A Relational Separation Logic for Effect Handlers

PAULO EMÍLIO DE VILHENA, Imperial College London, United Kingdom
SIMCHA VAN COLLEM, Radboud University, The Netherlands
INES WRIGHT, Aarhus University, Denmark
ROBBERT KREBBERS, Radboud University, The Netherlands

Effect handlers offer a powerful and relatively simple mechanism for controlling a program's flow of execution. Since their introduction, an impressive array of verification tools for effect handlers has been developed. However, to this day, no framework can express and prove *relational properties* about programs that use effect handlers in languages such as OCaml and Links, where programming features like mutable state and concurrency are readily available. To this end, we introduce *blaze*, the first *relational separation logic* for effect handlers. We build blaze on top of the Iris framework for concurrent separation logic in Rocq, thereby enjoying the rigour of a mechanised theory and all the reasoning properties of a modern fully-fledged concurrent separation logic, such as modular reasoning about stateful concurrent programs and the ability to introduce user-defined ghost state. In addition to familiar reasoning rules, such as the bind rule and the frame rule, blaze offers rules to reason modularly about programs that perform and handle effects. Significantly, when verifying that two programs are related, blaze *does not* require that effects and handlers from one program be in correspondence with effects and handlers from the other. To assess this flexibility, we conduct a number of case studies: most noticeably, we show how different implementations of an asynchronous programming library using effects are related to *truly-concurrent* implementations. As side contributions, we introduce two new, simple, and general reasoning rules for concurrent relational separation logic that are independent of effects: a *logical-fork rule* that allows one to reason about an arbitrary program phrase as if it had been spawned as a thread and a *thread-swap rule* that allows one to reason about how threads are scheduled.

1 Introduction

Effect handlers [38] are a powerful programming abstraction that separates the use of an effect from its implementation, allowing programmers to write effectful code independently of how these effects are implemented. Its programming interface offers the ability to *perform* and to *handle* effects. Performing an effect is similar to raising an exception: execution is suspended and control is transferred to an enclosing pre-installed handler. Handling an effect is also similar to handling an exception with the key difference that, in addition to the effect's payload, the effect handler also has access to a first-class representation of the suspended program, a *continuation*. When invoked, the continuation resumes the suspended program, but, as a first-class value, the continuation can also be discarded or stored in memory to be invoked later.

The ability to suspend and resume programs can be used to implement interesting features such as coroutines [13] and promise-style asynchronous-programming libraries [17]. However, the ability to manipulate continuations is also dangerous. A continuation can capture resources. It may also contain code that must eventually be called to free up these resources. So, if the continuation is discarded, if it becomes unreachable, or, if for some other reason it is not invoked, some resources may never be released. Users of effect handlers must also make sure that the operation of performing an effect is always enclosed by a handler, otherwise, like an uncaught exception, an *unhandled effect* would cause a runtime error. For these reasons, effect handlers are widely seen as an advanced programming feature to be used with great care.

Authors' Contact Information: Paulo Emílio de Vilhena, p.de-vilhena@imperial.ac.uk, Imperial College London, London, United Kingdom; Simcha Van Collem, simcha.vancollem@ru.nl, Radboud University, Nijmegen, The Netherlands; Ines Wright, Aarhus University, Aarhus, Denmark; Robbert Krebbers, mail@robbertkrebbers.nl, Radboud University, Nijmegen, The Netherlands.

An impressive range of tools to help programmers to reason about programs with effect handlers and to avoid these programming errors has thus been introduced. Programming languages such as Koka [32], Links [12, 22] and Effekt [9], for example, have type systems that statically ensure *effect safety*, the absence of unhandled effects. Multiple other type systems with similar guarantees, covering a fairly comprehensive portion of the design space of effect handlers, can be found in the literature [4–7, 10, 16, 27, 34, 43, 44, 50, 53].

In this paper, we are interested in expressing and verifying *relational properties* of programs with handlers, namely *program refinement* and *program equivalence*. These relational properties have several interesting applications. One could specify a complex but efficient algorithm or data structure in terms of a simple but inefficient counterpart, or express the correctness condition of *linearizability* for concurrent programs using program refinement [18]. Relational reasoning also plays a key role in compiler verification [2, 21].

The study of relational properties of programs with effect handlers is not new. Building on a logic to reason about equality of programs using effects described by an *algebraic theory* [37], Plotkin and Pretnar [39] introduce the notion of correctness of an effect handler as a relational property: the handler implementation must validate the equations of the corresponding algebraic theory. This seminal work has spawned a fertile investigation of relational logics for effect handlers [35, 41].

In prior work, however, relational reasoning is limited to a strictly functional setting deprived of imperative features. This limitation precludes the application of relational reasoning to programming languages like Links and OCaml, which, in addition to user-defined effects and handlers, have support for programming features such as concurrency and mutable state. Moreover, although user-defined effects and handlers offer a modular basis for effectful programming, it is often the case that the handler-based implementation of an effect is obscured by the combination of advanced programming patterns, whereas its handler-free implementation can be derived directly using imperative features. Therefore, from a reasoning perspective, it is desirable to establish a formal statement relating a user-defined effect to its imperative counterpart. For example, can the the operation `perform Fork task`, which performs the user-defined effect `Fork`, be seen as the operation `fork (task())`, which directly spawns a new thread?

To overcome this limitation and to address this question, we introduce *blaze*, the first relational logic for a language supporting effect handlers, heap-allocated state, and primitive concurrency and also the first *relational separation logic* for effect handlers. We build blaze on top of the Iris framework [24–26, 28–30] in the Rocq prover [47], thus providing users with the comfort of a proof assistant, the confidence of a mechanised theory, and the expressiveness of Iris, a modern higher-order concurrent separation logic with powerful features such as support for higher-order functions, user-defined ghost state, and invariants.

The blaze logic in a nutshell. The *refinement relation* $e_l \precsim e_r \langle \mathcal{T} \rangle \{R\}$ of blaze informally states that either e_l diverges, or both e_l and e_r terminate with values v_l and v_r that satisfy the postcondition R. The key ingredient is the parameter $\mathcal{T}$, which specifies the *relational theory* under which the refinement holds. This notion is inspired by Biernacki et al. [5], but, whereas they use pure (step-indexed) logic to express relational theories, we use separation logic, and, whereas they reason at the level of a *transparent* logical interpretation of types, we rely on abstract reasoning rules to manipulate an *opaque* notion of refinement. Separation logic allows us to express relational properties that involve the primitive effects of the language, and allows us to make the relations conditional on the ownership of locations in the heap or Iris-style ghost state.

Using relational theories we can relate user-defined effects to other user-defined effects, or relate user-defined effects to the native imperative features of the language. Concrete examples include:

(1) Relating the *state effect* to the composition of *reader* and *writer effects*.

(2) Relating the *state effect* effect to primitive load and store operations (§2).
(3) Relating a handler-based implementation of *concurrency* to *true concurrency* (§5.1).
(4) Expressing *algebraic laws*, for example that the non-deterministic choice operator (either implemented using a handler that collects a list of results or implemented using concurrency) satisfies monoid laws (§5.2).

Biernacki et al. [5, §4.2] already support (1). We port their result to blaze as part of our Rocq formalisation [1]. More crucially, by using separation logic to formulate our relational theories, blaze also supports (2) and (3). Another application of blaze is (4), which enables the formulation of a handler-correctness criterion in the style of Plotkin and Pretnar [39]. Expressing handler correctness in this style is novel in the context of primitive effects provided by the language.

We give a semantics to the refinement relation $e_l \lesssim e_r \langle \mathcal{T} \rangle \{R\}$ using an interpretation in Iris. At the basis, we use Iris's weakest precondition assertion to define *observational refinement* in the same way as ReLoC [20, §7.1]. Then, taking inspiration from Pitts and Stark [36]'s *biorthogonality* technique (used for the first time by Biernacki et al. [5] in the context of effect handlers) we define refinement, mutually inductively with two other relations, using Iris's guarded fixpoint operator.

While this layering of definitions makes it possible to bootstrap blaze, it also makes it infeasible to carry out refinement proofs directly by unfolding these definitions, let alone carry out these proofs in a compositional manner. We therefore take inspiration from ReLoC [19, 20] and Simuliris [21] to develop a *relational logic* with a range of high-level reasoning principles that abstract over the details of these definitions. Our high-level logic provides a number of novel features:

(1) Our novel *introduction* and *exhaustion rules* make it possible to abstractly manipulate a relational theory $\mathcal{T}$. If $\mathcal{T}$ contains a relation between e_l and e_r, then the introduction rule allows us to prove $e_l \lesssim e_r \langle \mathcal{T} \rangle \{R\}$. The exhaustion rule allows us to eliminate the dependency on a theory $\mathcal{T}$, provided that the relations included in $\mathcal{T}$ are correctly handled.
(2) The *bind rule* makes it possible to focus on a subexpression and then continue with the verification of the whole expression in which the subexpression are replaced with a value. It is well known in the context of logics for effect handlers that a restriction on the bind rule is necessary for soundness. We develop a new restriction that requires the bound contexts to be *traversable* with respect to the relational theory $\mathcal{T}$. This flexibility is crucial to support dynamic effects labels.

We show the versatility of our approach through various extensions. We add support for dynamic labels in the style of OCaml's `let exception` construct, following de Vilhena and Pottier [16]. Moreover, we add support for both one-shot and multi-shot continuations, taking inspiration from van Rooij and Krebbers [50]. Finally, as a side contribution needed to carry out some of our case studies, we introduce new relational rules for concurrency. These rules are independent of effect handlers and hold in any Iris-style relational logic such as ReLoC [19, 20].

Contributions. In sum, our contributions are the following:

(1) **Novel relational logic.** We introduce blaze, the first relational separation logic for handlers.
(2) **Case studies.** We conduct several challenging case studies including the verification that multiple effect-handler-based implementations of concurrency refine *truly concurrent* ones.
(3) **Novel reasoning rules.** Our case studies led us to discover novel, simple, and general reasoning rules in relational concurrent separation logic that are independent of handlers.
(4) **Correctness with respect to algebraic theories.** We show how the correctness of an effect handler with respect to an algebraic theory can be stated and proved in blaze.
(5) **Mechanised theory.** We mechanise all our results, including soundness, in the Rocq prover.

2 Overview

In this section, we discuss the main challenges in designing a relational separation logic with support for effect handlers. Our goal is to informally explain how blaze handles these challenges. The examples are written in λ-blaze, a calculus whose syntax and semantics we explain in §3. In this section, we assume familiarity with functional programming and effect handlers. For the unaccustomed reader, Pretnar [40] provides a tutorial introduction to effect handlers.

Let us start by considering the following example:

```
countdown ≜ fun timer.
  timer.set 10; while (timer.get() > 0) {timer.set (timer.get() - 1)}
```

The function *countdown* receives an object *timer* as an argument with two fields, `get` and `set`. It assumes these fields implement the functionality to respectively access and update *timer*'s memory. It uses this functionality to update the timer from `10` to `0` through decrements of `1`.

The definition of *countdown* is modular on the implementation of the timer. In a language with effect handlers, the programmer can exploit this generality by implementing `get` and `set` as user-defined effects and providing different handlers to customise the implementation of the effects performed by `get` and `set`. For example, assuming an effect `$Timer` is available, a generic implementation of `get` and `set` can be obtained as follows:

```
timer ≜ {get = fun _. perform $Timer (inl ()); set = fun y. perform $Timer (inr y)}
```

In this definition, the operations `get` and `set` simply perform the effect `$Timer`. They use *left* and *right injections* **`inl`** and **`inr`** to distinguish between a request sent by `get` and a request sent by `set`. A `$Timer` handler eventually assigns meaning to `get` and `set` by replying to these requests. Here are two possible instances of such handlers:

```
run_st_passing ≜ fun main.
  let run = fun ().
    handle main() with
    | effect $Timer request, k ⇒ fun x.
        match request with
        | inl () ⇒ k x x
        | inr y ⇒ k () y
    | y ⇒ fun _. y
  in run() 0
```

```
run_heap ≜ fun main.
  let r = ref 0 in
  handle main() with
  | effect $Timer request, k ⇒
      match request with
      | inl () ⇒ k (!r)
      | inr y ⇒ r ← y; k ()
  | y ⇒ y
```

Both receive a piece of client code *main* that performs `$Timer` effects. The function *run_st_passing* installs a handler that interprets `$Timer` effects in *state-passing style*, whereby the computation is transformed into a function that takes the current state of the timer and outputs the timer's final state. In contrast, the function *run_heap* interprets `$Timer` effects by storing the current state of the timer in a local reference r. This implementation is arguably simpler than the state-passing implementation of *run_st_passing* although presumably they implement the same functionality.

This observation motivates a key question: is it possible to show that *run_st_passing* is a *refinement* of *run_heap*? That is, can *run_heap* be used as a *specification* of *run_st_passing* and, therefore, as a *reference implementation* of `get` and `set`?

The notion of refinement is formalised in relational logics as the relation $e_l \precsim e_r \{R\}$, where e_l and e_r are expressions and the *postcondition* R is a relation on values. We refer to e_l as the expression on the *implementation side* and to e_r as the expression on the *specification side*. The refinement relation informally states that either e_l diverges or both e_l and e_r terminate with outputs v_l and v_r such that $R(v_l, v_r)$ holds, capturing the intuition that e_l implements the same functionality described by e_r, because, informally, every output of e_l corresponds to an output of e_r related by R.

The question can thus be reformulated as how to establish a refinement between *run_st_passing* and *run_heap*, such as the statement

$$\mathit{impl}_1 \precsim \mathit{impl}_2\ \{y_l\, y_r.\ y_l = y_r\}, \quad \begin{array}{rr} \text{where } \mathit{impl}_1 \triangleq & \mathit{run\_st\_passing}\,(\texttt{fun}\,().\ \mathit{countdown\ timer}) \\ \text{and } \mathit{impl}_2 \triangleq & \mathit{run\_heap}\,(\texttt{fun}\,().\ \mathit{countdown\ timer}), \end{array} \tag{1}$$

expressing the property that impl_1 and impl_2 have the same outputs.

To our knowledge, there are no relational logics with support for effect handlers *and* heap-allocated mutable state and therefore no logics where such a relation can be derived. Addressing this gap, we introduce blaze, the first relational separation logic with support for handlers. The choice of a separation logic enables modular reasoning about state-manipulating programs such as *run_heap*. The following subsections explain other interesting and novel aspects of blaze.

2.1 Modular reasoning about effects: handler versus handlee

In blaze, it is possible to state and prove Refinement 1. In fact, this refinement can be established in a *compositional* way, whereby the proof is split into two parts: a proof that the handlers installed by *run_st_passing* and *run_heap* are related and a proof that the handlees monitored by these handlers are related. In the current example, this creates the following two subgoals:

$$\mathit{countdown\ timer} \overset{?}{\precsim} \mathit{countdown\ timer} \tag{2}$$

$$\forall \mathit{main}_l,\ \mathit{main}_r.\ \mathit{main}_l\,() \overset{?}{\precsim} \mathit{main}_r\,() \mathrel{-\!\!*} \mathit{run\_st\_passing}\ \mathit{main}_l \precsim \mathit{run\_heap}\ \mathit{main}_r\ \{=\} \tag{3}$$

Refinement 2 relates the handlees and Refinement 3 relates *run_st_passing* and *run_heap* under the assumption they receive related arguments. For brevity, we write "=" in Refinement 3 for the postcondition $y_l\, y_r.\ y_l = y_r$. Moreover, we use $\overset{?}{\precsim}$ to denote a notion of relation that is yet to be defined. Recall that the informal reading of $\mathit{countdown\ timer} \precsim \_\ \{\_\}$ states *countdown timer* either diverges or terminates with a value. The standard notion of refinement $\_ \precsim \_\ \{\_\}$ is insufficient because, without a handler, the program *countdown timer* performs unhandled effects.

This limitation reveals a *key challenge*: to reason about the handlee independently of the handler, it is necessary to generalise the standard notion of refinement to account for unhandled effects.

The blaze logic solves this challenge by parameterising the refinement relation with a *relational theory*. A relational theory can be seen as a set of assumed refinements. Concretely, it is defined as a set of triples $(e_l,\ e_r,\ Q)$, where e_l and e_r are expressions and Q is a relation on pairs of expressions called the *return condition*. In short, the return condition describes the condition under which e_l and e_r can return. For example, the return condition $y_l\, y_r.\ y_l = y_r$ states e_l and e_r can return only when they terminate with the same values. In this case, the return condition Q can be seen as a postcondition. For the purposes of this section, this first approximation is enough as the reading of $(e_l,\ e_r,\ Q)$ then simply states e_l refines e_r with postcondition Q.

The general refinement relation in blaze has the form $e_l \precsim e_r\ \langle \mathcal{T} \rangle\ \{R\}$, where $\mathcal{T}$ is the parameterised relational theory. When a relational theory is empty, we write $e_l \precsim e_r\ \{R\}$ which has the same informal meaning as before. The informal reading of the general relation $e_l \precsim e_r\ \langle \mathcal{T} \rangle\ \{R\}$ states $K_l[e_l] \precsim K_r[e_r]\ \{R\}$ holds for every pair of contexts K_l and K_r that *validate the theory* $\mathcal{T}$. A pair of contexts K_l and K_r validate $\mathcal{T}$ when the refinements included in $\mathcal{T}$ hold under K_l and K_r; that is, if $\mathcal{T}$ relates two expressions e_l and e_r, then $K_l[e_l]$ refines $K_r[e_r]$. This general notion of refinement allows us to reason about programs e_l and e_r that perform unhandled effects, because, when $\mathcal{T}$ is well-chosen, the contexts K_l and K_r that validate $\mathcal{T}$ are precisely those that handle the effects performed by e_l and e_r. At the same time, the contexts K_l and K_r appear only in the definition of $e_l \precsim e_r\ \langle \mathcal{T} \rangle\ \{R\}$, which, during a verification task, need not be unfolded. The theory $\mathcal{T}$ can thus be seen as a logical abstraction of the contexts under which e_l and e_r occur.

