Building Extensible Program Logics through Effect Handlers

ZICHEN ZHANG, New York University, USA
SIMON ODDERSHEDE GREGERSEN, New York University, USA
JOSEPH TASSAROTTI, New York University, USA

One strategy for reasoning about programs that have certain kinds of effects is to use program logics that provide specialized rules for reasoning about these effects. However, *developing* program logics requires skills that are distinct from those needed for *using* program logics, making the development of new logics challenging and less accessible. Moreover, when developing new logics, it can be difficult to reuse components from prior logics or combine support for different effects.

In this paper, we propose an approach for building extensible program logics based on effect handlers. Our starting point is an expressive program logic for reasoning about programs written in a pure, sequential language with support for effect handlers. Within this language, we implement handlers that model concurrency, distributed execution, and crash-recovery behavior. Then, by proving properties about these handlers, we extend the program logic and derive expressive rules for reasoning about these effects. In some cases, this approach leads to stronger reasoning rules than those found in prior program logics targeting these features.

In addition, we develop a relational logic for proving contextual refinements between programs using effects. As with unary reasoning, handlers enable this relational logic to be developed in an extensible way.

1 Introduction

Program logics have proven to be a powerful tool for program verification. As a result, a variety of program logics have been developed for challenging program features, including pointers [43], concurrency [27, 38, 39], weak memory [16, 29, 31, 35, 52, 53], distributed execution [30, 44, 57], crash recovery [12–14, 37, 42], and randomness [2, 3, 5–8, 24, 48], among others.

Traditionally, a program logic is developed by proving that a collection of reasoning rules is sound with respect to an operational or denotational semantics for a language. However, following this traditional approach is challenging for several reasons. First, the development of program logics requires skills that are distinct from those needed for *using* program logics, making the development of new logics less accessible. Second, when developing new logics, it can be difficult to reuse components or port a particular feature from a different logic. In reality, many important computer systems *combine* many of the features described in the previous paragraph, yet developing logics that provide support for such combinations of features requires significant work.

Recently, Vistrup et al. [56] proposed an alternative approach to developing program logics that addresses the re-usability problem. Starting from a minimal, pure lambda calculus, they incrementally add effects to this language by giving a denotational semantics in terms of ITrees [59]. On the logic side, one gives rules that logically "interpret" or "handle" the events in the generated ITree. The soundness of the logic is established in a modular way by relating these logical handlers for events in the ITree to an interpretation function that "executes" the events. While this approach addresses the problem of re-usability, it does not address the first accessibility issue: it requires understanding the formalism of ITrees and their denotation, and the adequacy proofs require a form of reasoning that is different from the task of *using* the program logic to reason about programs. Moreover, it is unclear how to use ITrees to account for certain types of program effects or logic features, such as prophecy variables.

This paper advocates for an alternative approach to developing program logics by using *effect handlers* [40, 41] to model all program effects. Effect handlers are a language feature that enable

Authors' Contact Information: Zichen Zhang, New York University, New York, New York, USA, zichenzhang@nyu.edu; Simon Oddershede Gregersen, New York University, New York, New York, USA, s.gregersen@nyu.edu; Joseph Tassarotti, New York University, New York, New York, USA, jt4767@nyu.edu.

programmers to define custom effects in a modular and compositional way. Moreover, recent work has shown how to develop program logics for reasoning about effect handlers [17–20]. For example, using these logics, it is possible to verify an effect handler that implements a mutable state effect and derive a specification that resembles the usual separation logic rules for reasoning about pointers. As a result, reasoning about a client program that uses this state effect handler looks just like doing a standard separation logic proof about a program that uses built-in primitive state. However, prior program logics for effect handlers have considered a setting where effect handlers are added on top of a language that already has various other forms of primitive effects built in, such as mutable references and concurrency. This makes sense for verifying examples that involve a subtle interaction between primitive effects and effect handlers, but it means that the soundness proofs of these logics combine the complexity of primitive effects and effect handlers.

In this work, we instead use effect handlers to bootstrap an expressive program logic for a range of effects. Our starting point is a minimal core effect handler language called FicusLang that has just two primitive effects: non-deterministic choice and recording an action to a trace. For reasoning about programs written in FicusLang, we develop Ficus, a separation logic for effect handlers that is an adaptation of an earlier logic called Hazel [18]. Using FicusLang, we then implement handlers to model mutable state, shared-memory concurrency, distributed execution over unreliable networks, and crash-recovery with durable state. For each effect, we apply Ficus to verify the handler implementations and obtain proof rules that are analogous to the reasoning principles derived in prior specialized program logics for these effects.

The handler-based approach even allows us to derive *stronger* proof rules than prior work. There are two main reasons. First, with the handler-based approach, we can build up effects in a hierarchical way, using earlier effects in the definition of handlers for later effects. Then, when verifying those later effects, we can use the proof rules from the earlier effects. For example, we show in §5 how to derive local *prophecy variables* [1, 28] from a simpler global prophecy by implementing local prophecy variables as an effect handler. Later, when implementing the handlers for crashes and recovery in §6, we attach prophecy variables to non-deterministic choices made during crashes, which allows client proofs to reason about when crashes will occur.

A second source of stronger proof rules arises from using effect handlers to implement non-standard versions of effects that are easier to reason about. For example, our handler for concurrency generates fewer interleavings than a standard operational semantics for concurrency. As a result, the "invariant-opening" rule that we obtain for this handler is stronger than the standard rule from most concurrent separation logics (CSLs). To justify the use of these non-standard semantics, we must prove that they are equivalent in an appropriate sense to the standard semantics. To do so, we develop a new relational logic for effect handlers called Banyan.

Contributions. To summarize, our work makes the following contributions:

- We extend Hazel [18] to develop Ficus, an extensible unary program logic based on effect handlers (§2). Ficus uses *protocols* to decompose reasoning about handlers and client code using handlers, and adds support for an extensible notion of *worlds* to share resources between client code (§3).
- We develop a relational logic, Banyan, for proving contextual refinement in the presence of effect handlers (§4). Banyan adapts Ficus protocols to the relational setting, similarly allowing for reuse and extensibility.
- As case studies, we show how to reconstruct and extend features from existing program logics using our effect handler approach. This includes: (1) A derivation of *local* prophecy variables out of global prophecies, along with an approach to *implicitly* make prophecies without annotating a program with prophecy operations; (2) A logic for crash-recovery

$$
\begin{aligned}
v &::= () \mid \mathbf{rec}\ f\ x.\ e \mid \cdots \mid \mathbf{cont}\ N \\
e &::= v \mid x \mid e\ e \mid \cdots \mid \mathbf{do}\ e \mid \S(N)[v] \mid (\mathbf{try}\ e\ \mathbf{with}\ v\ k \Rightarrow\ e \mid \mathbf{ret}\ v \Rightarrow e) \mid \mathbf{pick} \mid \mathbf{observe}\ e \\
K &::= [\,] \mid e\ K \mid K\ v \mid \cdots \mid \mathbf{do}\ K \mid (\mathbf{try}\ K\ \mathbf{with}\ v\ k \Rightarrow\ e \mid \mathbf{ret}\ v \Rightarrow e) \\
N &::= [\,] \mid e\ N \mid N\ v \mid \cdots \mid \mathbf{do}\ N
\end{aligned}
$$

Fig. 1. Syntax of values v, expressions e, evaluation contexts K, and neutral evaluation contexts N.

reasoning with asynchronous durable storage that recovers the features of Perennial [12], but with a simpler model, and novel support for *crash-aware* prophecy variables; (3) A logic for distributed systems with IronFleet-style [25] atomic blocks.

Our work is mechanized in the Iris separation logic framework [27] and the Rocq Prover. Our formalization is available in the supplementary material.

2 Program Logics by Effect Handlers

This section provides an overview of how to build up a program logic using effect handlers. After introducing FicusLang, we describe the core features of Ficus, and use them to develop proof rules for reasoning about shared-memory concurrency. An inference rule with premises $P_1, \ldots, P_n$ and conclusion Q should be read as a separation logic entailment of the form $P_1 * \cdots * P_n \vdash Q$.

2.1 The FicusLang Calculus

FicusLang is a call-by-value ML-style lambda calculus with effect handlers. The syntax of FicusLang is shown in Figure 1. The expression $\mathbf{do}\ v$ raises an effect with value v, which will later be handled by the closest enclosing effect handler. Expression $\mathbf{try}\ e_0\ \mathbf{with}\ v_1\ k \Rightarrow\ e_1 \mid \mathbf{ret}\ v_2 \Rightarrow e_2$ installs a *shallow* effect handler for e_0: it evaluates e_0 until either e_0 raises an effect or becomes a value. For the first case, the handler will have access to the value v_1 raised by the effect and a continuation k that allows the handler to resume e_0 from the point the effect was raised. For the second case, the handler will obtain the result of e_0, a value v_2. This type of handler is called a shallow handler because it will disappear in both cases, and the interpreter must reinstall the handler if the expression e_0 may raise effects multiple times.[1] Continuations are used by applying them like functions.

In addition to effect handlers, FicusLang has two built-in primitive effects. The first is the **pick** expression, which non-deterministically evaluates to an arbitrary integer. The second is **observe** v, which performs a labeled transition with the label given by the value v. These **observe** statements are used as a form of ghost code and underlie our support for prophecy variables as discussed in §5.

Figure 2 shows an example of a handler implementing a global state effect supporting `read` and `write` operations. The handler uses a state-passing style. The recursive function `go` takes a continuation k, a value r to pass to the continuation, and the current global state σ. It runs the continuation under a handler that expects raised effects to be pairs of the form (η, v), where η is a *tag* indicating whether the operation is a `read` or `write`. Based on the tag, it recursively calls `go` with the appropriate return value and updated state. If the tag does not match `read` or `write`, it re-raises the effect to allow composition with another handler for other effects.

2.2 Core Ficus Logic

To reason about programs written in FicusLang, we make use of Ficus, a separation logic built on top of the Iris framework [27], adapted from the Hazel logic for effect handlers [18]. This section first presents the basic core of Ficus, which is essentially a subset of Hazel. Later sections will describe additional generalizations that go beyond Hazel.

[1] *Deep handlers*, which are re-installed after an effect is raised, can be simulated with shallow handlers and recursion.

$$
\begin{array}{l}
\mathtt{run}_{\mathtt{state}} \triangleq \lambda \mathit{main}\ \mathit{init}.\ \mathtt{go}\ \mathit{main}\ ()\ \mathit{init} \\
\text{where } \mathtt{go} \triangleq \mathbf{rec}\ \mathtt{go}\ k\ r\ \sigma. \\
\qquad \mathbf{try}\ k\ r\ \mathbf{with} \\
\qquad\quad v\ k \Rightarrow \mathbf{match}\ v\ \mathbf{with} \\
\qquad\qquad\qquad (\mathtt{read}, ()) \Rightarrow \mathtt{go}\ k\ \sigma\ \sigma \\
\qquad\qquad\quad \mid (\mathtt{write}, y) \Rightarrow \mathtt{go}\ k\ ()\ y \\
\qquad\qquad\quad \mid\quad (\eta, v) \Rightarrow \mathtt{go}\ k\ (\mathbf{do}\ (\eta, v))\ \sigma \\
\qquad \mid \mathbf{ret}\ v \Rightarrow v
\end{array}
$$

Fig. 2. Handler for a global state effect.

$$
\frac{}{\mathsf{ewp}\ v\ \langle\Psi\rangle\ \{\Phi\}}\ \Phi(v)\ \textsc{Ewp-Value}
\qquad
\frac{}{\mathsf{ewp}\ \mathbf{do}\ v\ \langle\Psi\rangle\ \{\Phi\}}\ \Psi(v,\Phi)\ \textsc{Ewp-Do}
$$

$$
\frac{\Psi \sqsubseteq \Psi' \qquad \forall v.\ \Phi(v) \mathrel{-\!\!*} \Phi'(v) \qquad \mathsf{ewp}\ e\ \langle\Psi\rangle\ \{\Phi\}}{\mathsf{ewp}\ e\ \langle\Psi'\rangle\ \{\Phi'\}}\ \textsc{Ewp-Mono}
$$

$$
\frac{R \qquad \mathsf{ewp}\ e\ \langle\Psi\rangle\ \{\Phi\}}{\mathsf{ewp}\ e\ \langle\Psi\rangle\ \{v.\ R * \Phi(v)\}}\ \textsc{Ewp-Frame}
\qquad
\frac{\mathsf{ewp}\ e\ \langle\Psi\rangle\ \{\Phi\} \qquad e' \rightarrow^* e}{\mathsf{ewp}\ e'\ \langle\Psi\rangle\ \{\Phi\}}\ \textsc{Ewp-Pure}
$$

$$
\frac{\mathsf{ewp}\ e\ \langle\Psi\rangle\ \{v.\ \mathsf{ewp}\ N[v]\ \langle\Psi\rangle\ \{\Phi\}\}}{\mathsf{ewp}\ N[e]\ \langle\Psi\rangle\ \{\Phi\}}\ \textsc{Ewp-Bind}
$$

Fig. 3. Selected reasoning rules about the effect weakest precondition ewp $e\ \langle\Psi\rangle\ \{\Phi\}$.

Ficus uses a weakest precondition assertion of the form ewp $e\ \langle\Psi\rangle\ \{\Phi\}$ for reasoning about programs. In this assertion, e is a program expression, Φ is a postcondition, and Ψ is a *protocol* that describes the specifications for effect handlers that are active as e executes. This assertion says that if e executes in an environment with handlers satisfying Ψ, then evaluating e will not get stuck, and if e terminates with value v, then the assertion $\Phi(v)$ will hold. More concretely, the protocol Ψ is a predicate of type $\mathit{Val} \to (\mathit{Val} \to \mathit{iProp}) \to \mathit{iProp}$, where the first argument is the value raised with an effect, and the second argument is the postcondition at the time the effect was raised. Throughout this paper, we require all protocols to be *monotonic* [17]. A protocol Ψ is monotonic if $(\forall w.\ \Phi(w) \mathrel{-\!\!*} \Phi'(w)) \vdash \Psi(v,\Phi) \mathrel{-\!\!*} \Psi(v,\Phi')$ for all v, Φ, and Φ'. This essentially enforces a one-shot continuation discipline in the logic which simplifies our presentation and suffices for our purposes.

Figure 3 lists a selection of reasoning rules for the ewp assertion. Most of these rules are similar to standard weakest precondition rules in separation logics. The key rule for reasoning about effects is Ewp-Do, which says that to raise value v, it suffices to show that the protocol holds for the value v and the current postcondition Φ.

For example, for the state handler in Figure 2, we use the STATE protocol

$$
\begin{aligned}
\mathsf{READ}^{\gamma}(v,\Phi) &\triangleq \exists x.\ v = (\mathtt{read}, ()) * S^{\gamma}(x) * (S^{\gamma}(x) \mathrel{-\!\!*} \Phi(x)) \\
\mathsf{WRITE}^{\gamma}(v,\Phi) &\triangleq \exists x, y.\ v = (\mathtt{write}, y) * S^{\gamma}(x) * (S^{\gamma}(y) \mathrel{-\!\!*} \Phi(())) \\
\mathsf{STATE}^{\gamma}(v,\Phi) &\triangleq \mathsf{READ}^{\gamma}(v,\Phi) \lor \mathsf{WRITE}^{\gamma}(v,\Phi)
\end{aligned}
$$

where $S^{\gamma}(x)$ is a predicate that uses a piece of *ghost state* with the name γ to assert that the current value of the global state σ is x. The first component of the protocol is READ, which says that when the effect tag is `read`, then the client must show $S^{\gamma}(x)$ for some x, in which case the protocol gives back $S^{\gamma}(x)$ for proving the postcondition Φ instantiated with the value x, indicating that the return value of the effect will be x. The second component is the WRITE protocol, which updates the given S^{γ} predicate from value x to the value y being written and returns back the unit value. Finally, STATE is the disjunction of these two protocols.

