n \in \llbracket \Gamma \vdash Q : \mathsf{Prop} \rrbracket_\gamma(r, s) \big) \Rightarrow n \in \llbracket \Gamma \vdash P : \mathsf{Prop} \rrbracket_\gamma(r, s)$$

The soundness statement of the logic reads

$$\Gamma \mid \Theta \vdash P \Rightarrow \llbracket \Gamma \mid \Theta \vdash P \rrbracket$$

A.7   Derived proof rules and other constructions

We will below abuse notation, using the *term* meta-variables like v to range over (bound) *variables* of the corresponding type. We omit type annotations in binders and equality, when the type is clear from context. We assume that the signature $\mathcal{S}$ embeds all the meta-level concepts we use, and their properties, into the logic. (The Coq formalization is a *shallow embedding* of the logic, so we have direct access to all meta-level notions within the logic anyways.)

Persistent/Relevant assertions.

Definition 39. *An assertion P is* persistent *or* relevant *if $P \vdash \Box P$.*

Of course, $\Box P$ is persistent for any P. Furthermore, by the proof rules given in §A.5, $t = t'$ as well as $\lceil \overline{[a]} \rceil$, $\mathcal{V}(a)$ and $\boxed{P}^{\iota}$ are persistent. Persistence is preserved by conjunction, disjunction, separating conjunction as well as universal and existential quantification.

In our proofs, we will implicitly add and remove $\Box$ from persistent assertions as necessary.

Affine assertions.

Definition 40. *An assertion P is* affine *if $P \vdash \mathcal{A}(P)$.*

In our proofs, we will implicitly add and remove $\mathcal{A}(-)$ from persistent assertions as necessary.

Timeless assertions. We can show that the following additional closure properties hold for timeless assertions:

$$\frac{\Gamma \vdash \mathsf{timeless}(P) \qquad \Gamma \vdash \mathsf{timeless}(Q)}{\Gamma \vdash \mathsf{timeless}(P \wedge Q)} \qquad \frac{\Gamma \vdash \mathsf{timeless}(P) \qquad \Gamma \vdash \mathsf{timeless}(Q)}{\Gamma \vdash \mathsf{timeless}(P \vee Q)}$$

$$\frac{\Gamma \vdash \mathsf{timeless}(P) \qquad \Gamma \vdash \mathsf{timeless}(Q)}{\Gamma \vdash \mathsf{timeless}(P * Q)} \qquad \frac{\Gamma \vdash \mathsf{timeless}(P)}{\Gamma \vdash \mathsf{timeless}(\Box P)}$$

Some similar rules apply for $\mathsf{atimeless}(\text{-})$.

Program logic Hoare triples and view shifts are syntactic sugar for weakest (liberal) preconditions and primitive view shifts, respectively:

$$P \xrightarrow{\mathcal{E}_1 \Rrightarrow \mathcal{E}_2} Q \triangleq \Box(P \Rightarrow {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q)$$

$$\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} \triangleq \mathcal{A}(\Box(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\lambda v.\, Q\})) \qquad P \xrightarrow{\mathcal{E}_1 \Rrightarrow \mathcal{E}_2} Q \triangleq \Box(P \Rightarrow {}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2} Q)$$

$$P \xrightarrow{\mathcal{E}_1 \Lleftarrow\!\!\Rrightarrow \mathcal{E}_2} Q \triangleq P \xrightarrow{\mathcal{E}_1 \Rrightarrow \mathcal{E}_2} Q \wedge Q \xrightarrow{\mathcal{E}_2 \Rrightarrow \mathcal{E}_1} P$$

We write just one mask for a view shift when $\mathcal{E}_1 = \mathcal{E}_2$. Clearly, all of these assertions are persistent. The convention for omitted masks is similar to the base logic: An omitted $\mathcal{E}$ is $\top$ for Hoare triples and $\emptyset$ for view shifts.

Derived Rules We omit many of the derived rules for Hoare triples, view shifts, and step shifts. The interested reader can consult the Coq mechanization.

$$(s, T) \to (s', T') \triangleq s \to s' \wedge \mathcal{L}(s) \uplus T = \mathcal{L}(s') \uplus T'$$

$$s \xrightarrow{\overline{T}} s' \triangleq \exists T_1, T_2.\, T_1 \;\#\; \mathcal{L}(s) \cup T \wedge (s, T_1) \to (s', T_2)$$

STS-ALLOC
$$\mathcal{A}(\varphi(s)) \Rrightarrow \exists \iota, \gamma.\, \mathsf{StsCtx}^\gamma(\mathcal{S}, \varphi) * \mathsf{StsSt}^\gamma(s, \mathcal{T} \setminus \mathcal{L}(s))$$

STS-ST-SPLIT
$$\mathsf{StsSt}^\gamma(s, T_1 \uplus T_2) \Leftrightarrow \mathsf{StsSt}^\gamma(s, T_1) * \mathsf{StsSt}^\gamma(s, T_2)$$

STS-OPEN
$$\frac{\begin{array}{c}(\text{additional side conditions omitted}) \qquad \mathsf{atomic}(e) \\ \forall s.\, s_0 \xrightarrow{\overline{T}}^* s.\, \{\mathcal{A}(\varphi(s)) * P\}\, e \\ \{v.\, \exists s', T'.\, (s, T) \to^* (s', T') * \mathcal{A}(\varphi(s')) * Q\}\end{array}}{\begin{array}{c}\mathsf{StsCtx}^\gamma(\mathcal{S}, \varphi) \vdash \{\mathsf{StsSt}^\gamma(s_0, T) * P\}\, e \\ \{v.\, \exists s', T'.\, \mathsf{StsSt}^\gamma(s', T') * Q\}\end{array}}$$

Global functor, ghost ownership, and namespaces For composability reasons, Iris makes it possible to combine a collection of CMRAs to get a larger "global" CMRA. This makes it possible to combine proofs that are done using a certain CMRA M with ones that are done doing another CMRA M'. We do omit the descriptions of these mechanisms; the interested reader should consult the original Iris 2.0 documentation.

A.8   Refinement RA

We now briefly describe the RA used to model source assetions. This entire section is new.

Fix a source language Λ. We say that a list of configurations, C is compatible with a list of thread indices, L, written $\mathsf{compat}(C, L)L$, if:

$$\mathsf{compat}([], []) \qquad \mathsf{compat}([\rho], []) \qquad \frac{\mathsf{compat}(C +\!\!+ [\rho], L) \qquad \rho \xrightarrow{i} \rho'}{\mathsf{compat}(C +\!\!+ [\rho, \rho'], L +\!\!+ [i])}$$

We define an RA REFINE(Λ):

$$\textsc{Refine}(\Lambda) \triangleq \mathit{View} \times \mathcal{P}^{\mathrm{fin}}(\mathbb{N}) \times \mathit{List\ Config} \times \mathit{List\ Nat}$$

$$\text{where } \mathit{View} \triangleq \{\mathsf{master}, \mathsf{snapshot}\}$$

$$\mathcal{V} \triangleq \left\{ (v, S, C, L) \in \textsc{Refine}(\Lambda) \;\middle|\; \begin{array}{l} (C = [] \wedge L = [] \wedge S = \emptyset) \vee \\ (\exists C', T, \sigma.\, C = C', ([T]; \sigma) \wedge \\ \quad (\forall i \in S, i < |T|) \wedge \\ \quad \mathsf{compat}(C, L)) \end{array} \right\}$$

$$|(v, S, C, L)| \triangleq (\mathsf{snapshot}, \emptyset, C, L)$$

$$(v, S, C, L) \cdot (v', S', C', L') \triangleq (\max(v, v'), S \uplus S', \max(C, C'), \max(L, L)')$$

where $\max(v, v')$ is master if either v or v' is master, and the maximum of two lists is just the longer of the two. We have an additional proviso stating that multiplication is only defined if all of the following hold:

1. Either $v = \mathsf{snapshot}$ or $v' = \mathsf{snapshot}$
2. If $v = v' = \mathsf{snapshot}$ then $\exists C'', L''$ such that either:
   (a) $C = C' +\!\!+ C''$, $L = L' +\!\!+ L''$ and, $\forall i \in S', i \notin L''$, or
   (b) $C' = C +\!\!+ C''$, $L' = L +\!\!+ L''$ and $\forall i \in S, i \notin L''$.
3. If $v = \mathsf{snapshot}$ and $v' = \mathsf{master}$ then $\exists C'', L''$ such that $C' = C +\!\!+ C''$, $L' = L +\!\!+ L''$ and $\forall i \in S', i \notin L''$.
4. If $v = \mathsf{master}$ and $v' = \mathsf{snapshot}$ then $\exists C'', L''$ such that $C = C' +\!\!+ C''$, $L = L' +\!\!+ L''$ and, $\forall i \in S, i \notin C''$.

Intuitively, the second component of an element, S represents a set of thread ID's "owned" by this element, and the C and L are some prefix of an execution of a source program. Then, condition one for multiplication being defined says there can be at most one master. Condition two says that, among two snapshots, one can be longer than the other, but the longer one cannot contain any additional steps by threads owned by the other. Condition three and four say that a snapshot must be a prefix of master, subject to the constraint that the master cannot contain any extra steps by threads owned by the snapshot.