In the running example of Refinements 2 and 3, it is now possible to substitute $\overset{?}{\lesssim}$ with a refinement relation parameterised by a relational theory, say $\mathit{Timer}_{\mathit{refl}}$:

$$\_ \overset{?}{\lesssim} \_ \quad \triangleq \quad \_ \lesssim \_ \, \langle \mathit{Timer}_{\mathit{refl}} \rangle \, \{=\}$$

There are two minimal requirements for the relational theory $\mathit{Timer}_{\mathit{refl}}$: (1) $\mathit{Timer}_{\mathit{refl}}$ must include sufficiently many relations so that $\mathit{countdown\ timer} \lesssim \mathit{countdown\ timer}\ \langle \mathit{Timer}_{\mathit{refl}} \rangle\ \{=\}$ holds and (2) $\mathit{Timer}_{\mathit{refl}}$ must be sufficiently small so that $\mathit{run\_st\_passing}\ \mathit{main}_l \lesssim \mathit{run\_heap}\ \mathit{main}_r\ \{=\}$ holds under the assumption that $\mathit{main}_l\,() \lesssim \mathit{main}_r\,()\ \langle \mathit{Timer}_{\mathit{refl}} \rangle\ \{=\}$ holds. A choice of $\mathit{Timer}_{\mathit{refl}}$ that satisfies both requirements is one that includes only the following refinement:

$$\forall v.\ \mathbf{perform}\ \mathtt{\$Timer}\ v \lesssim \mathbf{perform}\ \mathtt{\$Timer}\ v\ \langle \mathit{Timer}_{\mathit{refl}} \rangle\ \{=\} \tag{4}$$

This is sufficient to prove $\mathit{countdown\ timer} \lesssim \mathit{countdown\ timer}\ \langle \mathit{Timer}_{\mathit{refl}} \rangle\ \{=\}$, because, as the two expressions in this relation are same, every `$Timer` effect on one side corresponds to exactly one `$Timer` effect on the other side. Therefore, when reasoning about performing an effect, it suffices to apply Refinement 4 to conclude that, in both expressions, the results are the same.

Moreover, because $\mathit{Timer}_{\mathit{refl}}$ includes only Refinement 4, it follows that, if $e_l \lesssim e_r\ \langle \mathit{Timer}_{\mathit{refl}} \rangle\ \{=\}$ holds for arbitrary expressions e_l and e_r, then it must be the case that every `$Timer` effect in e_l corresponds to exactly one `$Timer` effect in e_r. This assumption can be exploited by the proof of $\mathit{run\_st\_passing}\ \mathit{main}_l \lesssim \mathit{run\_heap}\ \mathit{main}_r\ \{=\}$ to establish the relation between the two handlers.

2.2 Flexible reasoning: handler-based versus handler-free implementations

One of the motivations to establish the refinement between $\mathit{run\_st\_passing}$ and $\mathit{run\_heap}$ is that $\mathit{run\_heap}$ provides a simpler and more direct implementation of the timer when compared to the state-passing implementation of $\mathit{run\_st\_passing}$. However, $\mathit{run\_heap}$ does not exploit non-trivial functionalities of effect handlers as the effect branch always immediately resumes the continuation. This observation permits an implementation of the timer without effects:

$$\mathit{ref\_timer} \triangleq \mathbf{fun}\, r.\ \{\mathtt{get} = \mathbf{fun}\,().\ !r;\ \mathtt{set} = \mathbf{fun}\, y.\ r \leftarrow y\}$$

The question now is: can the refinement

$$\mathit{impl}_1 \lesssim \mathit{impl}_3\ \{=\},\ \text{where}\ \mathit{impl}_3 \triangleq \mathbf{let}\, r = \mathbf{ref}\, \mathtt{0}\ \mathbf{in}\ \mathit{countdown}\ (\mathit{ref\_timer}\, r), \tag{5}$$

be established in blaze? Moreover, if possible, can the proof be done in a compositional way like in the previous example, where reasoning about handlee and handler are carried out independently?

The answers to both previous questions are positive: the refinement can be established in blaze with a compositional proof. Indeed, the proof of $\mathit{impl}_1 \lesssim \mathit{impl}_3\ \{=\}$ is split into two subgoals:

$$\forall \ell.\ \ell \overset{1/2}{\mapsto}_{\mathtt{s}} \mathtt{0} \mathrel{-\!\!*} \mathit{countdown\ timer} \lesssim \mathit{countdown}\ (\mathit{ref\_timer}\ \ell)\ \langle \mathit{Timer}^{\ell}_{\mathit{spec}} \rangle\ \{=\} \tag{6}$$

$$\forall \ell, \mathit{main}_l, e_r.\ \ell \overset{1/2}{\mapsto}_{\mathtt{s}} \mathtt{0} \mathrel{-\!\!*} \mathit{main}_l\,() \lesssim e_r\ \langle \mathit{Timer}^{\ell}_{\mathit{spec}} \rangle\ \{=\} \mathrel{-\!\!*} \mathit{run\_st\_passing}\ \mathit{main}_l \lesssim e_r\ \{=\} \tag{7}$$

Refinement 6 relates the handlee $\mathit{countdown\ timer}$ to $\mathit{countdown}\ (\mathit{ref\_timer}\ \ell)$ under the theory $\mathit{Timer}^{\ell}_{\mathit{spec}}$, which we introduce shortly. Refinement 7 is stated in an interesting way. It relates $\mathit{run\_st\_passing}\ \mathit{main}_l$ to an arbitrary expression e_r. Intuitively, the expression represents the program $\mathit{countdown}\ (\mathit{ref\_timer}\ \ell)$, but, thanks to the theory $\mathit{Timer}^{\ell}_{\mathit{spec}}$, this specific program can be entirely abstracted: all the information needed to carry out Refinement 7 is that main_l refines e_r under $\mathit{Timer}^{\ell}_{\mathit{spec}}$.

The variable ℓ stands for the location bound by r in impl_3. As usual in relational separation logics, each of the two programs in a refinement relation manipulates its own heap. The *points-to predicate* $\_ \mapsto_{\mathtt{s}} \_$ describes the state of the heap of the program on the specification side, whereas $\_ \mapsto_{\mathtt{i}} \_$ describes the state of the heap of the implementation side. The fraction that appears on top of $\mapsto_{\mathtt{s}}$ in this relation represents a *fractional ownership* of ℓ: it grants read-only permission to ℓ. Full ownership can be retrieved by combining two $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} \_$ assertions. In the proof of 5, fractional assertions $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} \_$ are given to both the handlee and the handler. Full ownership is therefore retrieved when the handlee performs an effect and ownership of the handlee's fractional assertion $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} \_$ is temporarily transferred to the handler until the handlee is resumed.

Like $\mathit{Timer}_{\mathit{refl}}$, the theory $\mathit{Timer}^{\ell}_{\mathit{spec}}$ must fulfil two requirements: (1) the theory must be sufficiently relaxed so that 6 can be established and (2) it must be sufficiently small so that the terms $\mathit{main}_l\,()$ and e_r in 7 are tightly related. The first requirement now seems particularly challenging because Refinement 6 relates an effectful program to a non-effectful one. Fortunately, relational theories are not limited to relations between only effectful expressions like in $\mathit{Timer}_{\mathit{refl}}$. They can in fact express relations between arbitrary expressions. Taking advantage of this flexibility, the $\mathit{Timer}^{\ell}_{\mathit{spec}}$ includes a relation between the effectful implementation of `get` and `set` fields of *timer* and their heap-manipulating counterparts of *ref_timer*:

$$\forall x.\ \ell \overset{1/2}{\mapsto}_{\mathsf{s}} x \mathrel{-\!\!*} \texttt{perform}\ \texttt{\$Timer}\ (\texttt{inl}\ ()) \precsim\ !\ell\ \langle \mathit{Timer}^{\ell}_{\mathit{spec}} \rangle\ \{y_l\, y_r.\ y_l = y_r = x\ *\ \ell \overset{1/2}{\mapsto}_{\mathsf{s}} x\} \qquad (8)$$

$$\forall y.\ \ell \overset{1/2}{\mapsto}_{\mathsf{s}} \_ \mathrel{-\!\!*} \texttt{perform}\ \texttt{\$Timer}\ (\texttt{inr}\ y) \precsim \ell \leftarrow y\ \langle \mathit{Timer}^{\ell}_{\mathit{spec}} \rangle\ \{y_l\, y_r.\ y_l = y_r = ()\ *\ \ell \overset{1/2}{\mapsto}_{\mathsf{s}} y\} \qquad (9)$$

From the perspective of the handlee, these relations guarantee that performing the effect `$Timer` is similar to manipulating the memory location ℓ.

2.3 Context-local relational reasoning

A closer look at Refinements 8 and 9 reveals an important limitation. They apply only to pairs of programs e_l and e_r where e_l consists precisely of a single `$Timer` effect and e_r consists precisely of a single read or store operation. As such, they are insufficient to establish 6, because the calls to `get` and `set` in *countdown timer* and in *countdown* (*ref_timer* ℓ) occur in the context of a larger program, not as single operations.

The key missing principle to address this limitation is the *bind rule*. The bind rule allows the user to reason about a piece of code independently of the context under which this code is eventually executed. In standard relational logics, the bind rule is formally stated as follows:

$$\textsc{standard-bind} \qquad e_l \precsim e_r\ \{y_l\, y_r.\ K_l[y_l] \precsim K_r[y_r]\ \{R\}\} \vdash K_l[e_l] \precsim K_r[e_r]\ \{R\}$$

This rule is sound in blaze, but insufficient because it assumes the parameterised theory is empty. A natural fix would be to decorate every occurrence of the refinement relation with a theory $\mathcal{T}$:

UNSOUND-BIND ~~$e_l \precsim e_r\ \langle \mathcal{T} \rangle\ \{y_l\, y_r.\ K_l[y_l] \precsim K_r[y_r]\ \langle \mathcal{T} \rangle\ \{R\}\} \vdash K_l[e_l] \precsim K_r[e_r]\ \langle \mathcal{T} \rangle\ \{R\}$~~

The resulting rule is *unsound*. To see why, it suffices to consider the following counterexample, where the effect `$Id` is assumed to be available:

$$e_{\mathit{true}} \triangleq \texttt{handle}\ (\texttt{perform}\ \texttt{\$Id}\ \texttt{true})\ \texttt{with}\,\texttt{effect}\ \texttt{\$Id}\ x,\ k \Rightarrow k\,x \mid y \Rightarrow y$$

If we further assume there is a theory Neq that includes the refinement $\forall b \in \mathit{Bool}.\ \texttt{perform}\ \texttt{\$Id}\ b \precsim \texttt{perform}\ \texttt{\$Id}\ b\ \langle \mathit{Neq} \rangle\ \{\neq\}$, then, using UNSOUND-BIND with both K_l and K_r instantiated as the `$Id` handler, it is possible to establish the refinement $e_{\mathit{true}} \precsim e_{\mathit{true}}\ \langle \mathit{Neq} \rangle\ \{\neq\}$, which is false, because both sides of the refinement terminate with `true`.

This counterexample suggests that to enable sound context-local reasoning, there must some restriction on the evaluation contexts K_l and K_r. In particular, the rule should not be applicable when the contexts K_l and K_r contain handlers for effects described by the theory $\mathcal{T}$. In blaze, a sound bind rule integrating these restrictions is formulated as follows:

$$\textsc{bind} \qquad \begin{array}{c} \mathit{traversable}(K_l,\, K_r,\, \mathcal{T}) \qquad \mathcal{T} \sqsubseteq \mathcal{F} \\ e_l \precsim e_r\ \langle \mathcal{T} \rangle\ \{y_l\, y_r.\ K_l[y_l] \precsim K_r[y_r]\ \langle \mathcal{F} \rangle\ \{R\}\} \end{array} \vdash K_l[e_l] \precsim K_r[e_r]\ \langle \mathcal{F} \rangle\ \{R\}$$

$$\begin{aligned}
e ::=\ & v \mid x \mid e\, e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
\mid\ & \texttt{let effect}\ \mathsf{E}\ \texttt{in}\ e \mid \texttt{perform}\ \eta\ e \\
& \texttt{handle}\ e\ \texttt{with} \\
\mid\ & \texttt{| effect}\ \eta\ x,\ k \Rightarrow e \\
& \texttt{|}\ y \Rightarrow e \\
\mid\ & \texttt{ref}\ e \mid\ !e \mid e \leftarrow e \mid \texttt{fork}\ e \\
\eta ::=\ & \mathsf{E} \mid \$\mathsf{E}
\end{aligned}
\qquad
\begin{aligned}
v ::=\ & () \mid \texttt{rec}\ f\ x.\ e \mid (v,\ v) \mid \ell \mid \texttt{kont}\ K \\
K ::=\ & [] \mid e\, K \mid K\, v \\
\mid\ & \texttt{let}\ x = K\ \texttt{in}\ e \mid \texttt{let}\ x = v\ \texttt{in}\ K \\
& \texttt{handle}\ K\ \texttt{with} \\
\mid\ & \texttt{| effect}\ \$\mathsf{E}\ x,\ k \Rightarrow e \\
& \texttt{|}\ y \Rightarrow e \\
\mid\ & \texttt{perform}\ \$\mathsf{E}\ K \mid \texttt{ref}\ K \mid\ !K \mid e \leftarrow K \mid K \leftarrow v
\end{aligned}$$

(a) Syntax of expressions, values, and evaluation contexts. (Runtime terms are displayed in gray.)

EFFECT

$$\{\vec{e}[i \mapsto K[\texttt{let effect}\ \mathsf{E}\ \texttt{in}\ e]]\,;\ \sigma;\ \delta\} \quad \$\mathsf{E} \notin \delta \;\xrightarrow{\hspace{2em}}\; \{\vec{e}[i \mapsto K[e\{\$\mathsf{E}/\mathsf{E}\}]]\,;\ \sigma;\ \delta \uplus \{\$\mathsf{E}\}\}$$

FORK

$$\{\vec{e}[i \mapsto K[\texttt{fork}\ e]]\,;\ \sigma;\ \delta\} \quad n = |\vec{e}| \;\xrightarrow{\hspace{2em}}\; \{\vec{e}[i \mapsto K[()],\ n \mapsto e]\,;\ \sigma;\ \delta\}$$

ALLOC

$$\{\vec{e}[i \mapsto K[\texttt{ref}\ v]]\,;\ \sigma;\ \delta\} \quad \ell \notin \sigma \;\xrightarrow{\hspace{2em}}\; \{\vec{e}[i \mapsto K[\ell]]\,;\ \sigma[\ell \mapsto v]\,;\ \delta\}$$

PURE

$$e_1 \to_{\mathrm{p}} e_2 \quad \{\vec{e}[i \mapsto K[e_1]]\,;\ \sigma;\ \delta\} \;\xrightarrow{\hspace{2em}}\; \{\vec{e}[i \mapsto K[e_2]]\,;\ \sigma;\ \delta\}$$

(b) Operational rules.

BETA

$$(\texttt{rec}\ f\ x.\ e)\ v \to_{\mathrm{p}} e\{(\texttt{rec}\ f\ x.\ e)/f, v/x\}$$

MULTI-SHOT

$$(\texttt{kont}\ K)\ v \to_{\mathrm{p}} K[v]$$

HANDLE

$$\frac{\$\mathsf{E} \notin \mathscr{L}(K) \qquad H = \texttt{handle}\ []\ \texttt{with effect}\ \$\mathsf{E}\ x,\ k \Rightarrow h\ \texttt{|}\ y \Rightarrow r}{H[K[\texttt{perform}\ \$\mathsf{E}\ v]] \to_{\mathrm{p}} h\{v/x, \texttt{kont}\ H[K]/k\}}$$

(c) Pure-reduction rules.

Fig. 1. Syntax and semantics of λ-blaze.

The rule is applicable when there exists a theory $\mathcal{T}$ *included* in $\mathcal{F}$, such that *traversable*$(K_l,\ K_r,\ \mathcal{T})$ holds. The predicate *traversable*$(K_l,\ K_r,\ \mathcal{T})$ intuitively states that K_l and K_r do not conflict with $\mathcal{T}$, or, visually, that $\mathcal{T}$ can *traverse* K_l and K_r. It is defined in an abstract way with no reference to the handlers in K_l and K_r. In the case of $Timer_{refl}$, however, it is possible to show this predicate holds for any contexts K_l and K_r that contain no `$Timer` handler. In the case of $Timer^{\ell}_{spec}$, the predicate holds for any K_l that contains no `$Timer` handler. No condition is imposed on K_r in this case, because the expressions on the right-hand side of Refinements 8 and 9 do not perform effects. The blaze logic therefore enjoys a powerful context-local reasoning principle that is adjustable to the parameterised theory. As we are going to show in §4.2, this principle is especially important to support reasoning in the presence of multiple effect names.

3 Language

We introduce λ-*blaze*, an untyped calculus with formally defined syntax and semantics. The language has support for heap-allocated mutable state and concurrency, both deep and shallow handlers, both one-shot and multi-shot continuations, and dynamically allocated effect names. For most of the paper, only deep handlers that capture multi-shot continuations are used, so, for the sake of conciseness, we postpone the introduction of the syntax and semantics of one-shot continuations to §4.3, where we explain the extension of the logic with support for this feature.

3.1 Syntax

Figure 1a shows the syntax of expressions, values, and evaluation contexts. The definition of evaluation contexts reflects a right-to-left evaluation order. Every node in the syntax tree of an evaluation context K contains exactly one child, except for the empty context which contains none. Thanks to this observation, a context K can be seen as a list whose elements, called *frames*, are the nodes in its syntax tree. We use the notation $K[K']$ to denote the context obtained by concatenating these lists. The similar notation $K[e]$ is used to denote the expression obtained by the operation of *filling* K with e, characterised by the equations: $[\,][e] = e$ and $(K[K'])[e] = K[K'[e]]$.

Most of the syntactic constructs of the language are standard. In the following paragraphs, we explain two aspects that are unusual: the syntax of function definitions and the distinction between *effect names* and *effect labels.*

Function definitions. Functions are defined using the syntax $\mathbf{rec}\, f\, x.\ e$. The variable x is a formal argument of the function. Its scope is e. The variable f binds the function definition itself (that is, the entire term $\mathbf{rec}\, f\, x.\ e$). It can be used in the scope of e to write recursive definitions. When f does not occur in e, we use the simpler notation $\mathbf{fun}\, x.\ e$. For function definitions with more than one formal argument, we introduce the following syntactic sugar: $\mathbf{fun}\, \vec{x}.\ e \triangleq \mathbf{fun}\, x_0.\ \ldots\ \mathbf{fun}\, x_{n-1}.\ e$ and $\mathbf{rec}\, f\, \vec{x}.\ e \triangleq \mathbf{rec}\, f\, x_0.\ \mathbf{fun}\, \overrightarrow{x_1 \ldots x_{n-1}}.\ e$, where $n = |\vec{x}|$.