By applying Ewp-Do, we obtain the following derived rules for reasoning with this protocol:

Ewp-Read
$$\frac{S^\gamma(x)}{\mathsf{ewp}\ \mathbf{do}\ (\mathtt{read}, ())\ \langle \mathsf{STATE}^\gamma \rangle\ \{v.\, v = x\}}$$

Ewp-Write
$$\frac{S^\gamma(x)}{\mathsf{ewp}\ \mathbf{do}\ (\mathtt{write}, y)\ \langle \mathsf{STATE}^\gamma \rangle\ \{v.\, v = () * S^\gamma(y)\}}$$

Installing Handlers. So far, we have seen how a client can reason about effects when an appropriate protocol is part of the ewp assertion. Protocols are added to the ewp when a handler is installed using the Ewp-Try rule shown below. Using the Ficus approach, we think of the language and logic as being extended to support new effects by adding handlers, so applying this rule forms the core proof obligation of a developer trying to extend the program logic.

Ewp-Try
$$\frac{\mathsf{ewp}\ e\ \langle \Psi \rangle\ \{\Phi\} \qquad \begin{array}{l}(\forall v_2.\, \Phi(v_2) \mathrel{-\!\!*} \mathsf{ewp}\ e_2\ \langle \Psi' \rangle\ \{\Phi'\}) \wedge \\ (\forall v_1, k_1.\, \Psi(v_1, \lambda w.\, \mathsf{ewp}\ k_1\ w\ \langle \Psi \rangle\ \{\Phi\}) \mathrel{-\!\!*} \mathsf{ewp}\ e_1\ \langle \Psi' \rangle\ \{\Phi'\})\end{array}}{\mathsf{ewp}\ \mathbf{try}\ e\ \mathbf{with}\ v_1\ k_1 \Rightarrow\ e_1 \mid \mathbf{ret}\ v_2 \Rightarrow e_2\ \langle \Psi' \rangle\ \{\Phi'\}}$$

Specifically, in the Ewp-Try rule, we start with a protocol Ψ', and end up with a protocol Ψ when reasoning about the expression e that runs with the new handler available. This rule has two premises. The first premise requires proving an ewp about e with the new protocol Ψ. As a result, this premise will be proved by a client who may now reason as if e has access to the new effects.

Meanwhile, the second premise makes up the proof obligation that justifies extending the logic with this new protocol. This premise is a logical conjunction with two parts. The first conjunct is for the case where e evaluates to a value without raising an effect and requires showing that e's postcondition Φ implies an ewp about the remaining expression e_2. The second conjunct is for the case where e raises an effect. Recall that when e raises an effect, from the client's perspective it must establish the protocol Ψ. Conversely, that means that here in this proof rule, we get the protocol Ψ instantiated with the value v_1 raised with the effect, and a predicate that captures a specification for the continuation k_1. From this, we must prove an ewp about the handler code e_1 that will run. These two conjuncts are joined with $\wedge$ instead of $*$ because only one of these two outcomes will occur, so the rule does not require separate resources for each conjunct.

To apply this rule for the state handler from Figure 2, and thereby add the STATE protocol, we first need to more carefully define the $S^\gamma(x)$ assertion. To do so, we use the underlying Iris logic's support for defining ghost state using *resource algebras* [27]. In particular, we define it as the *fragment* copy of a *authoritative* resource algebra: $S^\gamma(x) \triangleq \boxed{\circ x}^\gamma$. Roughly speaking, this resource algebra comes with two types of ghost resources: an authoritative copy $\boxed{\bullet x}^\gamma$ and a fragment copy $\boxed{\circ x}^\gamma$. When these copies are combined together, they are guaranteed to agree, *i.e.*, $\boxed{\bullet x}^\gamma * \boxed{\circ y}^\gamma \vdash x = y$, and they can be updated to an arbitrary value y, with the rule $\boxed{\bullet x}^\gamma * \boxed{\circ x}^\gamma \vdash \Rrightarrow \boxed{\bullet y}^\gamma * \boxed{\circ y}^\gamma$, where $\Rrightarrow$ is the *basic update modality*. The update modality is the primitive for manipulating ghost resources in the Iris logic. The assertion $\Rrightarrow P$ says that we can update our ghost resources and obtain P. The modality can be eliminated at any suitable time during program verification.

The handler $\mathtt{run}_{\mathsf{state}}$ allocates ghost states $\boxed{\bullet init}^\gamma$ and $\boxed{\circ init}^\gamma$ at a fresh ghost location γ, where $init$ is the initial value for the state that is passed in. It keeps its authoritative copy $\boxed{\bullet init}^\gamma$, and passes the fragment copy $\boxed{\circ init}^\gamma$, which is $S^\gamma(init)$, to the client. Whenever the client raises an effect, it must show the value satisfies STATE^γ, which is then passed to the handler code. The handler proof uses the $S^\gamma(x)$ that is included in STATE^γ and combines it with its corresponding authoritative copy of the ghost state to carry out the read or write. Note that the handler recursively calls go, thereby re-installing the handler and running the continuation. To reason about this recursion, we use Löb induction from the underlying Iris logic [27]. Altogether, we obtain the

```
run_conc ≜ λmain. go {|(main, (), M)|}
where go ≜ rec go pool.
             let ((k, r, t), pool) := choose pool in
             try k r with
               v k    ⇒ match v with
                          (fork, e) ⇒ go ({|(e, (), C), (k, (), t)|} ⊎ pool)
                        |     (η, v) ⇒ go ({|(k, do (η, v), t)|} ⊎ pool)
             | ret v ⇒ if t = M^× then v
                       else go ({|((λ_. v), (), t^×)|} ⊎ pool)
```

Fig. 4. A handler for concurrency.

following derived rules for go and run, starting from a base protocol $\bot$ defined by $\bot(v, \Phi) \triangleq \text{False}$.

$$\frac{\boxed{\bullet\sigma}^{\gamma} \qquad \text{ewp } k\ r\ \langle \text{STATE}^{\gamma} \rangle\ \{\Phi\}}{\text{ewp go } k\ r\ \sigma\ \langle \bot \rangle\ \{\Phi\}}\ \textsc{Ewp-StateGo} \qquad \frac{\forall \gamma.\ S^{\gamma}(\mathit{init}) \mathrel{-\!\!*} \text{ewp } \mathit{main}\ ()\ \langle \text{STATE}^{\gamma} \rangle\ \{\Phi\}}{\text{ewp run}_{\text{state}}\ \mathit{main}\ \mathit{init}\ \langle \bot \rangle\ \{\Phi\}}\ \textsc{Ewp-StateRun}$$

Building a Hierarchy of Effects. The previous example showed how to go from no effects (represented by protocol $\bot$) to the state effect (protocol STATE^{γ}). In practice, we want to accumulate effects by nesting additional handlers within the handler for STATE^{γ}. To that end, as in Hazel, we define a combinator $\oplus$ on protocols by $(\Psi_1 \oplus \Psi_2)(v, \Phi) \triangleq \Psi_1(v, \Phi) \vee \Psi_2(v, \Phi)$. In other words, $\Psi_1 \oplus \Psi_2$ represents that a client may choose to use effects from either of Ψ_1 or Ψ_2, or both. For example, $\text{STATE}^{\gamma} = \text{READ}^{\gamma} \oplus \text{WRITE}^{\gamma}$. Applying this operation to protocols results in a "larger" protocol. This is formally captured by a preorder relation on protocols $\Psi_1 \sqsubseteq \Psi_1 \oplus \Psi_2$. Intuitively, $\Psi_1 \oplus \Psi_2$ is larger than Ψ_1 because the former permits the client to raise more kinds of effects. Using the Ewp-Mono rule, we can generalize Ewp-Read and Ewp-Write accordingly: rather than requiring exactly the protocol STATE^{γ}, we require that the protocol is some Ψ such that $\text{STATE}^{\gamma} \sqsubseteq \Psi$.

Similarly, the Ewp-StateGo and Ewp-StateRun specifications for installing the handler do not need to start from the base protocol $\bot$. Instead, they can start from an arbitrary protocol Ψ, and—so long as Ψ does not already handle the tags for `read` and `write`—the client code would then operates with protocol $\Psi \oplus \text{STATE}^{\gamma}$. This enables the state handler to be composed with an arbitrary context of previously installed handlers, allowing a logic developer to mixin rules for state with other effects. A protocol is said to handle a set of tags T if $\Psi(v, \Phi) \vdash \exists t, v.\ v = (t, v) \wedge t \in T$ for all v and Φ. We write $\text{tags}(\Psi)$ for the tags handled by Ψ. For the state protocol we would require $\texttt{read}, \texttt{write} \notin \text{tags}(\Psi)$. As another example of effects, we have implemented a handler for a heap effect with dynamically allocatable higher-order references and the ability to locally read and write from a given reference. This handler uses the STATE protocol to store a global map representing the heap, and provides its own HEAP protocol for clients to use.

In this way, we compositionally build logical support for a collection of effects starting from the $\bot$ protocol, extending the core pure logic with support for these effects. This approach is grounded in an adequacy theorem, which shows that the logic starting with the $\bot$ protocol is sound.

Theorem 2.1 (Adequacy, Core Ficus). *Let φ be a first-order predicate. If $\vdash$ ewp $e\ \langle \bot \rangle\ \{\varphi\}$ is derivable, then executing e will not get stuck, and if $e \to^* v$ then $\varphi(v)$ holds.*

3 Concurrency and Extensible Worlds

In the effects we have seen so far, the handler always immediately returns control back to the client that raised the effect. However, for other kinds of effects, the handler may instead pass control to other client code. To reason about these kinds of handlers, we need to go beyond the core features of Ficus inherited from Hazel. This section describes a new feature in Ficus called *extensible worlds*.

A key example of an effect where this mechanism is needed is preemptive concurrency. Figure 4 shows an implementation of a concurrency handler. It depends on a bag (*a.k.a.* multiset) library for the thread pool *pool*. Each thread in *pool* is represented by a triple of (continuation, result of last effect, thread type), where the thread type can be either a main thread $\mathcal{M}$, a child thread C, or their terminated variants $\mathcal{M}^\times$ and $C^\times$.

To execute one thread, the scheduler non-deterministically chooses one thread from *pool*, let it execute for as many pure steps as it can until it raises an effect or terminates. If the thread raises a `fork` effect, the scheduler will push both the new thread $(e, (), C)$ and the old thread $(k, (), t)$ to the thread pool. If the thread raises another effect, the scheduler will forward the effect to an outer handler by re-raising it, collect its result, and put the old thread back to the thread pool. Finally, if the thread terminates, the scheduler will not immediately terminate the whole system but mark the thread as terminated and put it back into *pool*. The scheduler will only exit when the main thread terminates the second time, allowing other threads to continue executing for some number of steps before the program exits.

Because the handler may pass control to other threads when an effect is raised, we now need to reason about coordination between threads, which is what extensible worlds will enable.

Background: Iris Invariants and Fancy Updates. To motivate extensible worlds, let us first recall how modern concurrent separation logics like Iris handle reasoning about interaction between different threads. By default, CSL allows for local reasoning about different threads in a concurrent system by dividing up state and resources into separate disjoint parts using separating conjunction, with each thread having ownership of some portion of state. However, in some cases, threads need to *share* ownership of state. To do so, CSLs make use of *invariants*. In Iris, an invariant assertion $\boxed{P}^{\mathcal{N}}$ says that P is an invariant that holds between all program steps. The $\mathcal{N}$ annotation is a *name* given to this invariant. These assertions are duplicable, meaning that $\boxed{P}^{\mathcal{N}} \vdash \boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}$, which allows each thread to have a copy of the assertion. When carrying out a proof about a thread, we access the underlying assertion P by "opening" the invariant using the following rule:

$$\textsc{Wp-InvAcc}\quad \frac{\boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \triangleright P \twoheadrightarrow \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{x.\ \triangleright P * \Phi(x)\} \qquad \mathsf{atomic}(e)}{\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

This rule allows us to prove the weakest precondition under the assumption that P holds (under a later modality $\triangleright$ [4, 11, 36], which we will ignore for now), so long as we re-establish P in the postcondition of e. Here, e must be atomic, meaning that it reduces to a value in a single step, so that by re-establishing P in the postcondition, we ensure that P will continue to hold before and after each step. The *mask* parameter $\mathcal{E}$ is a set that tracks invariants that have not yet been opened. The invariants in $\mathcal{E}$ are said to be *closed* or *enabled*, while all other invariants are *open* or *disabled*.

In fact, in Iris, Wp-InvAcc is a derived rule. Iris uses a more primitive mechanism called a *fancy update modality* of the form ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2}$ that encodes the process of opening and closing invariants. Informally, the assertion ${}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} P$ is an assertion stating that starting with all invariants in $\mathcal{E}_1$ being enabled, and then opening/closing invariants so as to end up with $\mathcal{E}_2$ being enabled, it is possible to prove P. Then the Wp-InvAcc rule can be derived from the following two rules.

$$\textsc{Fupd-InvAcc}\quad \frac{\boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E}}{\mathcal{E} \Rightarrow_{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \twoheadrightarrow {}_{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow_{\mathcal{E}} \mathsf{True})} \qquad \textsc{Wp-Atomic}\quad \frac{{}_{\mathcal{E}_1}\!\Rightarrow_{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e\, \{x.\ {}_{\mathcal{E}_2}\!\Rightarrow_{\mathcal{E}_1} \Phi(x)\} \qquad \mathsf{atomic}(e)}{\mathsf{wp}_{\mathcal{E}_1}\, e\, \{\Phi\}}$$

These rules are notationally heavy, but the rule on the left captures the process of opening an invariant with the update modality. Starting from masks in $\mathcal{E}$, we end up with masks in $\mathcal{E} \setminus \mathcal{N}$,

and get $\triangleright P$. Additionally, we get that by supplying $\triangleright P$ we can close the invariant, as represented by the ${}_{\mathcal{E}\setminus\mathcal{N}}\Rrightarrow_{\mathcal{E}}\mathsf{True}$. Meanwhile, the rule on the right is what allows us to actually use the fancy update modality to open invariants when reasoning about an atomic expression e, so long as the the postcondition also includes a modality to close those same invariants.

Under the hood, the semantic model for this ${}_{\mathcal{E}_1}\Rrightarrow_{\mathcal{E}_2}$ modality uses a mechanism called *world satisfaction*. Essentially, the Iris definition of ${}_{\mathcal{E}_1}\Rrightarrow_{\mathcal{E}_2}$ tracks the set of all of the enabled/disabled invariants, and requires that for each enabled invariant, there are resources ensuring the invariant holds. This bundle of resources is called a world.

Although the Iris invariant mechanism is very expressive and flexible, it has some limitations. As a result, some prior projects have found it necessary to modify this notion of invariants. For example, both Perennial [12] and Nola [34] have considered alternate forms of invariant assertions, the former to encode invariants that govern behavior when a program crashes, and the latter to reason about termination without needing the later modality. One key issue, for our purposes, is that the notion of atomicity and the way invariants can be used in a rule like Wp-Atomic is closely tied to the built-in preemptive concurrency in Iris. Instead, we want to allow handler implementers to define a notion of invariant suitable for the kind of effect they are modeling.

Extensible Worlds. To achieve this kind of extensibility, Ficus does not fix a single baked-in world in the interpretation of the fancy update. Instead, Ficus parameterizes the ewp and fancy update modalities by a customizable notion of world. The full version of the Ficus ewp assertion then has the form $\mathsf{ewp}_{\mathrm{W}_1,\mathrm{W}_2}\ e\ \langle\Psi\rangle\ \{\Phi\}$, where W_1 is an arbitrary Iris proposition representing the world at the start of e's execution, and W_2 is the world after e finishes. Meanwhile, the fancy update modality becomes the *world update modality* ${}_{\mathrm{W}_1}\Rrightarrow_{\mathrm{W}_2}$, stating that the update is possible starting from the world W_1 and ends up in the world W_2. When the starting world W is the same as the ending world, we simply write $\mathsf{ewp}_{\mathrm{W}}\ e\ \langle\Psi\rangle\ \{\Phi\}$ and $\Rrightarrow_{\mathrm{W}}$.