The definition of $\curvearrowright$ for this RA is somewhat complicated. Let us motivate it in words – a reader that wants details is advised to consult the Coq formalization. Only snapshots may take steps. Intuitively, the snapshot is obligated to step every thread it controls (i.e. every index in S) which can possibly take a step. But, since the snapshot is only a partial prefix of the program execution, other threads not controlled by this snapshot may have taken steps. Thus, we first non-deterministically speculate some steps performed by other threads and then perform all of the required steps for the owned threads. Finally, since performing those steps may fork off new threads, we add the thread ids of the new threads to S.

Of course, for extra flexibility, we are allowed to step each thread more than once. We do not bake delay steps into this monoid. Rather, if we want delay

steps, we first transform Λ into a language Λ' that has additional "stutter" steps for delay.

Finally, we can interpret source as:

$$\mathsf{source}(i, e) \triangleq \exists T, \sigma, C, L.\; \overline{(\mathsf{snapshot}, \{i\}, C + [[T]; \sigma], L)}^{\,\llcorner} \wedge (T[i] = e)$$

We also get an assertion $\mathsf{SPhy}(\sigma)$ for talking about the state of the assertion:

$$\mathsf{SPhy}(\sigma) \triangleq \exists T, C, L.\; \overline{(\mathsf{master}, \emptyset, C + [[T]; \sigma], L)}$$

We can use this "large footprint" assertion about source programs to derive smaller assertions, just as we do for the $\mathsf{Phy}(\sigma)$ assertion in Iris.

Finally, with some effort we can use this CMRA with the infinite adequacy theorem to get the refinement results stated in the body of the paper.

B   Case Studies

B.1   Session-Typed Language Translation

In this appendix we develop the logical relation used for our compiler correctness proof. The body of the paper has already explained the meaning of session types. But it avoided the crucial issue of showing that this relation is well-defined. What actually happens is we develop a state transition system for sessions which is *parameterized* by an interpretation of types. As we'll see, we then take a fixed point which is defined using this construction.

In our actual proof though, we do not work with the Hoare triple versions of our rules. Instead, we work with a more primitive form called weakest precondition, since it is much easier to work with in a proof assistant. We first sketch the connection between these and Hoare triples.

Then, we give this parameterized STS construction and state the mechanized weakest precondition proof rules using this STS for the message passing primitives. This time around we will be more explicit about the delay constants at first, but then show why it is OK to hide them. Next, we describe some additional Iris features we'll need and use them to define the logical relation (in particular, the $\triangleright$ modality we mentioned without explanation in the main text). We then prove that our logical relation is sound (that is, it implies refinement for closed terms). Finally, we prove the fundamental lemma, which shows that the logical relation holds between well-typed expressions and their translation. This completes the proof.

Weakest Precondition In Iris, Hoare triples are not a primitive form. Instead, they are defined in terms of a *weakest pre-condition* primitive as follows:

$$\{P\}\, e\, \{x.\, Q\} \triangleq \mathcal{A}(\Box(P \twoheadrightarrow \mathsf{wp}\, e\, \{x.\, Q\}))$$

The assertion $\mathsf{wp}\, e\, \{x.\, Q\}$ expresses ownership of resources that is strong enough to justify the safe execution of e, such that when e terminates, Q holds. This is

an *ephemeral* assertion in the sense that, like *e.g.* $l \hookrightarrow v$, it can be used at most once and is the invalidated.

The magic wand $P \twoheadrightarrow \mathsf{wp}\, e\, \{x.\, Q\}$ says that if we are given resources satisfying P, we have enough resources to satisfy the weakest pre-condition. The wand $\twoheadrightarrow$ works like an implication, but is right adjoint to the separating conjunction $*$ instead of plain conjunction $\wedge$.

Notice that in the example in §3, when we summarized the current state of the proof, we often said things like "Our current resources are P and we have to verify the following code e (with post-condition Q)". This exactly corresponds to a proof-state where our current logical context is P, and the goal is $\mathsf{wp}\, e\, \{Q\}$:

$$P \vdash \mathsf{wp}\, e\, \{Q\}$$

It should not be surprising that in carrying out Iris proofs in Coq, we generally work with weakest pre-conditions.

Next, we wrap the *always* modality $\square$ around the wand. This is to enforce that proofs of Hoare triples be *persistent*, *i.e.,* the modality makes sure that a Hoare triple, once established, will remain valid throughout the remaining verification. By default, assertions in Iris are ephemeral and hence can be used only once.

Finally, there is an affine modality wrapping the whole thing – there should be no other source hidden within such a Hoare triple. We only get to use ones in P.

The Session STS The states, tokens, and transitions we gave in the main text did not mention the logical relation, so there is no concern about circularity in their definition. The only problem was the definition of the state interpretation, which was given as:

$$\varphi_{S,c}(n_\mathsf{l}, n_\mathsf{r}, l_\mathsf{l}, l_\mathsf{r}) \triangleq \exists L_\mathsf{c}, L_\mathsf{h}.$$
$$\Big(c \hookrightarrow_\mathsf{s} (L_\mathsf{c}, []) * \mathsf{linklist}(L_\mathsf{h}, l_\mathsf{l}, l_\mathsf{r}) *$$
$$(L_\mathsf{h} \simeq^{\mathcal{L}} L_\mathsf{c} : S^{n_\mathsf{l}}) * n_\mathsf{l} + |L_\mathsf{c}| = n_\mathsf{r} \Big) \vee ...$$

This implicitly relied on the definition of the logical relation via the lifting of the relation to lists of values:

$$[] \simeq^{\mathcal{L}} [] : S \qquad\qquad \frac{\triangleright(v \simeq^{\mathcal{V}} V : \tau) * L_\mathsf{h} \simeq^{\mathcal{L}} L_\mathsf{c} : S}{v L_\mathsf{h} \simeq^{\mathcal{L}} V L_\mathsf{c} : ?\tau.\, S}$$

What we now do is have the lifting of lists take a *parameter*, Θ, which is a pre-existing interpretation of types (*i.e.,* a map from types to a relation between values of the target and source):

$$[] \simeq_\Theta^\mathcal{L} [] : S \qquad\qquad \frac{\Theta(\tau)(v, V) * L_\mathsf{h} \simeq_\Theta^\mathcal{L} L_\mathsf{c} : S}{v L_\mathsf{h} \simeq_\Theta^\mathcal{L} V L_\mathsf{c} : ?\tau. S}$$

Then, φ will also take this parameter and pass it to the list relation:

$$\varphi_{\Theta, S, c}(n_\mathsf{l}, n_\mathsf{r}, l_\mathsf{l}, l_\mathsf{r}) \triangleq \exists L_\mathsf{c}, L_\mathsf{h}.$$
$$\left(c \hookrightarrow_\mathsf{s} (L_\mathsf{c}, []) * *\mathsf{linklist}(L_\mathsf{h}, l_\mathsf{l}, l_\mathsf{r}) * \right.$$
$$\left. L_\mathsf{h} \simeq_\Theta^\mathcal{L} L_\mathsf{c} : S^{n_\mathsf{l}} * n_\mathsf{l} + |L_\mathsf{c}| = n_\mathsf{r} \right) \vee ...$$

We write $\uparrow (n_\mathsf{l}, -, l, -)$ for the set of all states that have first component n_l and third component l, and symmetrically for the right counts and heap pointer.

Finally, we can define an assertion $\mathsf{Session}_\Theta(l, c_s, S)$ which asserts (1) existence of an STS governing l and c, (2) comes equpped with the tokens needed for manipulating the end-point indicated by s, and (3) ensures that the current type of the end-point for s has type S:

$$\mathsf{Session}_\Theta(l, c_s, S) \triangleq$$
$$(s = \mathsf{left} \Rightarrow \exists S_0, n_\mathsf{l}, \gamma, \mathsf{StsCtx}^\gamma(\mathcal{S}, \varphi_{\Theta, S, c}(-))$$
$$* \mathsf{StsSt}^\gamma(\uparrow (n_\mathsf{l}, -, l, -), \{[\mathsf{Left}\ n] \mid n > n_\mathsf{l}\} * S_0^n = S)$$
$$\vee\ (s = \mathsf{right} \Rightarrow \exists S_0, n_\mathsf{r}, \gamma, \mathsf{StsCtx}^\gamma(\mathcal{S}, \varphi_{\Theta, S, c}(-))$$
$$* \mathsf{StsSt}^\gamma(\uparrow (-, n_\mathsf{r}, -, l), \{[\mathsf{Right}\ n] \mid n > n_\mathsf{l}\} * S_0^n = \overline{S})$$

Of course, we've now only deferred the problem – eventually we do need to plug in the desired logical relation for Θ. For now, the key is that we can still prove things about channels which use this invariant; its just that the proof rules are also parameterized by Θ.

To show these rules, we need to be a bit more precise about delay constants for a moment. The triple about the receive primitive mentioned in the body of the paper omitted delay constants, and in general we have not been explicit about such constants in the main text. Let us first be a bit more precise about this, so that we can justify why it is safe to ignore them subsequently.