Effect names and effect labels. Taking inspiration from previous work [7, 16, 53], the syntax of λ-blaze makes a clear distinction between effect names E and effect labels $\mathsf{\$E}$. Effect names are binders whose scope is delimited by the construct $\mathbf{let\, effect}\ \mathsf{E}\ \mathbf{in}\ e$ (following a syntax similar to OCaml's **let exception** construct). Effect labels are the values bound by effect names. They appear at runtime after the execution of a **let effect** construct. The motivation for introducing this distinction is to provide λ-blaze with mechanisms to avoid the issue of *colliding effect names* [7, 16, 53], when the same effect name is used in two unrelated pieces of code.

To give an example, consider the following implementation of an *ask effect* [5] and the client code *colliding*, which installs an Ask handler over calls to the function it receives as an argument:

$$\begin{aligned}
\mathit{run\_ask} \;&\triangleq\; \mathbf{fun}\, x\ \mathit{main}.\ \mathbf{let\, effect}\ \mathsf{Ask}\ \mathbf{in}\ \mathbf{let}\ \mathit{ask} = \mathbf{fun}\ \_.\ \mathbf{perform}\ \mathsf{Ask}\ ()\ \mathbf{in}\\
&\qquad \mathbf{handle}\ \mathit{main}\ \mathit{ask}\ \mathbf{with\, effect}\ \mathsf{Ask}\ ()\mathbf{,}\ k \Rrightarrow k\, x\ \mathbf{|}\ z \Rrightarrow z\\[1ex]
\mathit{colliding} \;&\triangleq\; \mathbf{fun}\, \mathit{ask}_0.\ \mathbf{let\, effect}\ \mathsf{Ask}\ \mathbf{in}\ \mathbf{let}\ \mathit{ask}_1 = \mathbf{fun}\ \_.\ \mathbf{perform}\ \mathsf{Ask}\ ()\ \mathbf{in}\\
&\qquad \mathbf{handle}\ \mathit{ask}_0\,() + \mathit{ask}_1\,()\ \mathbf{with\, effect}\ \mathsf{Ask}\ ()\mathbf{,}\ k \Rrightarrow k\, 1\ \mathbf{|}\ z \Rrightarrow z
\end{aligned}$$

The collision occurs during the execution of $\mathit{run\_ask}\ \mathtt{0}\ \mathit{colliding}$. The call to ask_0 in *colliding* performs an Ask effect that should be handled by *run_ask*, but, at this moment, the innermost handler is the one installed by *colliding*. If effect names were used to find the handlers, then the call to ask_0 would be handled by *colliding*'s handler.

This example illustrates the issue of collision of effect names: the name Ask is used with different purposes by two unrelated pieces of code. It would thus be desirable for the semantics of $\mathbf{let\, effect}\ \mathsf{E}\ \mathbf{in}\ e$ to take care of avoiding this collision of names. This is exactly what it does: when $\mathbf{let\, effect}\ \mathsf{E}\ \mathbf{in}\ e$ is executed, it allocates a fresh *effect label* $\mathsf{\$E}$ which is substituted for E in e. At runtime, it is $\mathsf{\$E}$ that is used to perform and handle effects. According to this semantics, the program $\mathit{run\_ask}\ \mathtt{0}\ \mathit{colliding}$ runs as expected, because, at runtime, the handlers installed by *run_ask* and *colliding* handle effects for different labels.

3.2 Semantics

The semantics is defined using three sorts of runtime terms: memory locations ℓ, created by **ref** instructions; multi-shot continuations **kont** K, created by handlers during the handling of an effect; and effect labels $E, which, as explained, are created by **let effect** instructions.

Memory operations follow a standard heap semantics [25]. The semantics of handlers that capture multi-shot continuations is also standard [40]. The semantics of **let effect** follows [16].

The semantics is formalised by a number of operational rules, of which the most relevant can be seen in Figure 1b. The rules define a reduction relation between *configurations* of the form $\{\vec{e};\, \sigma;\, \delta\}$, where $\vec{e}$ is a pool of running threads, σ is a store, and δ is a set of allocated effect labels. Rules ALLOC captures the semantics of memory allocation where a fresh location ℓ is non-deterministically chosen and initialised in σ with v. Rule FORK captures the semantics of **fork** e, allocates a thread to execute e and returns (). Rule EFFECT captures the semantics of **let effect**: to guarantee freshness, the non-deterministically chosen label $E must not be in the set of pre-allocated labels δ. Finally, Rule PURE captures the semantics of *pure reductions* $e \to_{\mathsf{p}} e'$, partially defined in Figure 1c. Rule BETA is the standard beta reduction. Rule MULTI-SHOT shows how the invocation of a multi-shot continuation **kont** K restores K as an evaluation context. Rule HANDLE shows how control is transferred to the effect branch h of a handler in case of an effect. The term $\mathscr{L}(K)$ denotes the labels of the handlers in K. The condition $\$\mathsf{E} \notin \mathscr{L}(K)$ ensures H is the innermost handler.

4 Logic

The logic consists of two main layers with independent notions of refinement and independent reasoning rules. The first layer, baze, offers a base logic built directly on top of Iris [25]. The baze logic offers an expressive notion of refinement for arbitrary λ-blaze programs. Reasoning about programs with multiple effect labels in baze however can be challenging, motivating the introduction of the second layer, blaze, which is built on top of baze and tailored for programs with dynamic labels. So far, we have used the name blaze for the collection of both logics. To avoid confusion, from now on, we use the term blaze to refer exclusively to this second layer.

4.1 baze: The base logic

The refinement statement in baze takes the form $e_l \precsim e_r \langle \mathcal{T} \rangle \{R\}$, where e_l and e_r are λ-blaze programs, $\mathcal{T}$ is a parameterised relational theory, and the postcondition R is a predicate on pairs of values. It intuitively means that, under any pair of contexts K_l and K_r that *validate* the theory $\mathcal{T}$, either $K_l[e_l]$ diverges or both $K_l[e_l]$ and $K_r[e_r]$ terminate with values v_l and v_r such that $R(v_l, v_r)$ holds. The key to formalise this notion of refinement is thus to precisely formulate what is a theory $\mathcal{T}$ and what it means for a pair of contexts to validate $\mathcal{T}$.

4.1.1 Relational theories. A theory $\mathcal{T}$ is modelled as a predicate of type[1]

$$iThy \triangleq (Expr \times Expr \times (Expr \times Expr \to iProp)) \to iProp,$$

where *iProp* is the type of Iris assertions.[2] Intuitively, the assertion $\mathcal{T}(e_l, e_r, Q)$ means that e_l is related to e_r and that e_l and e_r can be replaced with any pair of expressions e'_l and e'_r for which the *return condition* $Q(e'_l, e'_r)$ holds. Perhaps the simplest example of a relational theory is the *empty theory* $\bot$, which does not include any relations: $\bot(e_l, e_r, Q) \triangleq \mathsf{False}$.

[1] This type is similar to the type of semantic rows **Eff** from Biernacki et al. [5, §3.2] and to the type of *abstract protocols* from Allain et al. [2, §6]. See §6 for an in-depth discussion.

[2] Iris assertions include standard connectives and quantifiers, separation logic connectives (in particular, the *separating conjunction* $*$ and the *separating implication* $-\!\!*$), and modalities whose purpose and meaning we explain as they appear.

$$
\begin{aligned}
O(e_l,\, e_r,\, S) &\triangleq \forall i,\, K.\ \mathit{specCtx} \mathrel{-\!\!*} i \Mapsto K[e_r] \mathrel{-\!\!*} \mathit{wp}\ e_l\ \{v_l.\ \exists v_r.\ i \Mapsto K[v_r] \ *\ S(v_l,\, v_r)\}\\
e_l \lesssim e_r\ \langle \mathcal{T} \rangle\ \{R\} &\triangleq \forall K_l,\, K_r,\, S.\ \{R\}\, K_l \lesssim K_r\ \langle \mathcal{T} \rangle\ \{S\} \mathrel{-\!\!*} O(K_l[e_l],\, K_r[e_r],\, S)\\
\{R\}\, K_l \lesssim K_r\ \langle \mathcal{T} \rangle\ \{S\} &\triangleq \bigwedge \begin{cases} \forall v_l,\, v_r.\, R(v_l,\, v_r) \mathrel{-\!\!*} O(K_l[v_l],\, K_r[v_r],\, S) \\ \forall e_l,\, e_r.\, \mathcal{T} \blacktriangleleft e_l \lesssim e_r\ \{R\} \mathrel{-\!\!*} O(K_l[e_l],\, K_r[e_r],\, S) \end{cases}\\
\mathcal{T} \blacktriangleleft e_l \lesssim e_r\ \{R\} &\triangleq \exists Q.\ \mathcal{T}(e_l,\, e_r,\, Q) \ *\ \Box \triangleright \forall e_l',\, e_r'.\ Q(e_l',\, e_r') \mathrel{-\!\!*} e_l' \lesssim e_r'\ \langle \mathcal{T} \rangle\ \{R\}
\end{aligned}
$$

Fig. 2. Model of the base logic.

For a slightly more involved example, consider the definition of theory $\mathit{Timer}_{\mathit{refl}}$ from §2.1 relating the effect `$Timer`[3] to itself and asserting that both effects return the same output:

$$\mathit{Timer}_{\mathit{refl}}(\texttt{perform}\ \texttt{\$Timer}\ v,\ \texttt{perform}\ \texttt{\$Timer}\ v,\ Q) \ \triangleq\ \Box\, \forall w \in \mathit{Val}.\ Q(w,\, w)$$

To express that both **perform** `$Timer` v operations return the same output, the theory $\mathit{Timer}_{\mathit{refl}}$ asserts the return condition Q holds of any pair of copies of the same value: $\Box\, \forall w \in \mathit{Val}.\ Q(w,\, w)$. This assertion is guarded by Iris's *persistently modality* $\Box$. Typical separation-logic assertions, such as the points-to assertion $\ell \mapsto v$, declare ownership of resources, so, by default, they cannot be arbitrarily shared or duplicated. The persistently modality indicates when an assertion does not claim ownership of ephemeral resources and thereby *can* be duplicated. Here, it is used to indicate that the effect `$Timer` complies with a multi-shot policy whereby the operation **perform** `$Timer` v can return multiple times. The return condition must hold every time the operation returns.

4.1.2 Context-closure operation. As defined, the theory $\mathit{Timer}_{\mathit{refl}}$ suffers from the limitation highlighted in §2.3: $\mathit{Timer}_{\mathit{refl}}$ is limited to relations between single **perform** expressions, when, in fact, it is desirable for the theory to enjoy some form of context-local reasoning whereby **perform** expressions can be related under evaluation contexts. More abstractly, we wish to *close a theory* $\mathcal{T}$ *under contexts*, so that, along rough lines, if $\mathcal{T}(e_l,\, e_r,\, Q)$ holds, then so does $\mathcal{T}(K_l[e_l],\, K_r[e_r],\, P)$ for some return condition P. To this end, we introduce the *context-closure* operation:

$$
\begin{aligned}
&((\mathit{ls}_l,\, \mathit{ls}_r) \downharpoonleft\!\upharpoonright \mathcal{T})(e_l,\, e_r,\, P) \ \triangleq\\
&\qquad \exists e_l',\, e_r',\, K_l,\, K_r,\, Q.\ \begin{array}{l} e_l = K_l[e_l'] \ *\ e_r = K_l[e_r'] \ *\ \mathit{neutral}(\mathit{ls}_l, K_l) \ *\ \mathit{neutral}(\mathit{ls}_r, K_r) \ * \\ \mathcal{T}(e_l',\, e_r',\, Q) \ *\ \Box\, \forall e_l'',\, e_r''.\ Q(e_l'',\, e_r'') \mathrel{-\!\!*} P(K_l[e_l''],\, K_r[e_r'']) \end{array}
\end{aligned}
$$

The context-closure of a theory $\mathcal{T}$ enables the relation of expressions of the form $K_l[e_l']$ and $K_r[e_r']$, provided the subexpressions e_l' and e_r' are related. A common restriction on the contexts K_l and K_r under which e_l' and e_r' appear is that they contain no handlers for the effects performed by these expressions. To incorporate this restriction, the context-closure operation is parameterised by a pair of lists of labels ls_l and ls_r and includes the condition that K_l and K_r be respectively *neutral* for ls_l and ls_r. A context K is neutral for a list of labels ls, noted $\mathit{neutral}(\mathit{ls}, K)$, when K contains no handlers for labels in ls: $\mathscr{L}(K) \cap \mathit{ls} = \emptyset$.

In the example of $\mathit{Timer}_{\mathit{refl}}$, the context-closure $\mathit{Timer}'_{\mathit{refl}} \triangleq ([\texttt{\$Timer}],\, [\texttt{\$Timer}]) \downharpoonleft\!\upharpoonright \mathit{Timer}_{\mathit{refl}}$ enables the relation of expressions under contexts K_l and K_r respectively neutral for `$Timer`: $\mathit{Timer}'_{\mathit{refl}}(e_l,\, e_r,\, \lambda e_l'\, e_r'.\ Q(K_l[e_l'],\, K_r[e_r'])) \vdash \mathit{Timer}'_{\mathit{refl}}(K_l[e_l],\, K_r[e_r],\, Q)$.

[3] In §2.1, we assume `$Timer` is *available*. Formally, this assumption means `$Timer` is created by a **let effect** instruction placed at the global level.

4.1.3 Model. With the definition of *iThy*, it is now possible to formalise the notion of validation of a theory by a pair of contexts and consequently to formalise the notion of a refinement relation parameterised by a theory.

Figure 2 shows the definition of the refinement relation $e_l \precsim e_r \langle \mathcal{T} \rangle \{R\}$. The definition is recursive and relies on the validation of a theory by a pair of contexts $\{R\}\, K_l \precsim K_r \langle \mathcal{T} \rangle \{S\}$. The definition also relies on the notions of *observational refinement* $\mathcal{O}(e_l,\, e_r,\, S)$ and of *admissibility of a refinement by a theory* $\mathcal{T} \blacktriangleleft e_l \precsim e_r \{R\}$. The notion of admissibility is transparent to the user, whereas the notions of observational refinement and theory validation are opaque and used only in the model.

Observational refinement is defined exactly like in ReLoC [20, §7.1]. The intuitive reading of $\mathcal{O}(e_l,\, e_r,\, S)$ is that either e_l diverges or both e_l and e_r respectively terminate with values v_l and v_r such that $S(v_l,\, v_r)$ holds. The formal definition makes use of Iris's weakest precondition *wp* e_l $\{\ldots\}$, which expresses precisely the condition that e_l either diverges or terminates with a value. To express conditions on e_r, the definition makes use of the *ghost thread-pool* assertion $i \Mapsto K[e_r]$ to state that thread i on the specification side is about to execute e_r. The use of $i \Mapsto K[v_r]$ as a postcondition means that thread i finished executing e_r and that e_r returned output v_r. [4] The thread identifier i is universally quantified, because it is not particularly relevant which specific thread is related to e_l as long as it executes e_r. The evaluation context K under which e_r runs is universally quantified to endow observational refinement with context-local reasoning. [5]

Given the intuitive reading of observational refinement $\mathcal{O}(e_l,\, e_r,\, S)$ and assuming that the notion of validation of a theory $\mathcal{T}$ by a pair of contexts K_l and K_r is captured by $\{R\}\, K_l \precsim K_r \langle \mathcal{T} \rangle \{S\}$, the definition of the refinement relation $e_l \precsim e_r \langle \mathcal{T} \rangle \{R\}$ reads as: for every pair of contexts K_l and K_r, and for all postconditions S, if K_l and K_r validate $\mathcal{T}$, then, either $K_l[e_l]$ diverges or both $K_l[e_l]$ and $K_r[e_r]$ terminate with values by S. The universal quantification over contexts is inspired by Pitts and Stark [36] *biorthogonality* technique, used for the first time by Biernacki et al. [5] to define logical relations for a language with effect handlers.

The definition of $\{R\}\, K_l \precsim K_r \langle \mathcal{T} \rangle \{S\}$ consists of the conjunction of two clauses: (1) a clause relating K_l and K_r when filled with values related by R and (2) a clause relating K_l and K_r when filled with expressions e_l' and e_r' for which the admissibility condition $\mathcal{T} \blacktriangleleft e_l' \precsim e_r' \{R\}$ holds. This condition asserts that e_l' and e_r' are related with postcondition R under $\mathcal{T}$. Because R is a relation on values while return conditions in $\mathcal{T}$ are relations on expressions, admissibility existentially quantifies over a return condition Q such that $\mathcal{T}(e_l',\, e_r',\, Q)$ holds. To connect Q with R, the definition also claims that the refinement $e_l'' \precsim e_r'' \langle \mathcal{T} \rangle \{R\}$ holds for every pair of expressions e_l'' and e_r'' related by Q. This occurrence of the refinement relation makes the definition of $e_l \precsim e_r \langle \mathcal{T} \rangle \{R\}$ recursive. This explains the use of the *later modality* $\triangleright$, which is one of Iris's mechanisms to introduce recursive definitions: as long as the recursive occurrences are guarded by the later modality, the definition can be constructed in Iris. The use of the persistently modality is again related to the compliance with multi-shot continuations. [6]

4.1.4 Reasoning rules. The refinement relation enjoys a collection of powerful and high-level reasoning rules shown in Figure 3. Rules VALUE, STEP-L, and STEP-R are standard: Rules STEP-L and STEP-R provide the ability to partially execute code using pure reductions and Rule VALUE allows the user to end a refinement proof when both sides terminate with values that satisfy the postcondition. The remaining rules are novel.

[4] The assertion *specCtx* is used to momentarily give ownership of the specification side's resources. It is defined like in [20].

[5] There is no need to enclose e_l under a universally quantified context, because *wp* already enjoys context-local reasoning.

[6] In §4.3, we show how to extend the logic with support for one-shot continuations with no changes to the model.