Worlds are just normal Iris assertions, but it is nevertheless helpful to think of them more abstractly. The combination of two worlds, written $\mathrm{W}_1 \oplus \mathrm{W}_2$, is defined as $\mathrm{W}_1 * \mathrm{W}_2$. We impose a preorder $\sqsubseteq$ on worlds defined by $\mathrm{W}_1 \sqsubseteq \mathrm{W}_2 \triangleq \exists \mathrm{W}'.\ (\mathrm{W}_2 \dashv\vdash \mathrm{W}_1 \oplus \mathrm{W}')$. Here, larger worlds have more resources, and the minimal element $\bot$ is the proposition True. Thus, with an update like ${}_{\mathrm{W}_1}\Rrightarrow_{\mathrm{W}_2} P$, when $\mathrm{W}_2 \sqsubseteq \mathrm{W}_1$, we are shifting to a *smaller* world, and give the difference betweeen W_2 and W_1 to the proof of P. Conversely, shifting to a larger world with $\mathrm{W}_1 \sqsubseteq \mathrm{W}_2$ requires putting in the difference between W_1 and W_2

The rules we have seen previously for the ewp are generalized to account for worlds. A selection of the generalized rules about ewp and the world update modality are shown in Figure 5. Wupd-Intro introduces a world update ${}_{\mathrm{W}_1}\Rrightarrow_{\mathrm{W}_2} P$ by showing that, given access to the initial world W_1, we are able to prove P and the resulting world W_2, potentially performing ghost updates using the basic update modality. Wupd-Elim eliminates a world update modality from an assumption, updating the worlds on the goal accordingly. The Wupd-Frame allows for "framing out" an unnecessary world W that occurs in both the starting and ending world.

Unlike the Iris Wp-Atomic rule, which only allows masks to change around an atomic step, the Ewp-WupdPre rule allows us to apply a world update that changes the starting world for any expression, and Ewp-WupdPost changes the corresponding ending world. The is allowed because, unlike standard Iris, where the scheduler could preempt a thread at any point, in the effect handler approach, control can only be transferred when an effect is raised. This means the starting world does not need to be immediately restored. Instead, only when an effect is raised, must the world be in an appropriate configuration, depending on whether the protocol Ψ requires it or not. In Ewp-DoWupd, we start by shifting to the bottom world $\bot$, and then in the continuation passed to

$$\textsc{Wupd-Intro}\ \frac{\mathrm{W}_1 \twoheadrightarrow \Rightarrow (P * \mathrm{W}_2)}{{}_{\mathrm{W}_1}\Rightarrow_{\mathrm{W}_2} P} \qquad \textsc{Wupd-Elim}\ \frac{Q \vdash {}_{\mathrm{W}_2}\Rightarrow_{\mathrm{W}_3} P}{({}_{\mathrm{W}_1}\Rightarrow_{\mathrm{W}_2} Q) \vdash {}_{\mathrm{W}_1}\Rightarrow_{\mathrm{W}_3} P} \qquad \textsc{Wupd-Frame}\ \frac{{}_{\mathrm{W}_1}\Rightarrow_{\mathrm{W}_2} Q}{{}_{\mathrm{W}_1\oplus\mathrm{W}}\Rightarrow_{\mathrm{W}_2\oplus\mathrm{W}} Q} \qquad \textsc{Ewp-ValueWupd}\ \frac{{}_{\mathrm{W}_1}\Rightarrow_{\mathrm{W}_2} \Phi(v)}{\mathsf{ewp}_{\mathrm{W}_1,\mathrm{W}_2}\, v\, \langle\Psi\rangle\, \{\Phi\}}$$

$$\textsc{Ewp-WupdPre}\ \frac{{}_{\mathrm{W}_1}\Rightarrow_{\mathrm{W}_2} \mathsf{ewp}_{\mathrm{W}_2,\mathrm{W}_3}\, e\, \langle\Psi\rangle\, \{\Phi\}}{\mathsf{ewp}_{\mathrm{W}_1,\mathrm{W}_3}\, e\, \langle\Psi\rangle\, \{\Phi\}} \qquad \textsc{Ewp-WupdPost}\ \frac{\mathsf{ewp}_{\mathrm{W}_1,\mathrm{W}_2}\, e\, \langle\Psi\rangle\, \{v.\ {}_{\mathrm{W}_2}\Rightarrow_{\mathrm{W}_3} \Phi(v)\}}{\mathsf{ewp}_{\mathrm{W}_1,\mathrm{W}_3}\, e\, \langle\Psi\rangle\, \{\Phi\}}$$

$$\textsc{Ewp-DoWupd}\ \frac{{}_{\mathrm{W}_1}\Rightarrow_{\bot} \Psi(v, (\lambda r.\ {}_{\bot}\Rightarrow_{\mathrm{W}_2} \Phi(r)))}{\mathsf{ewp}_{\mathrm{W}_1,\mathrm{W}_2}\ \mathbf{do}\ v\, \langle\Psi\rangle\, \{\Phi\}} \qquad \textsc{Ewp-WorldFrame}\ \frac{\mathsf{ewp}_{\mathrm{W}_1,\mathrm{W}_2}\, e\, \langle\Psi\rangle\, \{\Phi\}}{\mathsf{ewp}_{\mathrm{W}_1\oplus\mathrm{W},\mathrm{W}_2\oplus\mathrm{W}}\, e\, \langle\Psi\rangle\, \{\Phi\}}$$

Fig. 5. Selected reasoning rules for ${}_{\mathrm{W}_1}\Rightarrow_{\mathrm{W}_2}$ and ewp with worlds.

the protocol Ψ, we must restore back to W_2. Finally, we can frame out an unused world in ewp with Ewp-WorldFrame.

Recovering Iris Invariants. It is straightforward to recover Iris-style impredicative invariants and the Iris fancy update modality in this more general world setting. As was described above, the standard Iris definition fixes some particular world in its definition of fancy updates, and uses ghost state to track the enabled invariants. Let us write $\mathsf{Tok}_{\mathrm{I}}(\mathcal{E})$ for the assertion that bundles the world with the ghost state saying that mask $\mathcal{E}$ is enabled. Then we recover the following analogue of the Fupd-InvAcc rule that we saw earlier.

$$\textsc{Wupd-InvAcc}\ \frac{\boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E}}{{}_{\mathsf{Tok}_{\mathrm{I}}(\mathcal{E})}\Rightarrow_{\mathsf{Tok}_{\mathrm{I}}(\mathcal{E}\setminus\mathcal{N})} \triangleright P * (\triangleright P \twoheadrightarrow {}_{\mathsf{Tok}_{\mathrm{I}}(\mathcal{E}\setminus\mathcal{N})}\Rightarrow_{\mathsf{Tok}_{\mathrm{I}}(\mathcal{E})} \mathsf{True})}$$

Moreover, by combining this rule with Ewp-WorldFrame, we can support Iris invariants while including other possible components in the world. As we will see in §6.1, this allows us to encode a mechanism similar to Perennial's crash borrows [49] while retaining standard Iris invariants.

Protocol for the Concurrency Handler. Now that we have an extensible mechanism for encoding invariants that hold across threads, we turn to the protocol for the concurrency handler.

One challenge is that the concurrency handler in Figure 4 is generic, in the sense that it does not know about the other effects that might be supported by outer handlers. It simply re-raises those effects to the outer handler and potentially transfers control to another thread when the effect returns. This means that the outer handlers could, say, implement shared memory or channel-based message passing concurrency, or some combination thereof. Ideally, the protocol we develop should similarly work for different kinds of outer effects.

To achieve this, the first ingredient is a *protocol transformer* $\mathsf{ATOM}^{\mathrm{W}}$ that lifts a protocol for these outer effects into a concurrent protocol, where W is a world that describes shared resources that can be accessed by different threads. To do this lifting, ATOM transforms Ψ to ensure that as part of raising an effect governed by Ψ, the thread must be able to restore the world W. In addition, when the handler returns control back to a thread, it promises that W will hold. Formally, this is captured through the following definition

$$\mathsf{ATOM}_{\mathrm{W}}(\Psi)(v, \Phi) \triangleq \Psi(v, \lambda r.\ {}_{\bot}\Rightarrow_{\mathrm{W}}\, {}_{\mathrm{W}}\Rightarrow_{\bot} \Phi(r))$$

The first ${}_{\bot}\Rightarrow_{\mathrm{W}}$ is an obligation that the thread raising the effect has to be able establish W after the effect completes. Meanwhile, because the second ${}_{\mathrm{W}}\Rightarrow_{\bot}$ precedes the continuation $\Phi(r)$, it effectively gives back access to W before the continuation's Φ must be proved. In particular, if we

instantiate W to be $\mathsf{Tok}_{\mathrm{I}}(\top)$, where $\top$ is the full mask saying that all invariants are enabled, then the above requires a thread to close all Iris invariants after the operation completes, just as in the Iris rule Wp-Atomic.

To get the final protocol CONC for the concurrency handler, we combine ATOM with a protocol FORK governing the `fork` effect. Because the forked thread will run in the scope of the concurrency handler, FORK must have a recursive dependence on CONC.

$$\mathsf{CONC}_{\mathrm{W}}(\Psi) \triangleq \mathsf{ATOM}_{\mathrm{W}}(\Psi \oplus \mathsf{FORK}_{\mathrm{W}}(\Psi))$$
$$\mathsf{FORK}_{\mathrm{W}}(\Psi)(v, \Phi) \triangleq \exists e.\, v = (\mathtt{fork}, \lambda\_.\, e) * \triangleright\, \mathsf{ewp}_{\mathrm{W}}\, e\, \langle \mathsf{CONC}_{\mathrm{W}}(\Psi)\rangle\, \{\_.\, \mathsf{True}\} * \Phi(())$$

Here, the recursive occurrence of CONC in FORK occurs under the later modality $\triangleright$, so that we can define the result as a *guarded fixed point* [15, 21, 27]. The protocol for FORK requires showing an appropriate ewp for the forked thread. Let us introduce a wrapper $\mathbf{fork}\ e \triangleq \mathbf{do}\ (\mathtt{fork}, \lambda\_.\, e)$. We can re-derive the standard **fork** rule from Iris with this protocol.

$$\textsc{Ewp-Fork}\ \frac{\mathsf{ewp}_{\mathrm{W}}\, e\, \langle \mathsf{CONC}_{\mathrm{W}}(\Psi)\rangle\, \{\_.\, \mathsf{True}\}}{\mathsf{ewp}_{\mathrm{W}}\, \mathbf{fork}\, e\, \langle \mathsf{CONC}_{\mathrm{W}}(\Psi)\rangle\, \{v.\, v = ()\}}$$

Finally, we have the specification for $\mathtt{run}_{\mathsf{conc}}$ from Figure 4, which installs the concurrency handler.

$$\textsc{Ewp-ConcRun}\ \frac{\mathtt{fork} \notin \mathsf{tags}(\Psi) \qquad \mathsf{ewp}_{\mathrm{W}}\, \mathit{main}\, ()\, \langle \mathsf{CONC}_{\mathrm{W}}(\Psi)\rangle\, \{\Phi\}}{\mathsf{ewp}_{\mathrm{W}}\, \mathtt{run}_{\mathsf{conc}}\, \mathit{main}\, \langle \Psi\rangle\, \{\Phi\}}$$

In addition to the ability to create threads with **fork**, we can also model primitive atomic instructions such as compare-and-swap (CAS) or fetch-and-add (FAA). To do so, we just need to define an additional handler on top of the heap handler and associated protocol ATOMHEAP that models these effects, much like the earlier HEAP protocol did. Combining HEAP and ATOMHEAP together, and applying the concurrency handler we obtain a protocol that models all of the operations one finds in the "standard" concurrent HeapLang distributed with Iris. Along the way, we have obtained a stronger version of the invariant opening rule, allowing us to keep invariants open for multiple pure steps.

However, a careful reader might object that what allowed us to derive this stronger invariant opening rule is the fact that the concurrency handler only transfers control to another thread when an effect is raised. In contrast, a standard operational semantics for preemptive concurrency typically allows for preemption at *every* step, thereby generating more possible interleavings of thread operations. Because our handler semantics for concurrency is not generating all of the interleavings that the standard semantics would, one might wonder whether this handler is really sound. Informally, the reason why the handler is sound in spite of this is that the intermediate pure steps in-between effects are not observable to other threads to the system, thus inserting additional preemption points would not change the possible outcomes of execution. In the next section, we introduce a *relational logic* that will allow us to prove this claim rigorously.

4 Contextual Equivalence of Effectful Programs

In this section, we develop Banyan, a relational logic that allows us to prove correspondences between the behavior of two effectful programs written in FicusLang. Just as with unary reasoning, we compositionally derive relational reasoning rules for a number of effects. Using the resulting logic, we define logical relations models that can prove contextual equivalences of programs written in typed subsets of FicusLang. We apply this logical-relations model to prove that inserting additional preemption points in our concurrency handler does not change the set of possible

program behaviors, thus justifying the semantics from the previous section where preemption only occurs when effects are raised.

4.1 Background: Embedding Relational Logics into Unary Logics

Banyan embeds relational reasoning into Ficus by encoding a second program as *ghost state*, as in CaReSL [51]. Let us first recall how this ghost state encoding works in modern Iris-based separation logics. For sequential programs, one first introduces two assertions: $\mathsf{spec}(e)$, which says that the second program (which we call the "spec" program) is currently represented by the expression e; and specCtx, which is an invariant that ensures that the $\mathsf{spec}(e)$ ghost state can only be updated in ways that represent valid transitions in the language's operational semantics. We further add in assertions to represent the state of this spec program. For example a spec program points-to assertion $l \mapsto_s v$ says that in the spec program, location l contains the value v, analogous to the standard points-to assertion. Using these assertions, one derives rules that allow for the spec program to be "executed" by applying ghost updates. To prove a relational property about two programs e_1 and e_2, it then suffices to derive a judgement of the form:

$$\mathsf{specCtx} * \mathsf{spec}(e_1) \vdash \mathsf{wp}\ e_2\ \{v_2.\ \exists v_1.\ \mathsf{spec}(v_1) * \varphi(v_1, v_2)\}$$

The soundness theorem of the encoding says that such a derivation implies that, for every execution of e_2 terminating in a value v_2, there exists a terminating execution of e_1 ending in some value v_1 such that $\varphi(v_1, v_2)$ holds. This basic approach can be generalized to account for concurrency as well by having multiple spec program resources, one for each thread in the concurrent program.

Banyan adapts this style of relational reasoning with ghost programs to the setting of effect handlers. A key challenge is that the usual ghost state encoding requires fixing the primitive effects of the language ahead of time, and requires special treatment in the concurrent case to introduce per-thread spec programs. In contrast, in Banyan these notions are derivable using protocols and worlds, just as with unary reasoning.