As we said when explaining the source assertion when we do a proof we need to fix a number D that will be an upper bound throughout for all delay constants. In the Coq development, we somewhat profligately proved things involving the above STS assuming that this upper bound was at least ≥ 100. So for concreteness, let us just fix this D now to be 100. Then we have proved the following rules (written in the weakest precondition style instead of Hoare triples):

$$\frac{1 < d \leq D \qquad d' \leq D}{\mathsf{source}(i, K[\mathsf{newch}], d) \vdash \mathsf{wp} \; \mathsf{heapNewch} \left\{ \begin{array}{l} (l, l). \exists c. \, \mathsf{source}(i, K[(c_{\mathsf{left}}, c_{\mathsf{right}})], d') \\ * \, \mathsf{Session}_{\Theta}(l, c_{\mathsf{left}}, S) * \mathsf{Session}_{\Theta}(l, c_{\mathsf{right}}, \overline{S}) \end{array} \right\}}$$

$$\frac{1 < d \leq D - 2 \qquad d' \leq D - 2 \qquad \forall v, V. \, \Theta(\tau)(v, V) \vdash \triangleright P(v, V)}{\begin{array}{l} \mathsf{source}(i, K[\mathsf{recv}(c_s)], d) * \mathsf{Session}_{\Theta}(l, c_s, ?\tau. \, S) \vdash \\ \quad \mathsf{wp} \; \mathsf{heapRecv} \; l \, \{(l', v). \, \exists V. \, \mathsf{source}(i, K[(c_s, V)], d') * P(v, V) * \mathsf{Session}_{\Theta}(l, c_s, S)\} \end{array}}$$

$$\frac{4 < d \leq D \qquad d' \leq D - 1}{\begin{array}{l} \mathsf{source}(i, K[\mathsf{send}(c_s, V)], d) * \Theta(\tau)(v, V) * \mathsf{Session}_{\Theta}(l, c_s, !\tau. \, S) \vdash \\ \quad \mathsf{wp} \; \mathsf{heapSend} \; l \; v \, \{l'. \, \mathsf{source}(i, K[c_s], d') * \mathsf{Session}_{\Theta}(l', c_s, S)\} \end{array}}$$

In addition, to the explicit delays, we also have this business about $\triangleright P$ in the second rule, which we can ignore for now. Note that if $4 < k \leq D - 2$, then k satisfies all of the side conditions placed on the delay constants that we start with in each rule (*i.e.*, d). Moreover, such a d satisfies all of the constraints placed on the *ending* delay constant (*i.e.*, d'). That means if we start with such a k, at the end we can continue with the same k. In particular, a choice of $k = 50$ works. It's also the case that 0 satisfies the conditions on the d' above; so whatever delay we start with, we can always end up with 0, if we wish.

Putting this together, we will define $\mathsf{source}(i, E)$ (that is, *without* the delay constant) as:

$$\mathsf{source}(i, E) \triangleq \mathsf{source}(i, E, 50) \qquad\qquad \mathsf{source}(i, V) \triangleq \mathsf{source}(i, V, 0)$$

That is, when we are working with an expression, we assume an implicit delay constant of 50; when the source thread is a value, it is 0'd out. Rewriting the rules above with this new form, we can ignore the delay constants:

$$\mathsf{source}(i, K[\mathsf{newch}]) \vdash \mathsf{wp} \; \mathsf{heapNewch} \left\{ \begin{array}{l} (l, l). \exists c. \, \mathsf{source}(i, K[(c_{\mathsf{left}}, c_{\mathsf{right}})]) \\ * \, \mathsf{Session}_{\Theta}(l, c_{\mathsf{left}}, S) * \mathsf{Session}_{\Theta}(l, c_{\mathsf{right}}, \overline{S}) \end{array} \right\}$$

$$\frac{\forall v, V. \, \Theta(\tau)(v, V) \vdash \triangleright P(v, V)}{\begin{array}{l} \mathsf{source}(i, K[\mathsf{recv}(c_s)]) * \mathsf{Session}_{\Theta}(l, c_s, ?\tau. \, S) \vdash \\ \quad \mathsf{wp} \; \mathsf{heapRecv} \; l \, \{(l', v). \, \exists V. \, \mathsf{source}(i, K[(c_s, V)]) * P(v, V) * \mathsf{Session}_{\Theta}(l, c_s, S)\} \end{array}}$$

$$\begin{array}{l} \mathsf{source}(i, K[\mathsf{send}(c_s, V)]) * \Theta(\tau)(v, V) * \mathsf{Session}_{\Theta}(l, c_s, !\tau. \, S) \vdash \\ \quad \mathsf{wp} \; \mathsf{heapSend} \; l \; v \, \{l'. \, \mathsf{source}(i, K[c_s]) * \mathsf{Session}_{\Theta}(l', c_s, S)\} \end{array}$$

To see that these implicit rules follow from the explicit delay constant form, observe that if (1) $K[E]$ is a value, E must in fact be a value, and (2) if $K[V]$ is a

value, then for all V', $K[V']$ is a value. This means for each of the above rules, we first determine whether the evaluation K will be a value after we substitute the return values in the post-condition; if it is, we apply the corresponding original rule taking $d' = 0$. If not, we take $d' = 50$.

The reason we chose the "implicit" delay for values to be 0 is so that $\mathsf{source}(i, V) \vdash \mathsf{Stopped}$, and so that our refinement rule can also be written as:

$$\frac{\{\mathsf{source}(i, E)\} \, e \, \{x. \, \exists V. \, \mathsf{source}(i, V) * x \approx V\}}{e \sqsubseteq E}$$

Logical Relation Now that we have the STS defined in this parameterized way, we can follow up by defining the logical relation in a non-circular way. We follow the standard set-up of defining an interpretation of types that relates values, then lifting this to a relation on closed expressions, and then using that to define a relation on open expressions. In general, given a function Θ which maps types to Iris relations on values of the target and source, we write $v \simeq^{\mathcal{V}}_{\Theta} V : \tau$ to say that v and V are related at the interpretation of τ under Θ.

Lifting relations on values to expressions. The expression lifting operation is generic with respect to the final interpretation on values, so let us first define that. Given Θ, we define its lifting between expressions of the target and source as:

$$e \simeq^{\mathcal{E}}_{\Theta} E : \tau \triangleq \forall i, K. \, \mathcal{A}(\mathsf{source}(i, K[E]) \mathbin{-\!\!*} \mathsf{wp} \, e \, \{v. \, \exists V. \, \mathcal{A}(v \simeq^{\mathcal{V}}_{\Theta} V : \tau) * \mathsf{source}(i, K[V])\})$$

This is an assertion saying that for any choice of thread i and evaluation context K, if we are given $\mathsf{source}(i, K[E])$ we can prove a weakest-precondition for e in which the executions end in related values according to Θ at the appropriate type. We wrap this wand in an affine modality to ensure there are no implicit other threads owned. Also, in the post-condition, the proof that the values are related must be affine for a similar reason.

Let us note that the standard way such lifting relations are defined (in for instance, the sequential case) is more or less to say that e and E are related if they evaluate to related values. Indeed, that's essentially what we are saying here, but using the weakest precondition and all of the other machinery we have built up so that we're implicitly talking about *concurrent* executions and fairness.

Note that so long as $v \simeq^{\mathcal{V}}_{\Theta} V : \tau$ is affine, $v \simeq^{\mathcal{V}}_{\Theta} V : \tau \vdash v \simeq^{\mathcal{E}}_{\Theta} V : \tau$

Logical relation on values. Iris has two important features which we did not explain in the main text. First, there is the modality $\triangleright P$ (called "later"), which we have asked the reader to ignore a few times. This describes resources which satisfy P at *one lower step-index*. So, $P \vdash \triangleright P$, but not conversely. Second, we can

construct fixed-points of arbitrary recursive definition of a predicate, so long as all recursive occurences are under the "later" modality. Such recursive definitions are called "guarded".

We will define our interpretation of types by taking a fixed point of a guarded recursive definition. First, we define the function F which we are going to take the fixed-point of. Given an interpretation of types, Θ (that is, a map from types to an Iris relation on values), we define $F(\Theta)$ as yet another interpretation of types, defined by structural induction on types:

$$n \simeq^{\mathcal{V}}_{F(\Theta)} n : \mathsf{Int} \qquad () \simeq^{\mathcal{V}}_{F(\Theta)} () : \mathsf{Unit} \qquad \frac{(v_1 \simeq^{\mathcal{V}}_{F(\Theta)} V_1 : \tau_1) * (v_2 \simeq^{\mathcal{V}}_{F(\Theta)} V_2 : \tau_2)}{(v_1, v_2) \simeq^{\mathcal{V}}_{F(\Theta)} (V_1, V_2) : \tau_1 \otimes \tau_2}$$

$$\frac{\forall v', V'. \mathcal{A}(v' \simeq^{\mathcal{V}}_{F(\Theta)} V' : \tau' \mathrel{-\!\!*} (\lambda x.\, e)\, v' \simeq^{\mathcal{E}}_{F(\Theta)} (\lambda x.\, E) V' : \tau)}{\lambda x.\, e \simeq^{\mathcal{V}}_{F(\Theta)} \lambda x.\, E : \tau' \multimap \tau}$$

$$\frac{\mathsf{Session}_{\triangleright\Theta}(l, c_s, S)}{l \simeq^{\mathcal{V}}_{F(\Theta)} c_s : S}$$

The first three rules are straight-forward. The third says that two values are related at $\tau' \multimap \tau$, if, when we apply them to values related at the intepretation of τ', the applications are related at τ. Since the applications are expressions and not values, we use the lifting of the type interpretation.[13] By $\triangleright\Theta$ we mean the type interpretation which applies a later after applying Θ to its arguments. Note that each occurence of $F(\Theta)$ in a premise is at a smaller type, so $F(\Theta)$ is well-defined. Second, the only occurence of Θ is in the rule for session types, where it occurs guarded by a later modality. Thus, the fixed point of F exists – call it Ω. Then, our logical relation at values is this $- \simeq^{\mathcal{V}}_{\Omega} - : \tau$ (which in the text we wrote without Ω annotation)

The fact that Ω is a fixed point of F means that

$$v \simeq^{\mathcal{V}}_{\Omega} V : \tau \mathrel{\dashv\vdash} v \simeq^{\mathcal{V}}_{F(\Omega)} V : \tau$$

Note that $v \simeq^{\mathcal{V}}_{\Omega} V : \tau$ is affine for all v, V, and τ, and so is the lifting of Ω to expressions.