VALUE

$$\frac{R(v_l,\, v_r)}{v_l \lesssim v_r \,\langle \mathcal{T} \rangle\, \{R\}}$$

INTRODUCTION

$$\frac{\mathcal{T} \blacktriangleleft e_l \lesssim e_r \,\{R\}}{e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\}}$$

BIND

$$\frac{\begin{array}{c} \mathit{traversable}(K_l,\, K_r,\, \mathcal{T}) \qquad \mathcal{T} \sqsubseteq \mathcal{F} \\ e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{v_l\, v_r.\ K_l[v_l] \lesssim K_r[v_r] \,\langle \mathcal{F} \rangle\, \{R\}\} \end{array}}{K_l[e_l] \lesssim K_r[e_r] \,\langle \mathcal{F} \rangle\, \{R\}}$$

EXHAUSTION

$$\frac{\begin{array}{c} e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\} \\ \wedge \begin{cases} \forall v_l,\, v_r.\ R(v_l,\, v_r) \mathrel{-\!\!*} K_l[v_l] \lesssim K_r[v_r] \,\langle \mathcal{F} \rangle\, \{S\} \\ \forall e_l',\, e_r'.\ \mathcal{T} \blacktriangleleft e_l' \lesssim e_r' \,\{R\} \mathrel{-\!\!*} K_l[e_l'] \lesssim K_r[e_r'] \,\langle \mathcal{F} \rangle\, \{S\} \end{cases} \end{array}}{K_l[e_l] \lesssim K_r[e_r] \,\langle \mathcal{F} \rangle\, \{S\}}$$

MONOTONICITY

$$\frac{\begin{array}{c} e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\} \qquad \mathcal{T} \sqsubseteq \mathcal{F} \\ \Box\, \forall v_l,\, v_r.\ R(v_l,\, v_r) \mathrel{-\!\!*} S(v_l,\, v_r) \end{array}}{e_l \lesssim e_r \,\langle \mathcal{F} \rangle\, \{S\}}$$

STEP-L

$$\frac{e_l \to_{\mathsf{p}} e_l' \qquad \triangleright K[e_l'] \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\}}{K[e_l] \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\}}$$

STEP-R

$$\frac{e_r \to_{\mathsf{p}} e_r' \qquad e_l \lesssim K[e_r'] \,\langle \mathcal{T} \rangle\, \{R\}}{e_l \lesssim K[e_r] \,\langle \mathcal{T} \rangle\, \{R\}}$$

Fig. 3. Reasoning rules of the base logic.

Rule INTRODUCTION states the admissibility of a refinement between e_l and e_r with postcondition R under the theory $\mathcal{T}$ implies $e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\}$. This rule is usually applied to reason about effectful operations independently of their handlers.

Rule EXHAUSTION incorporates a *case-analysis* principle into the logic whereby, if $e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\}$ holds, then the derivation of $K_l[e_l] \lesssim K_r[e_r] \,\langle \mathcal{F} \rangle\, \{S\}$ splits into two subgoals: one where e_l and e_r are replaced with values related by R and another one where e_l and e_r are replaced with expressions e_l' and e_r' such that $\mathcal{T} \blacktriangleleft e_l' \lesssim e_r' \,\{R\}$ holds. This rule allows one to reason about handlers independently of their handlees. It is typically applied when the contexts K_l and K_r contain handlers monitoring the handlees e_l and e_r. However, it is important to note that the rule is applicable to any contexts K_l and K_r. This flexibility allows the relation of programs where handlers on both sides of the relation do not necessarily match. In §4.1.5, we return to the example of *countdown* (§2) to show this principle in action.

Rule MONOTONICITY enables one to weaken the postcondition R and the parameterised theory $\mathcal{T}$ of a refinement $e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\}$. Such a reasoning principle is useful, for example, when the refinement $e_l \lesssim e_r \,\langle \mathcal{T} \rangle\, \{R\}$ is assumed, but $\mathcal{T}$ and R do not exactly match $\mathcal{F}$ and S. The weakening of R to S is captured by the condition $\Box\, \forall v_l,\, v_r.\ R(v_l,\, v_r) \mathrel{-\!\!*} S(v_l,\, v_r)$, where the persistently modality ensures this ordering does not rely on ephemeral resources and can thus be used multiple times in case the program is reified as a multi-shot continuation and resumed multiple times. The weakening of $\mathcal{T}$ to $\mathcal{F}$ is captured by the *theory ordering* $\mathcal{T} \sqsubseteq \mathcal{F}$, which is similarly defined: $\Box\, \forall e_l,\, e_r,\, Q.\ \mathcal{T}(e_l,\, e_r,\, Q) \mathrel{-\!\!*} \mathcal{F}(e_l,\, e_r,\, Q)$.

Finally, Rule BIND enables context-local reasoning about e_l and e_r independently of their enclosing evaluation contexts K_l and K_r. The only side-condition is that there must be a theory $\mathcal{T}$ *contained in* $\mathcal{F}$ (that is, $\mathcal{T} \sqsubseteq \mathcal{F}$) such that $\mathcal{T}$ can *traverse* the pair of contexts K_l and K_r. Informally, this says that whenever e_l and e_r are related by the theory $\mathcal{T}$, so are $K_l[e_l]$ and $K_r[e_r]$. The formal definition is:

$$\begin{array}{l} \mathit{traversable}(K_l,\, K_r,\, \mathcal{T}) \triangleq \Box\, \forall e_l,\, e_r,\, Q.\ \mathcal{T}(e_l,\, e_r,\, Q) \mathrel{-\!\!*} \\ \qquad \exists P.\ \mathcal{T}(K_l[e_l],\, K_r[e_r],\, P) * \Box\, \forall e_l',\, e_r'.\ P(K_l[e_l'],\, K_r[e_r']) \mathrel{-\!\!*} Q(e_l',\, e_r') \end{array}$$

This definition is transparent to the user. In other words, when applying Rule BIND, the user must find a theory $\mathcal{T}$ and prove this traversable condition. Fortunately, the predicate *traversable* works nicely in combination with the context-closure of a theory:

$$\forall \mathcal{T}, ls_l,\ ls_r,\ K_l,\ K_r.\ \mathit{neutral}(ls_l, K_l) \mathrel{-\!\!*} \mathit{neutral}(ls_r, K_r) \mathrel{-\!\!*} \mathit{traversable}(K_l,\ K_r,\ (ls_l,\ ls_r) \downharpoonleft\!\upharpoonright \mathcal{T}) \tag{10}$$

Using this theorem, it is possible to derive the following version of the bind rule, where the traversable condition is replaced with more explicit conditions on the contexts K_l and K_r:

$$\textsc{derived-bind}\quad \frac{\mathit{neutral}(ls_l, K_l) \quad \mathit{neutral}(ls_r, K_r) \quad (ls_l,\ ls_r) \downharpoonleft\!\upharpoonright \mathcal{T} \sqsubseteq \mathcal{F}}{e_l \precsim e_r \ \langle (ls_l,\ ls_r) \downharpoonleft\!\upharpoonright \mathcal{T} \rangle \ \{v_l\, v_r.\ K_l[v_l] \precsim K_r[v_r] \ \langle \mathcal{F} \rangle \ \{R\}\}} \vdash K_l[e_l] \precsim K_r[e_r] \ \langle \mathcal{F} \rangle \ \{R\}$$

4.1.5 Example. We briefly discuss how these rules can be used to derive Refinements 6 and 7 from §2.2. First, let us formally define the theory $\mathit{Timer}^{\ell}_{spec}$:

$$\begin{aligned}
&\mathit{Timer}^{\ell}_{spec} \triangleq ([\texttt{\$Timer}],\ []) \downharpoonleft\!\upharpoonright (\mathit{Get} \oplus \mathit{Set}) \\
&\mathit{Get}(\textbf{perform}\ \texttt{\$Timer}\ (\textbf{inl}\ ()),\ !\ell,\ Q) \triangleq \ell \overset{1/2}{\mapsto}_{\mathsf{s}} x \ast \square(\ell \overset{1/2}{\mapsto}_{\mathsf{s}} x \mathrel{-\!\!*} Q(x,\ x)) \\
&\mathit{Set}(\textbf{perform}\ \texttt{\$Timer}\ (\textbf{inr}\ y),\ \ell \leftarrow y,\ Q) \triangleq \ell \overset{1/2}{\mapsto}_{\mathsf{s}} \_ \ast \square(\ell \overset{1/2}{\mapsto}_{\mathsf{s}} y \mathrel{-\!\!*} Q((),\ ()))
\end{aligned}$$

The definition uses the *sum operator* $\oplus$, which combines relations from two theories:

$$(\mathcal{T} \oplus \mathcal{F})(e_l,\ e_r,\ Q) \ \triangleq\ \mathcal{T}(e_l,\ e_r,\ Q) \lor \mathcal{F}(e_l,\ e_r,\ Q)$$

The theory *Get* allows the handlee to establish a relation between **perform** $Timer (**inl** ()) and $!\ell$ in exchange for the fractional ownership $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} x$. This assertion appears as a premise to the return condition, which holds of the pair $(x,\ x)$. From the perspective of the handlee, this means that the fractional ownership $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} x$ can be reclaimed and that the expressions **perform** $Timer (**inl** ()) and $!\ell$ both return x. The reading of *Set* is analogous.

Because $\mathit{Timer}^{\ell}_{spec}$ is defined using the context-closure operator, the theory can be used in conjunction with DERIVED-BIND (where $\mathcal{F}$ is instantiated with $\mathit{Timer}^{\ell}_{spec}$) to carry out Refinement 6.

The key rule to establish Refinement 7 is Rule EXHAUSTION. It is applied using K_l instantiated with *run_st_passing*'s handler and K_r instantiated with $[]$. Furthermore, the theory $\mathcal{T}$ in the statement of Rule EXHAUSTION is taken to be $\mathit{Timer}^{\ell}_{spec}$ and $\mathcal{F}$ is instantiated with $\bot$. During the proof of the clause $\forall e'_l,\ e'_r.\ \mathit{Timer}^{\ell}_{spec} \blacktriangleleft e'_l \precsim e'_r \ \{=\} \mathrel{-\!\!*} K_l[e'_l] \precsim e'_r \ \{=\}$, the admissibility condition gives the fractional ownership $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} x$ to the handler. In combination with the other assertion $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} x$ initially given to the handler as a premise in 7, full ownership of ℓ is claimed by the handler, which can then update ℓ in case of a `set` request for example. The return condition can be interpreted in this proof as the condition under which the handler can resume the continuation: in the case of a `get` request, for example, both the value x and the fractional ownership $\ell \overset{1/2}{\mapsto}_{\mathsf{s}} x$ must be supplied.

4.2 blaze: A logic for effect handlers with dynamic labels

So far, we have exclusively considered examples where the effect labels have already been allocated. This observation incites the question: how to reason about programs like *run_ask* (§3.1) where effect labels are allocated locally to avoid collision of effect names? For instance, given two clients of the ask effect main_1 and main_2 and an integer x, is it possible to establish a refinement between *run_tick* x main_1 and main_2 (**fun** _. x) despite the fact that the implementation of the ask effect by *run_ask* is correct regardless of the effects main_1 might perform?

The verification of programs with local allocation of effects, such as *run_ask*, depends heavily on assumptions about fresh labels being distinct from previously allocated ones. The baze logic places the burden of keeping track of these assumptions entirely on the user. So, while baze *can* be used to reason about *run_ask*, it is not placed at the right level of abstraction. To address this limitation, we introduce blaze, a logic built on top of baze to facilitate reasoning about dynamic labels. In

$$e_l \lesssim_\star e_r \,\langle \mathcal{L} \rangle\, \{R\} \;\triangleq\; valid(\mathcal{L}) \mathrel{-\!\!*} e_l \lesssim e_r \,\langle interp(\mathcal{L}) \rangle\, \{R\}$$

$$\begin{array}{rl} interp([\,]) & \triangleq\; \bot \\ interp((ls_l,\, ls_r,\, \mathcal{T}) :: \mathcal{L}) & \triangleq \\ (ls_l,\, ls_r) \downharpoonleft\!\upharpoonright \mathcal{T} \oplus interp(\mathcal{L}) & \end{array} \qquad \begin{array}{l} valid(\mathcal{L}) \;\triangleq \\ \mathop{\ast}\begin{cases} distinct(labels_{\mathtt{i}}(\mathcal{L})) \ast \forall \$\mathsf{E} \in labels_{\mathtt{i}}(\mathcal{L}).\ label^{\square}_{\mathtt{i}}(\$\mathsf{E}) \\ distinct(labels_{\mathtt{s}}(\mathcal{L})) \ast \forall \$\mathsf{E} \in labels_{\mathtt{s}}(\mathcal{L}).\ label^{\square}_{\mathtt{s}}(\$\mathsf{E}) \end{cases} \end{array}$$

$$\begin{array}{rcl} labels_{\mathtt{i}}([\,]) & \triangleq & [\,] \\ labels_{\mathtt{i}}((ls_l,\, \_,\, \_) :: \mathcal{L}) & \triangleq & ls_l +\!\!+\, labels_{\mathtt{i}}(\mathcal{L}) \end{array} \qquad \begin{array}{rcl} labels_{\mathtt{s}}([\,]) & \triangleq & [\,] \\ labels_{\mathtt{s}}((\_,\, ls_r,\, \_) :: \mathcal{L}) & \triangleq & ls_r +\!\!+\, labels_{\mathtt{s}}(\mathcal{L}) \end{array}$$

Fig. 4. Model of blaze.

blaze, it is possible to establish (in a relatively straightforward way) a strong result about *run_ask* where assumptions about labels being distinct are hidden:

$$\begin{array}{l} \forall main_1, \\ main_2, \\ x,\, \mathcal{L},\, R. \end{array} \left(\begin{array}{l} \forall\, ask_1,\, ask_2,\, \mathcal{M}. \\ \square\, ask_1\,() \lesssim_\star ask_2\,()\, \langle \mathcal{M} \rangle\, \{v_l\, v_r.\ v_l = v_r = x\} \mathrel{-\!\!*} \\ main_1\, ask_1 \lesssim_\star main_2\, ask_2\, \langle \mathcal{L} +\!\!+ \mathcal{M} \rangle\, \{R\} \end{array} \right) \mathrel{-\!\!*} \begin{array}{l} run\_ask\, x\, main_1 \lesssim_\star \\ main_2\, (\mathbf{fun}\,\_.\ x)\, \langle \mathcal{L} \rangle\, \{R\} \end{array} \qquad (11)$$

The novelty of the refinement relation $e_l \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\}$ is the parameterised *list of theories* $\mathcal{L}$. The elements of $\mathcal{L}$ are triples of the form $(ls_l, ls_r, \mathcal{T})$, where ls_l and ls_r are lists of effect labels respectively allocated by the implementation and the specification sides, and $\mathcal{T}$ is a theory relating expressions that use these effects. Roughly speaking, the list $\mathcal{M}$ in Refinement 11 is used to relate **fun** _. **perform** Ask () to **fun** _. x. Its universal quantification reflects the fact that Ask is allocated locally by *run_ask*. The list $\mathcal{L}$ represents an ambient set of relational theories used to relate $main_1$ to $main_2$. The lists of theories $\mathcal{L}$ and $\mathcal{M}$ are *disjoint*, because the labels in $\mathcal{L}$ are allocated before Ask. This assumption however is implicit: it is not written in the statement of Refinement 11 and it does not appear in the verification.

4.2.1 Model. The formal definition of $e_l \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\}$ appears in Figure 4. It unfolds to a refinement in baze with parameterised theory $interp(\mathcal{L})$ and premise $valid(\mathcal{L})$.

The theory $interp(\mathcal{L})$ is constructed as the iterated sum of theories $(ls_l, ls_r) \downharpoonleft\!\upharpoonright \mathcal{T}$ for every triple $(ls_l, ls_r, \mathcal{T})$ in $\mathcal{L}$. In essence, this construction sacrifices the expressivity of general theories in baze's refinement relation to endow the blaze layer with context-local reasoning by default.

The premise $valid(\mathcal{L})$ is defined using the terms $labels_{\mathtt{i}}(\mathcal{L})$ and $labels_{\mathtt{s}}(\mathcal{L})$, which collect the labels in $\mathcal{L}$ that belong to the implementation side and to the specification side, respectively. The assertions $label^{\square}_{\mathtt{i}}(\$\mathsf{E}_1)$ and $label^{\square}_{\mathtt{s}}(\$\mathsf{E}_2)$ claim ownership of persistent resources obtained after the allocation of the effects $\$\mathsf{E}_1$ and $\$\mathsf{E}_2$.[7] Therefore, the premise $valid(\mathcal{L})$ asserts that the labels in $labels_{\mathtt{i}}(\mathcal{L})$ and in $labels_{\mathtt{s}}(\mathcal{L})$ have already been allocated and are pairwise distinct. In essence, this premise represents the assumption that the theories in $\mathcal{L}$ do not interfere with one another or new theories for newly allocated effects.

4.2.2 Reasoning rules. The reasoning rules of blaze appear in Figure 5.

Rule EFFECT-L-★ can be used in conjunction with Rule ADD-LABEL-L-★ to add a freshly allocated label to one of the triples in $\mathcal{L}$. The assertion $label_{\mathtt{i}}(\$\mathsf{E})$ works as an exchangeable token that is

[7] The assertion $label^{\square}_{\mathtt{i}}(\$\mathsf{E}_1)$ is defined in terms of a more general assertion $label_{\mathtt{i}}(\$\mathsf{E}_1, dq)$, where dq is a *discardable fraction* [51]. This general assertion, on its turn, is defined in Iris as the *fragmental ownership* [45] of a global ghost variable: $\boxed{\circ\, \{\$\mathsf{E}_1 \mapsto dq\}}^{labelsMap}$. (Full ownership is kept by the *state interpretation*.) Taking dq as the full fraction 1 gives the assertion $label_{\mathtt{i}}(\$\mathsf{E}_1)$, whereas taking dq as the discarded fraction gives the persistent assertion $label^{\square}_{\mathtt{i}}(\$\mathsf{E}_1)$. The assertion $label^{\square}_{\mathtt{s}}(\$\mathsf{E}_2)$ is defined analogously (with full ownership kept by *specCtx*).