4.2 Banyan: A Relational Logic for FicusLang

To enable extensibility, Banyan reasons about spec programs using an *effect specification resource* assertion $\mathsf{espec}_W\ e\ \langle\Psi\rangle$ that tracks a spec program e and a protocol Ψ in a world W. The program e can be updated and progressed according to the operational semantics of FicusLang. For example, $\mathsf{espec}_W\ ((\lambda x.\ e)\ v)\ \langle\Psi\rangle$ can be updated to $\mathsf{espec}_W\ e[v/x]\ \langle\Psi\rangle$ to reflect the execution of a beta reduction as justified by Espec-Pure shown below. As in Ficus, the protocol Ψ describes the effect handlers that are active as e executes. That is, $\mathsf{espec}_W\ N[\mathbf{do}\ v]\ \langle\Psi\rangle$ can be updated to $\mathsf{espec}_W\ N[w]\ \langle\Psi\rangle$ such that $\Phi(w)$ for some w by establishing $\Psi(v, \Phi)$ as enabled by Espec-Do.

$$\mathsf{espec}_W\ K[e]\ \langle\Psi\rangle * e \to^* e' \vdash \Rrightarrow_W \mathsf{espec}_W\ K[e']\ \langle\Psi\rangle \qquad \textsc{Espec-Pure}$$

$$\mathsf{espec}_W\ N[\mathbf{do}\ v]\ \langle\Psi\rangle * \Psi(v, \Phi) \vdash \Rrightarrow_W \exists w.\ \mathsf{espec}_W\ N[w]\ \langle\Psi\rangle * \Phi(w) \qquad \textsc{Espec-Do}$$

Using these rules, we obtain derived rules for raising specific effects like global state, much as in the unary case in §2, *e.g.*,

$$\mathsf{espec}_W\ (\mathtt{read}, ())\ \langle\Psi \oplus \mathsf{STATE}^\gamma\rangle * S^\gamma(v) \vdash \Rrightarrow_W \mathsf{espec}_W\ v\ \langle\Psi \oplus \mathsf{STATE}^\gamma\rangle * S^\gamma(v)$$

$$\mathsf{espec}_W\ (\mathtt{write}, w)\ \langle\Psi \oplus \mathsf{STATE}^\gamma\rangle * S^\gamma(v) \vdash \Rrightarrow_W \mathsf{espec}_W\ ()\ \langle\Psi \oplus \mathsf{STATE}^\gamma\rangle * S^\gamma(w)$$

Similarly, we have rules for installing handlers that capture these effects, such as

$$\mathsf{espec}_W\ \mathtt{run}_{\mathtt{state}}\ \mathit{main}\ \mathit{init}\ \langle\Psi\rangle \vdash \Rrightarrow_W \exists \gamma.\ S^\gamma(\mathit{init}) * \mathsf{espec}_W\ \mathit{main}\ ()\ \langle\Psi \oplus \mathsf{STATE}^\gamma\rangle.$$

The following adequacy theorem for Banyan holds, which requires that both the spec protocol and the ewp protocol are $\bot$.

Theorem 4.1 (Adequacy, Banyan). *Let φ be a first-order relation. If*

$$\mathsf{espec}_\bot\ e_2\ \langle\bot\rangle \vdash \mathsf{ewp}_\bot\ e_1\ \langle\bot\rangle\ \{v_1.\ \exists v_2.\ \mathsf{espec}_\bot\ v_2\ \langle\bot\rangle * \varphi(v_1, v_2)\}$$

and $e_1 \to^* v_1$ then there exists a value v_2 such that $e_2 \to^* v_2$ and $\varphi(v_1, v_2)$.

The key to proving this adequacy theorem lies in coming up with a suitable definition of the $\mathsf{espec}_W\ e\ \langle\Psi\rangle$ resource that validates the above rules.

Constructing the Effect Specification Resource. Much like the ewp assertion in Ficus, the specification resource $\mathsf{espec}_W\ e\ \langle\Psi\rangle$ tracks the behavior of the program e *under the assumption* that it executes in a program context satisfying the protocol Ψ and a logical world W. To define espec, we first define a more general construction genspec which we will specialize to obtain espec. The genspec assertion takes some abstract notion of a spec program and transforms it into a version that has protocols and worlds for reasoning about effects. Specifically, let spec : *Expr* → *iProp* be a predicate with the property that $\mathsf{spec}(e) \vdash \Rrightarrow_W \mathsf{spec}(e')$ when $e \to^* e'$. Given a choice of spec predicate, the genspec assertion is defined as

$$\mathsf{genspec}_W\ e\ \langle\Psi\rangle \triangleq \exists K.\ \mathsf{spec}(K[e]) * \mathsf{handler}_W(\Psi)(K)$$

where the assertion $\mathsf{handler}_W(\Psi)(K)$ captures that K is an evaluation context that realizes the protocol Ψ indefinitely from the perspective of a program e running inside that context. Formally, this is expressed using a *greatest fixpoint*:

$$\begin{aligned}&\mathsf{handler}_W(\Psi) \triangleq \mathsf{gfp}\ F, K.\\ &\quad \forall v.\ \mathsf{spec}(K[v]) \mathrel{-\!\!*} \Rrightarrow_W \mathsf{spec}(v)\\ \wedge\ &\quad \forall v, N, \Phi.\ \mathsf{spec}(K[\S(N)[v]]) * \Psi(v, \Phi) \mathrel{-\!\!*} \Rrightarrow_W \exists K', w.\ \mathsf{spec}(K'[N[w]]) * \Phi(w) * F(K')\end{aligned}$$

The two conjuncts of this definition require that

(1) if e is a value v then the context terminates with value v, and
(2) if e is a raised effect with continuation N and value v, *i.e.,* $e = \S(N)[v]$, and $\Psi(v, \Phi)$ holds, then N is reinstated with some value w in a context K' such that $\Phi(w)$ and $\mathsf{handler}_W(\Psi')(K')$ holds co-recursively.

We can derive generic versions of the Espec-Pure and Espec-Do rules for genspec, as well as examples like the STATE protocol. Then we obtain espec and specialized versions of these rules by instantiating genspec with $\mathsf{spec}(e) \triangleq e_0 \to^* e$, where e_0 is the initial expression that the ghost program starts as. Later, we will see how instantiating the definition with other choices of the base specification resource allow us to reason about thread-local effects in concurrent execution.

4.3 Concurrency

To reason relationally about effects that do not immediately transfer control back to the raising thread, we need to make use of extensible worlds, just as we did with the unary logic for concurrency. However, in the relational case, our specification of concurrency has an additional requirement: We want to be able to reason about each thread in the concurrent system individually. In the encoding of specification programs in CaReSL described above in §4.1, this is achieved by having a specification assertion per thread. Since each thread can raise effects and use other components of a protocol, we need these per-thread resources to have access to the protocol, just as in espec.

To construct this per-thread effect specification resource, we again use the genspec construction. To do so, we first need an underlying per-thread local specification resource $\mathsf{spec}_t^\gamma(e)$ that we will use to instantiate the construction with. Here, the γ parameter is a ghost name and t tracks whether

the thread is either a main thread $\mathcal{M}$ or a child thread C. Additionally, we also need as *specification context world* $\mathrm{CTX}^\gamma(\mathrm{W}, \Psi)$ that tracks the state of the concurrency handler.

$$\mathrm{spec}_t^\gamma(e) \triangleq \exists k, r. \boxed{\circ\{(k, r, t)\}}^\gamma * (\forall K.\, \mathrm{spec}(K[k\ r]) \mathrel{-\!\!*} \Rrightarrow_\bot \mathrm{spec}(K[e]))$$

$$\mathrm{CTX}^\gamma(\mathrm{W}, \Psi) \triangleq \exists B, \mathit{pool}.\, \mathrm{isBag}(B, \mathit{pool}) * \boxed{\bullet B}^\gamma * \mathrm{espec}_{\mathrm{W}}\ \mathsf{go}_{\mathsf{conc}}\ \mathit{pool}\ \langle \Psi \rangle$$

In these definitions, we assert that there is some instance of the concurrency handler from Figure 4 installed, and we use ghost state to track the thread pool in the handler. Specifically, the thread pool is tracked using ghost resources such that $\boxed{\bullet B}^\gamma * \boxed{\circ B'}^\gamma \vdash B' \subseteq B$ and $\boxed{\bullet B}^\gamma \vdash \Rrightarrow_{\mathrm{W}} \boxed{\bullet (B \uplus B')}^\gamma * \boxed{\circ B'}^\gamma$. In the definition of $\mathrm{spec}_t^\gamma(e)$ we use this ghost state to assert that a triple of the form (k, r, t) is stored in the pool, where k is the continuation representing the thread and r is the result of the last effect. Additionally, we require that there is some way to run $k\ r$ so that it will reach the expression e. In other words, the thread currently in the pool may not yet be e, but when it is next scheduled to run, it can execute to e. The $\mathrm{CTX}^\gamma(\mathrm{W}, \Psi)$ assertion enforces that in fact there is an underlying espec running the concurrency handler providing the protocol Ψ.

This thread-local specification resource fulfills the requirements needed to instantiate genspec, *i.e.,* when $e \to^* e'$ then $\mathrm{spec}_t^\gamma(e) \vdash \Rrightarrow_{\mathrm{W}} \mathrm{spec}_t^\gamma(e')$. When updating $\mathrm{spec}_t^\gamma(e)$ to $\mathrm{spec}_t^\gamma(e')$ in this derivation, instead of directly updating the underlying base effect specification resource in $\mathrm{CTX}^\gamma(\mathrm{W}, \Psi)$, we instead accumulate the evidence that there exists a thread in the pool that can be evaluated to the expression e' when it is next scheduled.

Let $\mathrm{espec}_{\mathrm{W}}^{\gamma;t}\ e\ \langle \Psi \rangle$ be notation for the result of instantiating genspec with this assertion. This thread-local effect specification resource gives us a unified mechanism to reason about both global and thread-local effects. It supports analogues of the Espec-Pure and Espec-Do rules. In addition, we derive a rule for forking threads,

Spec-Fork

$$\frac{\mathrm{espec}_{\mathrm{W}}^{\gamma;t}\ N[\mathbf{fork}\ e]\ \langle \mathrm{CONC}_s^\gamma(\Psi) \rangle \qquad \mathrm{CTX}^\gamma(\mathrm{W}', \Psi') \sqsubseteq \mathrm{W}}{\Rrightarrow_{\mathrm{W}} \mathrm{espec}_{\mathrm{W}}^{\gamma;t}\ N[()]\ \langle \mathrm{CONC}_s^\gamma(\Psi) \rangle * \mathrm{espec}_{\mathrm{CTX}^\gamma(\mathrm{W}', \Psi')}^{\gamma;C}\ e\ \langle \mathrm{CONC}_s^\gamma(\Psi') \rangle}$$

where $\mathrm{CONC}_s^\gamma(\Psi) \triangleq \mathrm{FORK}_s^\gamma \oplus \Psi$ and $\mathrm{FORK}_s^\gamma(v, \Phi) \triangleq \exists e.\, v = (\mathsf{fork}, \lambda\_.\, e) * (\mathrm{spec}_C^\gamma(e) \mathrel{-\!\!*} \Phi())$. Note that Spec-Fork assumes that the specification context is in the current world. The specification context is allocated when the concurrency handler is installed using the following rule.

Spec-Conc-Run

$$\frac{\mathrm{espec}_{\mathrm{W}}\ \mathsf{run}_{\mathsf{conc}}\ \mathit{main}\ \langle \Psi \rangle \qquad \mathsf{fork} \notin \mathrm{tags}(\Psi)}{\Rrightarrow_{\mathrm{W}} \exists \gamma.\, \mathrm{CTX}^\gamma(\mathrm{W}, \Psi) * \mathrm{espec}_{\mathrm{CTX}^\gamma(\mathrm{W}, \Psi)}^{\gamma;\mathcal{M}}\ \mathit{main}\ ()\ \langle \mathrm{CONC}_s^\gamma(\Psi) \rangle}$$

4.4 Logical Relation for Contextual Equivalence

Using Banyan, we next define a binary *program-logic based logical relation* [22] for proving contextual equivalence of programs written in typed subsets of FicusLang. Intuitively, an expression e_1 is contextually equivalent to another expression e_2 at a type τ, written $e_1 \simeq_{\mathrm{ctx}} e_2 : \tau$, if no well-typed contexts C can distinguish them. In other words, the behavior of a client program remains unchanged if we replace any occurrence of the sub-program e_1 with e_2. Contextual equivalence is defined as the symmetric interior of contextual refinement, denoted by $e_1 \lesssim_{\mathrm{ctx}} e_2 : \tau$. Intuitively refinement means that, for any context C the observable behavior of $C[e_1]$ is *included* in the observable behavior of $C[e_2]$, *relative to a closing handler context* H. Formally, we define

$$e_1 \lesssim_{\mathrm{ctx}}^H e_2 : \tau \quad \triangleq \quad \forall b \in \mathit{Bool}, C : \tau \to \mathbf{bool}.\, H[C[e_1]] \to^* b \Rightarrow H[C[e_2]] \to^* b.$$

As a consequence, both the context and the programs may interact through effects.

As an example, we consider a standard System-F-style type system $\Theta \mid \Gamma \vdash e : \tau$ with impredicative polymorphism, recursive types, and typing rules for CAS, FAA, and the fork operation (see, *e.g.,* Timany et al. [50] for a complete definition).

The logical relation is entirely standard and follows previous Iris-based models ([50]), *except* that we define the expression interpretation using Banyan instantiated with the atomic heap instructions and concurrency. The expression interpretation is

$$\llbracket \tau \rrbracket(e_1, e_2) \triangleq \forall N, t.\, \mathrm{espec}^{\gamma;t}_{\mathrm{W}_s}\ N[e_2]\ \langle \Psi_2 \rangle \twoheadrightarrow \\ \mathrm{ewp}_{\mathrm{W}}\ e_1\ \langle \Psi_1 \rangle\ \{v_1.\, \exists v_2.\, \mathrm{espec}^{\gamma;t}_{\mathrm{W}_s}\ N[v_2]\ \langle \Psi_2 \rangle * \llbracket \tau \rrbracket(v_1, v_2)\}$$

where $\Psi_2 \triangleq \mathrm{CONC}^{\gamma}_s(\Psi')$, $\Psi_1 \triangleq \mathrm{CONC}_{\mathrm{W}}(\Psi)$, $\mathrm{W}_s = \mathrm{CTX}^{\gamma}(\bot, \Psi_2)$, $\mathrm{W} = \mathrm{W}_s \oplus \mathrm{Tok}_{\mathrm{I}}(\top)$ for Ψ and Ψ' that describe the global stack of effects (state, heap, and atomic heap). The proof of the fundamental theorem of logical relations is immediate from the existing proofs since all our rules for reasoning about the atomic heap operations and concurrency are identical to the usual separation logic rules.

Theorem 4.2 (Fundamental). *If* $\Theta \mid \Gamma \vdash e : \tau$ *then* $\Theta \mid \Gamma \vDash e \lesssim e : \tau$.

To prove soundness, we consider the closing handler context

$$H_{\mathrm{CONC}} = \mathtt{run}_{\mathtt{state}}\ (\lambda\_.\, \mathtt{run}_{\mathtt{heap}}\ (\lambda\_.\, \mathtt{run}_{\mathtt{atomheap}}\ (\lambda\_.\, \mathtt{run}_{\mathtt{conc}}\ [])))$$

and use the handler rules, *e.g.,* Ewp-ConcRun and Spec-Conc-Run, and Theorem 4.1.

Theorem 4.3 (Soundness). *If* $\cdot \mid \cdot \vDash e_1 \lesssim e_2 : \tau$ *then* $e_2 \lesssim^{H_{\mathrm{CONC}}}_{\mathrm{ctx}} e_2 : \tau$.

Contextual Equivalence of Preemption. Using our logical relation, we prove that inserting additional preemption points in concurrent programs does not change program behaviors. This justifies the soundness of using a scheduler that only triggers preemption when effects are raised, since it shows that additional preemption would not affect observable behavior. Formally, we introduce an expression **yield** that triggers a preemption point by defining **yield** $\triangleq$ **fork** $()$, *i.e.,* a program that simply forks a thread that terminates immediately. By forking a thread, **yield** transfers control to the scheduler, which may choose another thread to continue.

To justify that **yield** has no effect on the computation, we show that it is contextually equivalent to the unit value, *i.e.,* **yield** $\simeq^{H_{\mathrm{CONC}}}_{\mathrm{ctx}} () : \mathbf{unit}$. The proof is an immediate consequence of the rules Ewp-Fork and Spec-Fork for the left-to-right and right-to-left refinements, respectively. As a corollary, for example, it then follows that $e_1;\ \mathbf{yield};\ e_2 \simeq^{H_{\mathrm{CONC}}}_{\mathrm{ctx}} e_1;\ e_2 : \tau$ for any well-typed e_1 and e_2. As we will see in §7, a similar technique can be used to justify stronger atomicity reasoning rules in the context of distributed execution.