Now we can explain the additional premise in the rule for receive. By default when we have $l \simeq^{\mathcal{V}}_{\Omega} c_s : {?}\tau.\, S$, this would unfold to $\mathsf{Session}_{\triangleright\Omega}(l, c_s, \mathsf{recv}(\tau)S)$ – looking at the definition of state interpretation, we might be afraid that the returned message values v and V, might merely be related at $\triangleright$ of $\Omega\,(\tau)$ in the postcondition, not at $\Omega\,(\tau)$. The key is that the premise above the line lets us "strip" off such a later: take P in that premise to be $- \simeq^{\mathcal{V}}_{\Omega} - : \tau$. Then, the premise of the rule holds when Θ is $\triangleright\, \Omega$, because $\triangleright(v \simeq^{\mathcal{V}}_{\Omega} V : \tau) \vdash \triangleright(v \simeq^{\mathcal{V}}_{\Omega} V : \tau)$, and so in the post condition we get out $v \simeq^{\mathcal{V}}_{\Omega} V : \tau$ without the later. This

---

[13] A lifting of an interpretation of types is defined as the point-wise lifting at each type.

justifies the rule we presented in the paper (and which we use below in the proof of the fundamental lemma).

We now lift this relation to one on open expressions in a given context. Given a map γ_h from a finite domain of variables to closed expressions in the target language, we write $[\gamma_h]e$ for the result of simultaneously substituting each variable in the domain of γ_h for its image under γ_h. $[\gamma_c]E$ is the analogous thing for source expressions.

We define the relation $\Gamma \vdash e \simeq E : \tau$, where Γ is a typing context, by:

$$\Gamma \vdash e \simeq E : \tau \triangleq \forall \gamma_h, \gamma_c. \, (\mathrm{dom}(\gamma_h) = \mathrm{dom}(\gamma_c) = \Gamma \wedge \mathsf{fv}(E) = \mathsf{fv}(e) \subseteq \mathrm{dom}(\Gamma)) \rightarrow$$

$$\left(\underset{x \in \Gamma}{\mathlarger{\ast}} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x) \right) \vdash [\gamma_h]e \simeq^{\mathcal{E}}_{\Omega} [\gamma_c]E : \tau$$

where we take $\mathlarger{\ast}$ over an empty set to be $\mathcal{A}(\mathsf{True})$. We now show that the logical relation is *sound*:

Lemma 41. *If $\emptyset \vdash e \simeq E : \tau$, then $e \sqsubseteq E$.*

Proof. By induction on τ:

- $\tau = \mathsf{Int}$: By assumption, we have that $\mathsf{Emp} \vdash e \simeq^{\mathcal{E}}_{\Omega} E : \mathsf{Int}$.
  By the refinement rule, it suffices to prove:

  $$\mathsf{Emp} \vdash \{\mathsf{source}(i, E)\} \, e \, \{x. \, \exists V. \, \mathsf{source}(i, V) * x \approx V\}$$

  Since Emp is affine and persistent, we can clear the implicit modalities in the Hoare triple, and move the pre-condition to the context by using the wand intro rule, so that we need to show:

  $$\mathsf{source}(i, E) \vdash \mathsf{wp} \, e \, \{x. \, \exists V. \, \mathsf{source}(i, V) * x \approx V\}$$

  By our assumption, we can rewrite this as:

  $$\mathsf{source}(i, E) * e \simeq^{\mathcal{E}}_{\Omega} E : \mathsf{Int} \vdash \mathsf{wp} \, e \, \{x. \, \exists V. \, \mathsf{source}(i, V) * x \approx V\}$$

  Unfolding the definition of the lifting of Ω, instantiating it with i and $[]$, clearing the afine modality, and then eliminating the resulting wand with $\mathsf{source}(i, e)$, we need

  $$\mathsf{wp} \, e \, \{x. \, \exists V. \, \mathsf{source}(i, V) * \exists n. \, x = V = n\} \vdash \mathsf{wp} \, e \, \{x. \, \exists V. \, \mathsf{source}(i, V) * x \approx V\}$$

  By the rule of consequence of weakest precondition (*i.e.*, , weakest precondition is covariant in the post condition), it suffices to show for aribtrary x,

  $$\exists V. \, \mathsf{source}(i, V) * \exists n. \, x = V = n \vdash \exists V. \, \mathsf{source}(i, V) * x \approx V$$

  Which follows from the definition of $\approx$.

The other cases are the same: in each case, we take the assumption, apply the refinement proof rule; then the ownership we get in the pre-condition of the refinement triple is the antecedent of the wand in the definition of the lifting of the type relation to expressions; eliminating that gives a weakest precondition, where the post condition implies that values are logically related; our logical relation implies $\approx$, so we're done.

We now prove what is called the fundamental theorem:

Lemma 42. *If $\Gamma \vdash E : \tau$ then $\Gamma \vdash \widehat{E} \simeq E : \tau$*

Proof. The proof is by induction on the derivation of $\Gamma \vdash E : \tau$.

– Case Var: $\widehat{x} = x$, so given closing substitutions γ_h and γ_c, we need to show

$$\left(\underset{x \in \Gamma}{\text{\Large$*$}} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x) \right) \vdash [\gamma_h]x \simeq^{\mathcal{E}}_{\Omega} [\gamma_c]x : \tau$$

Then $[\gamma_h]x = \gamma_h(x)$ and similarly for γ_c, so this becomes

$$\left(\underset{x \in \Gamma}{\text{\Large$*$}} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x) \right) \vdash \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \tau$$

Note that $\Gamma(x) = \tau$, so, manipulating $*$, this just becomes

$$\gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \tau * \left(\underset{x \in \Gamma \setminus \{x\}}{\text{\Large$*$}} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x) \right) \vdash \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \tau$$

Since the interpretation is affine, we can throw away $\left(\underset{x \in \Gamma \setminus \{x\}}{\text{\Large$*$}} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x) \right)$, and we are done.

– Case Int: Again, $\widehat{n} = n$, and these are closed, so we just have to show, given substitutions:

$$\left(\underset{x \in \Gamma}{\text{\Large$*$}} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x) \right) \vdash n \simeq^{\mathcal{E}}_{\Omega} n : \tau$$

We can throw away the context, since it is affine and not needed. Then, our goal follows immediately from the fact that $n \simeq^{\mathcal{V}}_{\Omega} n : \mathsf{Int}$, and we have already said that related values are related under the lifting, since Ω is affine.

– Case Fun-Intro: Once more, $\widehat{(\lambda x.\,E)} = \lambda x.\,\widehat{E}$. By our induction hypothesis, we have that $\Gamma, x : \tau_1 \vdash \widehat{E} \simeq E : \tau_2$. Given γ_h and γ_c, we know $[\gamma_c]\lambda x.\,E = \lambda x.\,[\gamma_c]E$ and $[\gamma_h]\lambda x.\,\widehat{E} = \lambda x.\,[\gamma_h]\widehat{E}$. We can push the substitutions under the binders because x is explicitly not in Γ hence not in the domain of the substitutions, and capture is not possible because they are substituting closed expressions. Then it suffices to show

$$\left(\underset{x \in \Gamma}{\bigstar} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x)\right) \vdash \lambda x.\, [\gamma_h]\widehat{E} \simeq^{\mathcal{V}}_{\Omega} \lambda x.\, [\gamma_c]E : \tau_1 \multimap \tau_2$$

Unfolding the definition on the right, we are given arbitrary v', V' which are related at τ_1, which we put in the context using the rule for for all introduction, affine introduction, and wand introduction; we then need to show:

$$v' \simeq^{\mathcal{V}}_{\Omega} V' : \tau_1 * \left(\underset{x \in \Gamma}{\bigstar} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x)\right)$$
$$\vdash (\lambda x.\, [\gamma_h]\widehat{E})\; v' \simeq^{\mathcal{E}}_{\Omega} (\lambda x.\, [\gamma_c]E)\; V' : \tau_1 \multimap \tau_2$$

After yet more unfolding, we need to show

$$\mathsf{source}(i, K[(\lambda x.\, [\gamma_c]E)\; V']) * v' \simeq^{\mathcal{V}}_{\Omega} V' : \tau_1 * \left(\underset{x \in \Gamma}{\bigstar} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x)\right)$$
$$\vdash \mathsf{wp}\,(\lambda x.\, [\gamma_h]\widehat{E})\; v'\, \{v''.\, \exists V''.\, \mathsf{source}(i, K[V'']) * v'' \simeq^{\mathcal{V}}_{\Omega} V'' : \tau_2\}$$

We step the source and target to do the beta reductions:

$$\mathsf{source}(i, K[[V'/x][\gamma_c]E]]) * v' \simeq^{\mathcal{V}}_{\Omega} V' : \tau_1 * \left(\underset{x \in \Gamma}{\bigstar} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : \Gamma(x)\right)$$
$$\vdash \mathsf{wp}\,[V'/x][\gamma_h]\widehat{E}\, \{v''.\, \exists V''.\, \mathsf{source}(i, K[V'']) * v'' \simeq^{\mathcal{V}}_{\Omega} V'' : \tau_2\}$$

We can then extend the substititons γ_h and γ_c to also map x to v' and V' respectively; call these γ'_h and γ'_c. Then, rewriting the above and regrouping our assertions to the left of the turnstile, we get:

$$\mathsf{source}(i, K[[\gamma'_c]E]) * \left(\underset{y \in \Gamma, x:\tau_1}{\bigstar} \gamma'_h(y) \simeq^{\mathcal{E}}_{\Omega} \gamma'_c(y) : \Gamma(y)\right)$$
$$\vdash \mathsf{wp}\,[\gamma'_h]\widehat{E}\, \{v''.\, \exists V''.\, \mathsf{source}(i, K[V'']) * v'' \simeq^{\mathcal{V}}_{\Omega} V'' : \tau_2\}$$

But now we can apply our induction hypothesis (suitably unrolling and instantiating everything).