EFFECT-L-★

$$\frac{\triangleright \forall \$\mathsf{E}.\ label_{\mathtt{i}}(\$\mathsf{E}) \mathrel{-\!\!*} K[e\{\$\mathsf{E}/\mathsf{E}\}] \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\}}{K[\mathtt{let\,effect}\ \mathsf{E}\ \mathtt{in}\ e] \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\}}$$

EFFECT-R-★

$$\frac{\forall \$\mathsf{E}.\ label_{\mathtt{s}}(\$\mathsf{E}) \mathrel{-\!\!*} e_l \lesssim_\star K[e\{\$\mathsf{E}/\mathsf{E}\}] \langle \mathcal{L} \rangle \{R\}}{e_l \lesssim_\star K[\mathtt{let\,effect}\ \mathsf{E}\ \mathtt{in}\ e] \langle \mathcal{L} \rangle \{R\}}$$

ADD-LABEL-L-★

$$\frac{label_{\mathtt{i}}(\$\mathsf{E}) \qquad e_l \lesssim_\star e_r \langle (\$\mathsf{E} :: ls_l,\ ls_r,\ \mathcal{T}) :: \mathcal{L} \rangle \{R\}}{e_l \lesssim_\star e_r \langle (ls_l,\ ls_r,\ \mathcal{T}) :: \mathcal{L} \rangle \{R\}}$$

ADD-LABEL-R-★

$$\frac{label_{\mathtt{s}}(\$\mathsf{E}) \qquad e_l \lesssim_\star e_r \langle (ls_l,\ \$\mathsf{E} :: ls_l,\ \mathcal{T}) :: \mathcal{L} \rangle \{R\}}{e_l \lesssim_\star e_r \langle (ls_l,\ ls_r,\ \mathcal{T}) :: \mathcal{L} \rangle \{R\}}$$

NEW-THEORY-★

$$\frac{e_l \lesssim_\star e_r \langle ([\,],\ [\,],\ \bot) :: \mathcal{L} \rangle \{R\}}{e_l \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\}}$$

INTRODUCTION-★

$$\frac{(ls_l,\ ls_r,\ \mathcal{T}) \in \mathcal{L} \qquad (ls_l,\ ls_r,\ \mathcal{T}) \blacktriangleleft e_l \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\}}{e_l \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\}}$$

EXHAUSTION-★

$$\frac{\begin{array}{c} \mathscr{L}(K_l) \subseteq ls_l \qquad \mathscr{L}(K_r) \subseteq ls_r \\ e_l \lesssim_\star e_r \langle \mathcal{M} \rangle \{R\} \qquad \mathcal{M} = (ls_l,\ ls_r,\ \mathcal{T}) :: \mathcal{L} \qquad \mathcal{N} = (ls_l,\ ls_r,\ \mathcal{F}) :: \mathcal{L} \\ \wedge \begin{cases} \Box\, \forall v_l,\ v_r.\ R(v_l,\ v_r) \mathrel{-\!\!*} K_l[v_l] \lesssim_\star K_r[v_r] \langle \mathcal{N} \rangle \{S\} \\ \Box\, \forall e'_l,\ e'_r.\ (ls_l,\ ls_r,\ \mathcal{T}) \blacktriangleleft e'_l \lesssim_\star e'_r \langle \mathcal{M} \rangle \{R\} \mathrel{-\!\!*} K_l[e'_l] \lesssim_\star K_r[e'_r] \langle \mathcal{N} \rangle \{S\} \end{cases} \end{array}}{K_l[e_l] \lesssim_\star K_r[e_r] \langle \mathcal{N} \rangle \{S\}}$$

BIND-★

$$\frac{\begin{array}{c} \mathscr{L}(K_l) \subseteq labels_{\mathtt{i}}(\mathcal{M}) \qquad \mathscr{L}(K_r) \subseteq labels_{\mathtt{s}}(\mathcal{M}) \qquad \mathcal{L} \mathbin{+\!\!+} \mathcal{M} \sqsubseteq_\star \mathcal{N} \\ e_l \lesssim_\star e_r \langle \mathcal{L} \rangle \{v_l\, v_r.\ K_l[v_l] \lesssim_\star K_r[v_r] \langle \mathcal{N} \rangle \{R\}\} \end{array}}{K_l[e_l] \lesssim_\star K_r[e_r] \langle \mathcal{N} \rangle \{R\}}$$

Fig. 5. Reasoning rules of blaze.

forged by Rule EFFECT-L-★ and consumed by Rule ADD-LABEL-R-★. Rules EFFECT-R-★ and ADD-LABEL-R-★ enable analogous reasoning for the specification side. The order of the triples in $\mathcal{L}$ is not important. New theories can be added with Rule NEW-THEORY-★.

The statement of Rule INTRODUCTION-★ is similar to Rule INTRODUCTION. The theory $\mathcal{T}$ can be chosen among any of the list $\mathcal{L}$. Admissibility must be shown with respect to the triple $(ls_l,\ ls_r,\ \mathcal{T})$:

$$\begin{array}{l} (ls_l,\ ls_r,\ \mathcal{T}) \blacktriangleleft e_l \lesssim_\star e_r \langle \mathcal{L} \rangle \{R\} \ \triangleq \\ \qquad \exists e'_l,\ e'_r,\ K_l,\ K_r,\ Q.\ \begin{array}{l} e_l = K_l[e'_l] \ * \ e_r = K_r[e'_r] \ * \ neutral(ls_l, K_l) \ * \ neutral(ls_r, K_r) \ * \\ \mathcal{T}(e',\ e',\ Q) \ * \ \Box \triangleright \forall e''_l,\ e''_r.\ Q(e''_l,\ e''_r) \mathrel{-\!\!*} K_l[e''_l] \lesssim_\star K_r[e''_r] \langle \mathcal{L} \rangle \{R\} \end{array} \end{array}$$

The ability to relate expressions under arbitrary contexts K_l and K_r in this definition closes the theory $\mathcal{T}$ under neutral contexts (for ls_l and ls_r). This design choice makes it possible to state Rule BIND-★ in a similar fashion to Rule DERIVED-BIND with explicit side conditions on K_l and K_r. Namely, the condition $\mathscr{L}(K_l) \subseteq labels_{\mathtt{i}}(\mathcal{M})$ restricts the handlers in K_l to the effect labels in $labels_{\mathtt{i}}(\mathcal{M})$. The condition $\mathscr{L}(K_r) \subseteq labels_{\mathtt{s}}(\mathcal{M})$ is analogous. The condition $\mathcal{L} \mathbin{+\!\!+} \mathcal{M} \sqsubseteq_\star \mathcal{N}$ is defined as the multiplicity-preserving inclusion of $\mathcal{L} \mathbin{+\!\!+} \mathcal{M}$ in $\mathcal{N}$. It guarantees the labels in $\mathcal{L}$ are disjoint from the labels in $\mathcal{M}$. In combination these conditions restrict the handlers in K_l and K_r to not capture effects with labels from $\mathcal{L}$.

Finally, Rule EXHAUSTION-★ incorporates the exhaustion principle into blaze. The expressions e_l and e_r are related under a list of theories $\mathcal{M}$, but the user needs to choose only one of the theories $\mathcal{T}$ in $\mathcal{M}$ with which to perform the case-analysis reasoning; that is, the premise requiring a relation between $K_l[e'_l]$ and $K_r[e'_r]$ assumes the admissibility with respect only to the theory $\mathcal{T}$. Intuitively, this is possible because of the conditions $\mathscr{L}(K_l) \subseteq ls_l$ and $\mathscr{L}(K_r) \subseteq ls_r$, which guarantee that the remaining theories in $\mathcal{M}$ are irrelevant in the context of K_l and K_r. The persistently modality is needed because these remaining theories can still relate effects that cause K_l or K_r to be captured in a multi-shot continuation.

4.2.3 Example. We now show how these reasoning rules can be applied to derive Refinement 11. The proof starts with the application of Rules NEW-THEORY-★, EFFECT-L-★, and ADD-LABEL-L-★, in this order. This sequence of rules has the effect of adding the fresh label `$Ask` to a new entry in the ambient list of theories $\mathcal{L}$. Initially this new entry has the form $([\texttt{\$Ask}], [], \bot)$. The core of the proof is the application of Rule EXHAUSTION-★, where e_l is instantiated with $main_1$ (**perform** `$Ask` ()), the expression e_r with $main_2$ (**fun**_. x), and the theory $\mathcal{T}$ with $AskT(\textbf{perform}\ \texttt{\$Ask}\ (),\ x,\ Q) \triangleq \square\, Q(x, x)$. The refinement between $main_1$ and $main_2$ directly follows from the premise of Refinement 11 with the abstract theory list $\mathcal{M}$ instantiated with $[([\texttt{\$Ask}], [], AskT)]$. The other conditions of Rule EXHAUSTION-★ are relatively straightforward.

4.3 Support for one-shot continuations

The introduction of one-shot continuations is motivated by the fact that, in languages like OCaml, the violation of a one-shot discipline causes a runtime error. We thus follow [15, 50] to introduce one-shot continuations in a way that enables the logic to rule out such runtime errors. The idea is to represent one-shot continuations with a construct **cont** ℓ K [8] that, in addition to the reified context K, carries a location ℓ which triggers a runtime error if the contination is resumed twice. By the soundness theorem of the logic (§4.4), a verified program either diverges or terminates, but it cannot resume a one-shot continuation twice as runtime errors are guaranteed to be absent.

Another motivation is that, as we have seen, from a logical perspective, allowing continuations to be resumed multiple times results in the addition of persistently modalities in some of the reasoning rules, most notably, in Rule MONOTONICITY. To provide the logic with a strong monotonicity reasoning principle applicable to fragments of code that abide by a one-shot discipline, we take inspiration from van Rooij and Krebbers [50], by introducing the *one-shot* operator $\bigcirc\mathcal{T}$, a semantic version of the *flip-bang* operator: [9]

$$(\bigcirc\mathcal{T})(e_l, e_r, Q) \ \triangleq\ \exists P.\, \mathcal{T}(e_l, e_r, P) \ast \triangleright \forall e_l', e_r'.\, P(e_l', e_r') \mathrel{-\!\!\ast} Q(e_l', e_r')$$

The key property of this definition is that it closes a theory $\mathcal{T}$ under a monotonicity principle on return conditions: if $(\bigcirc\mathcal{T})(e_l, e_r, Q)$ and $\forall v_l, v_r.\, Q(v_l, v_r) \mathrel{-\!\!\ast} P(v_l, v_r)$ hold, then $(\bigcirc\mathcal{T})(e_l, e_r, P)$ holds. Using the notations $\bigcirc_{\mathsf{ms}}\mathcal{T} \triangleq \mathcal{T}$, $\bigcirc_{\mathsf{os}}\mathcal{T} \triangleq \bigcirc\mathcal{T}$, $\square_{\mathsf{ms}}A \triangleq \square\, A$, and $\square_{\mathsf{os}}A \triangleq A$, it is then possible to incorporate a generalised monotonicity principle into the logic:

$$\text{GEN-MONOTONICITY}\qquad \begin{matrix} (\square_m \forall v_l, v_r.\, R(v_l, v_r) \mathrel{-\!\!\ast} S(v_l, v_r)) \\ e_l \lesssim e_r \langle \mathcal{T} \rangle \{R\} \qquad \mathcal{T} \sqsubseteq \mathcal{F} \end{matrix} \ \vdash\ e_l \lesssim e_r \langle \bigcirc_m \mathcal{F} \rangle \{S\}$$

Taking $m = \mathsf{ms}$ yields Rule MONOTONICITY, while taking $m = \mathsf{os}$ eliminates the persistently modality, thereby allowing ephemeral resources to be used in the proof that R implies S.

The blaze logic admits a similar generalised monotonicity rule and a generalised exhaustion rule that makes use of the one-shot operator to eliminate the persistently modalities in EXHAUSTION-★. Both rules are included in the Appendix (Figure 11).

[8] In our Rocq formalisation [1], we have an extended syntax k (**as multi**)$^?$ for the continuation binder in the effect branch of a handler. The keywords **as multi** are optional. Their presence indicates a multi-shot semantics. Their absence indicates the handler captures the handlee in a one-shot continuation. The construct **cont** ℓ K is introduced at runtime by such handlers.

[9] This definition can also be seen as a generalisation of the *upward closure* [14, §2.2.2] to a binary setting.

4.4 Soundness

Soundness of baze is shown by a standard *adequacy statement* [20, Thm. 7.1] that relates the notion of refinement to the underlying operational semantics of λ-blaze:[10]

Theorem 4.1. *If* $\vdash e_l \lesssim e_r \langle \bot \rangle \{\mathsf{True}\}$, *then either* e_l *diverges or both* e_l *and* e_r *terminate.*

Soundness of blaze follows as a corollary by $e_l \lesssim_\star e_r \langle [] \rangle \{\mathsf{True}\} \vdash e_l \lesssim e_r \langle \bot \rangle \{\mathsf{True}\}$.

Another common corollary of adequacy is *contextual refinement* [20, Lem. 7.2]. We cannot write the statement of contextual refinement, because it depends on types, whereas λ-blaze is untyped. Extending λ-blaze with types is one of our directions for future work (§7).

5 Case Studies

To assess the usability of the logic, we verify refinement statements for a number of interesting effects including *concurrency* (§5.1), *Haskell-like non-determinism* (§5.2), and *state*, where, like Biernacki et al. [5, §4.2], we show state can be implemented in terms of two independent effects: a *reader effect* and a *writer effect*. In the interest of space, we do not discuss this state effect in detail. Its implementation can be found in the Appendix C.1. Mechanised proofs of all case studies are included in our Rocq formalisation [1].

5.1 Concurrency

Effect handlers enable the implementation of *cooperative-concurrency* libraries. In such libraries, multiple tasks can be spawned and their execution is monitored by a scheduler making sure at most one task runs at a time. It is an important and interesting application of effect handlers, serving as the "*primary motivation*" for the addition of effect handlers to OCaml [33, §24.5]. Here is the handler-based implementation of a *fork effect* [7, Fig. 11] in λ-blaze:

```
run_fork ≜ fun main.
  let effect Fork in let q = new_queue() in
  let run = rec run task. handle task() with
    | effect Fork task′, k ⇒ push q k; run task′
    | _ ⇒ if empty q then () else (let k = pop q in k())
  in run (fun _. main (fun task′. perform Fork task′))
```

The function *run_fork* supplies a piece of client code *main* with the functionality to fork tasks by monitoring the execution of *main* with a handler for the Fork effect. The handling of a Fork effect with payload *task′* pushes the paused continuation k to a queue q. This queue is allocated at the beginning of *run_fork*'s execution. It is initially empty, and, as an invariant, it stores continuations that can be readily resumed with (). Updates to q maintain this invariant, because, thanks to a deep-handler semantics, the continuation k includes the Fork handler at its top-most frame. After this update, the handling of Fork terminates by running *task′* under a new Fork handler. When a task terminates, if the handler finds q non-empty, it pops a continuation k from q representing a previously paused task and resumes the execution of this task. If q is empty then all scheduled tasks have executed, so the function *run_fork* terminates.

The implementation of *run_fork* is concise, but relies on advanced programming features, notably, the ability to reify contexts as first-class continuations using handlers and the ability to place these continuations in the store. The complexity of *run_fork*'s operational behaviour motivates the question: is it possible to show that the fork functionality implemented by *run_fork* can be abstracted as a real concurrent fork instruction?

[10] Theorem B.3 rephrases adequacy in a slightly more formal way in terms of the predicates *safe* and *terminates*.

$$\begin{array}{l} \mathit{runForkSpec} \triangleq \square\, \forall\, \mathit{main}_1,\ \mathit{main}_2. \\ \quad \begin{pmatrix} \forall \mathit{fork}_1,\ \mathit{fork}_2,\ \mathcal{L}. \\ \mathit{forkSpec}(\mathit{fork}_1,\ \mathit{fork}_2,\ \mathcal{L}) \mathrel{-\!\!*} \\ \mathit{main}_1\, \mathit{fork}_1 \lesssim_\star \mathit{main}_2\, \mathit{fork}_2\ \langle \mathcal{L} \rangle\ \{\mathsf{True}\} \end{pmatrix} \mathrel{-\!\!*} \begin{array}{l} \mathit{run\_fork}\ \mathit{main}_1 \lesssim_\star \\ \mathit{main}_2\ (\mathbf{fun}\ \mathit{task}'.\ \mathbf{fork}\ (\mathit{task}'()))\ \{\mathsf{True}\} \end{array} \end{array}$$

$$\begin{array}{l} \mathit{forkSpec}(\mathit{fork}_1,\ \mathit{fork}_2,\ \mathcal{L}) \triangleq \square\, \forall \mathit{task}_1,\ \mathit{task}_2. \\ \quad \mathit{task}_1() \lesssim_\star \mathit{task}_2()\ \langle \mathcal{L} \rangle\ \{\mathsf{True}\} \mathrel{-\!\!*} \mathit{fork}_1\ \mathit{task}_1 \lesssim_\star \mathit{fork}_2\ \mathit{task}_2\ \langle \mathcal{L} \rangle\ \{\mathsf{True}\} \end{array}$$

Fig. 6. Fork case study: Specification.

In this case study, we answer this question positively by verifying in blaze that the functionality implemented by *run_fork* refines the primitive **fork** construct of λ-blaze. The formal statement is written in Figure 6. The specification of *run_fork*, the assertion *runForkSpec*, states a refinement between the application of *run_fork* to a client main_1 and the application of a client main_2 to a function **fun** *task'*. **fork** (*task'*()) that directly forks *task'*. The clients main_1 and main_2 are universally quantified in this specification. It is assumed that main_1 and main_2 can be related when respectively supplied with abstract fork implementations fork_1 and fork_2. It is the obligation of the user of the library to show the relation between $\mathit{main}_1\, \mathit{fork}_1$ and $\mathit{main}_2\, \mathit{fork}_2$. To establish this relation, the user can rely on a relational specification of fork_1 and fork_2, the assertion *forkSpec*, stating a relation between the application of fork_1 to a task task_1 and the application of fork_2 to a task task_2. To use this specification, it is again an obligation of the user to establish the relation between $\mathit{task}_1()$ and $\mathit{task}_2()$. In establishing this relation, the user can still rely on *forkSpec* to relate further calls to fork_1 and fork_2 in the tasks task_1 and task_2. The refinement between main_1 and main_2 is carried out under an abstract theory list $\mathcal{L}$. Intuitively, this list represents the internal relational theory that is used by *run_fork* to relate Fork to **fork**. Apart from $\mathcal{L}$, which is abstract to the user, the specification *runForkSpec* assumes an empty ambient theory to relate effects in main_1 and main_2. In other words, the specification disallows main_1 and main_2 to perform unhandled effects. This limitation is necessary, because forking tasks that perform unhandled effects is a runtime error.

5.1.1 Relational reasoning about concurrency. Before presenting the proof of *runForkSpec*, we explain how we extend blaze with support for reasoning about native concurrency. [11] Despite the substantial literature on relational concurrent separation logic [19, 20, 51, 52], we devise *novel* relational reasoning rules to tackle three key challenges.