5 Case Study: Prophecy Variables

When verifying certain concurrent programs in a forward-reasoning style, at some points in the proof it is necessary to know how later operations will be non-deterministically ordered. Prophecy variables [1, 28] are a logical mechanism that allows for "speculating" or "predicting" these future outcomes during a proof. In Iris, these prophecy variables are ghost code, and a prover must instrument a program to attach prophecy variables to operations whose values need to be predicted. New prophecy variables are allocated using a command **newproph**, and then attached to a program value using **resolve**. The proof rules for prophecy variables tell us *at the time of allocation* what the future resolved value will be. Iris comes with an additional proof showing that these prophecy variable operations can be erased from the program without affecting the outcome.

In this section, we show how to extend Ficus with prophecy variables. Our starting point is a single *global* prophecy variable. On top of this global prophecy variable, we implement effect handlers that allow for dynamically allocatable local prophecy variables, with an interface similar

to that of Iris. Finally, we observe that by instrumenting the *handlers* for heap operations with prophecy variables, we can automatically extend all heap operations to have prophecies, without requiring the *client* program to be directly instrumented with prophecies.

Global Prophecy. Recall that FicusLang has a ghost expression `observe` v, which records the value v on a global trace. The entire future value of this trace is predicted in a global prophecy assertion $\mathsf{primProph}\ (\vec{v})$, which is used when performing an `observe`.

$$\textsc{Ewp-Obs}\quad \frac{\mathsf{primProph}\ (\vec{v})}{\mathsf{ewp}_{\mathrm{W}}\ \texttt{observe}\ w\ \langle \Psi \rangle\ \{\_.\ \exists \vec{v}'.\ \vec{v} = w :: \vec{v}' * \mathsf{primProph}\ (\vec{v}')\}}$$

By making an observation of w, we immediately learn that the first element of $\vec{v}$ is indeed w, so $\vec{v}$ must equal $w :: \vec{v}'$ for some unknown $\vec{v}'$.[2] The adequacy theorem of Ficus is extended to provide this prophecy assertion.

Theorem 5.1 (Adequacy, Ficus). *Let φ be a first-order predicate. If* $\mathsf{primProph}\ (\vec{v}) \vdash \mathsf{ewp}\ e\ \langle \bot \rangle\ \{\varphi\}$ *is derivable for all $\vec{v}$, then executing e will not get stuck, and if $e \to^* v$ then $\varphi(v)$ holds.*

However, because this prophecy variable is global, it is awkward to use when trying to do local reasoning about data structures that need prophecies.

Encoding Local Prophecy Variables. We recover Iris-style local prophecy variables by using handlers on top of the global `observe`. Formally, our local prophecy variables are specified by the following protocols

$$\begin{aligned}\mathsf{NEWPROPH}(u, \Phi) &\triangleq u = (\texttt{newproph}, ()) * (\forall p, \vec{v}.\ \mathsf{proph}\ (p, \vec{v}) \mathrel{-\!\!*} \Phi(p))\\ \mathsf{RESOLVE\_PROPH}(u, \Phi) &\triangleq \exists v, p, w, \vec{v}.\ u = (\texttt{resolve\_proph}, (v, p, w)) * \mathsf{proph}\ (p, \vec{v}) *\\ &\qquad (\forall \vec{v}'.\ \vec{v} = (v, w) :: \vec{v}' * \mathsf{proph}\ (p, \vec{v}') \mathrel{-\!\!*} \Phi(v))\end{aligned}$$

A new prophecy variable is created by raising the effect `newproph`, which returns back a fresh prophecy variable p. Assertion $\mathsf{proph}\ (p, \vec{v})$ is the local version of $\mathsf{primProph}\ (\vec{v})$, which says that the trace of resolutions that will occur on prophecy variable p is $\vec{v}$. The effect `resolve_proph` resolves p to a pair of values (v, w). The first component v is the primary value that we want to observe, while the second value is used for "tagging" meta-data to certain kinds of prophecies.

Under the hood, these assertions work by slicing the observation trace of the global prophecy into traces of individual prophecy variables. This is done by making every call to *ghostcodekeywordobserve* have the format $(p, (v, w))$ where p is the identifier of the prophecy variable that the corresponding observation is for. Then we can (tentatively) define $\mathsf{proph}\ (p, \vec{v}) \stackrel{?}{=} \exists \vec{v}_0.\ \mathsf{primProph}\ (\vec{v}_0) * \vec{v} = \mathsf{filter}(p, \vec{v}_0)$, where filter is the least fixed-point of

$$\begin{aligned}\mathsf{filter}(p, (p', (v, w)) :: \vec{v}) &\triangleq \text{if } p = p' \text{ then } (v, w) :: \mathsf{filter}(p, \vec{v}) \text{ else } \mathsf{filter}(p, \vec{v})\\ \mathsf{filter}(p, \_) &\triangleq \varepsilon\end{aligned}$$

The filter projects out the observations that are associated with the indicated prophecy variable, and it returns ε as a default value, if the observations in the trace do not match the expected format.

Recall that resources in CSL are exclusive to one thread, so to actually have mutually independent prophecy variables, we need to decompose ownership of the global prophecy $\mathsf{primProph}\ (\vec{v})$ into

[2] As usual with prophecy reasoning, if the predicted value at the head of the sequence $\vec{v}$ was *not* w, then we derive a contradiction from this rule, and no longer have to reason about this moot execution with a misprediction.

ownership of individual prophecy variables using an authoritative ghost resource algebra.

$$\text{proph}\,(p, \vec{v}) \triangleq \boxed{\circ\{p \mapsto \vec{v}\}}^{\gamma_p}$$
$$I_p \triangleq \exists \vec{v}_0, M_p.\ \text{primProph}\,(\vec{v}_0) * \boxed{\bullet M_p}^{\gamma_p} * \mathop{*}_{p \mapsto \vec{v} \in M_p} \vec{v} = \text{filter}(p, \vec{v}_0)$$

Here γ_p is an arbitrary but fixed global variable. Invariant I_p connects individual prophecy variables to the global prophecy and is maintained by the handler. The underlying resource algebra guarantees that $\circ\{p \mapsto \vec{v}\}$ is always an element in the map M_p.

The handler $\texttt{run}_{\texttt{proph}}$ provides protocols NEWPROPH and RESOLVE_PROPH. The handler maintains I_p. To create new prophecy variables, it internally maintains a monotonically increasing counter for the next fresh prophecy ID so that it can always "slice out" an unused resolving sequence from the global prophecy for a new prophecy variable. To resolve prophecy variable p to (v, w), it makes an observation of $(p, (v, w))$ and updates the ghost resources accordingly. In practice, the handler $\texttt{run}_{\texttt{proph}}$ is installed first so that other handlers can use prophecy variables.

Atomic Prophecies. The resolve effect we have seen so far resolves a value that is returned by evaluating some expression. In other words, the prophecy is resolved *after* the expression finishes. However, in some scenarios, it is necessary to atomically execute the expression and prophecy resolution at the same time, particularly for atomic heap operations like CAS and FAA. Iris provides support for this so-called atomic prophecy resolution, and we can also implement this on top of the local prophecy variables through another effect handler with the following protocol:

$$\text{RESOLVE}(\Psi)(v, \Phi) \triangleq \exists e, p, w, \vec{v}.\ v = (\texttt{resolve}, (e, p, w)) * \text{proph}\,(p, \vec{v}) * \Psi(e, (\lambda r.\ \forall \vec{v}'.\ \vec{v} = (r, w) :: \vec{v}' \mathrel{-\!\!*} \text{proph}\,(p, \vec{v}') \mathrel{-\!\!*} \Phi(r)))$$

The handler $\texttt{run}_{\texttt{atomproph}}$ provides this protocol. To handle **do** $(\texttt{resolve}, (e, p, w))$, it executes **do** $(\texttt{resolve\_proph}, (\textbf{do}\ e, p, w)$. By installing this handler *after* the $\texttt{run}_{\texttt{proph}}$ and the handlers for heap operations, but *before* the handler $\text{run}_{\texttt{conc}}$ for concurrency, we ensure that **do** e and **do** $(\texttt{resolve\_proph}, \ldots)$ behave as if they executed together atomically, because the additional **raise** in $\texttt{run}_{\texttt{atomproph}}$ does not trigger an additional preemption.

Implicit Prophecies. As in Iris, the above interface for prophecies still requires a proof developer to annotate a program with calls to create new prophecies and to resolve them at relevant points. However, we can use effect handlers to make it so that *every* heap location has an associated prophecy variable that predicts the full trace of operations that will be performed on that heap location. This "prophetic heap" handler interposes on all of heap related effect tags, and adds an extra `resolve` operation before re-raising the effect. With this protocol, when a location is allocated, in addition to the standard points-to assertion $l \mapsto v$, we also get a proph $(l, \vec{v})$ assertion. When a heap operation on l occurs, we also pass this proph $(l, \vec{v})$ assertion, allowing us to deduce that the value read/written to the location matches the head value in the trace $\vec{v}$. This allows for proofs with prophecies *without* having to annotate a client program with explicit prophecy operations. Appendix B describes the protocols in further detail.

6 Case Study: Crash-Recovery Reasoning

Many software systems that store data on durable media such as disks must be *crash safe*, meaning that the system must be able to recover from a crash caused by externally generated events such as power failures. When a crash occurs, any data that the system has in volatile memory, such as RAM, will be wiped, but data in durable storage will be preserved. After the system restarts, it will typically re-run a recovery procedure that restores system invariants. A number of program logics and verification frameworks have been developed for reasoning about such systems [12, 14, 37, 42].

```
run_crash_trigger ≜ λmain.
  try main () with
    v k ⇒ let r := do v in
          if nondet_bool () then do (crash, ())
          else k r
  | ret v ⇒ v
```

```
run_crash ≜ rec run main.
  try main () with
    v k ⇒ match v with
          (crash, ()) ⇒ observe (); run main
        | (η, v) ⇒ do (η, v)
  | ret v ⇒ v
```

Fig. 6. A model of crash and recovery execution.

In this section, we show how to model crashes and recovery with effect handlers, and apply Ficus to derive protocols for reasoning about these systems in the style of Perennial [12], a separation logic for reasoning about the combination of concurrency and crash safety.

The process of crashing and recovering is modeled by the pair of handlers in Figure 6. The handler $\texttt{run}_{\texttt{crash\_trigger}}$ is installed at the inner-most level of a stack of effects for each thread, allowing it to interpose on every **do**.[3] It handles these effects by non-deterministically choosing to either trigger a crash by raising `crash`, or by simply re-raising the effect and returning the result to the continuation. The second handler, $\texttt{run}_{\texttt{crash}}$ responds to this trigger by throwing away the captured continuation k and re-starting the system by running *main*. Note that this handler does not directly deal with wiping the volatile state of the system. Instead, this is handled implicitly: by installing handlers for volatile state (such as the `heap` handler) *after* this crash handler (*i.e.,* as part of *main*), this volatile state will be effectively thrown away as a result of re-running *main* from scratch. In contrast, durable state can be preserved by installing these handlers *before* installing the crash handler at an outer level.

6.1 Managing the Crash Invariant

In order to establish that a system is crash safe, it is essential to show that when the system restarts, the *main* procedure finds itself in a state that satisfies its precondition. Perennial maintains a global *crash invariant* $\mathcal{R}$ that must hold before and after each step of execution, and which describes the durable state that the system needs upon restart.

Local Crash Conditions. Reasoning about a global crash invariant would run counter to the principle of local reasoning in concurrent separation logic. To recover per-thread reasoning about the crash invariant, Perennial extends the weakest precondition of each thread with an assertion called a *crash condition*. The crash condition enforces the portion of the global crash invariant that a given thread owns. In Ficus, rather than changing the weakest precondition to add an additional component, we can instead capture this local crash condition through worlds and a protocol transformer called DURA. We write $\mathsf{Tok}_{\mathrm{C}}(R_c)$ for a world stating a thread is responsible for ensuring that the local crash invariant R_c holds before and after each step it takes. The DURA protocol forces a thread to show that this R holds before and after each step of execution.

$$\mathsf{DURA}_{\mathrm{W}}(\Psi)(v,\Phi) \triangleq \exists R_c.\, \Psi(v, \lambda r.\ {}_{\perp}\Rrightarrow_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(R_c)} (R_c \wedge {}_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(R_c)}\Rrightarrow_{\perp} \Phi(r)))$$

Notice that R_c and the postcondition is connected by a logical conjunction $\wedge$ because the program can only either crash or continue so only one of R_c and the postcondition will be used.

The derived proof rules for the crash handlers then require a specification for the top level *main* procedure of the following form.

$$\mathcal{R} \vdash \mathsf{ewp}_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(\blacklozenge\mathcal{R}),\perp}\ \mathit{main}\ ()\ \langle \mathsf{DURA}_{\mathrm{W}}(\Psi)\rangle\ \{r.\ \exists R_c.\ {}_{\perp}\Rrightarrow_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(R_c)} R_c \wedge \Phi(v)\}$$

Here, $\mathcal{R}$ is the global crash invariant that also serves as the precondition for *main*, and $\blacklozenge$ is the *post-crash modality* [47, 55] that captures how crashing modifies volatile and durable resources.

[3] To install $\texttt{run}_{\texttt{crash\_trigger}}$ for every child thread, our Rocq mechanization actually integrates $\texttt{run}_{\texttt{crash\_trigger}}$ into $\texttt{run}_{\texttt{conc}}$.

Intuitively, this says that the main thread starts with precondition $\mathcal{R}$, and the initial crash condition requires $\mathcal{R}$ holds after a crash.

Concurrency with Crashes. To add support for concurrency, we install the concurrency handler below the outer crash handler. Because of the crash condition, we need to strengthen the CONC protocol to a protocol called CRASHCONC.

$$\mathsf{CRASHCONC}_{\mathrm{W}}(\Psi) \triangleq \mathsf{DURA}_{\mathrm{W}}(\Psi \oplus \mathsf{FORK}'_{\mathrm{W}}(\Psi))$$

$$\mathsf{FORK}'_{\mathrm{W}}(\Psi)(v, \Phi) \triangleq \exists e.\, v = (\texttt{fork}, \lambda\_.\, e) * \\ \triangleright\, \mathsf{ewp}_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(\mathrm{True}),\bot}\ e\ \langle \mathsf{CRASHCONC}_{\mathrm{W}}(\Psi)\rangle\ \{\exists R_e.\ \bot \Rrightarrow_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(R_e)} R_e\} * \Phi(())$$

CRASHCONC follows the same structure of CONC and is also a guarded fixed point. However, a forked child thread starts with a trivial crash condition and can terminate with any crash condition.

When forking a child thread, one would naturally like to move some resources from the parent thread to the child thread. Similarly, synchronization primitives like locks are logically thought of as transferring ownership of resources in CSL. However, in order to transfer ownership of durable resources that might be part of the crash condition, we also need a mechanism to transfer the *obligation* to maintain that part of the crash condition. Crash borrows in Perennial [49] provide a mechanism to "borrow" part of the crash condition as an ownable resource and transfer this resource to the child thread. A crash borrow $\boxed{P \mid R}$ has *content* P that describes the resources currently contained, and an associated *crash obligation* R, where $\square(P \twoheadrightarrow R)$.

The crash borrow can be understood as a box that packages up a resource P while preserving the obligation R in the event of a crash. They are used through the following two key rules.

$$\frac{\triangleright P \qquad \square(P \twoheadrightarrow R)}{\mathsf{Tok}_{\mathrm{C}}(R_c * R) \Rrightarrow_{\mathsf{Tok}_{\mathrm{C}}(R_c)} \boxed{P \mid R}}\ \textsc{Wupd-CBrwAlloc} \qquad \frac{\boxed{P \mid R}}{\mathsf{Tok}_{\mathrm{C}}(R_c) \Rrightarrow_{\mathsf{Tok}_{\mathrm{C}}(R_c * R)} \triangleright P}\ \textsc{Wupd-CBrwReturn}$$

Rule Wupd-CBrwAlloc creates a crash borrow $\boxed{P \mid R}$. It consumes a resource P that is stronger than R and removes R from the crash condition. Rule Wupd-CBrwReturn opens the box $\boxed{P \mid R}$ to extract resource P, in exchange, it adds R to the crash condition. In Perennial, this crash borrow mechanism is encoded on top of standard Iris invariants in a complex manner that requires extensive use of later credits [45] to avoid inconsistencies from impredicative circularities. In Ficus, the encoding is considerably simpler, because we are able to use a separate world for managing crash conditions and crash borrows. The complete model can be found in Appendix C.