– Case Fun-Elim: Our induction hypothesis says: $\Gamma \vdash \widehat{E} \simeq E : \tau_1 \multimap \tau_2$ and $\Gamma \vdash \widehat{E'} \simeq E' : \tau_1$ and we need to show, given closing substitutions γ_h and γ_c:

$$\left(\underset{x \in \Gamma \uplus \Gamma'}{\bigstar} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : (\Gamma \uplus \Gamma')(x)\right) \vdash [\gamma_h]\widehat{E}\, [\gamma_h]\widehat{E'} \simeq^{\mathcal{E}}_{\Omega} [\gamma_c]E\, [\gamma_c]E' : \tau_2$$

But we note now that E and E' (and their translations) must only have free variables within Γ and Γ' respectively. Let χ_h and χ'_h be the restrictions of γ_h to Γ and Γ' respectively; similarly for χ_c and χ'_c. Then $[\gamma_c]E = [\chi_c]E$ and $[\gamma_h]\widehat{E} = [\chi_c]\widehat{E}$ and similarly for the primed versions. Then substituting, and splitting our $\bigast$, the above becomes:

$$\left(\bigast_{x \in \Gamma} \chi_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi_c(x) : \Gamma(x) \right) * \left(\bigast_{x \in \Gamma'} \chi'_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi'_c(x) : \Gamma'(x) \right)$$
$$\vdash [\chi_h]\widehat{E}\,[\chi_h]\widehat{E'} \simeq^{\mathcal{E}}_{\Omega} [\chi_c]E\,[\chi_c]E' : \tau_2$$

Unfolding this becomes:

$$\mathsf{source}(i, K[[\chi_c]E\,[\chi'_c]E']) * \left(\bigast_{x \in \Gamma} \chi_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi_c(x) : \Gamma(x) \right) * \left(\bigast_{x \in \Gamma'} \chi'_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi'_c(x) : \Gamma'(x) \right)$$
$$\vdash \mathsf{wp}\,[\chi_h]\widehat{E}\,[\chi'_h]\widehat{E'}\,\{v''.\,\exists V''.\,\mathsf{source}(i, K[V'']) * v'' \simeq^{\mathcal{V}}_{\Omega} V'' : \tau_2\}$$

Using the bind rule to focus on the left expression of each computation, we use the induction hypothesis for E to get that there exists v, V, for which it suffices to show that:

$$\mathsf{source}(i, K[V\,[\chi'_c]E']) * v \simeq^{\mathcal{V}}_{\Omega} V : \tau_1 \multimap \tau_2 * \left(\bigast_{x \in \Gamma'} \chi'_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi'_c(x) : \Gamma'(x) \right)$$
$$\vdash \mathsf{wp}\,v\,[\chi'_h]\widehat{E'}\,\{v''.\,\exists V''.\,\mathsf{source}(i, K[V'']) * v'' \simeq^{\mathcal{V}}_{\Omega} V'' : \tau_2\}$$

Doing the same thing for the other induction hypothesis we get that there exists v', V' for which we need to show that:

$$\mathsf{source}(i, K[V\,V']) * v \simeq^{\mathcal{V}}_{\Omega} V : \tau_1 \multimap \tau_2 * v' \simeq^{\mathcal{V}}_{\Omega} V' : \tau_1$$
$$\vdash \mathsf{wp}\,v\,v'\,\{v''.\,\exists V''.\,\mathsf{source}(i, K[V'']) * v'' \simeq^{\mathcal{V}}_{\Omega} V'' : \tau_2\}$$

But this follows from the definition of $v \simeq^{\mathcal{V}}_{\Omega} V : \tau_1 \multimap \tau_2$.

– Case PAIR-INTRO: We have $\Gamma_1 \vdash \widehat{E_1} \simeq E_1 : \tau_1$ and $\Gamma_2 \vdash \widehat{E_2} \simeq E_2 : \tau_2$, and must show, given γ_h and γ_c that:

$$\left(\bigast_{x \in \Gamma_1 \uplus \Gamma_2} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : (\Gamma_1 \uplus \Gamma_2)(x) \right) \vdash ([\gamma_h]\widehat{E_1}\,[\gamma_h]\widehat{E_2}) \simeq^{\mathcal{E}}_{\Omega} ([\gamma_c]E_1, [\gamma_c]E_2) : \tau_1 \otimes \tau_2$$

As in the previous case, we can restrict the substitions to Γ_1 and Γ_2; for γ_h, call those χ_h and χ'_h respectively and similarly for γ_c, and since E_i has free variables only appearing in Γ_i, we can rewrite the above to be:

$$\left(\bigast_{x \in \Gamma_1} \chi_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi_c(x) : \Gamma_1(x) \right) * \left(\bigast_{x \in \Gamma_2} \chi'_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi'_c(x) : \Gamma_2(x) \right)$$
$$\vdash ([\chi_h]\widehat{E_1}, [\chi'_h]\widehat{E_2}) \simeq^{\mathcal{E}}_{\Omega} ([\chi_c]E_1, [\chi'_c]E_2) : \tau_1 \otimes \tau_2$$

Now, we unfold the lifting on the right and introduce the corresponding source ownership assertion:

$$\mathsf{source}(i, K[([\chi_c]E_1, [\chi'_c]E_2)]) * \left(\underset{x \in \Gamma_1}{\bigstar} \chi_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi_c(x) : \Gamma_1(x) \right) * \left(\underset{x \in \Gamma_2}{\bigstar} \chi'_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi'_c(x) : \Gamma_2(x) \right)$$

$$\vdash \mathsf{wp}\,([\chi_h]\widehat{E_1}, [\chi'_h]\widehat{E_2})\,\{(v_1, v_2).\, \exists V_1, V_2.\, \mathsf{source}(i, K[(V_1, V_2)]) * (v_1, v_2) \simeq^{\mathcal{V}}_{\Omega} (V_1, V_2) : \tau_1 \otimes \tau_2\}$$

As in the previous case, we use the bind rule to focus on the left and then right subexpressions; applying our induction hypothesis about E_1 and E_2 then lets us trade our context assumptions to end up with some v_1, V_1, v_2, V_2 for which it suffices to show:

$$\mathsf{source}(i, K[(V_1, V_2)]) * v_1 \simeq^{\mathcal{V}}_{\Omega} V_1 : \tau_1 * v_2 \simeq^{\mathcal{V}}_{\Omega} V_2 : \tau_2$$

$$\vdash \mathsf{source}(i, K[(V_1, V_2)]) * (v_1, v_2) \simeq^{\mathcal{V}}_{\Omega} (V_1, V_2) : \tau_1 \otimes \tau_2$$

But this follows immediately from the unfolding of Ω at pair type.

– Case PAIR-ELIM: The argument is similar to Fun-Intro, in the way that PAIR-INTRO corresponds to FUN-ELIM, so we omit it.

– Case FORK: Surprisingly, this too is similar to the cases for pair and function! The reader might have expected that this would in fact be a hard case; but it is not *precisely* because our lifting from value relation to expression relation is usiing the weakest precondition, so we'll be able to use the fork rule. To wit, we have by induction hypothesis that $\Gamma_1 \vdash \widehat{E_{\mathsf{f}}} \simeq E_{\mathsf{f}} : \tau'$ and $\Gamma_2 \vdash \widehat{E} \simeq E : \tau$. We must show:

$$\left(\underset{x \in \Gamma_1 \uplus \Gamma_2}{\bigstar} \gamma_h(x) \simeq^{\mathcal{E}}_{\Omega} \gamma_c(x) : (\Gamma_1 \uplus \Gamma_2)(x) \right) \vdash \mathsf{fork}\{[\gamma_h]\widehat{E_{\mathsf{f}}}\}; \widehat{[\gamma_h]E} \simeq^{\mathcal{E}}_{\Omega} \mathsf{fork}\{[\gamma_c]E_{\mathsf{f}}\}; [\gamma_c]E : \tau$$