Limitation to refinements where **fork***s match.* In previous work (for example [20, §4.1]), it is assumed that **fork** instructions on both sides of a refinement match. This is clearly not the case for the refinement *runForkSpec* because only the specification side forks threads directly. To overcome this limitation, we follow Vindum et al. [52] in exposing the ghost thread-pool assertion $i \Mapsto e$ [12] in the logic. Recall that its reading simply states thread i at the specification side runs e. Using this resource, we can split a traditional relational *fork rule* into Rules FORK-L-★ and FORK-R-★, shown in Figure 7. Rule FORK-R-★ forges a new resource $i \Mapsto e_r$. There are many ways to spend this resource. Rule FORK-L-★ consumes it to allow reasoning about a **fork** e_l instruction on the implementation side. As a condition to this rule, the expressions e_l and e_r must be related under the

[11] We focus on blaze but similar reasoning principles can be achieved in baze (Figure 13).

[12] Vindum et al. [52] in fact present this resource as a *right refinement*. In our logic, the user does not explicitly manipulate this resource, so we keep its standard notation.

$$\textsc{fork-l-}\star\;\frac{i \Mapsto e_r \qquad e_l \lesssim_\star e_r\,\langle \mathcal{L}^\perp\rangle\,\{\mathsf{True}\} \qquad K_l[\,()\,] \lesssim_\star e'_r\,\langle \mathcal{L}\rangle\,\{R\}}{K_l[\texttt{\textbf{fork}}\ e_l] \lesssim_\star e'_r\,\langle \mathcal{L}\rangle\,\{R\}} \qquad \textsc{fork-r-}\star\;\frac{\forall i.\ i \Mapsto e_r \mathrel{-\!\!*} e_l \lesssim_\star K_l[\,()\,]\,\langle \mathcal{L}\rangle\,\{R\}}{e_l \lesssim_\star K_l[\texttt{\textbf{fork}}\ e_r]\,\langle \mathcal{L}\rangle\,\{R\}}$$

$$\textsc{logical-fork-}\star\;\frac{i \Mapsto K_r[e_r] \qquad e_l \lesssim_\star e_r\,\langle \mathcal{L}^\perp\rangle\,\{R\} \qquad \forall v_l, v_r.\ R(v_l, v_r) \mathrel{-\!\!*} i \Mapsto K_r[v_r] \mathrel{-\!\!*} K_l[v_l] \lesssim_\star e'_r\,\langle \mathcal{L}\rangle\,\{S\}}{K_l[e_l] \lesssim_\star e'_r\,\langle \mathcal{L}\rangle\,\{S\}}$$

$$\textsc{thread-swap-}\star\;\frac{i \Mapsto K[e_r] \qquad \forall j, K'.\ j \Mapsto K'[e'_r] \mathrel{-\!\!*} e_l \lesssim_\star e_r\,\langle \mathcal{L}^\perp\rangle\,\{v_l\ \_.\ \exists v'_r.\ j \Mapsto K'[v'_r] \ast R(v_l, v'_r)\}}{e_l \lesssim_\star e'_r\,\langle \mathcal{L}\rangle\,\{R\}}$$

Fig. 7. Reasoning rules for concurrency.

theory list $\mathcal{L}^\perp$, which sets every theory in $\mathcal{L}$ to $\perp$. This condition guarantees the forked threads do not perform unhandled effects.

Explicit operational reasoning about thread-pool assertions. The reasoning rules introduced by Vindum et al. [52, Fig. 8] require the user to explicitly manipulate thread-pool resources; that is, the user must inspect the shape of the expression e_r in an assertion $i \Mapsto e_r$ and select one of their rules allowing e_r to be partially executed. This is a strong limitation for the verification of *runForkSpec*, because the only assumption on forked tasks $task_1$ and $task_2$ is that $task_1()$ refines $task_2()$. The specific shape of $task_2$ is unknown. To overcome this limitation, we introduce Rule logical-fork-★ (Figure 7). This rule consumes a thread-pool resource $i \Mapsto K_r[e_r]$ and, as a condition, the user must supply a subexpression e_l that refines e_r. In return, the user can reclaim the assertion $i \Mapsto K_r[v_r]$ where e_r is replaced with its result v_r, obtained with no explicit manipulation of the thread-pool assertion. This rule can be used in conjunction with Rule fork-r-★ to derive the refinement $e_1 \lesssim e'_1\,\{\mathsf{True}\} \mathrel{-\!\!*} e_2 \lesssim e'_2\,\{\mathsf{True}\} \mathrel{-\!\!*} e_1\,;\,e_2 \lesssim \texttt{\textbf{fork}}\ (e'_1)\,;\,e'_2\,\{\mathsf{True}\}$, which cannot be shown using the rules in [52, Fig. 8] without breaking the abstraction of their refinement relation.

Access to thread-pool resource describing the main thread. With the rules discussed so far, the only way to obtain new thread-pool resources is by means of Rule fork-r-★. In other words, thread-pool resources can only describe forked threads but not the *main thread* e_r on the specification side of the refinement. As we are going to see, the proof of *runForkSpec* needs access to the thread-pool resource describing the main thread. Rule rel-split from Vindum et al. [52, Fig. 8] supports this very feature. However, the statement relies on the fact that ReLoC's notion of refinement $\Delta \vDash e_l \lesssim e_r : \tau$ is defined using $i \Mapsto e_r$ as a premise. This makes the adaption of rel-split to blaze particularly difficult, because blaze's model hides thread-pool assertions under multiple layers of abstraction. [13] Instead, we introduce Rule thread-swap-★ (Figure 7), which allows the user to trade a thread-pool resource $i \Mapsto K[e_r]$ in exchange for a thread-pool resource $j \Mapsto K'[e'_r]$ describing the main thread e'_r under an abstract context K'. The expression e_r becomes the thread on the specification side and the postcondition is updated to require the termination of e'_r, which is part of the implicit requirements of the original refinement.

5.1.2 *Verification.* After the allocation of an effect label `$Fork` by *run_fork*, the crux of the proof is (1) the introduction of a relational theory *Fork* to relate `$Fork` effects to **fork** and (2) the definition of the queue invariant in blaze. These definitions appear in Figure 8.

[13] The same holds for baze.

Relational theory.

$$\begin{array}{l} \mathit{Fork}(\textbf{perform}\ \texttt{\$Fork}\ \mathit{task}_1,\ \textbf{fork}\ (\mathit{task}_2()),\ Q) \triangleq \\ \quad \triangleright\, \mathit{task}_1() \precsim_\star \mathit{task}_2()\ \langle[([\texttt{\$Fork}],\ [],\ \mathit{Fork})]\rangle\ \{\mathsf{True}\} \ast Q((),\ ()) \end{array}$$

Invariants and predicates.

$$\begin{array}{l} \mathit{queueInv}(q,\ ks,\ ks') \triangleq \mathit{isQueue}(q,\ ks.1) \ast \\ \quad \left(\mathop{\Large\ast}_{(k,\,(j,K))\in ks} .\ \exists e_r.\ j \Rrightarrow K[e_r] \ast \mathit{ready}(q,\ k(),\ e_r)\right) \ast \left(\mathop{\Large\ast}_{(\_,\,(j,K))\in ks'} .\ \exists v_r.\ j \Rrightarrow K[v_r]\right) \end{array}$$

$$\begin{array}{l} \mathit{ready}(q,\ e_l,\ e_r) \triangleq \forall ks,\ ks'.\ \triangleright\, \mathit{queueInv}(q,\ ks,\ ks') \mathrel{-\!\!\ast} \\ \quad e_l \precsim_\star e_r\ \langle[([\texttt{\$Fork}],\ [],\ \bot)]\rangle\ \{\mathit{queueInv}(q,\ [],\ ks \mathbin{+\!\!+} ks')\} \end{array}$$

Fig. 8. Fork case study: Internal logical definitions.

The theory *Fork* requires $task_1$ to refine $task_2$ as naturally expected. To allow `$Fork` effects in $task_1$, the refinement between $task_1$ and $task_2$ assumes the theory *Fork* itself. The later modality ▷ guards this recursive occurrence of *Fork* to facilitate the definition in Iris. The return condition asserts that both the `$Fork` effect and **`fork`** return `()`.

Recall that, according to the informal explanation of *run_fork*, the queue stores continuations that can be readily resumed. The definition of the queue invariant, the predicate *queueInv*, formalises this description. The term q represents the queue identifier. The term ks describes the contents of q. Concretely, it is a list of triples $(k,\ (j,\ K))$, where k is one of the continuations in q. This connection is captured by $\mathit{isQueue}(q,\ ks.1)$, which asserts q contains the collection of continuations in ks. Because the continuation k is created by a running task that performs an effect, there must be a corresponding task on the specification side that k refines. The thread identifier j and the context K are used to describe the state of this task: it is an expression e_r such that $j \Rrightarrow e_r$. Finally, the term ks' in *queueInv* represents the tasks on the specification side that have terminated and that were once used in the description of continuations in ks.

During the handling of a `$Fork` effect with payload $task'_1$, the specification side is a program of the form $K_r[\textbf{fork}\ (task'_2())]$. After the application of Rule FORK-R-★, the newly obtained resource $i \Rrightarrow task'_2()$ is immediately traded, via Rule THREAD-SWAP, for a thread-pool resource $j \Rrightarrow K'[K_r[()]]$ describing the main thread. This resource is used to show the queue invariant is preserved after pushing k. The proof then carries on with *run* $task'_1$ on the implementation side and the specification side correctly adjusted to $task'_2()$. Upon termination of a task, if the queue is non-empty, a continuation k is taken from the queue. At this point, Rule LOGICAL-FORK is used in conjunction with the thread-pool resource and the *ready* assumption retrieved from the queue invariant, thus concluding the proof.

5.1.3 Async/await. We prove a similar refinement statement for an *asynchronous-computation* library offering *async* and *await* effects [15, 17]. The implementation run_coop_1, which appears in Figure 9 is the translation to λ-blaze of the OCaml implementation from Dolan et al. [17, Fig. 1].

In addition to a queue of ready continuations, run_coop_1 also stores continuations in *promises*. Abstractly, a promise p represents the result of a running task. The continuations in p wait for this result. The continuations can be readily resumed once the task finishes, so they are transferred to the queue. We show that run_coop_1 refines run_coop_2, which offers a more direct implementation of async using **`fork`** instead of storing continuations in a queue. The implementation of await by run_coop_2 still relies on a handler and also uses promises to manage waiting threads. To avoid races, run_coop_2 uses locks to protect accesses to promises.

The proof that run_coop_1 refines run_coop_2 relies on a queue invariant similar to *queueInv*

```
run_coop₁ ≜ fun main.
  let effect Coop in
  let q = new_queue() in
  let next = fun _.
    if empty q then () else (pop q) ()
  in
  let run = rec run p task. handle task() with
     | effect Coop request, k ⇒
         match request with
         | inl task′ ⇒
            let p′ = ref (inr []) in
            push q (fun _. k p′) ; run p′ task′
         | inr p′ ⇒ match !p′ with
            | inl x ⇒ k x
            | inr ks ⇒ p′ ← inr (k :: ks) ; next()
     | y ⇒
         let (inr ks) = !p in p ← inl y;
         iter (fun k. push q (fun _. k y)) ks;
         next()
  in
  let async = fun task′. perform Coop (inl task′) in
  let await = fun p′. perform Coop (inr p′) in
  let p = ref (inr []) in
  run p (fun _. main async await)
```

```
run_coop₂ ≜ fun main.
  let effect Await in
  let new_promise = fun _.
    (ref (inr []), new_lock())
  in
  let run = rec run p task.
    handle task() with
    | effect Await p′, k ⇒
        acquire p′.2; match !p′.1 with
        | inl x ⇒ release p′.2; k x
        | inr ks ⇒ p′.1 ← inr (k :: ks);
          release p′.2
    | y ⇒ acquire p.2;
        let (inr ks) = !p.1 in
        p.1 ← inl y; release p.2;
        iter (fun k. fork (k y)) ks
  in let async = fun task′.
    let p′ = new_promise() in
    fork (run p′ task′); p′
  in
  let await = fun p′. perform Await p′ in
  let p = new_promise() in
  run p (fun _. main async await)
```

Fig. 9. Async/await implementations.

(Figure 8). Other logical definitions used internally in the proof are adapted from de Vilhena and Pottier [15] (who carry out the verification of a similar asynchronous library in a unary setting in Iris). The complete list of definitions is included in Appendix C.2.1.

Finally, we also prove the negative result that *run_coop*$_1$ *does not* refine the following handler-free implementation of async and await by *run_coop*$_3$, where async is implemented using **fork** and await is implemented by *busy waiting*:

```
deadlock ≜ fun async await.
  let r = ref (inl ()) in
  let p = async (rec f ().
    match !r with
    | inl () ⇒ async (fun _. ()); f ()
    | inr p ⇒ await p
  ) in
  r ← inr p;
  await p
```

```
run_coop₃ ≜ fun main.
  let async = fun task.
    let p = ref (inl ()) in
    fork (let y = task() in p ← (inr y)); p
  in
  let await = rec await p.
    match !p with inl () ⇒ await p | inr v ⇒ v
  in
  main async await
```

The key idea is to adapt the *deadlock* example from de Vilhena [14, Fig. 4.2] to exhibit a client that terminates when using the handler-based library but diverges otherwise. [14] In short, the client *deadlock* creates a cyclic dependency between *p* and itself. With the implementation of async and await by *run_coop*$_3$, when *deadlock* executes the final instruction *await p*, it diverges, because *p*

[14] The precise statement is included in Appendix C.2.2 using the formally defined predicates *terminates* and *diverges*.

```
run_nd_pure ≜ fun main.
  handle main() with
  | effect $ND request, k ⇒
      match request with
      | inl (t₁, t₂) ⇒ k t₁ ++ k t₂
      | inr () ⇒ []
  | y ⇒ [y]
```

```
run_nd_rand ≜ fun main. handle main() with
  | effect $ND request, k ⇒
      match request with
      | inl (t₁, t₂) ⇒ let b = ref true in
        fork (b ← false); if !b then k t₁ else k t₂
      | inr () ⇒ (rec f (). f())()
  | y ⇒ y
```

Fig. 10. Non-determinism handlers.

is never fulfiled. With the implementation of async and await by *run_coop*$_1$, on the other hand, when *deadlock* executes the final instruction *await p*, it is captured in a continuation and stored in *p*. The internal queue managed by *run_coop*$_1$ becomes empty, so it terminates.

5.2 Algebraic effects: Haskell-like non-determinism

In this case study, we are interested in evaluating how relational theories can be used to reason about *algebraic effects* [37]. As an illustration, we consider the pair of constructs `or` and `fail`, where e_1 `or` e_2 models the functionality to non-deterministically run e_1 or e_2, and `fail` represents a failed execution path. Here is how `or` and `fail` are written in λ-blaze (with a global effect `$ND`): e_1 `or` e_2 $\triangleq$ `(`**`perform`** `$ND (`**`inl`** `(`**`fun`** `_.` e_1`,` **`fun`** `_.` e_2`)))()` and `fail` $\triangleq$ **`perform`** `$ND (`**`inr`** `()).`

Because λ-blaze has strict evaluation, the construct e_1 `or` e_2 performs a `$ND` effect with thunked versions of e_1 and e_2. After one of them is non-deterministically chosen by the handler, its execution is forced with `()`. The construct `fail` just performs a `$ND` effect.

Plotkin and Pretnar [39] show that `or` and `fail` can be described by the *algebraic theory* of a *monoid*: e_1 `or` $(e_2$ `or` $e_3)$ = $(e_1$ `or` $e_2)$ `or` e_3 and e `or fail` = `fail or` $e = e$. Such an algebraic theory can be used not only to reason about `or` and `fail` but also to state the correctness of an effect handler providing an implementation of these effects. In short, a handler is correct when the separate handling of two programs, that are equal according to the algebraic theory, yields equal results.

This equational correctness criterion suits a pure setting well, but precludes its application to cases where the effects `or` and `fail` are implemented using native non-determinism. For example, consider the two handler implementations that appear in Figure 10. The implementation of e_1 `or` e_2 provided by *run_nd_pure* uses a list to collect the results of returning e_1 and the results of returning e_2. Paths signalled by `fail` are not added to this list. [15] The implementation of e_1 `or` e_2 provided by *run_nd_rand* chooses the expression to run by reading a location b that holds `true` initially but is non-deterministically set to `false` by a forked thread. [16] The handling of `fail` diverges.

The correctness criterion of Plotkin and Pretnar [39] can be used to justify *run_nd_pure* provides a correct implementation of `or` and `fail` with respect to their algebraic theory. However, *run_nd_rand* falls out of the scope of their approach. Using relational theories of blaze, it is possible to introduce a similar handler-correctness criterion applicable to both *run_nd_pure* and *run_nd_rand*:

$$runNdCorrect(run) \triangleq \begin{array}{l}\forall main_1,\\ main_2,\, \mathcal{L}.\end{array} \begin{cases} main_1() \lesssim_\star main_2()\ \langle([\texttt{\$ND}],\ [\texttt{\$ND}],\ Nd) :: \mathcal{L}\rangle\ \{=\} \twoheadrightarrow \\ run\ main_1 \lesssim_\star run\ main_2\ \langle([\texttt{\$ND}],\ [\texttt{\$ND}],\ \bot) :: \mathcal{L}\rangle\ \{=\} \end{cases}$$

The predicate $runNdCorrect(run)$ asserts the correctness of a handler *run* with respect to a relational theory *Nd*. It states that the handling of two handlees $main_1$ and $main_2$ yields the same

[15] This implementation is similar to the list instance of `MonadPlus`'s constructs `mplus` and `mzero` from Haskell [11].