6.2 Asynchrony and Crash-Aware Prophecies

Many durable storage media are *asynchronous*, meaning that when a write is performed, the written value does not immediately become durable. Instead, the written value is first stored in some volatile buffer and only later made durable. If a crash occurs while the value is still in the volatile buffer, then the write is lost. Reasoning about asynchrony is challenging when trying to prove that a concurrent durable data structure satisfies *durable linearizability* [26], because it makes the durability of an operation *future dependent*.

To deal with this challenge, Perennial introduced a *prophetic disk points-to* assertion of the form $l \mapsto^{\mathrm{d}} [v_c]v$ which says that the disk address l currently stores the value v, and after a crash occurs, the stored address will be v_c. In other words, this assertion bundles a normal points-to with a form of prophecy about the post-crash state. However, in Perennial, this primitive could not re-use the existing support for prophecies in Iris, and instead has an ad-hoc soundness proof. The issue is that, with standard Iris prophecy variables, there is no way to make a prophecy about whether an event will happen before or after a crash occurs.

In contrast, in Ficus it is easy to handle this by incorporating prophecy resolution as part of the implementation of the crash handler. We use the `observe ()` statement in $\texttt{run}_{\texttt{crash}}$ to effectively record that a crash has occurred in the trace of every prophecy variable. Formally, for every prophecy variable p, $\text{proph}\,(p, \vec{v}) \vdash \blacklozenge(\vec{v} = \varepsilon)$. The definition of filter in §5 ensures that this truncates the trace of events in the prophecy stream for all variables. Thus, when inspecting the prophecy stream, it is possible to determine whether a crash will occur before the prophecy is resolved. We call these resulting prophecy variables *crash-aware*. Using this mechanism, we implement an asynchronous disk with prophetic points-to assertions by resolving a crash-aware prophecy whenever an asynchronous disk operation is performed. More details can be found in Appendix C.

7 Case Study: Distributed Systems with IronFleet-Style Atomic Blocks

In this section, we consider a distributed system with multiple nodes connected by an unreliable network, in which nodes communicate through messages that may be dropped, delayed, duplicated, or re-ordered. On top of the network, a global scheduler decides the order of execution of nodes.

Network. The network provides two operations. $\text{NETWORK} \triangleq \text{SEND} \oplus \text{RECV}$, where

$$\text{SEND}(v, \Phi) \triangleq \exists s, t, m, M.\, v = (\texttt{send}, (s, t, m)) * t \rightsquigarrow M * (t \rightsquigarrow \{(s, t, m)\} \cup M \mathrel{-\!\!*} \Phi(()))$$

$$\text{RECV}(v, \Phi) \triangleq \exists t, M.\, v = (\texttt{recv}, t) * t \rightsquigarrow M * \left(\forall x.\, t \rightsquigarrow M * \begin{pmatrix} x = \texttt{inl}\ () \lor (\exists s, m.\, x = \texttt{inr} \\ (s, t, m) \land (s, t, m) \in M) \end{pmatrix} \mathrel{-\!\!*} \Phi(x)\right)$$

Assertion $t \rightsquigarrow M$ says that M is the set of messages that have ever been sent to address t. Since messages can be arbitrarily duplicated, this set is monotonically increasing *w.r.t.* the subset relation. The SEND protocol expects a package of (source address, destination address, message) as input, and adds this package to the message history of the destination address. The RECV protocol expects the destination address t as input and non-deterministically chooses to either not return a message or to return an arbitrary message that has been sent to t. The dropping of a message is implicitly modeled as just never having it be selected for receipt. Protocol NETWORK is provided by the handler $\texttt{run}_{\texttt{network}}$ which implements the network as a soup of messages [30, 58].

Scheduler with IronFleet-Style Atomic Blocks. Next, we need a scheduler $\texttt{run}_{\texttt{dist}}$ that specifies how nodes run concurrently through a $\text{DISTR}^t_{\text{W}}$ protocol.

$$\text{DISTR}^t_{\text{W}} \triangleq \text{ATOM}_{\text{W}}(\text{SEND} \oplus \text{START}^t_{\text{W}}) \oplus \text{RECV}$$

$$\text{START}^t_{\text{W}}(v, \Phi) \triangleq \exists e.\, v = (\texttt{start}, \lambda\_.\, e) * \left(t = C \lor \triangleright\, \text{ewp}_{\text{W}}\, e\, \langle \text{DISTR}^C_{\text{W}} \rangle\, \{\_.\, \text{True}\}\right) * \Phi(())$$

The $\texttt{run}_{\texttt{dist}}$ handler resembles $\texttt{run}_{\texttt{conc}}$ and potentially transfers control to different nodes when an effect is raised by a node. The START protocol is used for initially creating nodes. In the protocol above, the RECV effect is *outside* the ATOM protocol transformer. The reason for this is that $\texttt{run}_{\texttt{dist}}$ scheduler does *not* transfer control to another node when processing an `recv` operation. In other words, a node can receive a series of messages without transferring control to another node. This modeling choice is inspired by IronFleet [25] which uses a movers-based parallel reduction proof [32] to treat a sequence of receives followed by a sequences of sends as an atomic step. Our global scheduler permits the prover to view a series of `recv` operations, followed by a series node-local processing operations, followed by one `send` operation as an atomic block.[4] As a result, because the RECV is not included in the ATOM component, we do *not* need to close shared invariants when performing a RECV operation.

Even though this scheduler does not include preemptions at RECV, the absence of these preemptions does not affect the overall set of possible behaviors of the program. To prove this, we

[4]We preempt after a single send because a node could diverge after sending. IronFleet avoids this by proving total correctness.

apply a similar technique as in §4, and use Banyan to prove that an explicit **yield** preemption is contextually equvialent to the unit value. Thus, adding in additional preemptions does not change the program's behavior. To carry out this proof, we develop a node-local specification resource $\mathsf{specn}_t^\gamma(e)$, similar to the thread-local version described in §4. It also uses the evidence accumulation technique described in §4.3, but additionally accumulates the evidence that a node can delay message receipt without changing behavior. Intuitively, if a message m is received received at some time T, then if we delay that receipt to some later time T', it is still possible to receive m. We capture this evidence using a *monotonic* resource algebra to track the set M of messages. More details can be found in Appendix D.

8 Related Work

Program Logics for Effect Handlers. The most closely related work is the Hazel logic for effect handlers [18]. As discussed in §2 and §3, Ficus extends Hazel with support for extensible worlds.

Hazel only handles unary reasoning, whereas Banyan supports relational reasoning through an encoding into Ficus. Recently, de Vilhena et al. [20] developed Blaze, a relational logic for effect handlers. Like Banyan, Blaze builds on a unary logic and represents a specification program via ghost state. However, unlike Banyan, in which the unary logic and the specification program have separate protocols, Blaze instead provides a judgement with a *relational* protocol. They use this to prove refinements in which the interpretation of effects is different between the two programs. In contrast, our examples keep effects the same on both sides and prove that client programs under these effects are equivalent.

Among other examples, de Vilhena et al. [20] use Blaze to prove that a handler implementation of concurrency refines a primitive concurrency effect. This refinement is in some sense the opposite of the direction that motivated our refinement proof in §4: it essentially shows that for every execution of the concurrency handler (which only preempts at effects), there is a corresponding execution using primitive concurrency (which preempts at every step). In contrast, we show that inserting additional preemption points when using the concurrency handler does not generate new behaviors. This is morally equivalent to showing that the concurrency handler already covers all possible heaviors that could be generated by a full interleaving semantics. It would be interesting to apply Blaze's approach to the kinds of applications we have considered here to justify the soundness of alternate implementations of effects that allow for deriving stronger reasoning rules.

Our logical relations are for type systems with a fixed collection of effects and do not provide rules for typing general effect handlers. Tes [19] and Affect [54] use logics to construct unary logical-relations models for type systems for effect handlers. Biernacki et al. [9, 10] directly construct a binary logical-relations model for effects and handlers using biorthogonality and step indexing.

Extensible Program Logics. As described in the introduction, Vistrup et al. [56] develop an approach to extensible program logics using ITrees [59]. They use a mechanism called *logical effect handlers* to interpret ITree events for an effect, which has similarities to the way Hazel and Ficus's protocols give a logical interpretation of what a raised effect will do. Their soundness proofs relate these logical effect handlers to interpretations of the effects. In contrast, the corresponding soundness of a protocol in Ficus is justified by the rule for **try** that installs an effect handler and makes the protocol accessible. Since the handlers are themselves just programs written in FicusLang, one uses Ficus itself to prove these handlers implement the protocol. Thus there is no distinction between verifying a *program* and proving the soundness of an *extension* to the logic. Another difference is that using the the effect handler approach, we are able to develop a relational logic by representing a specification program as ghost state. Vistrup et al. [56] do not develop a relational

logic. On the other hand, they show how to model other features, such as total correctness and angelic non-determinism, which we do not consider.

Like Ficus's extensible worlds, Matsushita and Tsukada [34] parameterize the Iris update modality and Hoare triples by a notion of a world. However, they require that the update shifts to the same world before and after, *i.e.,* only considering shifts of the form ${}_W\Rightarrow_W$. Hence, they cannot model the use of extensible worlds in §3, in which invariants are kept open across non-preempting steps.

Dijkstra Monads [46] offer a framework for deriving pre- and postcondition reasoning about dependently-typed programs with monadic effects, and, more recently, some aspects of algebraic effect handlers [33]. However, Dijkstra Monads have not been applied to effects like concurrency, crashes, or distributed execution, which might be challenging to encode as monads in a compositional way. Existing work on Dijkstra Monads does not support relational reasoning.

References

[1] Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. 165–175. doi:10.1109/LICS.1988.5115

[2] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, Shin-ya Katsumata, and Tetsuya Sato. 2021. Higher-order probabilistic adversarial computations: categorical semantics and program logics. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. doi:10.1145/3473598

[3] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, ICFP (2024), 284–316. doi:10.1145/3674635

[4] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. 109–122. doi:10.1145/1190216.1190235

[5] Jialu Bao, Emanuele D'Osualdo, and Azadeh Farzan. 2025. Bluebell: An Alliance of Relational Lifting and Independence for Probabilistic Reasoning. *Proc. ACM Program. Lang.* 9, POPL (2025), 1719–1749. doi:10.1145/3704894

[6] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. 387–401. doi:10.1007/978-3-662-48899-7_27

[7] Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. A probabilistic separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 55:1–55:30. doi:10.1145/3371123

[8] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29. doi:10.1145/3290347

[9] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.* 2, POPL, Article 8 (Dec. 2017), 30 pages. doi:10.1145/3158096

[10] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. doi:10.1145/3371116

[11] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.* 8, 4 (2012). doi:10.2168/LMCS-8(4:1)2012

[12] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. 243–258. doi:10.1145/3341301.3359632

[13] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

[14] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 18–37. doi:10.1145/2815400.2815402

[15] Krzysztof Ciesielski. 2007. On Stefan Banach and some of his results. *Banach Journal of Mathematical Analysis* 1, 1 (2007), 1–10. doi:10.15352/bjma/1240321550

[16] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 792–808. doi:10.1145/3519939.3523451

[17] Paulo de Vilhena. 2022. *Proof of Programs with Effect Handlers. (Preuve de Programmes avec Effect Handlers).* Ph. D. Dissertation. Paris Cité University, France. https://tel.archives-ouvertes.fr/tel-03891381

[18] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL, Article 33 (Jan. 2021), 28 pages. doi:10.1145/3434314

[19] Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings.* 225–252. doi:10.1007/978-3-031-30044-8_9

[20] Paulo Emílio de Vilhena, Simcha van Collem, Ines Wright, and Robbert Krebbers. 2025. A Relational Separation Logic for Effect Handlers. (November 2025). https://devilhena-paulo.github.io/files/blaze.pdf.

[21] Pietro Di Gianantonio and Marino Miculan. 2003. A Unifying Approach to Recursive and Co-recursive Definitions. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 148–161.

[22] Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* 7, 2 (2011). doi:10.2168/LMCS-7(2:16)2011

[23] Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL (2024), 332–361. doi:10.1145/3632854

[24] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL (2024), 753–784. doi:10.1145/3632868

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* 1–17. doi:10.1145/2815400.2815428

[26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9888)*, Cyril Gavoille and David Ilcinkas (Eds.). Springer, 313–327. doi:10.1007/978-3-662-53426-7_23

[27] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

[28] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. doi:10.1145/3371113

[29] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain.* 17:1–17:29. doi:10.4230/LIPICS.ECOOP.2017.17

[30] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* 336–365. doi:10.1007/978-3-030-44914-8_13

[31] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II.* 311–323. doi:10.1007/978-3-662-47666-6_25

[32] Richard J. Lipton. 1975. Reduction: A New Method of Proving Properties of Systems of Processes. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, USA, January 1975*, Robert M. Graham, Michael A. Harrison, and John C. Reynolds (Eds.). ACM Press, 78–86. doi:10.1145/512976.512985

[33] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *Proc. ACM Program. Lang.* 3, ICFP (2019), 104:1–104:29. doi:10.1145/3341708

[34] Yusuke Matsushita and Takeshi Tsukada. 2025. Nola: Later-Free Ghost State for Verifying Termination in Iris. *Proc. ACM Program. Lang.* 9, PLDI (2025), 98–124. doi:10.1145/3729250

[35] Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP, Article 66 (Aug. 2021), 29 pages. doi:10.1145/3473571

[36] Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. 255–266. doi:10.1109/LICS.2000.855774

[37] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. 2015. Fault-Tolerant Resource Reasoning. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. 169–188. doi:10.1007/978-3-319-26529-2_10

[38] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. doi:10.1016/j.tcs.2006.12.035

[39] Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340. doi:10.1007/BF00268134

[40] Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. 1–24. doi:10.1007/3-540-45315-6_1

[41] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 80–94. doi:10.1007/978-3-642-00590-9_7

[42] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 151:1–151:28. doi:10.1145/3428219

[43] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 55–74. doi:10.1109/LICS.2002.1029817

[44] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. 113–129. doi:10.1145/3600006.3613172

[45] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP, Article 100 (Aug. 2022), 29 pages. doi:10.1145/3547631

[46] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 387–398. doi:10.1145/2491956.2491978

[47] Joseph Tassarotti, Tej Chajed, and Perennial contributors. 2021. Crash Borrow in Perennial's Mechanization. https://github.com/mit-pdos/perennial/blob/c4c806b6ede0580dc8cb72ca873d25e3fb7e564d/src/goose_lang/crash_modality.v

[48] Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 64:1–64:30. doi:10.1145/3290377

[49] Joseph Tassarotti and Perennial contributors. 2022. Crash Borrow in Perennial's Mechanization. https://github.com/mit-pdos/perennial/blob/5189a4eccc8583a7042ccccbbb3bb13cfb57e4c5/src/goose_lang/crash_borrow.v

[50] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

[51] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 377–390. doi:10.1145/2500365.2500600

[52] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 691–707. doi:10.1145/2660193.2660243

[53] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 867–884. doi:10.1145/2509136.2509532

[54] Orpheas van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. *Proc. ACM Program. Lang.* 9, POPL, Article 5 (Jan. 2025), 29 pages. doi:10.1145/3704841

[55] Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. 2025. The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 83–97. doi:10.1145/3703595.3705876

[56] Max Vistrup, Michael Sammler, and Ralf Jung. 2025. Program Logics à la Carte. *Proc. ACM Program. Lang.* 9, POPL (2025), 300–331. doi:10.1145/3704847