Doing the same routine of restricting the substitutions, splitting the context assumptions, unfolding the lifting relation, etc. we get that we need to show

$$\mathsf{source}(i, K[\mathsf{fork}\{[\chi_c]E_{\mathsf{f}}\}; [\chi'_c]E]) * \left(\underset{x \in \Gamma_1}{\bigstar} \chi_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi_c(x) : \Gamma_1(x) \right) * \left(\underset{x \in \Gamma_2}{\bigstar} \chi'_h(x) \simeq^{\mathcal{E}}_{\Omega} \chi'_c(x) : \Gamma_2(x) \right)$$

$$\vdash \mathsf{wp}\,(\mathsf{fork}\{[\chi_h]\widehat{E_f}\}; [\chi'_h]\widehat{E})\,\{v.\, \exists V.\, \mathsf{source}(i, K[V]) * v \simeq^{\mathcal{V}}_{\Omega} V : \tau\}$$

We start by applying the fork rule; we then do a step shift in the source to get $\mathsf{source}(j, [\chi_c]E_f)$ for some new j, and the parent thread becomes $\mathsf{source}(i, E)$. We now split the context, passing the assumptions about Γ_1 the j source thread to a proof of weakest precondition for the child target thread; the rest is used in the target parent thread's proof. So we need to prove the following two entailments:

$$\mathsf{source}(j, [\chi_c]E_\mathsf{f}) * \left(\underset{x \in \Gamma_1}{\text{\Large $*$}} \chi_h(x) \simeq_\Omega^{\mathcal{E}} \chi_c(x) : \Gamma_1(x) \right)$$

$$\vdash \mathsf{wp}\, [\chi_h] \widehat{E_\mathsf{f}} \, \{v.\, \mathsf{Stopped}\}$$

$$\mathsf{source}(i, K[[\chi_c']E]) \left(\underset{x \in \Gamma_2}{\text{\Large $*$}} \chi_h'(x) \simeq_\Omega^{\mathcal{E}} \chi_c'(x) : \Gamma_2(x) \right)$$

$$\vdash \mathsf{wp}\, [\chi_h'] \widehat{E} \, \{v.\, \exists V.\, \mathsf{source}(i, K[V]) * v \simeq_\Omega^{\mathcal{V}} V : \tau\}$$

The latter follows from the induction hypothesis. For the former, using the induction hypothesis and the rule of consequence for weakest precondition means we just need to show, for all v, V:

$$\mathsf{source}(j, V) * v \simeq_\Omega^{\mathcal{V}} V : \tau' \vdash \mathsf{Stopped}$$

But the right side of the conjunction is affine, hence can be thrown away, and we have mentioned above that a source of a value entails $\mathsf{Stopped}$.

– Case NEWCHTYP: At last we come to a case involving a channel. This expression is closed, it suffices to show:

$$\mathsf{source}(i, K[\mathsf{newch}]) * \left(\underset{x \in \Gamma}{\text{\Large $*$}} \chi_h(x) \simeq_\Omega^{\mathcal{E}} \chi_c(x) : \Gamma_1(x) \right)$$

$$\vdash \mathsf{wp}\, \mathsf{heapNewch}\, \{v.\, \exists V.\, \mathsf{source}(i, K[V]) * v \simeq_\Omega^{\mathcal{V}} V : S \otimes \overline{S}\}$$

Throwing away the Γ piece of the context and rewriting our proof rule for newch on the remaining $\mathsf{source}(i, K[\mathsf{newch}])$ yields:

$$\mathsf{wp}\, \mathsf{heapNewch}\, \{(l,l).\exists c.\, \mathsf{source}(i, K[(c_\mathsf{left}, c_\mathsf{right})]) * \mathsf{Session}_{\triangleright\Omega}(l, c_\mathsf{left}, S) * \mathsf{Session}_{\triangleright\Omega}(l, c_\mathsf{right}, \overline{S})\}$$

$$\vdash \mathsf{wp}\, \mathsf{heapNewch}\, \{v.\, \exists V.\, \mathsf{source}(i, K[V]) * v \simeq_\Omega^{\mathcal{V}} V : S \otimes \overline{S}\}$$

Applying the rule of consequence, we merely need to show, for all l and c:

$$\mathsf{source}(i, K[(c_\mathsf{left}, c_\mathsf{left})]) * \mathsf{Session}_{\triangleright\Omega}(l, c_\mathsf{left}, S) * \mathsf{Session}_{\triangleright\Omega}(l, c_\mathsf{right}, \overline{S})$$

$$\vdash \mathsf{source}(i, K[(c_\mathsf{left}, c_\mathsf{right})]) * (l,l) \simeq_\Omega^{\mathcal{V}} (c_\mathsf{left}, c_\mathsf{right}) : S \otimes \overline{S}$$

Unfolding the definition of value relation at pair type and session type, the left side matches the right side so we are done.

– Case SEND: Our induction hypothesis gives us that $\Gamma \vdash \widehat{E_1} \simeq E_1 : !\tau.\, S$ and $\Gamma \vdash \widehat{E_2} \simeq E_2 : \tau$ and we need to show, given closing substitutions γ_h and γ_c:

$$\left(\underset{x \in \Gamma_1 \uplus \Gamma_2}{\text{\Large $*$}} \gamma_h(x) \simeq_\Omega^{\mathcal{E}} \gamma_c(x) : (\Gamma_1 \uplus \Gamma_2)(x) \right) \vdash \mathsf{heapSend}\, [\gamma_h]\widehat{E_1}\, [\gamma_h]\widehat{E_2} \simeq_\Omega^{\mathcal{E}} \mathsf{send}([\gamma_c]E_1, [\gamma_c]E_2) : S$$

Doing the usual routine of noting that the two subexpressions only have free variables in their respective contexts, taking restrictions, and unfolding everything:

$$\mathsf{source}(i, K[\mathsf{send}([\chi_c]E_1, [\chi_c']E_2])) * \left(\underset{x \in \Gamma_1}{\text{\Large $*$}} \chi_h(x) \simeq_\Omega^\mathcal{E} \chi_c(x) : \Gamma_1(x) \right) * \left(\underset{x \in \Gamma_2}{\text{\Large $*$}} \chi_h'(x) \simeq_\Omega^\mathcal{E} \chi_c'(x) : \Gamma_2(x) \right)$$
$$\vdash \mathsf{wp} \ \mathsf{heapSend} \ [\chi_h]\widehat{E_1} \ [\gamma_h]\widehat{E_2} \ \{v. \exists V. \mathsf{source}(i, K[V]) * v \simeq_\Omega^\mathcal{V} V : S\}$$

Notice that because the implementation of heapSend let-binds a pair containing the two "arguments", they are both evaluated first before substitution into the "body" of the send primitive. Hence, we first evaluate the left and then right subexpressions using our induction hypothesis, and then just have to show for all values v_1, v_2, V_1, V_2:

$$\mathsf{source}(i, K[\mathsf{send}(V_1, V_2)]) * v_1 \simeq_\Omega^\mathcal{V} V_1 : \ !\tau. \ S * v_2 \simeq_\Omega^\mathcal{E} V_2 : \tau$$
$$\vdash \mathsf{wp} \ \mathsf{heapSend} \ v_1 \ v_2 \ \{v. \exists V. \mathsf{source}(i, K[V]) * v \simeq_\Omega^\mathcal{V} V : S\}$$

But then, the relation of v_1 and V_1 unfolds to give us:

$$\mathsf{source}(i, K[\mathsf{send}(V_1, V_2)]) * \mathsf{Session}_{\triangleright\Omega}(v_1, V_1, !\tau. \ S) * v_2 \simeq_\Omega^\mathcal{E} V_2 : \tau$$
$$\vdash \mathsf{wp} \ \mathsf{heapSend} \ v_1 \ v_2 \ \{v. \exists V. \mathsf{source}(i, K[V]) * v \simeq_\Omega^\mathcal{V} V : S\}$$

So we can use the proof rule for send primitive on the left to get

$$\mathsf{wp} \ \mathsf{heapSend} \ v_1 v_2 \ \{l'. \mathsf{source}(i, K[c_s]) * \mathsf{Session}_{\triangleright\Omega}(v_1, V_1, S)\}$$
$$\vdash \mathsf{wp} \ \mathsf{heapSend} \ v_1 \ v_2 \ \{v. \exists V. \mathsf{source}(i, K[V]) * v \simeq_\Omega^\mathcal{V} V : S\}$$

Applying the rule of consequence, and unfolding the interpretation at session type, we are done.
– Case RECV: Our induction hypothesis gives us that $\Gamma \vdash \widehat{E} \simeq E : ?\tau. \ S$ and we need to show, given closing substitutions γ_h and γ_c:

$$\left(\underset{x \in \Gamma}{\text{\Large $*$}} \gamma_h(x) \simeq_\Omega^\mathcal{E} \gamma_c(x) : (\Gamma)(x) \right) \vdash \mathsf{heapRecv} \ [\gamma_h]\widehat{E} \simeq_\Omega^\mathcal{E} \mathsf{recv}([\gamma_c]E) : \tau \otimes S$$

Unfolding we have:

$$\mathsf{source}(i, K[\mathsf{recv}([\gamma_c]E)]) * \left(\underset{x \in \Gamma}{\text{\Large $*$}} \gamma_h(x) \simeq_\Omega^\mathcal{E} \gamma_c(x) : \Gamma(x) \right)$$
$$\vdash \mathsf{wp} \ \mathsf{heapRecv} \ [\gamma_h]\widehat{E} \ \{v. \exists V. \mathsf{source}(i, K[V]) * v \simeq_\Omega^\mathcal{V} V : \tau \otimes S\}$$