[16] This implementation is originally given by Frumin et al. [20, §6.4].

results assuming $main_1$ and $main_2$ are related under the theory $\mathcal{T}$ for $ND. The handler *run* cannot itself rely on $ND and it must not intercept other effects related by $\mathcal{L}$. The theory *Nd* enables algebraic reasoning about `or` and `fail`, written as the sum of several theories expressing their algebraic laws:

$$
\begin{array}{l}
Nd \triangleq Assoc_1 \oplus Assoc_2 \oplus Unit_1 \oplus Unit_2 \oplus Unit_3 \oplus Unit_4 \oplus Refl_1 \oplus Refl_2 \\
Assoc_1(e_{11}\ \texttt{or}\ (e_{12}\ \texttt{or}\ e_{13}),\ (e_{21}\ \texttt{or}\ e_{22})\ \texttt{or}\ e_{23},\ Q) \triangleq \Box Q(e_{11}, e_{21}) * \Box Q(e_{12}, e_{22}) * \Box Q(e_{13}, e_{23}) \\
Assoc_2((e_{11}\ \texttt{or}\ e_{12})\ \texttt{or}\ e_{23},\ e_{21}\ \texttt{or}\ (e_{22}\ \texttt{or}\ e_{23}),\ Q) \triangleq \Box Q(e_{11}, e_{21}) * \Box Q(e_{12}, e_{22}) * \Box Q(e_{13}, e_{23}) \\
Unit_1(e_1\ \texttt{or}\ \texttt{fail},\ e_2,\ Q) \triangleq Unit_2(\texttt{fail}\ \texttt{or}\ e_1,\ e_2,\ Q) \triangleq \Box Q(e_1, e_2) \\
Unit_3(e_1,\ e_2\ \texttt{or}\ \texttt{fail},\ Q) \triangleq Unit_4(e_1,\ \texttt{fail}\ \texttt{or}\ e_2,\ Q) \triangleq \pounds 1 * \Box Q(e_1, e_2) \\
Refl_1(e_{11}\ \texttt{or}\ e_{12},,\ e_{21}\ \texttt{or}\ e_{22},\ Q) \triangleq \Box Q(e_{11}, e_{21}) * \Box Q(e_{12}, e_{22}) \qquad Refl_2(\texttt{fail},\ \texttt{fail},\ \_) \triangleq \texttt{True}
\end{array}
$$

The theory $Assoc_1$ captures the associativity of `or`. The return condition Q is used to express the condition that the relation holds up a relation of the subexpressions. The other theories are written in a similar style, except for $Unit_3$ and $Unit_4$, which charge the user one *later credit* [42], part of Iris's machinery to avoid cyclic proofs. Without the charge of one later credit, the theory $Unit_4$ could be used, for example, to relate a terminating e_1 to a diverging e_2 such as $(\texttt{\textbf{rec}}\ f\ ().\ \texttt{fail}\ \texttt{or}\ f())()$. We prove this claim in our Rocq formalisation [1].

Unlike Plotkin and Pretnar [39]'s algebraic theory, which is closed under congruence rules, *Nd* supports only symmetry and reflexivity, but not transitivity. This is due to a known limitation of step-indexed relational logics [8, 23]. *Nd* is however sufficiently expressive to relate non-trivial examples of handlees (Appendix C.3). Moreover, using the *runNdCorrect* correctness criterion it is possible to prove both *run_nd_pure* and *run_nd_rand* are correct with respect to *Nd*: both *runNdCorrect*(*run_nd_pure*) and *runNdCorrect*(*run_nd_rand*) hold.

6 Related Work

To our knowledge, this is the first work to introduce a relational separation logic for effect handlers. In the following paragraphs, we discuss work within closely related topics.

Relational reasoning about effect handlers. Building on the notion of *algebraic effects*, where an effect is described by an equational theory, Plotkin and Pretnar [39] introduce the notion of correctness of handlers whereby the handler of an effect is correct if the handler *respects* the equations describing this effect. This equational approach is well-suited to strictly functional programs but has never been extended to languages with concurrency and mutable state. We follow a different approach, namely relational separation logic, but take inspiration from equational reasoning to introduce a notion of handler correctness that supports these features (§5.2).

Biernacki et al. [5] introduce binary logical relations for effect handlers. Their *biorthogonal-closed* [36] style of relations inspires similar definitions by several authors [7, 35, 53]. Such binary logical relations can be used as an intermediary step in the proof of contextual refinement. Biernacki et al. [5] explore this approach to establish interesting examples of refinement, including a statement about the *ask effect* [5, §4.1], similar to the one studied in §4.2.3, and one about the *state effect* [5, §4.2], ported to our system in our Rocq formalisation.

The main limitation of the logical-relations approach as a foundation for relational reasoning is the lack of high-level reasoning principles. To carry out refinement proofs, these works [5, 7, 35, 53] reason directly in terms of the definition of the logical relations. In contrast, our relational separation logic offers a notion of refinement that, thanks to a collection of reasoning rules, is never unfolded.

Relational reasoning about continuations in Iris. Timany and Birkedal [48] devise binary logical relations for programs that manipulate *undelimited continuations* captured by `callcc`. They use these logical relations to verify multiple challenging examples of refinement, one of which is similar to the fork library we verify in §5.1. Namely, they show that the `callcc`-based implementation of fork written in a sequential language refines the **`fork`** construct of a language with native cooperative concurrency. The refinement therefore relates programs written in different languages. To carry out this refinement, they devise *cross-language* logical relations. Like previous works exploiting logical relations, and unlike our work, the lack of high-level reasoning rules necessitates the proofs to be carried out at the level of Iris's weakest precondition *wp*, which, in their setting, is inconvenient because, in the presence of `callcc`, the bind rule for their version of *wp* is unsound. They mitigate this inconvenience by introducing the *context-local weakest precondition*, which admits the bind rule for the price of reduced support for `callcc`. (Although notions of weakest precondition that admit the bind rule while keeping support for `callcc` exist [14, §6.3.2].)

Relational theories. de Vilhena and Pottier [15] introduce *protocols* as a mechanism to allow modular reasoning about programs with effect handlers in a unary setting. The domain of relational theories *iThy* (§4.1.1) can be seen as a generalisation to a binary setting of the domain of protocols [15, Fig. 4] $(\mathit{Val} \to (\mathit{Val} \to \mathit{iProp})) \to \mathit{iProp}$. An immediate generalisation is to replace *Val* with a binary type $\mathit{Val} \times \mathit{Val}$. A more subtle generalisation is to subsequently replace *Val* with *Expr*. This is needed to allow relations between effectful and non-effectful expressions. For the same reason [5] introduce a similar domain of *semantic effects* **Eff** [5, §3.2], defined, approximately, as a predicate of type $(\mathit{Expr}^2 \times (\mathit{Expr}^2 \to \mathit{SProp})) \to \mathit{SProp}$, where *SProp* is a type of *step-indexed assertions*. Allain et al. [2] introduce a domain of protocols in Iris that coincides exactly with *iThy*. However, their focus is on the proof of correctness of compiler optimisations in a fragment of OCaml without handlers. Consequently, they derive a notion of simulation that admits a general bind rule with no conditions on contexts. To validate this rule, their simulation relation, by default, closes protocols under arbitrary evaluation contexts. In baze, we opt for a more flexible context-local reasoning principle where the user can choose when and under-which contexts to close theories via the context-closure operator (§4.1.2). This flexibility is key in the layered construction of blaze.

Reasoning about dynamic labels. de Vilhena and Pottier [16] introduce TesLogic, a unary logic for effect handlers with dynamic labels in a language similar to λ-blaze. The model of blaze is inspired by how TesLogic builds on top of Hazel [15], a unary logic for handlers which, like baze, lacks the abstraction principles for dynamic labels. The rules of TesLogic [14, Fig. 7.2], however, differ from the ones in blaze in key ways: whereas they have an explicit rule to reason about handlers, Rule EXHAUSTION-★ can be applied to contexts without handlers; and, whereas their bind rule is limited to neutral contexts, Rule BIND-★ can be applied to contexts with handlers.

Flexible relational reasoning rules for concurrency. Like Vindum et al. [52], we notice limitations of the reasoning rules for concurrency provided by standard relational separation logic. We have already compared the differences between our approaches in §5.1.1. In short, we both rely on *ghost thread-pool* assertions $i \Mapsto e$ describing the state of thread i on the specification side. However, while their rules require the user to explicitly execute e, our rules use the assertions $i \Mapsto e$ merely as tokens that can be forged, spent, or exchanged during the construction of a proof.

7 Future Work

Limitations of our current framework indicate directions for future work. An important deficiency is the lack of a type system. In a relational setting, a type system is particularly useful, because it offers a syntax-directed approach to prove refinements of the form $e \lesssim e$. In the future, we would like to remedy this deficiency by extending λ-blaze with a type system for handlers with dynamic labels, such as Tes [16]. It would be interesting to see how blaze could be used to devise a binary-logical-relations interpretation of Tes and whether the resulting interpretation could be used to show Tes enforces abstraction principles for programming with handlers, such as the *absence of accidental handling* [53, 54]. Another current limitation of our framework is the lack of support for Iris's invariants [25, 26]. Invariants are *not* needed in our case studies, but logical relations in Iris typically use invariants in the interpretation of reference types [49]. Although relational logics with support for Iris's invariants exist [19, 20], they do not support control effects. Recent work demonstrates how to support Iris's invariants in a relational logic for exceptions with reasoning rules for opening invariants that are modular with respect to the notion of atomicity [3]. In the future, we would like to add support for invariants following their approach. Finally, we would like to explore alternative definitions of the model. We suspect the later modality in the definition of baze's refinement relation can be eliminated by using an alternative method for constructing recursive definitions, namely Iris's *greatest fixpoint* operator [31, 46]. Following recent work [2, 3, 21], we would also like to investigate the implications of generalising the type of postconditions to a predicate on pairs of expressions. We believe this generalisation could improve context-local reasoning by allowing our bind rules (BIND and BIND-★) to focus on pairs of expressions that do not necessarily terminate synchronously.

References

[1] 2025. Rocq formalisation. Submitted as *Supplemental Material.*

[2] Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. In *Principles of Programming Languages (POPL)*, Vol. 9. ACM Press. https://doi.org/10.1145/3704915

[3] Anonymous Authors. 2025. Representation Independence for ML-Style Exceptions. Paper submitted to POPL 2026.

[4] Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10, 4 (2014). https://arxiv.org/pdf/1306.6316.pdf

[5] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 8:1–8:30. https://doi.org/10.1145/3158096

[6] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 6:1–6:28. https://www.ii.uni.wroc.pl/~mpirog/papers/biernacki-al-popl19.pdf

[7] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

[8] Lars Birkedal and Aleš Bizjak. 2012. A note on the transitivity of step-indexed logical relations. (Nov. 2012). https://abizjak.github.io/documents/notes/step-indexed-transitivity.pdf

[9] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming* 30 (2020), e8. https://ps.informatik.uni-tuebingen.de/publications/brachthaeuser19effekt-revision.pdf

[10] Edwin C. Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *International Conference on Functional Programming (ICFP)*. 133–144. https://www.type-driven.org.uk/edwinb/papers/effects.pdf

[11] Haskell Community. 2023. Alternative and MonadPlus. https://en.wikibooks.org/wiki/Haskell/Alternative_and_MonadPlus

[12] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science, Vol. 4709)*. Springer, 266–296. https://homepages.inf.ed.ac.uk/slindley/papers/links-fmco06.pdf

[13] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Transactions on Programming Languages and Systems* 31, 2 (Feb. 2009), 1–31. https://doi.org/10.1145/1462166.1462167

[14] Paulo Emílio de Vilhena. 2022. *Proof of Programs with Effect Handlers.* Ph. D. Dissertation. Université Paris Cité. https://inria.hal.science/tel-03891381

[15] Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021). https://doi.org/10.1145/3434314

[16] Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 13990)*. Springer, 225–252. https://doi.org/10.1007/978-3-031-30044-8_9

[17] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming (TFP) (Lecture Notes in Computer Science, Vol. 10788)*. Springer, 98–117. https://kcsrk.info/papers/system_effects_feb_18.pdf

[18] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *TCS* 411, 51-52 (2010), 4379–4398. doi:10.1016/J.TCS.2010.09.021

[19] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Logic in Computer Science (LICS)*. 442–451. https://iris-project.org/pdfs/2018-lics-reloc-final.pdf

[20] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* 17, 3 (2021). https://arxiv.org/abs/2006.13635v3

[21] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

[22] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *International Workshop on Type-Driven Development (TyDe@ICFP)*. 15–27. https://homepages.inf.ed.ac.uk/slindley/papers/links-effect.pdf

[23] Chung-Kil Hur, Derek Dreyer Georg, Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *Principles of Programming Languages (POPL)*. ACM Press, 59–72. https://doi.org/10.1145/2103656.2103666

[24] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *International Conference on Functional Programming (ICFP)*. 256–269. https://iris-project.org/pdfs/2016-icfp-iris2-final.pdf

[25] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

[26] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages (POPL)*. 637–650. https://plv.mpi-sws.org/iris/paper.pdf

[27] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. 94–105. doi:10.1145/2804302.2804319

[28] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

[29] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 696–723. https://iris-project.org/pdfs/2017-esop-iris3-final.pdf

[30] Robert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*. https://cs.au.dk/~birke/papers/ipm-conf.pdf

[31] Robbert Krebbers, Luko van der Maas, and Enrico Tassi. 2025. Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic. In *Interactive Theorem Proving (ITP)*. https://robbertkrebbers.nl/research/articles/iris_inductive.pdf

[32] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Workshop on Mathematically Structured Functional Programming (MSFP)*, Vol. 153. 100–126. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/paper-20.pdf

[33] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. 2025. The OCaml system: Documentation and user's manual. https://ocaml.org/manual/5.3/index.html

[34] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Principles of Programming Languages (POPL)*. https://homepages.inf.ed.ac.uk/slindley/papers/frankly.pdf

[35] Craig McLaughlin. 2020. *Relational reasoning for effects and handlers*. Ph. D. Dissertation. University of Edinburgh, UK. doi:10.7488/ERA/537

[36] Andrew Pitts and Ian Stark. 1999. *Operational reasoning for functions with local state*. Cambridge University Press, 227—-274.

[37] Gordon D. Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *Logic in Computer Science (LICS)*. 118–129. https://homepages.inf.ed.ac.uk/gdp/publications/Logic_Algebraic_Effects.pdf

[38] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 5502)*. Springer, 80–94. https://homepages.inf.ed.ac.uk/gdp/publications/Effect_Handlers.pdf

[39] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec. 2013). https://lmcs.episciences.org/705

[40] Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. In *Mathematical Foundations of Programming*

Semantics (Electronic Notes in Theoretical Computer Science, Vol. 319). Elsevier, 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

[41] Alex Simpson and Niels Voorneveld. 2019. Behavioural Equivalence via Modalities for Algebraic Effects. *ACM Transactions on Programming Languages and Systems* 42 (Nov. 2019). doi:10.1145/3363518

[42] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3547631

[43] Wenhao Tang, Daniel Hillerström, Sam Lindley, and Garrett J. Morris. 2024. Soundly Handling Linearity. In *Principles of Programming Languages (POPL)*, Vol. 8. ACM Press, 1600—-1628. doi:10.1145/3632896

[44] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types, In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). *Proceedings of the ACM on Programming Languages* 9. doi:10.1145/3720476

[45] The Iris Team. 2024. The Iris Documentation. https://plv.mpi-sws.org/iris/appendix-4.3.pdf.

[46] The Iris Team. 2025. Iris Fixpoint Operators. https://gitlab.mpi-sws.org/iris/iris/-/blob/master/iris/bi/lib/fixpoint_mono.v

[47] The Rocq Prover development team. 2025. *The Rocq Prover*. https://rocq-prover.org/

[48] Amin Timany and Lars Birkedal. 2019. Mechanized Relational Verification of Concurrent Programs with Continuations. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 105:1–105:28. https://doi.acm.org/10.1145/3341709

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

[50] Orpheas van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. In *Principles of Programming Languages (POPL)*, Vol. 9. ACM Press, Article 5, 29 pages. doi:10.1145/3704841

[51] Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90. https://cs.au.dk/~birke/papers/2021-ms-queue-final.pdf

[52] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *Certified Programs and Proofs (CPP)*. ACM Press, 100—-115. https://doi.org/10.1145/3497775.3503689

[53] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. In *Principles of Programming Languages (POPL)*, Vol. 3. ACM Press, 5:1–5:29. https://www.cs.cornell.edu/andru/papers/tunnel-eff/tunnel-eff.pdf

[54] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting blame for safe tunneled exceptions. In *Programming Language Design and Implementation (PLDI)*. 281–295. https://www.cs.cornell.edu/andru/papers/exceptions/exceptions-pldi16.pdf

A Language

$$
\begin{array}{rl}
e ::= & v \mid x \mid e\,e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid (e,\, e) \\
\mid & e.1 \mid e.2 \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \\
\mid & \begin{array}{l}\mathtt{match}\ e\ \mathtt{with} \\ \mathtt{|}\ \mathtt{inl}\ x \Rightarrow e \\ \mathtt{|}\ \mathtt{inr}\ y \Rightarrow e\end{array} \quad \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e \\
\mid & \mathtt{let\,effect}\ \mathsf{E}\ \mathtt{in}\ e \mid \mathtt{perform}\ \mathsf{E}\ e \\
\mid & \begin{array}{l}\mathtt{handle}\ e\ \mathtt{with} \\ \mathtt{|\,effect}\ \mathsf{E}\ x,\ \mathtt{rec}^{?}\ k\ \mathtt{as\,multi}^{?} \Rightarrow e \\ \mathtt{|}\ y \Rightarrow e\end{array} \\
\mid & \mathtt{ref}\ e \mid {!e} \mid e \leftarrow e \mid \mathtt{fork}\ e \mid \mathtt{cas}\,(e, e, e) \\
\mid & \begin{array}{l}\mathtt{handle}\ e\ \mathtt{with} \\ \mathtt{|\,effect}\ \$\mathsf{E}\ x,\ \mathtt{rec}^{?}\ k\ \mathtt{as\,multi}^{?} \Rightarrow e \\ \mathtt{|}\ y \Rightarrow e\end{array} \\
\mid & \mathtt{perform}\ \$\mathsf{E}\ e
\end{array}
$$

$$
\begin{array}{rl}
v ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \mathtt{rec}\,f\,x.\ e \mid (v,\, v) \\
\mid & \mathtt{inl}\ v \mid \mathtt{inr}\ v \mid \ell \mid \mathtt{cont}\ \ell\ K \mid \mathtt{kont}\ K \\
K ::= & [\,] \mid e\,K \mid K\,v \mid \mathtt{let}\ x = K\ \mathtt{in}\ e \\
\mid & \mathtt{let}\ x = v\ \mathtt{in}\ K \mid (e,\, K) \mid (K,\, v) \\
\mid & K.1 \mid K.2 \mid \mathtt{if}\ K\ \mathtt{then}\ e\ \mathtt{else}\ e \\
\mid & \begin{array}{l}\mathtt{match}\ K\ \mathtt{with} \\ \mathtt{|}\ \mathtt{inl}\ x \Rightarrow e \\ \mathtt{|}\ \mathtt{inr}\ y \Rightarrow e\end{array} \quad \mid \mathtt{inl}\ K \mid \mathtt{inr}\ K \\
\mid & \mathtt{perform}\ \$\mathsf{E}\ K \\
\mid & \begin{array}{l}\mathtt{handle}\ K\ \mathtt{with} \\ \mathtt{|\,effect}\ \$\mathsf{E}\ x,\ \mathtt{rec}^{?}\ k\ \mathtt{as\,multi}^{?} \Rightarrow e \\ \mathtt{|}\ y \Rightarrow e\end{array} \\
\mid & \mathtt{ref}\ K \mid {!K} \mid e \leftarrow K \mid K \leftarrow v \\
\mid & \mathtt{cas}\,(e, e, K) \mid \mathtt{cas}\,(e, K, v) \mid \mathtt{cas}\,(K, v, v)
\end{array}
$$

(a) Syntax of values, expressions, and evaluation contexts. (Runtime terms are displayed in gray.)