[57] James R. Wilcox, Ilya Sergey, and Zachary Tatlock. 2017. Programming Language Abstractions for Modularly Verified Distributed Systems. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. 19:1–19:12. doi:10.4230/LIPICS.SNAPL.2017.19

[58] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 357–368. doi:10.1145/2737924.2737958

[59] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. doi:10.1145/3371119

A Ficus

A.1 Semantics

Head reduction rules. $e \xrightarrow{\vec{\kappa}}_{\mathsf{h}} e'$

$$\textsc{Hd-Beta}\quad (\mathbf{rec}\ f\ x.\ e)\ v \xrightarrow{\varepsilon}_{\mathsf{h}} e[(\mathbf{rec}\ f\ x.\ e), v/f, x] \qquad \textsc{Hd-Cont}\quad (\mathbf{cont}\ N)\ v \xrightarrow{\varepsilon}_{\mathsf{h}} N[v]$$

$$\textsc{Hd-EffApR}\quad e_1\ \S(N)[v_2] \xrightarrow{\varepsilon}_{\mathsf{h}} \S(e_1\ N)[v_2] \qquad \textsc{Hd-Do}\quad \mathbf{do}\ v \xrightarrow{\varepsilon}_{\mathsf{h}} \S([])[v]$$

$$\textsc{Hd-EffApL}\quad \S(N)[v_1]\ v_2 \xrightarrow{\varepsilon}_{\mathsf{h}} \S(N\ v_2)[v_1] \qquad \textsc{Hd-Pick}\quad \mathbf{pick} \xrightarrow{\varepsilon}_{\mathsf{h}} z$$

$$\textsc{Hd-EffDo}\quad \mathbf{do}\ \S(N)[v] \xrightarrow{\varepsilon}_{\mathsf{h}} \S(\mathbf{do}\ N)[v] \qquad \textsc{Hd-Obs}\quad \mathbf{observe}\ v \xrightarrow{[v]}_{\mathsf{h}} ()$$

$$\textsc{Hd-TryEff}\quad \mathbf{try}\ \S(N)[v_0]\ \mathbf{with}\ v_1\ k \Rightarrow e_1 \mid \mathbf{ret}\ v_2 \Rightarrow e_2 \xrightarrow{\varepsilon}_{\mathsf{h}} e_1[v_0, \mathbf{cont}\ N/v_1, k]$$

$$\textsc{Hd-TryVal}\quad \mathbf{try}\ v_0\ \mathbf{with}\ v_1\ k \Rightarrow e_1 \mid \mathbf{ret}\ v_2 \Rightarrow e_2 \xrightarrow{\varepsilon}_{\mathsf{h}} e_2[v_0/v_2]$$

Pure Reduction and Its Reflexive Transitive Closure. $e \xrightarrow{\vec{\kappa}} e'$ and $e \xrightarrow{\vec{\kappa}}{}^{*} e'$

$$e \xrightarrow{\vec{\kappa}} e' \triangleq \exists K, \tilde{e}, \tilde{e}'.\ e = K[\tilde{e}] \wedge e' = K[\tilde{e}'] \wedge \tilde{e} \xrightarrow{\vec{\kappa}}_{\mathsf{h}} \tilde{e}'$$

$$e \xrightarrow{\vec{\kappa}}{}^{*} e' \triangleq (e = e' \wedge \vec{\kappa} = \varepsilon) \vee (\exists e'', \vec{\kappa}_1, \vec{\kappa}_2.\ \vec{\kappa} = \vec{\kappa}_1 +\!\!+ \vec{\kappa}_2 \wedge e \xrightarrow{\vec{\kappa}_1}_{\mathsf{h}} e'' \wedge e'' \xrightarrow{\vec{\kappa}_2}{}^{*} e')$$

A.2 Reasoning Rules

$$\textsc{Ewp-Value}\ \frac{{}_{W_1}\!\Rightarrow_{W_2} \Phi(v)}{\mathsf{ewp}_{W_1,W_2}\ v\ \langle\Psi\rangle\ \{\Phi\}} \qquad \textsc{Ewp-Do}\ \frac{\Psi(v, \Phi)}{\mathsf{ewp}_{W}\ \mathbf{do}\ v\ \langle\Psi\rangle\ \{\Phi\}} \qquad \textsc{Ewp-DoFupd}\ \frac{{}_{W_1}\!\Rightarrow_{\bot} \Psi(v, (\lambda r.\ {}_{\bot}\!\Rightarrow_{W_2} \Phi(r)))}{\mathsf{ewp}_{W_1,W_2}\ \mathbf{do}\ v\ \langle\Psi\rangle\ \{\Phi\}}$$

$$\textsc{Ewp-Mono}\ \frac{\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{\Phi\} \qquad \Psi \sqsubseteq \Psi' \qquad \forall v.\ \Phi(v) \mathrel{-\!\!*} \Rightarrow_{W_2} \Phi'(v)}{\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi'\rangle\ \{\Phi'\}} \qquad \textsc{Ewp-WorldMono}\ \frac{\mathsf{ewp}_{W}\ e\ \langle\Psi\rangle\ \{\Phi\} \qquad W \sqsubseteq W'}{\mathsf{ewp}_{W'}\ e\ \langle\Psi\rangle\ \{\Phi\}}$$

$$\textsc{Ewp-Frame}\ \frac{R \qquad \mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{\Phi\}}{\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{v.\ R * \Phi(v)\}} \qquad \textsc{Ewp-WupdPre}\ \frac{{}_{W_1}\!\Rightarrow_{W_2} \mathsf{ewp}_{W_2,W_3}\ e\ \langle\Psi\rangle\ \{\Phi\}}{\mathsf{ewp}_{W_1,W_3}\ e\ \langle\Psi\rangle\ \{\Phi\}}$$

$$\textsc{Ewp-WupdPost}\ \frac{\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{v.\ {}_{W_2}\!\Rightarrow_{W_3} \Phi(v)\}}{\mathsf{ewp}_{W_1,W_3}\ e\ \langle\Psi\rangle\ \{\Phi\}} \qquad \textsc{Ewp-Pure}\ \frac{\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{\Phi\} \qquad e' \to^{*} e}{\mathsf{ewp}_{W_1,W_2}\ e'\ \langle\Psi\rangle\ \{\Phi\}}$$

$$\textsc{Ewp-Bind}\ \frac{\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{v.\ \mathsf{ewp}_{W_2,W_3}\ N[v]\ \langle\Psi\rangle\ \{\Phi\}\}}{\mathsf{ewp}_{W_1,W_3}\ N[e]\ \langle\Psi\rangle\ \{\Phi\}} \qquad \textsc{Ewp-WorldFrame}\ \frac{\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{\Phi\}}{\mathsf{ewp}_{W_1 \oplus W, W_2 \oplus W}\ e\ \langle\Psi\rangle\ \{\Phi\}}$$

$$\textsc{Ewp-Obs}\ \frac{\mathsf{primProph}\ (\vec{v})}{\mathsf{ewp}_{W}\ \mathbf{observe}\ w\ \langle\Psi\rangle\ \{\_.\ \exists \vec{v}'.\ \vec{v} = w :: \vec{v}' * \mathsf{primProph}\ (\vec{v}')\}}$$

A.3 Model

$$\mathsf{ewp}_{W_1,W_2}\ v\ \langle\Psi\rangle\ \{\Phi\} \triangleq {}_{W_1}\!\Rightarrow_{W_2} \Phi(v)$$

$$\mathsf{ewp}_{W_1,W_2}\ \S(N)[v]\ \langle\Psi\rangle\ \{\Phi\} \triangleq {}_{W_1}\!\Rightarrow_{\bot} \Psi(v, \lambda w.\ \Rightarrow_{\bot} \triangleright(\mathsf{ewp}_{\bot,W_2}\ N[w]\ \langle\Psi\rangle\ \{\Phi\}))$$

$$\mathsf{ewp}_{W_1,W_2}\ e\ \langle\Psi\rangle\ \{\Phi\} \triangleq \forall \vec{\kappa}_1, \vec{\kappa}_2.\ \mathsf{primProph}^{\bullet}\ (\vec{\kappa}_1 +\!\!+ \vec{\kappa}_2) \mathrel{-\!\!*} {}_{W_1}\!\Rightarrow_{\bot} (\exists e'.\ e \to e') *$$

$$\forall e'.\, e \xrightarrow{\vec{\kappa}_1} e' \twoheadrightarrow \Rightarrow_{\bot} \triangleright \Rightarrow_{\bot} \mathsf{primProph}^{\bullet}\ (\vec{\kappa}_2) * \mathsf{ewp}_{\bot,\mathrm{W}_2}\ e'\ \langle\Psi\rangle\ \{\Phi\}$$

$$\mathsf{primProph}^{\bullet}\ (\vec{v}) \triangleq \boxed{\bullet Ex(\vec{v})}^{\gamma_P} \qquad \mathsf{primProph}\ (\vec{v}) \triangleq \boxed{\circ Ex(\vec{v})}^{\gamma_P}$$

B Prophetic Heap

$$\begin{aligned}
\mathsf{PROPH_ALLOC}(v,\Phi) &\triangleq \exists x.\, v = (\mathtt{alloc}, x) * (\forall l, \vec{v}.\, l \mapsto x * \mathsf{proph}\ (l,\vec{v})\ \mathsf{prophE}\ (l) \twoheadrightarrow \Phi(l))\\
\mathsf{PROPH_LOAD}(v,\Phi) &\triangleq \exists l, x, \vec{v}.\, v = (\mathtt{load}, l) * l \mapsto x * \mathsf{proph}\ (l,\vec{v}) * \mathsf{prophE}\ (l) *\\
&\quad (\forall \vec{v}'.\, l \mapsto x * v = (x, \mathsf{load}) :: \vec{v}' * \mathsf{proph}\ (l,\vec{v}') * \mathsf{prophE}\ (l) \twoheadrightarrow \Phi(x))\\
\mathsf{PROPH_LOAD}'(v,\Phi) &\triangleq \exists l, q, x.\, v = (\mathtt{load}, l) * l \overset{q}{\mapsto} x * \mathsf{prophD}\ (l) * (l \overset{q}{\mapsto} x \twoheadrightarrow \Phi(x))\\
\mathsf{PROPH_STORE}(v,\Phi) &\triangleq \exists l, x, y, \vec{v}.\, v = (\mathtt{store}, (l, y)) * l \mapsto x * \mathsf{proph}\ (l,\vec{v}) * \mathsf{prophE}\ (l) *\\
&\quad (\forall \vec{v}'.\, l \mapsto y * \vec{v} = ((), \mathsf{store}(y)) :: \vec{v}' * \mathsf{proph}\ (l,\vec{v}') * \mathsf{prophE}\ (l) \twoheadrightarrow \Phi(()))\\
\mathsf{PROPH_CAS}(v,\Phi) &\triangleq \exists l, w, x, y, \vec{v}.\, v = (\mathtt{cas}, (l, x, y)) * l \mapsto w * \mathsf{proph}\ (l,\vec{v}) * \mathsf{prophE}\ (l) *\\
&\quad (\forall \vec{v}'.\, l \mapsto (\text{if } w = x \text{ then } y \text{ else } w) * \vec{v} = (w = x, \mathsf{cas}(x, y)) :: \vec{v}' *\\
&\quad \mathsf{proph}\ (l,\vec{v}') * \mathsf{prophE}\ (l) \twoheadrightarrow \Phi(w, w = x))\\
\mathsf{PROPH_CAS}'(v,\Phi) &\triangleq \exists l, q, w, x, y.\, v = (\mathtt{cas}, (l, x, y)) * l \overset{q}{\mapsto} w * w \neq x * \mathsf{prophD}\ (l) *\\
&\quad (l \overset{q}{\mapsto} w \twoheadrightarrow \Phi(w, \mathtt{false}))
\end{aligned}$$

C Crash Recovery System

This section uses the complete $\mathsf{Tok}_{\mathrm{C}}(\mathcal{E}, R_c)$ token. Compared to $\mathsf{Tok}_{\mathrm{C}}(R_c)$ used in §6, it has one extra parameter for the enabled crash borrows. The connection between two tokens is $\mathsf{Tok}_{\mathrm{C}}(R_c) = \mathsf{Tok}_{\mathrm{C}}(\top, R_c)$.

C.1 Post-crash Modality

$$\begin{aligned}
\mathsf{crashed} &\triangleq \exists M\colon \mathit{PID} \to \mathit{List}(\mathit{Val} \times \mathit{Val}).\, \boxed{M}^{\gamma_{\mathsf{proph}}} * \forall p \mapsto \vec{v} \in M.\, \vec{v} = \varepsilon\\
\blacklozenge P &\triangleq \mathsf{crashed} \twoheadrightarrow \mathsf{crashed} * P
\end{aligned}$$

C.2 Protocol

$\mathcal{R}$ is the global crash invariant. There is no requirement on Φ because crash will never return. CRASHCONC intentionally uses the same tag as the regular CONC protocol to prevent having two schedulers. As said by the FORK′ protocol, a child thread is permitted to change the crash condition during execution, as long as it is consistent with the $\mathsf{Tok}_{\mathrm{C}}(\mathcal{E}, R)$ token. The crash condition cannot be violated even if a thread terminates.

$$\begin{aligned}
\mathsf{CRASH}(v,\Phi) &\triangleq v = (\mathtt{crash}, ()) * (\Rightarrow \mathcal{R})\\
\mathsf{DURA}_{\mathrm{W}}(\Psi)(v,\Phi) &\triangleq \exists R.\, \Psi(v, (\lambda r.\ {}_{\bot}\Rightarrow_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(\top,R)} (R \wedge {}_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(\top,R)}\Rightarrow_{\bot} \Phi(r))))\\
\mathsf{CRASHCONC}_{\mathrm{W}}(\Psi) &\triangleq \mathsf{DURA}_{\mathrm{W}}(\Psi \oplus \mathsf{FORK}'_{\mathrm{W}}(\Psi))\\
\mathsf{FORK}'_{\mathrm{W}}(\Psi)(v,\Phi) &\triangleq \exists e.\, v = (\mathtt{fork}, \lambda_.\, e) *\\
&\quad \triangleright\, \mathsf{ewp}_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(\top,\mathsf{True}),\bot}\ e\ \langle\mathsf{CRASHCONC}_{\mathrm{W}}(\Psi)\rangle \left\{\exists R_e.\ {}_{\bot}\Rightarrow_{\mathrm{W}\oplus\mathsf{Tok}_{\mathrm{C}}(\top,R_e)} R_e\right\} * \Phi(())
\end{aligned}$$

Interaction between protocols:

$$\mathsf{DURA}_{\mathrm{W}}(\Psi) \sqsubseteq \mathsf{ATOM}_{\mathrm{W}}(\Psi) \sqsubseteq\!\!\!\!\sim\ \Psi \qquad\qquad \mathsf{ATOM}_{\mathrm{W}}(\Psi) \sqsubseteq \mathsf{CONC}_{\mathrm{W}}(\Psi)$$

$$\mathsf{DURA}_{\mathrm{W}}(\Psi) \sqsubseteq \mathsf{CRASHCONC}_{\mathrm{W}}(\Psi)$$

Ewp-Seq-Atom

$$\frac{\mathsf{ewp}_{\mathrm{W}_I}\ e\ \langle\Psi\rangle\ \{\Phi\}}{\mathsf{ewp}_{\mathrm{W}_I \oplus \mathrm{W}}\ e\ \langle\mathsf{ATOM}_{\mathrm{W}}(\Psi)\rangle\ \{\Phi\}}$$

Ewp-Atom-Dura

$$\frac{\square(R' \mathrel{-\!\!*} R) \qquad R' \qquad \mathsf{ewp}_{\mathrm{W}}\ e\ \langle\mathsf{ATOM}_{\mathrm{W}}(\Phi)\rangle\ \{v.\ R' \mathrel{-\!\!*} \Phi(v)\}}{\mathsf{ewp}_{\mathrm{W} \oplus \mathsf{Tok}_{\mathrm{C}}(\top, R)}\ e\ \langle\mathsf{DURA}_{\mathrm{W}}(\Phi)\rangle\ \{\Phi\}}$$

Ewp-Conc-Crashconc

$$\frac{\mathsf{ewp}_{\mathrm{W}}\ e\ \langle\mathsf{CONC}_{\mathrm{W}}(\Phi)\rangle\ \{\Phi\}}{\mathsf{ewp}_{\mathrm{W} \oplus \mathsf{Tok}_{\mathrm{C}}(\top, \mathsf{True})}\ e\ \langle\mathsf{CRASHCONC}_{\mathrm{W}}(\Phi)\rangle\ \{\Phi\}}$$

C.3 Crash Hoare Logic

$$\mathsf{ewpc}^0_{(\mathcal{E}_1, R_1), (\mathcal{E}_2, R_2)}\ e\ \langle\Psi\rangle\ \{\Phi\} \triangleq \mathsf{ewp}_{\mathrm{W}_1, \mathrm{W}_2}\ e\ \left\langle\mathsf{CRASHCONC}_{\mathsf{Tok}_{\mathrm{I}}(\top)}(\Psi)\right\rangle\ \{\Phi\}$$
$$\text{where} \quad \mathrm{W}_i \triangleq \mathsf{Tok}_{\mathrm{I}}(\mathcal{E}_i) \oplus \mathsf{Tok}_{\mathrm{C}}(\mathcal{E}_i, R_i)$$

The ewpc assertion does not support the monotonic rule, but this will not become a restriction in practice because one can always use the upward closure of a non-mask-changing ewpc.

$$\mathsf{ewpc}_{(\mathcal{E}, R)}\ e\ \langle\Psi\rangle\ \{\Phi\} \triangleq \forall R', \Phi'.\ (\forall v.\ \Phi(v) \mathrel{-\!\!*} \Phi'(v)) \wedge (R \mathrel{-\!\!*} R') \mathrel{-\!\!*} \mathsf{ewpc}^0_{(\mathcal{E}, R')}\ e\ \langle\Psi\rangle\ \{\Phi'\}$$

The logical conjunction $\wedge$ between Φ and R precisely captures the monotonicity in a crash system. Only one part of this conjunction is needed at a time. The Φ part is used during normal execution and the R part is used when the system crashes.