We first evaluate the subexpression using our induction hypothesis, and then just to show for all values v, v:

$$\mathsf{source}(i, K[\mathsf{recv}(V)]) * v \simeq^{\mathcal{V}}_{\Omega} V : ?\tau.\, S$$
$$\vdash \mathsf{wp}\ \mathsf{heapRecv}\ v\ \{v'.\, \exists V'.\, \mathsf{source}(i, K[V]) * v \simeq^{\mathcal{V}}_{\Omega} V : \tau \otimes S\}$$

Again, the relation of v and V unfolds to give us:

$$\mathsf{source}(i, K[\mathsf{recv}(V)]) * \mathsf{Session}_{\rhd\Omega}(v, V, ?\tau.\, S)$$
$$\vdash \mathsf{wp}\ \mathsf{heapRecv}\ v\ \{v'.\, \exists V'.\, \mathsf{source}(i, K[V]) * v \simeq^{\mathcal{V}}_{\Omega} V : \tau \otimes S\}$$

So we can use the proof rule for receive primitive on the left to get

$$\mathsf{wp}\ \mathsf{heapRecv}\ v\ \{(l', v').\, \exists V'.\, \mathsf{source}(i, K[(c_s, V')]) * (v' \simeq^{\mathcal{V}}_{\Theta} V' : \tau) * \mathsf{Session}_{\rhd\Omega}(l', V', S)\}$$
$$\mathsf{wp}\ \mathsf{heapRecv}\ v\ \{v'.\, \exists V'.\, \mathsf{source}(i, K[V']) * v' \simeq^{\mathcal{V}}_{\Omega} V' : \tau \otimes S\}$$

Applying the rule of consequence, and unfolding the interpretation at tensor and session type, we are done.

This completes the proof of the fundamental lemma. Then, Theorem 1 follows by combining the fundamental lemma and the soundness of the logical relation (Lemma 41): every well typed term is logically related to its translation, and the left side of a logically related pair refines the right side.

B.2   Craig-Landin-Hagersten Lock

In this part, we describe our second case study, which shows that the Craig-Landin-Hagersten queue lock[11, 31] refines a fair ticket lock [32]. Like the first case study, the results mentioned here have all been mechanized in Coq. Our goal here is simply to give a feel for what the result is about; we do not describe the actual proofs or the state transition systems we use. The interested reader should consult the Coq source.

Lock Implementations The code for the two types of locks appears in Figure 5. The $\mathsf{FAI}(-)$ operation is a fetch-and-increment: it takes a location as a parameter and atomically increments that location and returns the previous value. Meanwhile, $\mathsf{swap}(-, -)$ takes a location and a value and atomically loads the previous value of the location and stores the second parameter as the new value.

Each ticket lock consists of two references storing integers: the "owner" counter and the "next" counter, called o and n in the code, so ticketnew just allocates these two references. To acquire the lock, a thread atomically fetches and increments the next counter to get a number, which we call a "ticket" (ticketacq). It then spins on the owner counter waiting until the owner counter matches the thread's ticket number (ticketwait). Once it matches, the thread

CLHnew $\triangleq$
   $\lambda\_.$ let $d = $ ref False in
      $($ref $d,$ ref $d)$

CLHwait $\triangleq$ rec $loop\ me\ prev\ lk$
   let $w = !prev$ in
   if w then
     $loop\ me\ prev\ lk$
   else
     $($fst $lk) := me$

CLHacq $\triangleq \lambda lk.$
   let $me = $ ref True in
   let $prev = $ swap$($snd $lk, me)$ in
   CLHwait $me\ prev\ lk$

CLHrel $\triangleq \lambda lk.$
   $!($fst $lk) := $ False

ticketnew $\triangleq$
   $\lambda\_.\ ($ref $0,$ ref $0)$

ticketwait $\triangleq$ rec $loop\ x\ lk$
   let $o = !($fst $prev)$ in
   if $x = o$ then
     $()$
   else
     $loop\ x\ lk$

ticketacq $\triangleq \lambda lk.$
   let $n = $ FAI$($snd $lk)$ in
   ticketwait $n\ lk$

ticketrel $\triangleq \lambda lk.$
   $($fst $lk) := !($fst $lk) + 1$

Fig. 5. Code for CLH and ticket locks.

enters the critical section. To release the lock, the thread increments the owner counter (ticketrel).

The lock is fair in the sense that once a thread calls ticketacq and completes its fetch-and-increment operation to get its ticket, it is guaranteed to enter the critical section *before* other threads that subsequently call ticketacq. Thus, if every thread which acquires the lock eventually releases it, every thread that tries to acquire the lock will eventually enter the critical section.

However, one drawback to this design is that every thread waiting to acquire the lock spins on the same owner counter location. Depending on the machine, this can cause poor performance for memory related reasons [32]. In contrast, in the CLH lock, every waiting thread spins on a *different* memory location. The rather extensive benchmarking done by David et al. [13] suggests that in some settings the CLH lock performs better than the ticket lock.

Conceptually, we can think of the CLH lock as consisting of a queue of threads, in the order that they tried to acquire the lock. The lock has two fields: a pointer to the head of the queue, and a pointer to the tail of the queue. Each node in the queue consists of a single boolean value. While this value is true, the node that owns that node is either (1) still waiting to enter the critical section, or (2) is in the critical section, but has not yet released the lock. When creating a new CLH lock, we create a dummy node in which this field is set to false, and set the head and tail to point to it (CLHnew). To acquire the lock, a thread allocates a new node and inserts itself at the back of the list by atomically swapping the location of the new node with the current tail (CLHacq) – the value returned by this atomic swap is the node of its predecessor in the list. The thread then spins on this node's boolean value until it is false – this indicates that the predecessor has released the lock, signalling that the thread is now at the head of the queue and may enter the critical section (CLHwait). Before entering the critical section, the thread therefore updates the head pointer to point to its node. To release the lock, the thread looks at the node pointed to by the head field, and sets that node to false (CLHrel).

Observe that, as claimed, each waiting thread spins on a different location: the node of its predecessor in the list.

There is one important difference between our implementation and typical versions. Usually, there is no head pointer, and the API is designed so that the acquire method take two parameters: (1) the lock, and (2) the new node to be added to the list, while the release method takes only the releaser's node. This means conventionally it is not a drop-in replacement for the ticket-lock, because in the ticket lock, both acquire/release take the lock structure as their only parameter. That said, the head pointer may have other uses and is included in some implementations [26].

Refinement Specification Note that the CLH lock is fair in the same sense as the ticket lock: once a thread completes its atomic swap to insert itself into the queue, it must enter the critical section before threads that later call CLHacq.

This is what makes it possible to establish a fair termination preserving refinement.

However, let us note that if we take a closed program e and replace all instances of the ticket lock primitives with CLH primitives to get a program e^*, it is *not* necessarily the case that e^* refines e. First, one can violate the "abstraction" of the lock datatype, like so:

$$\text{let } l = (\text{ref } 0, \text{ref } 0) \text{ in}$$
$$\text{ticketacq } l$$

This program does not trigger a fault, because l happens to represent a valid ticket lock, but if we replace ticketacq with CLHacq it gets stuck. Of course, we could use a type system in which the lock primitives manipulate values of some abstract type "lock", and this would rule out such examples. Then we might hope to prove that the refinement holds when the original program is well-typed.

However, this is not true. Consider the following example:

$$\text{let } l = \text{ticketnew } () \text{ in}$$
$$\text{ticketrel } l$$
$$\text{ticketacq } l$$

This program diverges, because the first call to ticketrel will increment the owner counter to 1, but during the call to ticketacq, the ticket the thread gets will be number 0, so it will loop forever waiting for the owner counter to be 0. In contrast, if we use CLHnew, CLHrel, and CLHacq in the above program, the result terminates! The call to CLHrel will set the dummy node's boolean to False, but it was already False, so this does nothing. Thus, the thread will see the dummy node is False during the call to CLHacq and will not spin. Thus, the CLH version is not a termination-preserving refinement of the source.

Of course, this example uses the lock "incorrectly" because it releases the lock before acquiring it. What's important is that as long as the lock is used "correctly" the refinement should hold. This is captured by the following set of rules that we prove about the CLH primitives:

$$\frac{5 < d \le D \qquad d' \le D}{\begin{aligned}&\mathsf{source}(i, K[\mathsf{ticketnew}\,()], d) * \mathcal{A}(R) \vdash \\ &\quad \mathsf{wp}\ \mathsf{CLHnew}\,()\ \{lk.\exists lks, \gamma.\ \mathsf{isLock}(\gamma, lks, lk, R) * \mathsf{source}(i, K[lks], d')\}\end{aligned}}$$

$$\frac{5 < d \le D \qquad d' \le D - 2}{\begin{aligned}&\mathsf{source}(i, K[\mathsf{ticketacq}\ lks], d) * \mathsf{isLock}(\gamma, lks, lk, R) \vdash \\ &\quad \mathsf{wp}\ \mathsf{CLHacq}\,()\ \{v.v = () * \mathsf{locked}(\gamma, lks, lk) * \mathsf{source}(i, K[()], d') * \mathcal{A}(R)\}\end{aligned}}$$

$$\frac{5 < d \le D \qquad d' \le D - 2}{\begin{aligned}&\mathsf{source}(i, K[\mathsf{ticketrel}\ lks], d) * \mathsf{isLock}(\gamma, lks, lk, R) * \mathsf{locked}(\gamma, lks, lk) * \mathcal{A}(R) \vdash \\ &\quad \mathsf{wp}\ \mathsf{CLHrel}\,lk\ \{v.v = () * \mathsf{source}(i, K[()], d')\}\end{aligned}}$$

where $\mathsf{isLock}(-,-,-,-)$ and $\mathsf{locked}(-,-,-)$ are certain predicates defined in the logic. If one ignores all the parts of the above specification that have to do with our extensions (delay constants, refinement resources, and the affine modality), this looks like a standard specification for a lock primitive in a higher-order concurrent separation logic (indeed, this kind of specification is proved about the ticket lock in the original Iris repository). Such a specification says that for an arbitrary assertion R, if one initially owns R, it can be given up to create a lock protecting R – in exchange, one gets an assertion $\mathsf{isLock}(-,-,-,-)$ which is duplicable. Then, a thread can use this assertion to call acquire, and afterward it back R, and an assertion $\mathsf{locked}(-,-,-)$ which signifies that it is the owner of the lock. Both R and $\mathsf{locked}(-,-,-)$ are needed to then call release. Thus this typical formulation of a lock specification rules out the kind of counter-examples we illustrated above. Re-reading the specification with the parts relevant to our extension, we see that each shows a refinement from the corresponding ticket lock primitive.