EFFECT

$$\frac{\{\vec{e}[i \mapsto K[\mathtt{let\,effect}\ \mathsf{E}\ \mathtt{in}\ e]];\ \sigma;\ \delta\} \quad \$\mathsf{E} \notin \delta}{\{\vec{e}[i \mapsto K[e\{\$\mathsf{E}/\mathsf{E}\}]];\ \sigma;\ \delta \uplus \{\$\mathsf{E}\}\}} \longrightarrow$$

FORK

$$\frac{\{\vec{e}[i \mapsto K[\mathtt{fork}\ e]];\ \sigma;\ \delta\} \quad n = |\vec{e}|}{\{\vec{e}[i \mapsto K[()],\ n \mapsto e];\ \sigma;\ \delta\}} \longrightarrow$$

ALLOC

$$\frac{\{\vec{e}[i \mapsto K[\mathtt{ref}\ v]];\ \sigma;\ \delta\} \quad \ell \notin \sigma}{\{\vec{e}[i \mapsto K[\ell]];\ \sigma[\ell \mapsto v];\ \delta\}} \longrightarrow$$

PURE

$$\frac{e_1 \rightarrow_{\mathrm{p}} e_2 \quad \{\vec{e}[i \mapsto K[e_1]];\ \sigma;\ \delta\}}{\{\vec{e}[i \mapsto K[e_2]];\ \sigma;\ \delta\}} \longrightarrow$$

HANDLE-OS

$$\frac{\begin{array}{c} H = \mathtt{handle}\ [\,]\ \mathtt{with\,effect}\ \$\mathsf{E}\ x,\ \mathtt{rec}^{?}\ k \Rightarrow h\ \mathtt{|}\ y \Rightarrow r \\ \$\mathsf{E} \notin \mathscr{L}(K') \qquad \ell \notin \sigma \qquad \sigma' = \sigma[\ell \mapsto \mathsf{true}] \\ K'' = \mathit{if}\ \mathit{deep}(H)\ \mathit{then}\ H[K']\ \mathit{else}\ K' \qquad w = \mathtt{cont}\ \ell\ K'' \end{array}}{\{\vec{e}[i \mapsto K[H[K'[\mathtt{perform}\ \$\mathsf{E}\ v]]]];\ \sigma;\ \delta\} \longrightarrow \{\vec{e}[i \mapsto K[h\{v/x, w/k\}]];\ \sigma';\ \delta\}}$$

(b) Operational rules.

BETA

$$(\mathtt{rec}\,f\,x.\ e)\ v \rightarrow_{\mathrm{p}} e\{(\mathtt{rec}\,f\,x.\ e)/f, v/x\}$$

MULTI-SHOT

$$(\mathtt{kont}\ K)\ v \rightarrow_{\mathrm{p}} K[v]$$

HANDLE-MS

$$\frac{\begin{array}{c} H = \mathtt{handle}\ [\,]\ \mathtt{with\,effect}\ \$\mathsf{E}\ x,\ \mathtt{rec}^{?}\ k\ \mathtt{as\,multi} \Rightarrow h\ \mathtt{|}\ y \Rightarrow r \\ \$\mathsf{E} \notin \mathscr{L}(K) \qquad K' = \mathit{if}\ \mathit{deep}(H)\ \mathit{then}\ H[K]\ \mathit{else}\ K \end{array}}{H[K[\mathtt{perform}\ \$\mathsf{E}\ v]] \rightarrow_{\mathrm{p}} h\{v/x, \mathtt{kont}\ K'/k\}}$$

(c) Pure-reduction rules.

Fig. 11. Syntax and semantics of λ-blaze.

B Logic

B.1 blaze

MONOTONICITY-GEN-★

$$\frac{e_l \precsim_\star e_r \, \langle \mathcal{L} \rangle \, \{R\} \qquad \mathcal{L} \sqsubseteq_\star \mathcal{M} \qquad \Box_m \forall v_l, v_r.\, R(v_l, v_r) \twoheadrightarrow S(v_l, v_r)}{e_l \precsim_\star e_r \, \langle \bigcirc_m \mathcal{M} \rangle \, \{S\}}$$

EXHAUSTION-GEN-★

$$\frac{\begin{array}{c} \mathscr{L}(K_l) \subseteq ls_l \qquad \mathscr{L}(K_r) \subseteq ls_r \\ e_l \precsim_\star e_r \, \langle \mathcal{M} \rangle \, \{R\} \qquad \mathcal{M} = (ls_l, ls_r, \mathcal{T}) :: \mathcal{L} \qquad \mathcal{N} = (ls_l, ls_r, \mathcal{F}) :: (\bigcirc_m \mathcal{L}) \\ \wedge \begin{cases} \Box_m \forall v_l, v_r.\, R(v_l, v_r) \twoheadrightarrow K_l[v_l] \precsim_\star K_r[v_r] \, \langle \mathcal{N} \rangle \, \{S\} \\ \Box_m \forall e_l', e_r'.\, ((ls_l, ls_r) \Downarrow\!\upharpoonright \mathcal{T}) \blacktriangleleft e_l' \precsim_\star e_r' \, \langle \mathcal{M} \rangle \, \{R\} \twoheadrightarrow K_l[e_l'] \precsim_\star K_r[e_r'] \, \langle \mathcal{N} \rangle \, \{S\} \end{cases} \end{array}}{K_l[e_l] \precsim_\star K_r[e_r] \, \langle \mathcal{N} \rangle \, \{S\}}$$

$$\begin{aligned} \bigcirc_m [] &\triangleq [] \\ \bigcirc_m (ls_l, ls_r, \mathcal{T}) :: \mathcal{L} &\triangleq (ls_l, ls_r, \bigcirc_m \mathcal{T}) :: \bigcirc_m \mathcal{L} \end{aligned}$$

Fig. 12. Generalised reasoning rules in blaze.

B.2 Soundness

Definition B.1 (Safe).

$$\mathit{safe}(e, \phi) \triangleq \forall e', \vec{e_f}, \sigma, \delta. \begin{pmatrix} \{[0 \mapsto e] ; \emptyset ; \emptyset\} \rightarrow^* \\ \quad \{[0 \mapsto e'] \uplus \vec{e_f} ; \sigma ; \delta\} \end{pmatrix} \implies \vee \begin{cases} e' \in \mathit{Val} \wedge \phi \\ \exists e''.\, \begin{array}{l} \{[0 \mapsto e'] \uplus \vec{e_f} ; \sigma ; \delta\} \rightarrow \\ \quad \{[0 \mapsto e''] \uplus \_ ; \_ ; \_\} \end{array} \end{cases}$$

Definition B.2 (Terminates).

$$\mathit{terminates}(e) \triangleq \exists v.\, \{[0 \mapsto e] ; \emptyset ; \emptyset\} \rightarrow^* \{[0 \mapsto v] \uplus \_ ; \_ ; \_\}$$

THEOREM B.3. *If* $\vdash e_l \precsim e_r \, \langle \bot \rangle \, \{\mathsf{True}\}$, *then* $\mathit{safe}(e_l, \mathit{terminates}(e_r))$.

C Case studies

C.1 State

Examples adapted from Biernacki et al. [5, §4.2]:

```
run_ask_tell ≜ fun main.
  let effect Ask in
  let effect Tell in
  let ask = fun _. perform Ask () in
  let tell = fun y. perform Tell y in
  let run_ask ≜ fun y main′. handle main′ () with effect Ask (), k ⇒ k y | z ⇒ z in
  let run_tell ≜ fun main′.
    handle main′ () with effect Tell y, k ⇒ run_ask y (fun _. k ()) | z ⇒ z
  in run_tell (fun _. run_ask 0 (fun _. main ask tell))
```

```
run_cell ≜ fun main.
  let effect Cell in
  let get = fun _. perform Cell (inl ()) in
  let set = fun y. perform Cell (inr y) in
  let run = fun main.
    handle main() with
    | effect Cell request, k ⇒ fun x.
        match request with
        | inl () ⇒ k x x
        | inr y ⇒ k () y
    | y ⇒ fun _. y
  in run (fun _. main get set) 0
```

C.2 Concurrency

$$\dfrac{i \Mapsto e_r \qquad e_l \precsim e_r\ \{\mathsf{True}\} \qquad K_l[\,()\,] \precsim e_r'\ \langle \mathcal{T} \rangle\ \{R\}}{K_l[\mathbf{fork}\ e_l] \precsim e_r'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textsc{Fork-L} \qquad \dfrac{\forall i.\ i \Mapsto e_r \mathrel{-\!\!*} e_l \precsim K_l[\,()\,]\ \langle \mathcal{T} \rangle\ \{R\}}{e_l \precsim K_l[\mathbf{fork}\ e_r]\ \langle \mathcal{T} \rangle\ \{R\}}\ \textsc{Fork-R}$$

$$\dfrac{i \Mapsto K_r[e_r] \qquad e_l \precsim e_r\ \{R\} \qquad \forall v_l, v_r.\ R(v_l, v_r) \mathrel{-\!\!*} i \Mapsto K_r[v_r] \mathrel{-\!\!*} K_l[v_l] \precsim e_r'\ \langle \mathcal{T} \rangle\ \{S\}}{K_l[e_l] \precsim e_r'\ \langle \mathcal{T} \rangle\ \{S\}}\ \textsc{Logical-Fork}$$

$$\dfrac{i \Mapsto K[e_r] \qquad \forall j, K'.\ j \Mapsto K'[e_r'] \mathrel{-\!\!*} e_l \precsim e_r\ \{v_l\ \_.\ \exists v_r'.\ j \Mapsto K'[v_r'] \ast R(v_l, v_r')\}}{e_l \precsim e_r'\ \langle \mathcal{T} \rangle\ \{R\}}\ \textsc{Thread-Swap}$$

(a) Rules in baze.

Fig. 13. Reasoning rules for concurrency.

$$\begin{aligned}
&\mathit{runCoopSpec} \triangleq \Box\, \forall \mathit{main}_1, \mathit{main}_2. \\
&\quad \left(\begin{array}{l} \Box\, \forall \mathit{async}_1, \mathit{async}_2, \mathit{await}_1, \mathit{await}_2, \mathit{promise}, \mathcal{L}. \\ \mathit{asyncSpec}(\mathit{async}_1, \mathit{async}_2, \mathit{promise}, \mathcal{L}) \mathrel{-\!\!*} \\ \mathit{awaitSpec}(\mathit{await}_1, \mathit{await}_2, \mathit{promise}, \mathcal{L}) \mathrel{-\!\!*} \\ \mathit{main}_1\ \mathit{async}_1\ \mathit{await}_1 \precsim_\star \mathit{main}_2\ \mathit{async}_2\ \mathit{await}_2\ \langle \mathcal{L} \rangle\ \{\mathsf{True}\} \end{array} \right) \mathrel{-\!\!*} \begin{array}{l} \mathit{run\_coop}_1\ \mathit{main}_1 \precsim_\star \\ \mathit{run\_coop}_2\ \mathit{main}_2\ \{\mathsf{True}\} \end{array}
\end{aligned}$$

$$\begin{aligned}
&\mathit{asyncSpec}(\mathit{async}_1, \mathit{async}_2, \mathit{promise}, \mathcal{L}) \triangleq \Box\, \forall \mathit{task}_1, \mathit{task}_2, S. \\
&\quad \mathit{task}_1\,() \precsim_\star \mathit{task}_2\,()\ \langle \mathcal{L} \rangle\ \{v_l\, v_r.\ \Box\, S(v_l, v_r)\} \mathrel{-\!\!*} \\
&\quad \mathit{async}_1\ \mathit{task}_1 \precsim_\star \mathit{async}_2\ \mathit{task}_2\ \langle \mathcal{L} \rangle\ \{p_1\, p_2.\ \Box\, \mathit{promise}(p_1, p_2, S)\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{awaitSpec}(\mathit{await}_1, \mathit{await}_2, \mathit{promise}, \mathcal{L}) \triangleq \Box\, \forall p_1, p_2, S. \\
&\quad \mathit{promise}(p_1, p_2, S) \mathrel{-\!\!*} \mathit{await}_1\, p_1 \precsim_\star \mathit{await}_2\, p_2\ \langle \mathcal{L} \rangle\ \{v_l\, v_r.\ \Box\, S(v_l, v_r)\}
\end{aligned}$$

Fig. 14. Async/await case study: Specification.

Relational theory.

$$Coop \triangleq Async \oplus Await$$

$$\begin{aligned}&Async(\mathbf{perform}\ \mathtt{\$Coop}\ (\mathbf{inl}\ task_1),\ \mathbf{fork}\ (task_2()),\ Q) \triangleq \exists S.\\ &\quad \triangleright\, task_1() \lesssim_\star task_2()\ \langle[([\mathtt{\$Coop}],\ [\mathtt{\$Await}],\ Coop)]\rangle\ \{v_l\, v_r.\ \Box\, S(v_l,\ v_r)\} \ast\\ &\quad \forall p_1,\ p_2.\ promise(p_1,\ p_2,\ S) \mathrel{-\!\!*} Q(p_1,\ p_2)\end{aligned}$$

$$\begin{aligned}&Await(\mathbf{perform}\ \mathtt{\$Coop}\ (\mathbf{inr}\ p_1),\ \mathbf{perform}\ \mathtt{\$Await}\ p_2,\ Q) \triangleq \exists S.\\ &\quad promise(p_1,\ p_2,\ S) \ast \forall v_l,\ v_r.\ \Box\, S(v_l,\ v_r) \mathrel{-\!\!*} Q(v_l,\ v_r)\end{aligned}$$

$$promise(p_1,\ p_2,\ S) \triangleq \exists \tau.\ inMap(p_1,\ p_2,\ \tau,\ S)$$

Ghost resources.

$$token(\tau) \triangleq \boxed{\mathrm{ex}(\bullet)}^{\tau} \qquad \begin{aligned}inMap(p_1,\ p_2,\ \tau,\ S) &\triangleq \boxed{\circ\,\{(p_1,\ p_2,\ \tau) \mapsto S\}}^{map}\\ isMap(M) &\triangleq \boxed{\bullet M}^{map}\end{aligned}$$

Invariants and predicates.

$$\begin{aligned}&queueInv(q,\ ks,\ ks') \triangleq isQueue(q,\ ks.1) \ast\\ &\quad \Big(\mathop{\Huge *}_{(k,\,(j,K))\in ks}.\ \exists e_r.\ j \Rrightarrow K[e_r] \ast ready(q,\ k(),\ e_r)\Big) \ast \Big(\mathop{\Huge *}_{(\_,\,(j,K))\in ks'}.\ \exists v_r.\ j \Rrightarrow K[v_r]\Big)\end{aligned}$$

$$\begin{aligned}&promiseInv \triangleq \exists M.\ isMap(M) \ast\\ &\quad \mathop{\Huge *}_{\{(p_1,p_2,\tau)\mapsto S\}\in M}.\ \bigvee \begin{cases}\exists v_l,\ v_r. & \begin{pmatrix}p_1 \mapsto_{\mathtt{i}} \mathbf{inl}\ v_l \ast \Box\, S(v_l,\ v_r)\\ p_2 \mapsto_{\mathtt{s}} \mathbf{inl}\ v_r \ast token(\tau)\end{pmatrix}\\ \exists ks. & \begin{pmatrix}p_1 \mapsto_{\mathtt{i}} \mathbf{inr}\ ks.1\\ p_2 \mapsto_{\mathtt{s}} \mathbf{inr}\ ks.2\\ \mathop{\Huge *}_{(k_1,k_2)\in ks}.\ waiting(q,\ S,\ k_1,\ k_2)\end{pmatrix}\end{cases}\end{aligned}$$

$$\begin{aligned}&ready(q,\ e_l,\ e_r) \triangleq \forall ks,\ ks'.\ \triangleright\, promiseInvq \mathrel{-\!\!*} \triangleright\, queueInv(q,\ ks,\ ks') \mathrel{-\!\!*}\\ &\quad e_l \lesssim_\star e_r\ \langle[([\mathtt{\$Coop}],\ [\mathtt{\$Await}],\ \bot)]\rangle\ \{queueInv(q,\ [],\ ks \mathbin{+\!\!+} ks')\}\end{aligned}$$

$$waiting(q,\ S,\ k_1,\ k_2) \triangleq \forall v_l,\ v_r.\ \Box\, S(v_l,\ v_r) \mathrel{-\!\!*} ready(q,\ (k_1\, v_l),\ (k_2\, v_r))$$

Fig. 15. Async/await case study: Internal logical definitions.

C.2.1 *Async/await – Part I.*

C.2.2 *Async/await – Part II.*

Theorem C.1.

$$terminates(run\_coop_1\ deadlock)$$

Definition C.2.

$$diverges(e) \triangleq safe(e,\ \mathrm{False})$$

Theorem C.3.

$$diverges(run\_coop_3\ deadlock)$$

Corollary C.4.

$$\neg(\vdash run\_coop_1\ deadlock \lesssim_\star run\_coop_3\ deadlock\ \{\mathrm{True}\})$$

C.3 Non-determinism

Example of a derivable refinement using the relational theory *Nd*:

$$
\begin{aligned}
&(\mathbf{let}\ x = \mathtt{0}\ \mathtt{or}\ (\mathtt{1}\ \mathtt{or}\ \mathtt{2})\ \mathbf{in}\ \mathbf{if}\ (\mathtt{fail}\ \mathtt{or}\ \mathtt{true})\ \mathbf{then}\ x\ \mathtt{or}\ (x+\mathtt{1})\ \mathbf{else}\ \mathtt{fail}) \lesssim_{\star} \\
&(\mathbf{let}\ x = (\mathtt{0}\ \mathtt{or}\ \mathtt{1})\ \mathtt{or}\ \mathtt{2}\ \mathbf{in}\ x\ \mathtt{or}\ (x+\mathtt{1}))\ \langle([\mathtt{\$ND}],\ [\mathtt{\$ND}],\ \mathit{Nd}) :: \mathcal{L}\rangle\ \{=\}
\end{aligned}
$$

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009