C.4 Crash Borrow

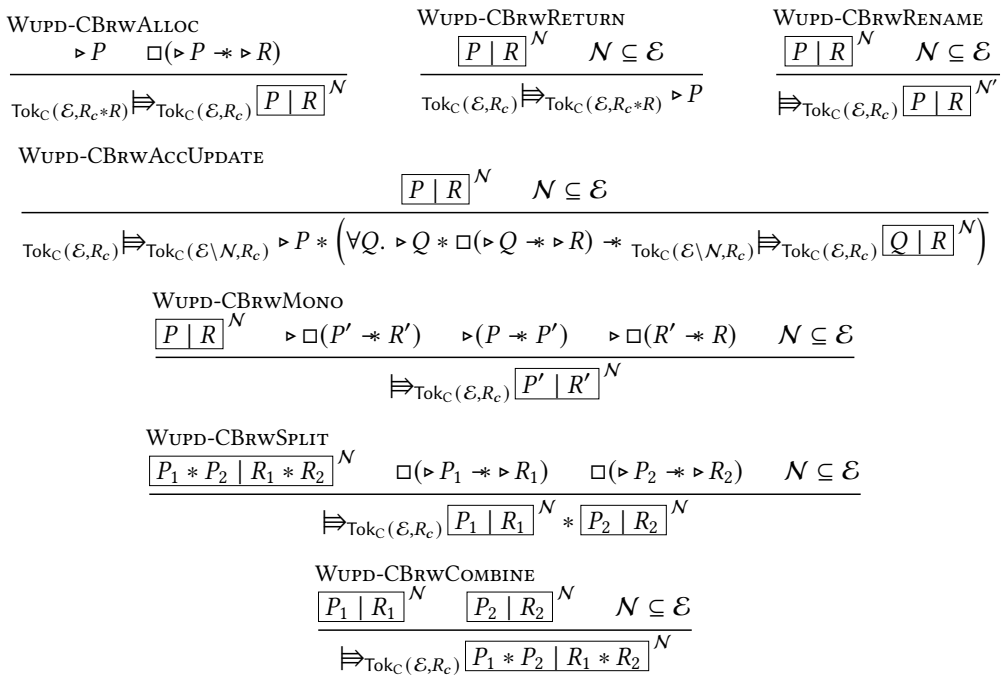
See Figures 7 and 8.

C.5 Asynchronous Disk

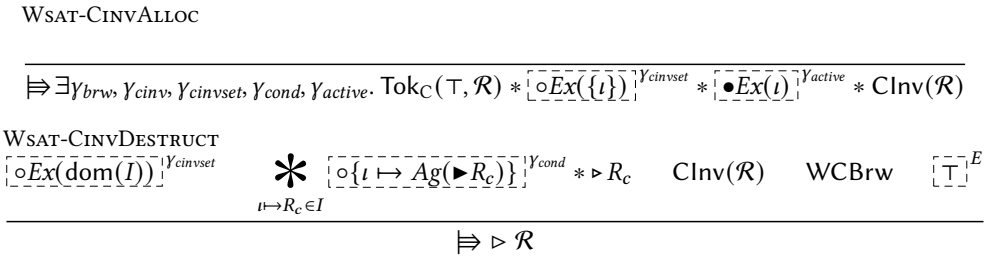
The client rules about the asynchronous disk are specified by protocols in Figure 9a. Resource $l \mapsto^{\mathrm{d}} [v_c]v$ declares the ownership of an asynchronous disk location l. There are two values associated with one location: v is the value visible to the system before crash, and v_c is the value visible to the system after crash. Using the post-crash modality, this means $l \mapsto^{\mathrm{d}} [v_c]v \vdash \blacklozenge l \mapsto^{\mathrm{d}} v_c$. Notice that because asynchronous disk is essentially a synchronous disk plus a software buffer, the points-to assertion will become a regular disk points-to $l \mapsto^{\mathrm{d}} v_c$ after crash. Only at the recovery stage will the asynchronous disk points-to assertion be recreated: $l \mapsto^{\mathrm{d}} v_c \vdash \lozenge l \mapsto^{\mathrm{d}} [v_c]v_c$, where $\lozenge$ is called *setup modality*.

The ADISK_LOAD protocol is standard. According to protocol ADISK_STORE, an `adisk_store` effect immediately updates the before-crash value at location l to w, but the after-crash value v'_c could be either w or v_c depending on whether the buffer will be written back before the next crash. An `adisk_barrier` effect issues a global write barrier that writes-back the *whole* buffer to the disk. Therefore, the client can use this effect to write-back an arbitrary number of locations. Because the buffer was indeed written back before crash, we now know that v_c must have equaled to v.

Use of Prophecy Variables in the Effect Handler. The handler for asynchronous disk is shown in Figure 9b and important logic constructions used to verify it are listed in Figure 9c. It uses a volatile state to store the buffer. For each buffered location, the handler associates a prophecy variable to it, indicating whether this location will be written back before crash. For buffer item $l \mapsto (v, p)$, the value v will be written back iff $\exists \vec{v}'.\ \mathsf{proph}\ (p, ((), \mathtt{true}) :: \vec{v}')$, which is formalized



(a) Client rules.



(b) Handler rules.

Fig. 7. Reasoning rules of crash borrows and crash conditions.

by the willWB assertion. For a `adisk_load` effect, the handler returns the cached value if location l is in the buffer (line 4), otherwise, it uses **`disk_load`** operation to load the value directly from the physical disk and buffers the result (line 5). For a `adisk_store` effect, the handler always writes the result to the buffer, but if the location is already in the buffer, the handler will resolve the associated prophecy variable to **`false`**, meaning that the old value in the buffer will never be written back (as it has been overwritten by the new value). For a `barrier` effect, the handler writes back the whole buffer and resolves each prophecy variable to **`true`**. In the event of a crash, all prophecy variables will become ε and because $\varepsilon \neq ((), \textbf{true}) :: \_$, we learn that remaining values in the buffer will never be written back.

Concretely, the asynchronous disk points-to assertion $l \mapsto^{\mathrm{d}} [v_c]v$ is defined as a view of the $\mathrm{adp}(B, l, v_c, v)$ assertion, which B is the buffer. The assertions consists of two cases. If l is not in the buffer, then v is in the physical disk and $v_c = v$. If l is in the buffer, then v is the buffered value and

$$\boxed{P \mid R}^{\mathcal{N}} \triangleq \exists i.\, i \in \mathcal{N} * \boxed{\circ\{i \mapsto Ag(\blacktriangleright(P, R))\}}^{\gamma_{brw}} \qquad \mathsf{CInv}(\mathcal{R}) \triangleq \boxed{\circ Ex(\blacktriangleright \mathcal{R})}^{\gamma_{cinv}}$$

$$\mathsf{Tok}_C(\mathcal{E})R_c \triangleq \mathsf{WCBrw} \oplus \boxed{\mathcal{E}}^{E} \oplus \left(\exists \iota.\, \boxed{\circ Ex(\iota)}^{\gamma_{active}} * \boxed{\circ\{\iota \mapsto Ag(\blacktriangleright R_c)\}}^{\gamma_{cond}}\right)$$

$$\mathsf{WCBrw} \triangleq \exists B \colon \mathbb{N} \xrightarrow{\text{fin}} iProp \times iProp, C \colon \mathbb{N} \xrightarrow{\text{fin}} iProp, \mathcal{R} \colon iProp.$$

$$\boxed{\bullet \mathsf{ag}(\mathsf{next}(B))}^{\gamma_{brw}} * \boxed{\bullet \mathsf{ag}(\mathsf{next}(C))}^{\gamma_{cond}} * \boxed{\bullet Ex(\mathsf{dom}(C))}^{\gamma_{cinvset}} * \boxed{\bullet Ex(\blacktriangleright \mathcal{R})}^{\gamma_{cinv}} *$$

$$\left(\mathop{\Large *}_{i \mapsto (P,R) \in B} \left(\triangleright P * \boxed{\{i\}}^{D} \vee \boxed{\{i\}}^{E}\right) * \Box(\triangleright P \mathrel{-\!\!*} \triangleright R)\right) * \left(\left(\mathop{\Large *}_{(\_,R) \in B} \triangleright R\right) * \left(\mathop{\Large *}_{R_c \in C} \triangleright R_c\right) \mathrel{-\!\!*} \triangleright \mathcal{R}\right)$$

Fig. 8. Model of crash borrows and crash conditions.

the value in the physical disk depends on willWB. If willWB, then the current value in the physical disk is unknown but also unimportant because it will eventually become v; otherwise, the value in the physical disk is v_c. The connection between $\mathsf{adp}(B, l, v_c, v)$ and $l \mapsto^{\mathrm{d}} [v_c]v$ is enforced by the authoritative resource algebra.

D Distributed System

$$\begin{aligned}
\mathsf{specn}^{\gamma}_{t}(e) \triangleq\ & \exists k, r.\, \boxed{\{k, r, t\}}^{\gamma} * \mathsf{loop}(k\ r, e) \\
& \mathsf{loop} \triangleq \mathsf{lfp}\ \mathsf{loop}\ (e_0, e).\ (\forall K.\, \mathsf{spec}(K[e_0]) \mathrel{-\!\!*} \Rrightarrow_{\perp} \mathsf{spec}(K[e])) \\
& \vee (\forall K.\, \mathsf{spec}(K[e_0]) \mathrel{-\!\!*} \exists H, t, M.\, \mathsf{spec}(K[\mathtt{eff}\ (\mathtt{recv}, t)\ H]) * t \rightsquigarrow^{\mathrm{lb}} M * \mathsf{loop}(H[\mathtt{inl}\ ()], e)) \\
& \vee (\forall K.\, \mathsf{spec}(K[e_0]) \mathrel{-\!\!*} \exists H, s, t, m, M.\, \mathsf{spec}(K[\mathtt{eff}\ (\mathtt{recv}, t)\ H]) * t \rightsquigarrow^{\mathrm{lb}} M * (s, t, m) \in M \\
& \qquad * \mathsf{loop}(H[\mathtt{inr}\ (s, t, m)], e))
\end{aligned}$$

It also uses the evidence accumulation technique described in §4.3, but for a given thread, in addition to accumulating the evidence it can be executed to e via pure steps, we now also need to accumulate the evidence that the thread may raise recv effects later on while executing to e. Intuitively, the point is that if a recv executed at some point x can return a message m, then if we delay that recv to some later time x', it is still possible for it to return that same message m. This is because the set of messages is monotonically growing. This evidence that a later recv can return a given message is accumulated in the least fixed point loop. Intuitively, $\mathsf{loop}(e_0, e)$ allows e_0 to execute to e via three ways: (1) Some pure steps. (2) First raising a recv effect that receives nothing and then continuing with the result of recv. (3) First raising a recv effect that receives some messages (s, t, m) and then continuing with the result of recv. Assertion $t \rightsquigarrow^{\mathrm{lb}} M$ in cases (2) and (3) is a lower-bound resource of $t \rightsquigarrow \cdot$, meaning that M is a subset of messages that have ever been sent to address t.

$$\mathsf{ADISK\_LOAD}(u,\Phi) \triangleq \exists l, v_c, v.\ u = (\texttt{adisk\_load}, l) * l \mapsto^{\mathrm{d}} [v_c]v * (l \mapsto^{\mathrm{d}} [v_c]v \mathrel{-\!\!*} \Phi(v))$$

$$\mathsf{ADISK\_STORE}(u,\Phi) \triangleq \exists l, v_c, v, w.\ u = (\texttt{adisk\_store}, (l, w)) * l \mapsto^{\mathrm{d}} [v_c]v * (\forall v_c' \in \{w, v_c'\}.\ l \mapsto^{\mathrm{d}} [v_c']w \mathrel{-\!\!*} \Phi(()))$$

$$\mathsf{BARRIER}(u,\Phi) \triangleq \exists M.\ u = (\texttt{barrier}, ()) * \left(\mathop{\Large *}_{l \mapsto (v_c, v) \in m} l \mapsto^{\mathrm{d}} [v_c]v \right) * \left(\left(\mathop{\Large *}_{l \mapsto (v_c, v) \in m} v_c = v * l \mapsto^{\mathrm{d}} [v_c]v \right) \mathrel{-\!\!*} \Phi(()) \right)$$

(a) Protocol.

$\mathsf{run}_{\mathsf{adisk}} \triangleq \lambda main.$

```
1  write ∅;
2  try main () with ret v ⇒ v | v k ⇒ match v with
3   (adisk_load, l) ⇒ let buf := read in
4       (if l ∈ buf then k (fst buf[l])
5       else let p := newproph, v := disk_load l in write ([l ↦ (v, p)]buf); k (v))
6  | (adisk_store, (l, w)) ⇒ let buf := read in
7       (if l ∈ buf then resolve_proph (snd buf[l]) to false);
8       let p := newproph in write([l ↦ (w, p)]buf); k ()
9  | (barrier, ()) ⇒ let buf := read in
10      iter (λl (v, p). resolve_proph p to true; disk_store l v) buf; write ∅
11 | (η, v) ⇒ do (η, v)
```

(b) Implementation.

$$\mathsf{willWB}(\vec{v}) \triangleq \exists \vec{v}'.\ \vec{v} = ((), \texttt{true}) :: \vec{v}' \qquad \mathsf{adp}(B, l, v_c, v) \triangleq l \notin B * l \mapsto^{\mathrm{d}} v * v_c = v \lor$$

$$\exists p.\ B[l] = (v, p) * \exists \vec{v}.\ \mathsf{proph}\ (p, \vec{v}) * (\text{if } \mathsf{willWB}(\vec{v}) \text{ then } (\exists x.\ l \mapsto^{\mathrm{d}} x) * v_c = v \text{ else } l \mapsto^{\mathrm{d}} v_c)$$

(c) Verification.

Fig. 9. Asynchronous Disk.