Of course, one can use these rules, along with the rest of the logic, to prove that for a given program, if one replaces the ticket primitives with the CLH versions, the result is a refinement. If the reader is familiar with the "usual" specification of lock primitives in higher-order CSL, hopefully the above description is compelling evidence that this refinement specification is "good enough". In the next section, we describe a particular use of these proof rules to show that for a large class of well-typed programs, the desired refinement holds.

Type-directed translation We now describe a simple type-directed translation from programs using the ticket primitives to ones using the CLH primitives. We then construct a logical relation which uses the weakest precondition specification described above to show that the translated programs refine their sources.

The translation is shown in Figure 6. We write $\Gamma \vdash e \rightsquigarrow e' : \tau$ to indicate that in context Γ, the expression e translates to e' at type τ. The type system used in the translation is a standard simple type system for MiniML

with references, extended with an abstract type Lock for locks. In contrast to the session-typed language, the type system here is fully structral, not affine. The primitive ticketnew translates to CLHnew and returns a value of type Lock (Lock-Intro). This lock can be used with the command ticketsync, which takes a ticket lock and a function of type Unit $\to \tau$ and synchronizes execution of the function with the lock (Lock-Elim). This translates to CLHsync, which does the same thing but using the CLH lock implementation.

Because these "synchronize" commands properly acquire and then release the lock, a the type system rules out the kinds of bad programs we mentioned above. Of course, this is a very restricted way of using the lock. Our point is just to show that the weakest precondition specification supports this kind of use; it in fact is more flexible, but a type system that demonstrated this would be more complex (at which point, one might want to just appeal to the program logic directly to prove the refinement).

As usual, to state our refinement result, we need to give a notion of observational equivalence on values. Here, we restrict our attention to booleans and say that given two booleans b and b', $b \approx b'$ if and only if $b = b'$. With this notion of value equivalence as our foundation for refinement, we have mechanized the following result:

Theorem 43. *If* $\cdot \vdash e \rightsquigarrow e' : \textit{Bool then } e' \sqsubseteq e$.

To prove this theorem, we once again construct a logical relation.[14] This is very much like the relation given for the session typed language, with two notable exceptions (1) the type system is not affine, so all the interpretations of the types in the relation are both *affine* and *relevant* propositions (meaning they can be duplicated and thrown away, freely), (2) we need to model reference and lock types. For references and locks we have:

$$\frac{\mathsf{isLock}(\gamma, lk, lk', \mathsf{True})}{lk \simeq^{\mathcal{V}} lk' : \mathsf{Lock}} \qquad \frac{\boxed{\exists v, v'. \, l \hookrightarrow v * l' \hookrightarrow_{\mathsf{s}} v' * v \simeq^{\mathcal{V}} v' : \tau}}{l \simeq^{\mathcal{V}} l' : \mathsf{Ref}\,\tau}$$

where the boxed assertion in the premise of the second rule is an Iris *invariant*. These were not described in the main text, so the reader unfamiliar with the original work on Iris can think of them as a single-state STS[15]. The first rule means that lock types are just interpreted as the isLock$(-, -, -, -)$ assertion we saw before. The fourth parameter is True because for purposes of the type translation, we don't care what resource the programmer is intending to protect with the lock. The second says that two heap locations are related at reference

---

[14] The logical relation in our Coq formalization is actually formulated generically with respect to a pair of lock implementations, assumed to satisfy weakest precondition specifications like the ones given above for the ticket and CLH lock. The mechanization then instantiates this generic formulation with the particular results about the ticket and CLH locks.

[15] In fact, the STS construct is *derived* from invariants.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \rightsquigarrow x : \tau} \text{ Var} \qquad \text{Int} \quad \Gamma \vdash n \rightsquigarrow n : \mathsf{Int} \qquad \text{Unit} \quad \Gamma \vdash () \rightsquigarrow () : \mathsf{Unit} \qquad \text{Bool} \quad \Gamma \vdash b \rightsquigarrow b : \mathsf{Bool}$$

$$\text{Pair-Intro} \quad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash (e_1, e_2) \rightsquigarrow (e_1', e_2') : \tau_1 \times \tau_2} \qquad \text{Pair-Fst} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ e \rightsquigarrow \mathsf{fst}\ e' : \tau_1}$$

$$\text{Pair-Snd} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ e \rightsquigarrow \mathsf{snd}\ e' : \tau_2} \qquad \text{Sum-Left} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \tau_1}{\Gamma \vdash \mathsf{inl}\ e \rightsquigarrow \mathsf{inl}\ e' : \tau_1 + \tau_2} \qquad \text{Sum-Right} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \tau_2}{\Gamma \vdash \mathsf{inr}\ e \rightsquigarrow \mathsf{inr}\ e' : \tau_1 + \tau_2}$$

$$\text{Sum-Elim} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \tau_1 + \tau_2 \qquad \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \rightarrow \tau \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 \rightarrow \tau}{\Gamma \vdash \mathsf{case}(e, e_1, e_2) \rightsquigarrow \mathsf{case}(e', e_1', e_2') : \tau}$$

$$\text{Seq} \quad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash e_1; e_2 \rightsquigarrow e_1'; e_2' : \tau_2} \qquad \text{Fun-Intro} \quad \frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e \rightsquigarrow e' : \tau_2}{\Gamma \vdash \mathsf{rec}\ f\ x.\ e \rightsquigarrow \mathsf{rec}\ f\ x..\ e' : \tau_1 \rightarrow \tau_2}$$

$$\text{Fun-Elim} \quad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_1}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_2} \qquad \text{Fork} \quad \frac{\Gamma \vdash e_\mathsf{f} \rightsquigarrow e_\mathsf{f}' : \tau}{\Gamma \vdash \mathsf{fork}\{e_\mathsf{f}\} \rightsquigarrow \mathsf{fork}\{e_\mathsf{f}'\} : \mathsf{Unit}}$$

$$\text{Load} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \mathsf{Ref}\ \tau}{\Gamma \vdash !e \rightsquigarrow !e' : \tau} \qquad \text{Store} \quad \frac{\Gamma \vdash e \rightsquigarrow e' : \mathsf{Ref}\ \tau \qquad \Gamma \vdash e_s \rightsquigarrow e_s' : \tau}{\Gamma \vdash e := e_s \rightsquigarrow e' := e_s' : \mathsf{Unit}}$$

$$\text{Send} \quad \frac{\Gamma_1 \vdash E_1 : !\tau.\ S \qquad \Gamma_2 \vdash E_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{send}(E_1, E_2) : S} \qquad \text{Lock-Intro} \quad \Gamma \vdash \mathsf{ticketnew} \rightsquigarrow \mathsf{CLHnew} : \mathsf{Unit} \rightarrow \mathsf{Lock}$$

$$\text{Lock-Elim} \quad \frac{\Gamma \vdash e_l \rightsquigarrow e_l' : \mathsf{Lock} \qquad \Gamma \vdash e \rightsquigarrow e' : \mathsf{Unit} \rightarrow \tau}{\Gamma \vdash \mathsf{ticketsync}\ e_l\ e \rightsquigarrow \mathsf{CLHsync}\ e_l'\ e' : \tau}$$

$$\mathsf{ticketsync}\ e_l\ e \triangleq \lambda lk, f.\ \mathsf{ticketacq}\ lk; \mathsf{let}\ z = e\,()\ \mathsf{in}\ (\mathsf{ticketrel}\ lk; z)$$

$$\mathsf{CLHsync}\ e_l'\ e' \triangleq \lambda lk, f.\ \mathsf{CLHacq}\ lk; \mathsf{let}\ z = e'\,()\ \mathsf{in}\ (\mathsf{CLHrel}\ lk; z)$$

Fig. 6. Type directed translation between programs using ticket lock and CLH locks.

type $\mathsf{Ref}\,\tau$ if there is an invariant enforcing that they must always point to values that are related at the interpretation of τ.

Using these definitions, we can prove a fundamental lemma showing that if $\cdot \vdash e \rightsquigarrow e' : \tau$, then $e \simeq^{\mathcal{E}} e' : \tau$ holds. Then, we show a soundness lemma that states that if $e \simeq^{\mathcal{E}} e' : \mathsf{Bool}$, then $e \sqsubseteq e'$. By composing these two results, we obtain Theorem 43.